



THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

Research Commons

<http://researchcommons.waikato.ac.nz/>

Research Commons at the University of Waikato

Copyright Statement:

The digital copy of this thesis is protected by the Copyright Act 1994 (New Zealand).

The thesis may be consulted by you, provided you comply with the provisions of the Act and the following conditions of use:

- Any use you make of these documents or images must be for research or private study purposes only, and you may not make them available to any other person.
- Authors control the copyright of their thesis. You will recognise the author's right to be identified as the author of the thesis, and due acknowledgement will be made to the author where appropriate.
- You will obtain the author's permission before publishing any material from the thesis.

Algorithmic Mapping of Software-Defined Networking Pipelines

A thesis

submitted in fulfillment

of the requirements for the degree of

Doctor of Philosophy

at

The University of Waikato

by

Christopher Lorier



THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

For Min and Hazel. Love the both of you to bits!

Abstract

Networks with a consistent software stack reduce the complexity of monitoring, testing and automation, and reduce the mental burden on operators. However, when network software is bundled with the hardware, operators face being locked into a single vendor's hardware, when they might otherwise be able to use cheaper or more suitable hardware from another vendor.

Ostensibly, Software Defined Networking (SDN) gives operators the freedom to operate hardware from different vendors using the same software stack. However, producing SDN software that controls hardware from different vendors is hampered by differences in the packet processing pipelines of the Application-Specific Integrated Circuits (ASICs) each vendor uses.

This thesis presents the design and evaluation of Shoehorn, a system for improving the portability of SDN control-plane software. Shoehorn finds mappings from virtual pipelines (defining the packet processing requirements of control-plane software), to physical pipelines (defining the packet processing pipeline of a physical device). Shoehorn improves on current approaches by ensuring that the mappings are suitable for real-time translation of control-channel instructions, by ensuring a one-to-one mapping of virtual pipeline table entries to physical pipeline table entries. This also ensures that the mappings do not significantly increase the memory usage or power consumption of the

pipelines.

This thesis evaluates Shoehorn by mapping 25 virtual pipelines, based on real SDN control-plane software for managing diverse networks, to a variety of physical pipelines, based on real hardware SDN implementations. The evaluation finds that all but 6 virtual pipelines are supported by multiple physical pipelines, and that in every case where Shoehorn could not find a mapping, it was due to a virtual table that no table in the physical pipeline could support.

Acknowledgements

I have been very fortunate to benefit from the experience of Richard Nelson, Matthew Luckie, and Marinho Barcellos as my supervisors during this PhD. Thank-you for your guidance and support.

I would also like to thank other members of the WAND research group: Brad Cowie, Richard Sanger, Dimeji Fayomi, and Florin Zaicu.

I would like to thank the network operators who graciously helped provide me with data: The Scinet team, REANNZ, NZNoG, Josh Bailey, and Marc Bruyere.

I also wish to express my gratitude to my parents: Mary and Michel, for their support, for giving me a place to stay in Hamilton, and with Hazel—but also more generally, for far more than this page could ever do justice to.

어머님, 멀리 한국에서 오셔서 저희 가족을 잘 보살펴주셔서 감사합니다. 특히, 어머님이 다인이를 잘 보살펴 주신 덕분에 저의 논문을 무사히 잘 마칠 수 있었습니다. 진심으로 감사드립니다. 그리고 아버님, 한국에서 많은 응원을 보내주셔서 감사합니다. 두 분 모두 항상 건강하시고, 행복하세요.

And—most importantly—to Min: thank-you so much for your patience and support. To say *“I couldn’t have done it without you”* feels like a cliché, but I honestly can’t think of a truer thing I have ever said.

This work was partially funded by a University of Waikato doctoral scholarship.

Contents

Abstract	iii
Acknowledgements	v
1 Introduction	1
1.1 Problem Statement	1
1.2 Contributions	3
1.3 Thesis Structure	4
2 Background	6
2.1 Software-Defined Networking	6
2.1.1 Terminology	7
2.2 OpenFlow	8
2.2.1 OpenFlow 1.0	8
2.2.2 Multi-table OpenFlow	9
2.2.3 Apply-Actions and Write-Actions	10
2.2.4 Metadata	11
2.2.5 Group Tables	11
2.2.6 OpenFlow Table Type Patterns	12
2.2.7 Table-Features Messages	13
2.2.8 Other OpenFlow Features	14

2.3	P4	15
2.3.1	P4 Architectures	16
2.3.2	Control Blocks	16
2.3.3	Extern Objects	17
2.3.4	Tables, Actions and Match Kinds	17
2.3.5	Annotations	18
2.3.6	Portable Switch Architecture	19
2.4	Other SDN Standards	20
2.4.1	Switch Abstraction Interface	20
2.4.2	P4 Integrated Network Stack	20
2.4.3	FoRCES	21
2.4.4	Broadcom OpenNSL and SDKLT	21
2.4.5	NPLang	22
3	Related Work	23
3.1	Alternative Approaches	23
3.1.1	Standardised Pipelines	23
3.1.2	Programmable Hardware	24
3.1.3	Device Drivers	25
3.1.4	NOSIX	25
3.1.5	TableVisor	26
3.2	Algorithmic Mapping of SDN Pipelines	26
3.2.1	Table Entry Growth	27
3.2.2	FlowAdapter and FlowConvertor	28
3.2.3	Sanger, Luckie and Nelson	29
3.3	Summary	32

4	Target Hardware	33
4.1	Broadcom	33
4.1.1	OF-DPA	34
4.1.2	OpenNSL	36
4.1.3	Other OpenFlow Implementations on Broadcom ASICs	36
4.2	Nvidia	38
4.3	Cisco	39
4.4	Aruba	39
4.5	SAI	40
4.6	Other OpenFlow Implementations	41
4.7	Discussion	41
4.7.1	Features	41
4.7.2	Common Table Structures	43
5	Target Controllers	45
5.1	Features of Interest	45
5.2	Production Deployments	46
5.2.1	Data Collected	47
5.2.2	Identifying Entry Classes and Match Kinds	48
5.2.3	Identifying Transactions	49
5.2.4	Production Deployments	50
5.2.5	Updates	51
5.3	Research Projects	57
5.3.1	Actions	59
5.3.2	Match Fields	60
5.3.3	Tables	61
5.3.4	Table Entries and Updates	63
5.4	Summary	64

6	Shoehorn Overview	66
6.1	Mapping	68
6.2	Packet Recirculation	69
6.2.1	Hardware Support	70
6.2.2	Metadata	70
6.2.3	Throughput	71
6.3	Aggregating Components	72
6.3.1	Cartesian Product Aggregation	72
6.3.2	Aggregating Conditionals	74
6.3.3	Aggregating Mutually Exclusive Tables	76
6.3.4	Concatenating Ternary Tables	76
6.3.5	Aggregation after Recirculation	77
6.4	Table Reordering	77
6.5	MAC Learning	79
6.6	Summary	80
7	Architectures	82
7.1	Packet Paths	82
7.2	Parser	84
7.3	Metadata	85
7.4	Action Modules	86
7.5	Counters	88
7.6	Actions	89
7.7	Conditionals	90
7.8	Match Kinds	91
7.9	Annotations	91
7.10	MAC Learning Externs	92

8	Shoehorn Mapping Algorithm	93
8.1	Stage 1: Identifying Supporting Components	94
8.2	Stage 2: Finding Mappings	96
8.2.1	Mapping Pipelines	96
8.2.2	Mapping Control Blocks	100
8.3	Stage 3: Resolving Goto Actions	109
8.4	Populating Tables	111
8.5	Discussion	112
8.5.1	Splitting Physical Control Blocks	112
8.5.2	Wide Mappings	113
9	Evaluation	114
9.1	Mapping Application	114
9.2	Physical Pipelines	115
9.2.1	Aruba	116
9.2.2	Cisco	117
9.2.3	OF-DPA	118
9.2.4	SAI	118
9.2.5	Nvidia	119
9.3	Virtual Pipelines	119
9.3.1	Implementation Details	120
9.3.2	Summary of Controllers	129
9.4	Results	129
9.4.1	Physical Pipelines	131
9.4.2	Recirculations	134
9.4.3	Run Time	135
9.5	Discussion	135
9.5.1	Likely Causes for Failure	135

9.5.2	Programming Pipelines	136
9.5.3	Ideal Hardware	137
9.5.4	Other Algorithmic Mapping Systems	138
10	Conclusion	140
10.1	Summary	140
10.2	Future Work	142
	References	144
A	Controllers	158
B	Ethics Approval	165

List of Figures

3.1 Example Software-Defined Networking (SDN) Architecture using Translation Software	26
3.2 An Example of Merging Two Virtual Tables Resulting in a Cartesian Product of Entries	27
4.1 The OF-DPA Flow Table Pipeline	34
4.2 Group Tables Used by the OF-DPA	36
5.1 Percentage of Updates to Datapaths by Entry Class	52
5.2 Maximum Number of Entries for Each Entry Class	54
5.3 Percentage of Entry Classes Performing each Action	55
5.4 Number of Controllers using Each Action	58
5.5 Number of Controllers Matching Each Field	61
5.6 Number of Tables used by Controllers	62
5.7 Number of Tables with High Update Rates	63
5.8 Number of High Entry Tables	64
6.1 Example Shoehorn Architecture	67
6.2 Example Pipelines that Cannot be Aggregated	73
6.3 Key for Table Aggregation Diagrams	73
6.4 Aggregating Drop Tables	74
6.5 Aggregating Conditionals	75

6.6	Aggreagting Mutually Exclusive Tables	75
6.7	Concatenating Ternary Tables	76
6.8	Aggregation after Recirculation	77
6.9	Example of Tables that Cannot Be Recirculated without Additional Knowledge	78
7.1	Example SVA Packet Path	83
7.2	Example SPA Packet Path	83
8.1	Table Update Rates	95
8.2	During Stage 1 Shoehorn Disregards Excess Matches	96
8.3	Example Entry Tree	102
8.4	Invalid Match Kinds Caused by Incremental Aggregation	106
8.5	Resolving goto Actions	110
8.6	Demonstration of Translating Table Entries	111

List of Tables

4.1 Support for Pipeline Features in the Target Hardware	42
4.2 Support for Common Table Types in the Target Hardware	43
7.1 Header Fields Set by the Shoehorn Parser	84
7.2 Shoehorn Control Block Input Metadata	85
7.3 Shoehorn Intrinsic Control Block Action Metadata	86
7.4 Shoehorn Primitive Action Externs	89
9.1 Overview of Target Controllers	129
9.2 Number of Recirculations Required to Support Each Virtual Pipeline	130

List of Algorithms

1 Stage 2 of the Shoehorn Mapping Algorithm	97
2 Map Components from a Virtual Control Block to a Physical Control Block	100

Publications

Christopher Lorier, Matthew Luckie, Marinho Barcellos, and Richard Nelson.
“Shoehorn: Towards Portable P4 for Low Cost Hardware”. In: *2022 IFIP Networking Conference (IFIP Networking)*. IEEE. 2022, pp. 1–9

List of Acronyms

ACL	Access Control List
AMA	Action Module Action
API	Application Programming Interface
ARP	Address Resolution Protocol
ASIC	Application-Specific Integrated Circuit
DAG	Directed Acyclic Graph
DoS	Denial of Service
EAP	Extensible Authentication Protocol
ForCES	Forwarding and Control Element Separation
gRPC	gRPC Remote Procedure Call
HSDN	Hierarchical SDN
JSON	JavaScript Object Notation
LACP	Link Aggregation Control Protocol
LLDP	Link Layer Discovery Protocol
MPLS	Multiprotocol Label Switching
MPTCP	MultiPath TCP
NAT	Network Address Translation
NIC	Network Interface Card
NPLang	Network Programming Language
OCP	Open Compute Project

OF-DPA	OpenFlow Data Plane Abstraction
ONL	Open Network Linux
ONF	Open Networking Foundation
ONOS	Open Network Operating System
OpenNSL	Open Network Switch Library
PCAP	Packet Capture
PINS	P4 Integrated Network Stack
PSA	Portable Switch Architecture
QoS	Quality of Service
REANNZ	Research and Education Advanced Network New Zealand
SAI	Switch Abstraction Interface
SDKLT	Software Development Kit Logical Table
SDMZ	Science DMZ
SDN	Software-Defined Networking
SPA	Shoehorn Physical Architecture
STP	Spanning Tree Protocol
SVA	Shoehorn Virtual Architecture
SRAM	Static Random Access Memory
TCAM	Ternary Content-Addressable Memory
TPA	Transport Protocol Address
TTP	Table Type Pattern
VLAN	Virtual LAN
VID	VLAN ID
VPN	Virtual Private Network
VRF	Virtual Routing and Forwarding
WIX	Wellington Internet Exchange

Chapter 1

Introduction

1.1 Problem Statement

The promise of SDN is that by decoupling the control plane from the data plane, network operators will have the freedom to combine the software of their choice with the hardware of their choice [34]. However, the creation of SDN control-plane software that is portable across a wide variety of hardware is hindered by incompatible implementations of SDN standards by different vendors [18, 25, 43, 62, 70, 75, 93].

To support the packet processing rates required for high speed networking most low-cost hardware uses fixed-function ASICs. Fixed-function ASICs are cheap to produce, consume very little power, and can process billions of packets per second. However, fixed-function ASICs process packets with a fixed pipeline of logic, look-up tables, and other components—which may not be consistent with the pipelines used by other ASICs. Software intended for one ASIC is unlikely to be able to control a different ASIC without modification, even when the two ASICs support the same SDN standard.

OpenFlow [60], the first SDN standard to have widespread industry support, uses a model that ostensibly allows the creation of an arbitrary pipeline of tables. In practice, however, most hardware implementations apply heavy restrictions to tables, in order to fit their ASIC pipelines. Hardware implementations specify which matches, masks, and actions can be used in each table. The restrictions for each vendor are contradictory, meaning control-plane software developers must customise their software to different hardware—a time-consuming, expensive, and error-prone process.

To overcome this, researchers have proposed a translation layer in-between the control- and data-planes [89, 90, 100, 114]. Control plane software defines a virtual pipeline to control, and translation software transforms commands for the virtual pipeline to produce identical behaviour from an otherwise incompatible physical pipeline. The translation software can use vendor provided drivers [114], or an algorithm to find mappings to arbitrary physical pipelines [89, 100].

However, current algorithms are not suitable for real-time translation of live control-channel commands in production networks. Sanger, Luckie, and Nelson [100], and Pan et al. [89] both present algorithms where, in the worst case, the rate at which controllers can update table entries reduces exponentially with the number of tables in a virtual pipeline.

This thesis investigates whether it is possible to algorithmically map from a virtual to a physical pipeline, in a manner that does not significantly impact the rate at which controllers can update table entries.

1.2 Contributions

The primary contribution of this thesis is Shoehorn, a novel system for finding mappings between virtual and physical pipelines. Shoehorn finds mappings that are suitable for real-time translation by ensuring they do not negatively impact the rate at which entries in the virtual pipeline can be updated. Shoehorn differs from previous algorithmic approaches by guaranteeing that the number of table entries required in the physical pipeline is no greater than the number of entries in the virtual pipeline (unless explicitly permitted), Shoehorn is able to map pipelines to physical pipelines with very flexible tables, and Shoehorn maps P4 pipelines, rather than populated OpenFlow tables.

The components of Shoehorn are:

- The Shoehorn Virtual Architecture (SVA), a P4 architecture to be used by the developers of control-plane software, to define virtual pipelines for their software to target,
- The Shoehorn Physical Architecture (SPA), a P4 architecture to be used by hardware vendors, to define the physical pipelines used by their ASICs,
- A mapping algorithm, that, when given a virtual and physical pipeline, defined in the SVA and SPA respectively, can find a mapping between the two, suitable for use by real-time translation software.

The evaluation of Shoehorn uses a proof-of-concept implementation of the mapping algorithm, which is publicly available [24].

In addition, this thesis provides an in-depth investigation of the pipelines used by existing SDN controllers, applications and hardware. This analysis informs the design of Shoehorn, but also produced a novel dataset containing details

of the pipelines used by a number of devices, controllers and controller applications, released alongside this thesis [23].

1.3 Thesis Structure

Chapter 2 provides relevant background to the work described in this thesis. It covers the evolution of SDN, explaining relevant technical details of SDN standards, and how this has impacted the creation of portable control-plane software.

Chapter 3 describes previous work to improve the portability of control-plane software, and summarises the limitations of each.

Chapters 4 and 5 investigate the use of existing SDN standards in control-plane software and hardware respectively. Mapping between pipelines is an NP-complete problem; these chapters investigate common features in control-plane software and hardware that Shoehorn can exploit to limit the scale of the potential solution space. Chapter 4 analyses the published details of various vendors implementations of SDN standards. It identifies low-cost hardware suitable for Shoehorn to target, and identifies barriers to portability within that hardware, as well as common features that may simplify the process of finding mappings. Chapter 5 analyses production SDN deployments, as well as control-plane software described in research. Chapter 5 also describes a technique for identifying transactions in control-channel traffic.

Chapters 6, 7, and 8 describe Shoehorn. Chapter 6 introduces Shoehorn, providing a high-level overview of the design of Shoehorn. This chapter justifies design decisions using the analysis performed in chapters 4 and 5. Chapter 7 provides a detailed description of the SVA and the SPA. Chapter 8 describes

the algorithm for finding mappings between pipelines.

To evaluate Shoehorn, this thesis created virtual pipelines in the SVA based on 25 example projects chosen from those analysed in chapter 5, and created 5 physical pipelines based on the target hardware analysed in chapter 4, and attempted to map each virtual pipeline to each physical pipeline. This is described in detail in chapter 9.

Finally, chapter 10 summarises this thesis, and provides an overview of future work.

Chapter 2

Background

This chapter provides a detailed description of the evolution of SDN and the most widely used SDN standards. It describes features of the standards that are difficult to support with fixed-function hardware, and therefore present a challenge to creating portable SDN control-plane software.

This chapter primarily focusses on OpenFlow (section 2.2) and P4 (section 2.3), the two standards used throughout this thesis. These sections also contain technical details of the standards that are relevant to this thesis.

2.1 Software-Defined Networking

Traditional networking equipment does not allow operators to modify the software that controls the forwarding behaviour of the device. Devices are only able to run software provided by the vendors, which operators can configure but not modify. SDN is an approach to networking that separates the control software from the forwarding hardware, allowing network operators to choose their own software to control their networks.

In the SDN model, network devices are split into two planes: the *data-plane* handles the packet (data) forwarding, and the *control-plane* runs the protocols that control the behaviour of the data-plane. The two planes communicate via an open API allowing the control-plane to be run on commodity hardware separate from the specialised hardware of the data-plane.

With an open interface to the data-plane, the control-plane software can be modified or replaced by an operator. This enables the creation of innovative new applications, but the potential benefits go far beyond that. Portable control-plane software allows operators to control a multi-vendor network with consistent control-plane software. This reduces complexity, ensures interoperability, simplifies monitoring, and makes troubleshooting easier. When running a Software-Defined Network, having the ability to introduce hardware from multiple vendors, without having to modify the control-plane software, allows operators to choose the ideal hardware for their requirements, ensures resilience of supply, and can reduce costs.

2.1.1 Terminology

The components of the SDN model are referred to by different names in different standards. The remainder of this thesis uses the names used in the OpenFlow specification [84]. A *datapath* is a device that performs data-plane forwarding, a *controller* is the control-plane software that controls the behaviour of the datapath, and the *control channel* is the communication channel between the controller and datapath. In some models, the term *controller* refers more specifically to a framework that handles communication with a datapath, and sits below a *controller application* which is responsible for determining the datapath's behaviour. As this work only requires that the control-

plane software defines a virtual pipeline, and is otherwise agnostic to how the control-plane software is structured, the remainder of this thesis uses the term *controller* to refer to any software that defines a virtual pipeline.

2.2 OpenFlow

OpenFlow [60] is the most widely supported SDN standard. It was first created to enable researchers to run experiments on University campus networks. Devices could run in a hybrid SDN–legacy mode, allowing production traffic to continue to be controlled by software running on the device, while experimental traffic could be controlled using OpenFlow. OpenFlow instigated a great deal of network research, and was quickly used to control production networks as well [8, 47].

The original OpenFlow version 1.0 standard [79] was designed to be simple for vendors to implement on legacy hardware, so that it could be added to existing campus networks without the expense of replacing devices. As a consequence, OpenFlow version 1.0 is very limited in the functionality it provides. This was addressed with later versions of OpenFlow [80, 81, 82, 83, 84], but this led to a protocol that was significantly harder for vendors to implement.

2.2.1 OpenFlow 1.0

In OpenFlow version 1.0, the packet handling behaviour of a device is represented as a single look-up table. The controller modifies the behaviour of the datapath by adding and removing entries in this table.

Entries in the table have a set of *Matches*, a set of *Actions*, and a *Priority*. Matches identify the packets to which entries apply. Matches specify a header

field, such as the IPv4 destination address or the Virtual LAN (VLAN) Identifier, and a masked value for that field. Packets match the highest priority entry where every match value is equal to the corresponding masked field in the header of the packet. An entry's actions define how the datapath should process matching packets. Possible actions include dropping the packet, modifying fields in the packet's header, sending the packet to the controller for further processing, or outputting the packet to a port.

As each packet can only ever match one entry, the single table approach scales poorly when different actions are associated with different fields. For instance, if a controller wishes to push a VLAN tag to every packet arriving on specific ports, and determine the egress port by matching the Ethernet destination, then the table must include an entry for every ingress port and Ethernet destination combination.

This resulted in a common paradigm of *reactive* controllers [35]. In this paradigm, traffic arriving at a datapath is forwarded to the controller by default using a `Packet-In` message. The controller then determines the correct course of action for all subsequent packets belonging to that flow, and installs a corresponding flow table entry. However, forwarding all traffic to the controller can be inefficient [35, 99] and creates a Denial of Service (DoS) attack vector.

2.2.2 Multi-table OpenFlow

OpenFlow 1.1 [80] introduced a series of tables. Datapaths perform a look-up in the first table for every packet, and entries can send packets to a subsequent table for additional look-ups with a `Goto-Table` instruction. Each table is numbered, and the entries in a table can not specify a lower numbered table with the `Goto-Table` instruction, thereby preventing loops.

Having multiple tables greatly improves the ability to support complicated controllers without needing reactive flow installation. However, this made it significantly harder to fully support OpenFlow on physical hardware. Fully supporting multi-table OpenFlow version 1.1 requires supporting a pipeline of up to 256 tables, that can be re-arranged at runtime, each able to support masked matches for at least 17 fields, with arbitrary priorities [80]. The only datapaths to fully support OpenFlow 1.1 relied on software flow tables [3, 91] or expensive, flexible hardware like NPUs [76].

Most hardware implementations support multi-table OpenFlow by applying heavy restrictions to tables. They specify which matches, masks, instructions and actions each table can use, in order to match their ASIC pipelines. The restrictions for each vendor are contradictory, meaning control-plane software developers must customise their software to different hardware—a time-consuming, expensive, and error-prone process.

2.2.3 Apply-Actions and Write-Actions

A multiple table pipeline raises the issue of where the pipeline should apply actions. For instance, if a table rewrites a header field, and a subsequent table matches that header field, should the table match the original value, or the rewritten value? To address this issue, OpenFlow uses two types of action instructions:

- The **Write-Actions** instruction writes an action to an *action set*, and the pipeline applies the actions in the action set after it has completed all flow table lookups for a packet. If a **Write-Actions** instruction writes an action that has been written by a previous instruction, the earlier instruction will be overwritten.

- The **Apply-Actions** instruction applies actions immediately, before any subsequent table lookups. If an **Apply-Actions** instruction sets a header field, then subsequent table lookups will use the new value. **Apply-Actions** can be very difficult for hardware to support as their pipelines often only allow rewriting packets to occur at the end of the pipeline.

2.2.4 Metadata

Metadata allows pipelines to carry additional information with packets. Table entries can write metadata with a **Write-Metadata** instruction, which can be matched by entries in subsequent tables. OpenFlow has a single 64Bit field, and the **Write-Metadata** instruction includes a mask, allowing it to write to arbitrary bits without overwriting other bits. Very few ASICs support masked writes, instead most vendors either do not support metadata, or specify a set of masks that can be used, effectively partitioning the metadata into multiple fields.

2.2.5 Group Tables

OpenFlow 1.1 also introduced group tables [80]. Group tables are applied at the end of pipelines, and allow associating groups of actions together allowing them to be applied in more sophisticated ways. Flow table entries can specify an entry in the group table with the **group** action. After the packet has passed through the flow tables, the pipeline applies the actions specified in the group table entry, according to the entries group type. The types of group are:

- **Indirect** groups combine a common set of actions together that multiple table entries can reference.

- **All** groups define sets of actions and apply each to copies of the packet.
- **Select** groups distribute flows between multiple sets of actions for the same table entry. For instance, select groups can load balance traffic across multiple interfaces.
- **Fast-Failover** groups specify backup sets of actions when a liveness check for the primary set of actions fails.

Groups can reference other groups. For instance, a pipeline could include an **Indirect** group for each VLAN interface, indicating whether the packet should be output with a VLAN tag, and an **All** group to flood packets. By referencing the VLAN interface groups, the flood group does not need to be updated when a VLAN interface is changed from tagged to untagged.

2.2.6 OpenFlow Table Type Patterns

Table Type Patterns (TTPs) [85] were proposed by the Open Networking Foundation (ONF) as a method of reducing the burden created by incompatible OpenFlow implementations, by enabling the standardisation of OpenFlow pipelines. TTPs are a framework for defining a set of restrictions for OpenFlow tables for a device or a controller. TTPs are very detailed, allowing very fine-grained specification of exactly what entries can be added to each table. TTPs can specify multiple different entry types within a table, each entry type with its own set of restrictions. TTPs allow specifying exact values for fields and masks that entries must match. Alternatively TTPs can specify match types, including:

- **exact**, an unmasked match;
- **mask**, a match with an arbitrary mask;

- `prefix`, a match with a prefix mask; or
- `all_or_exact`, a match that either matches an exact value or any value.

TTPs are intended to follow a life-cycle whereby a developer identifies a use case, and defines a TTP to support it, then vendors can choose to implement the TTP in their hardware or not. This leads to the obvious problem of having no guarantee that a vendor will be able to implement a TTP, and even if they can, whether they will choose to do so in a timely manner. Consequently, TTPs have not been widely adopted. As of July 2022 the ONF's TTP repository [78] was last updated in 2017 and only contains three TTPs (excluding examples and drafts): a TTP defining the pipeline used by Broadcom's OpenFlow Data Plane Abstraction (OF-DPA) [18], and two TTPs defined by the ONF.

2.2.7 Table-Features Messages

Vendors with flexible hardware have used `Table-Features` messages to limit the entries that can be added to tables at run-time [25, 43]. The controller sends a `Table-Features` message to the datapath, indicating the matches, masks, actions and next tables that entries in each table can use. The datapath then configures its tables to support only the entries specified in the `Table-Features` message. `Table-Features` messages are not as detailed as a TTP, however. For instance, a `Table-Features` message can only indicate whether a field is maskable, but not whether the mask will be `all_or_exact` or a `prefix`.

2.2.8 Other OpenFlow Features

This section briefly describes some other features of OpenFlow relevant to this thesis.

2.2.8.1 Vendor Extensions

Vendor extensions allow vendors to implement arbitrary features that are not defined in the specification. By definition, vendor extensions are only supported by a single vendor, and consequently are a barrier to portability.

2.2.8.2 Barrier Messages

Datapaths may apply commands from a controller out of order. Controllers can use `Barrier` messages to enforce an order between commands. Datapaths must process all messages received before the `Barrier` before attempting to process any messages received after the `Barrier`. However, `Barrier` messages are poorly supported by hardware [97].

2.2.8.3 Packet-In and Packet-Out Messages

When a controller wants to send a packet, for instance, to reply to an Address Resolution Protocol (ARP) request, it can use a `Packet-Out` message, to instruct the datapath to generate a packet. Likewise, when the controller wishes to receive packets from the data-plane, it can instruct the datapath to output packets to the `CONTROLLER` port. When a packet is output to the `CONTROLLER` port, the datapath will encapsulate the packet in a `Packet-In` message, and forward it to the controller.

2.2.8.4 The **NORMAL** Port

When a datapath is deployed in hybrid mode, packets can be sent to the legacy pipeline to be processed normally by outputting the packet to the **NORMAL** port.

2.3 P4

P4 (Programming Protocol-Independent Packet Processors) [13] is a programming language for specifying the packet-processing behaviour of a switch. It allows developers to define the behaviour of:

1. the parser, which extracts header fields from packets;
2. the deparser, which controls how packets are emitted;
3. the control flow, including logic and look-up tables;

as well as other aspects of a datapath with much more detail than OpenFlow.

When compiled, a P4 program produces a device-specific configuration for the datapath, and the Application Programming Interface (API) by which the datapath can be controlled. The communication protocol is unspecified—P4 only defines the interface. Commonly used communication protocols include the OpenFlow protocol, or P4 Runtime [87], a control protocol designed specifically for P4 datapaths based on gRPC.

Having control of the parser and deparser means that P4 programs are not tied to any existing protocols or header fields. The look-up tables expose functionality similar to OpenFlow but defining the control flow allows these to be designed much more efficiently. For instance, P4 programs can determine the IP version of a packet with a switch statement rather than a table look-up.

There are two versions of the P4 standard, P4₁₄ [106], and P4₁₆ [13], which is the version used in this thesis.

2.3.1 P4 Architectures

P4 programs specify the behaviour of programmable blocks in a datapath's pipeline, but this leaves the question of what programmable blocks are available, how they fit together, and their interfaces to the datapath itself. Such information is defined in a P4 architecture.

P4 architectures include definitions of P4 package objects, the object representing a programmable datapath. P4 programmes consist of an instantiation of a package object, which takes as arguments the parsers, programmeable control blocks, and any externs representing the behaviour of the datapath.

Code written for a P4 architecture should be portable across all hardware that supports that architecture. The Portable Switch Architecture (PSA) [88] is an example architecture that is designed to be portable across a variety of switches. The PSA contains few restrictions on the code that can be used within a control block, and is, consequently, impossible to support with fixed-function hardware.

2.3.2 Control Blocks

Control blocks define how the datapath processes packets, once the headers have been extracted by the parser. Control blocks contain control flow definitions, and can invoke tables and extern objects.

Control blocks take structs of metadata and packet headers as arguments. Each argument must have a direction: `in`, `out`, or `inout`, indicating whether

the metadata is input, output or both.

2.3.3 Extern Objects

In P4, control of queues and stateful functions (such as registers or counters) are supported using *extern* objects. Externs allow support of any component or function of a device without it having to be supported natively in P4. Architectures define the interface to externs, and typically provide a written description of their expected behaviour—the implementation details are left unspecified. Datapaths must support all extern objects faithfully in order to support the Architecture. Externs can be instantiated as part of a package, or can be invoked from within control blocks.

2.3.4 Tables, Actions and Match Kinds

In P4, table definitions include specifications of the matches and actions that the table can use. The actions are defined separately, and are themselves programmable. Actions can include logic, and typically write metadata or invoke extern objects to direct the behaviour of the device. Tables include a list of actions that entries in that table can apply. The matches a table can use are specified in a dictionary of header field to *match kind* mappings.

Match kinds indicate the type of look-up required for each field in a table. The core P4 library defines three match kinds:

1. **exact**, for fields that must match an exact value with no mask;
2. **lpm**, for longest prefix matches; and
3. **ternary**, for fields with arbitrary masks.

Different match kinds use different types of memory, and have different update rates and power consumption. `exact` matches are typically the fastest to update and use the least power, as they can be implemented with a hash table in Static Random Access Memory (SRAM). `lpm` and `ternary` tables are typically implemented with Ternary Content-Addressable Memory (TCAM). TCAM returns the matching entry with the lowest index, meaning that adding new entries may require moving existing entries [101, 112], thereby slowing down the update rate. TCAM also reads the entire table in parallel for every look-up, meaning TCAM look-ups consume considerably more power than SRAM look-ups.

P4 programs invoke tables by calling their `apply` method. The `apply` method performs a table look-up, applies the resulting actions, and returns a struct containing a `hit` field, indicating whether the packet matched an entry in the table, and a list of the actions that the table applied.

2.3.5 Annotations

Annotations allow architectures to extend the P4 language. P4 annotations provide additional information to the compiler without modifying the P4 grammar. There are 5 standard annotations in the P4 specification, and include the `@tableonly` and `@defaultonly` annotations that, when added to actions in a table's action list, indicate that an action cannot be used as the default action, or that an action can only be used as the default action, respectively.

2.3.6 Portable Switch Architecture

The Portable Switch Architecture (PSA) is a P4 Architecture for a generic programmable switch created by the P4.org Architecture Working Group [88]. As the name suggests, it is intended to be portable across a variety of P4 switch hardware. However, the PSA allows developers to define an arbitrary pipeline, and therefore cannot be supported by fixed-function devices.

The PSA defines a variety of new externs, those relevant to this work are described below.

2.3.6.1 *ActionProfiles*

`ActionProfiles` are analogous to the `indirect` group table in OpenFlow. Table entries can specify a reference to a `ActionProfile` rather than specifying the actions directly, allowing updating multiple entries with the same actions more efficiently.

2.3.6.2 *ActionSelectors*

`ActionSelectors` implement behaviour similar to OpenFlow `select` groups. They are similar to `ActionProfiles`, but allow referencing multiple actions. The `ActionSelector` takes an algorithm definition as an argument, which is used to determine which action the `ActionSelector` applies.

2.3.6.3 *DirectCounters*

`DirectCounters` are how the PSA implements table counters. `DirectCounters` associate with one table, and can only be updated by actions called by that table. Actions update counters by invoking the `DirectCounter`'s `count` method.

`DirectCounters` have a counter associated with each entry in the associated table. When an action invokes the `count` method, the counter associated with the matched entry is increased.

2.4 Other SDN Standards

This section provides a summary of other SDN standards, and how each approaches the issue of portability.

2.4.1 Switch Abstraction Interface

SAI is a C-like API to a generic switch, created by the Open Compute Project (OCP), based on a fixed pipeline of tables [77]. Switch Abstraction Interface (SAI) is widely supported by a variety of hardware from different vendors and a number of Network Operating Systems, such as SONiC [113] and Open Network Linux (ONL) [11].

SAI performs basic router functions, such as switching, spanning tree, Access Control Lists (ACLs), and routing, and has many proposed features such as Network Address Translation (NAT) or in-band telemetry. SAI exposes a small subset of the functionality of most of the target hardware, but still includes features that are not universally supported.

2.4.2 P4 Integrated Network Stack

P4 Integrated Network Stack (PINS) is a project to augment the SAI using P4 [86]. PINS enables control of the SAI pipeline using P4 runtime and introduces new programmable blocks within the pipeline. Control of SAI using P4

runtime requires a P4 definition of the SAI pipeline, but this definition remains a work in progress and currently omits many important features, most notably layer 2 switching.

2.4.3 FoRCES

Forwarding and Control Element Separation (ForCES) is a precursor to OpenFlow that proposed a separation of the data- and control-planes of network devices [28]. It also provides the protocol for communication between controllers and datapaths (called forwarding elements and control elements in the ForCES model). Unlike OpenFlow, ForCES does not strictly define the expected behaviour of a datapath, meaning it is possible that two datapaths controlled in exactly the same manner may process the same packet in a different way.

2.4.4 Broadcom OpenNSL and SDKLT

Broadcom offer a number of different ways to program their switches [17, 18, 19, 20]. They provided OpenFlow support with the OF-DPA, but because they could not offer the full feature set of their switch chips, they have created new specifications.

Open Network Switch Library (OpenNSL) [19] is a library that exposes the full functionality of Broadcom's switches. OpenNSL uses the abstraction of interconnected internal components they refer to as *devices*. Devices can be attached to ports allowing control of the packets arriving on that port.

Software Development Kit Logical Table (SDKLT) [20] is a table based abstraction that sits above the OpenNSL. There are two table types in SDKLT *modeled tables* are controlled by the controller and *interactive tables* are con-

trolled by the device itself. Interactive tables expose information from the switch to the controller such as link state.

As Broadcom produces ASICs with different capabilities, code written using OpenNSL or SDKLT may not be portable.

2.4.5 NPLang

Network Programming Language (NPLang) [17] is a language for programmable network hardware created by Broadcom. It is similar to P4, but is more closely modeled to the design of physical hardware. The NPLang architecture involves multiple stages, each reading from, and writing to, a logical bus.

NPLang allows programs to create functions to build keys for table look-ups. This allows tables to match similar fields from different headers (eg. IPv4 Source and ARP Transport Protocol Address (TPA)), or hashes of multiple different sets of fields. It also allows a table to perform multiple look-ups with the same packet (eg. to look-up the packet's Ethernet Source and its Ethernet Destination).

NPLang performs table look-ups in parallel. To resolve scenarios where multiple tables write to the same bus entries, NPLang uses a `strength_resolve` construct. The different table look-up results have associated strength values, and the look-up with the highest strength value writes its values to the bus entry.

Chapter 3

Related Work

This chapter describes previous work on enabling portable SDN controllers. This chapter describes a variety of different approaches, and compares these to algorithmic mapping. Then it describes previous work on algorithmic mapping approaches in detail, and discusses the limitations of each.

3.1 Alternative Approaches

This thesis investigates using algorithmic mapping to improve the portability of SDN controllers, but alternative approaches have been proposed. This section describes the limitations of these other approaches, when compared with algorithmic mapping.

3.1.1 Standardised Pipelines

Creating standard pipelines that only expose functionality that can be universally supported by the target hardware ensures that all software will be portable [77, 85]. However, by not exposing the full functionality of the hard-

ware, this approach unnecessarily restricts the controllers it can support.

Furthermore, this approach may detrimentally impact performance, when compared with algorithmically mapping pipelines. If the standardised pipeline differs from the pipeline used by the underlying hardware, the hardware may make performance concessions to support the standardised pipeline. Likewise, the controller may have to make performance concessions to enable support from the standardised pipeline. Consequently, controllers that might be directly mappable to the physical pipeline, instead incur an unnecessary penalty to performance.

3.1.2 Programmable Hardware

Researchers have proposed creating flexible hardware, capable of adapting its pipeline to the needs of diverse control-plane software [12, 50, 105] and this has been adopted by hardware vendors [2, 16].

Programmable hardware primarily targets high throughput data centre networks, however, and would be prohibitively expensive for many enterprise networks. Many enterprise networks are likely to use hardware with fixed-function ASICs for the foreseeable future.

Programmable hardware does not necessarily eliminate all challenges to portability. P4 Transformer [42] identified three barriers to portability with programmable hardware, and presented techniques for supporting controllers with diverse hardware in spite of these barriers. Hyper4 [40] proposed a P4 program that can be used as a target for other P4 programs, thereby guaranteeing portability, and improving composability, for all hardware that can support Hyper4.

3.1.3 Device Drivers

Another approach is to create drivers for individual devices that provide a standard higher-level interface to control-plane software [10, 114]. Like standardised pipelines, these do not expose the full functionality of the device.

Open Network Operating System (ONOS) [10], a network operating system created by the ONF, circumvents this problem by including a high-level driver interface, as well as exposing the hardware pipeline directly. This allows controllers to choose between portability and having access to the full functionality of the hardware, but offering this choice does not resolve the underlying issue: the controller must either limit its functionality, or it cannot be portable.

3.1.4 NOSIX

Defining a virtual pipeline that is subsequently mapped to the physical pipelines of a device was first proposed by Yu, Wundsam, and Raju, presenting their lightweight SDN portability layer NOSIX [114]. NOSIX allows controllers to define annotated virtual pipelines, which would be mapped to hardware using vendor provided drivers. The annotations allowed controllers to indicate their expectations of each table, such as the maximum number of entries, the rate entries are updated, or the traffic volumes the table is expected to support. The drivers can then make decisions on how best to support each table, including possibly supporting low traffic tables in software. NOSIX does not include a mapping algorithm, it leaves it to vendors to produce drivers, able to map from the virtual pipelines to their hardware.

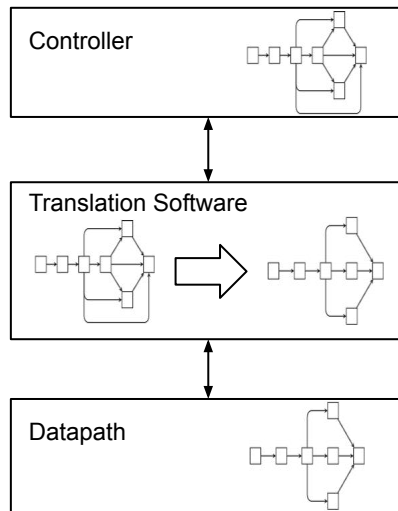


Figure 3.1: An example SDN architecture using translation software. The controller defines a hardware-agnostic virtual pipeline. The translation software sits in between the controller and datapath, rewriting the control plane messages to create equivalent behaviour from the physical pipeline used by the datapath.

3.1.5 TableVisor

Geissler et al. proposed TableVisor [36], a system for overcoming hardware limitations by aggregating multiple physical devices to act as a single logical datapath. However, their approach does not consider the internal structure of pipelines, instead it assumes that tables will always be reachable by packets delivered from another device. It also requires the use of multiple devices to achieve what might otherwise be achievable with a single device.

3.2 Algorithmic Mapping of SDN Pipelines

An algorithmic mapping approach allows control-plane software to define a hardware-agnostic virtual pipeline indicating the desired behaviour of datapaths. The algorithm finds a mapping from the virtual pipeline to the pipeline used by the physical datapath that causes packets to be processed as though

Virtual Table 1			Virtual Table 2	
VLAN ID:	Ethernet Destination:	Actions:	IPv6 Destination:	Actions:
111	00:00:00:00:01:11	goto_table 2	2001:db8:1::/48	next_hop 1
222	00:00:00:00:02:22	goto_table 2	2001:db8:2::/48	next_hop 2
			2001:db8:3::/48	next_hop 3

Merged Table				
VLAN ID:	Ethernet Destination:	IPv6 Destination:	Actions:	
111	00:00:00:00:01:11	2001:db8:1::/48	next_hop 1	
222	00:00:00:00:02:22	2001:db8:1::/48	next_hop 1	
111	00:00:00:00:01:11	2001:db8:2::/48	next_hop 2	
222	00:00:00:00:02:22	2001:db8:2::/48	next_hop 2	
111	00:00:00:00:01:11	2001:db8:3::/48	next_hop 3	
222	00:00:00:00:02:22	2001:db8:3::/48	next_hop 3	

Figure 3.2: An example of merging two virtual tables resulting in a Cartesian product of entries in the physical table. The merged table requires an entry for every combination of entries in the two virtual tables. Removing one entry in Virtual Table 1 requires removing 3 entries in the merged table, one for each entry in Virtual Table 2.

they were processed by the virtual pipeline. Then translation software can be used to rewrite the control plane messages between the controller and datapath. The controller behaves as though the datapath uses the virtual pipeline, and the translation software modifies the controllers commands to ensure the physical datapath will process packets in an equivalent manner. Figure 3.1 illustrates a potential architecture using this method.

3.2.1 Table Entry Growth

Previous work on algorithmic mapping converts virtual pipelines into a graph and then maps from the graph to the physical pipeline [89, 90, 100]. These approaches result in equivalent behaviour, but can increase the number of table entries used in the physical pipeline. This increases the memory usage, and slows the rate that tables can be updated, as updating a single entry in the virtual pipeline requires updating every associated entry in the physical pipeline. In the worst case scenario, merging two tables can result in a physical table with an entry for every combination of entries in the two virtual tables. This is illustrated in figure 3.2. Updating a single entry in one of the virtual

tables would require updating as many entries in the merged table as there are in the other virtual table. With non-trivial table sizes, this would result in an impractical reduction in performance when translating table updates in real-time.

3.2.2 FlowAdapter and FlowConvertor

Pan et al. [90] proposed FlowAdapter, a tool for translating control plane messages for a virtual OpenFlow pipeline to a physical pipeline in real time. This work was then augmented by Pan et al. with the publishing of FlowConvertor [89], an improved mapping algorithm for use by FlowAdapter. FlowConvertor works by mapping flow table entries—it does not require prior knowledge of any constraints to the virtual pipeline. Controllers can add arbitrary entries, with any matches or actions, and FlowConvertor will find a mapping for that entry (if possible).

FlowConvertor converts the virtual OpenFlow pipeline into a Directed Acyclic Graph (DAG) structure, and then maps that structure to a constrained physical pipeline. Each node in the DAG represents a flow table entry, and each node has an edge to every entry in the subsequent table that could be matched by the same packet. The DAG structure allows efficient incremental updates, which is necessary to support real time updates. Adding a flow table entry to the virtual pipeline only requires adding a single entry in the DAG, with edges for every entry in the previous table, and every entry in the subsequent table. In the worst case scenario, updating the DAG scales linearly in the number of table entries.

FlowConvertor maps from the DAG to the physical pipeline incrementally. When a node is added to the DAG, FlowConvertor finds every path from

a root node to a leaf node that passes through that node. For each path, FlowConvertor iterates through the tables in the target physical pipeline, and assigns values for every match field that the path matches, until all fields are assigned. FlowConvertor then uses metadata to ensure that packets adhere to the paths used in the virtual pipeline, and (with a few exceptions) assigns actions in the final table. This method is unsuitable for some hardware, as metadata is difficult to support in physical hardware, and actions may not be available in all tables.

Furthermore, while updating the DAG scales linearly with the number of table entries, the number of paths through the tree scales exponentially with the number of tables in the pipeline, as the paths through the tree branch for every link to a subsequent node.

In their evaluation, Pan et al. mapped four synthetic virtual pipelines to three physical pipelines, and found that the mapping resulted in only 1.5ms of overhead to table updates. However, the longest virtual pipeline had only four tables, and they did not indicate the number of table entries they used. It is inevitable that with longer pipelines, or with larger table sizes, FlowConvertor could not sustain real-time translation.

3.2.3 Sanger, Luckie and Nelson

Sanger, Luckie, and Nelson [100] proposed an alternative algorithm for mapping from a virtual pipeline to a physical pipeline. Like FlowConvertor, their algorithm works on a populated flow table pipeline, mapping flow table entries rather than a constrained virtual pipeline. Unlike FlowConvertor, they do not consider their algorithm suitable for finding a mapping in real time. Instead, they propose using an example populated flow table pipeline to find a mapping

offline, which effectively constrains the controller to only adding entries similar to those in the example.

Sanger, Luckie, and Nelson use the following steps in their algorithm:

1. Convert the full flow table pipeline into a compressed single table.
2. Find every practical mapping to the physical pipeline for each entry in the single table.
3. Use a Boolean Satisfiability (SAT) solver to find a combination of mappings that maps every entry in a manner that does not interfere with the mapping of other entries.

Converting the full pipeline into a single table involves taking the Cartesian product of all the tables in the pipeline. The algorithm then compresses that table by eliminating all entries that differ only in the value of the fields matched. In other words, all sets of entries that match the same fields, with the same masks, the same priority, and apply the same actions are considered similar enough that they will almost certainly be mapped in the same tables. Consequently, the algorithm only needs to map one representative entry. The algorithm ensures that it preserves dependencies between sets of entries. So that if one set of entries may match the same packets as another set of entries, the representative entry of the higher priority set will correctly overlap the representative entry of the lower priority set. The authors evaluated their flow table compression algorithm against real world data sets and found that they achieved a greater than 80% reduction in the number of flow table entries in every case. Converting the original pipeline to a single table eliminates any knowledge of how entries can be updated. Consequently, it is impossible for this approach to preserve the update rate of the original pipeline.

For step 2, the algorithm can either map an entry directly, or can map the entry

to multiple tables that are linked together with `Goto-Table` instructions. The authors refer to this as a *split transformation*. In some cases, when a split transformation maps part of an entry to a table, it may be possible to map multiple fields to that table. In such a case the algorithm always maps the maximum number of fields, reducing the number of potential mappings to search, and minimising the risk of an entry mapping conflicting with another mapping. This approach is significantly more flexible than the approach used by FlowConvertor, as it does not require the use of metadata to find potential mappings.

Step 3 creates a SAT expression representing a successful mapping, consisting of criteria that a successful mapping must meet. For instance, a successful mapping must include one mapping for every entry, and must not allow overlapping entries with the same priority in the same table. The algorithm uses a SAT solver to find a successful mapping based on these criteria.

Sanger, Luckie, and Nelson evaluated their system by converting between a two table pipeline and a five table pipeline using synthetic data. Although they were able to map from the five table pipeline (with 20 total table entries) to the two table pipeline in less than 150ms, they considered their algorithm impractical for real-time translation. The authors noted that with real world pipelines, mapping can be prevented by complications such as whether untagged packets are assigned a default VLAN tag within a pipeline. They also noted that their approach does not work with flexible pipelines, as the number of potential mappings is too great.

3.3 Summary

This chapter has identified a variety of approaches to creating portable SDN controllers. The most promising of these is algorithmically mapping from a virtual pipeline to a physical pipeline, as other approaches either place arbitrary limitations on what the hardware can do, or cannot be achieved with existing enterprise hardware.

While previous work to algorithmically map SDN pipelines has shown potential, it remains impractical for use in production. Most notably, real-time translation is severely impeded by the mappings potentially increasing the total number of entries the pipeline uses. This is exacerbated by the fact that these solutions map pipelines on an entry by entry basis. It would be considerably easier for controllers to define a set of constraints their tables will follow, and then map tables directly, rather than table entries. This fits well with the P4 model, where controllers write a program to define the pipeline for their target datapaths.

Chapter 4

Target Hardware

Creating a system that balances the goal of achieving portability without unduly limiting what software can achieve requires a clear understanding of what the target hardware can and cannot support. This chapter investigates the capabilities of a variety of SDN implementations, and identifies 5 targets, suitable for a mapping algorithm.

The investigation focuses primarily on OpenFlow implementations, as they are common and clearly define the behaviour of the device, but also covers other relevant SDN implementations.

4.1 Broadcom

Broadcom is the world's largest producer of merchant silicon networking chips. Their chips are used by a variety of networking hardware vendors, including those that also produce their own ASICs, such as Cisco. Broadcom ASICs support a variety of SDN standards, either natively [17, 18, 19, 20], or through operating systems created by other vendors [3, 4, 93].

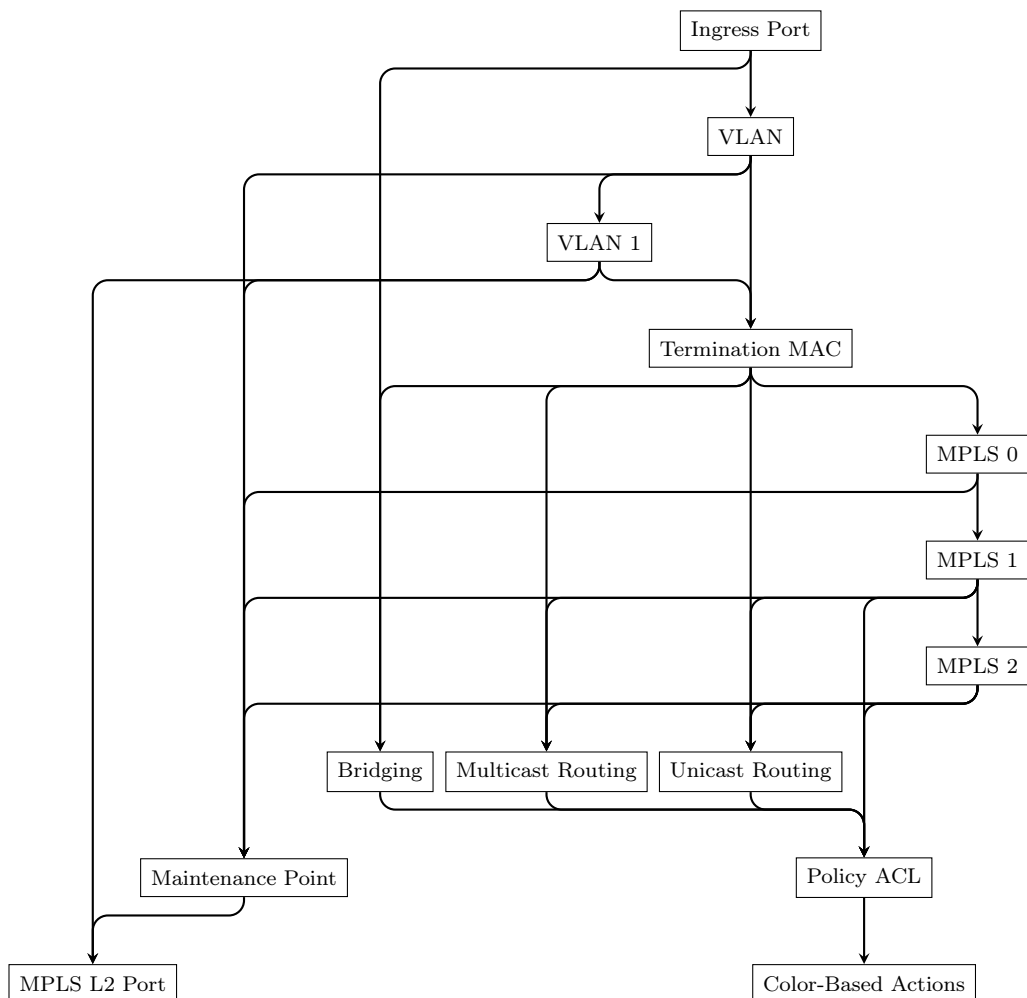


Figure 4.1: The OF-DPA flow table pipeline. Arrows indicate the potential targets of Goto-Table instructions.

4.1.1 OF-DPA

The OF-DPA is an interface to Broadcom ASICs that can be used by vendors to produce an OpenFlow implementation that closely models the underlying ASIC pipelines [18].

The OF-DPA can support multiple different pipelines. Most are non contradictory and can be used in parallel, with a few minor scenarios where that is not possible. An OF-DPA pipeline using all possible tables is shown in figure 4.1. Each table has a set of entry types, and each entry type has its own set of matches and actions that can be used. This results in tables that do not make

sense when translated directly to P4—for instance, the unicast routing table contains both IPv4 and IPv6 routes.

The entry types used in OF-DPA are very strictly defined. For instance, the *Untagged Packet Port VLAN Assignment* entry type in the VLAN table has the following constraints:

- entries must Match Ingress Port,
- entries must match a VLAN VID of 0,
- entries must assign a VLAN VID to matching packets,
- entries must direct packets to the Termination MAC table, and
- entries may, optionally, set VRF metadata.

There are 9 different entry types in the VLAN table, and each has similar constraints to the *Untagged Packet Port VLAN Assignment* type.

The bridging table uses a vendor extension to perform MAC learning. Every time the bridging or routing tables are applied to a packet, the OF-DPA performs a look-up of the packet's Ethernet Source address against the entries in the bridging table, comparing the ingress port of the packet with the associated port in the bridging table. If the two do not match, or the Ethernet Source address is not in the bridging table, then the datapath will either notify the controller, or automatically update the entry in the bridging table, depending on configuration.

Most tables, by default, forward packets to the next stage in the pipeline. This is advantageous when reordering tables or recirculating packets, as it means packets can easily reach the intended next table without needing to match entries in the intervening tables.

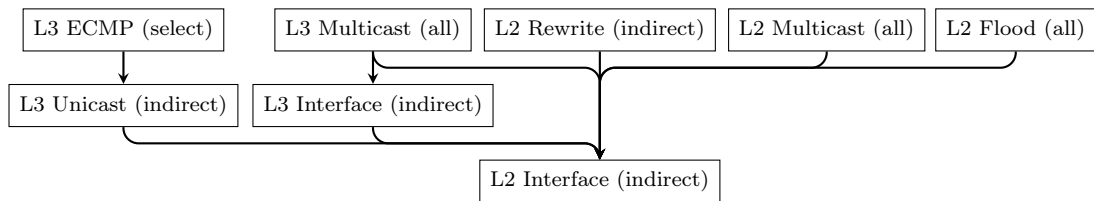


Figure 4.2: The group tables used by the OF-DPA for bridging and routing, and their group types. Arrows indicate groups that reference other groups.

The OF-DPA makes extensive use of group tables (§2.2.5). Figure 4.2 shows the groups the OF-DPA uses for bridging and routing. Groups are also used by the Multiprotocol Label Switching (MPLS) pipeline, including fast-failover groups for protected circuits.

The only writeable metadata used in the OF-DPA is a 16-bit non-maskable Virtual Routing and Forwarding (VRF) field, limiting how a mapping algorithm can use metadata to control the flow of packets through the pipeline.

4.1.2 OpenNSL

Broadcom ASICs also support other standards, including OpenNSL [19]. OpenNSL provides a very detailed, low-level interface to program Broadcom ASICs, and includes some features not included in OF-DPA. For instance, the OpenNSL field processor has the ability to mirror packets, both at ingress and egress. This thesis uses such features to infer the capabilities of Broadcom hardware.

4.1.3 Other OpenFlow Implementations on Broadcom ASICs

Broadcom does not create operating systems for its ASICs, it sells ASICs to vendors who produce their own operating systems. As such, an OpenFlow implementation using the OF-DPA is not supported by all hardware using Broadcom ASICs, the operating system has to include support for OpenFlow.

Some vendors that support OpenFlow on Broadcom ASICs do so without using the OF-DPA.

Allied-Telesis AlliedWare Plus provides support OpenFlow version 1.3 on a variety of Enterprise devices that use Broadcom ASICs [3]. Allied-Telesis support OpenFlow by running Open vSwitch on the chips, with a cache of flows in hardware. Packets are processed by a software implementation of Open vSwitch by default, and for every new flow the software processes, it installs a rule in hardware applying the same actions to all subsequent traffic belonging to that flow.

The number of hardware flows is limited, from 117 on the IE210L series switches to 8183 on the x950 series switches. Traffic processed by software is orders of magnitude slower than hardware, so this gives an upper limit on the amount of devices that can be active on the device at a time, depending on how many flows each device creates. Because IPv6 addresses are so long, matching IPv6 addresses requires two entries per flow, effectively halving the number of available hardware flows.

Some features are not able to be supported in hardware, depending on the device. For instance, the x530 series (Enterprise switches with 20–40 1Gb ports and 4 10Gb ports) support the hardware flows using an ACL system, and therefore cannot match MPLS headers, or modify MAC addresses for multicast traffic in hardware. These flows are handled in software instead (suffering a significant reduction in performance).

Arista EOS supports OpenFlow version 1.0, and allows for the use of a routing recirculation-interface to process packets multiple times. This sets a port into MAC loopback mode, meaning all packets output to the port will immediately be redirected to the switch. This does not require a transceiver to be present,

and can recirculate packets at the maximum bandwidth for the port.

4.2 Nvidia

Nvidia supports OpenFlow with the Onyx Operating system [62] on Spectrum ASICs. The Nvidia pipeline consists of 250 flexible ACL tables, able to support a variety of matches and actions, followed by two fixed function tables: *FDB*, for performing layer 2 switching, and *Router*, for performing prefix matching on IP destination addresses. The matches used in the ACL tables must be manually defined in the device configuration on a per table basis, but this does not allow configuration of masks, meaning all matches are ternary.

The Nvidia implementation always runs in hybrid mode, and determines what packets are controlled by OpenFlow by VLAN. This means that the devices apply VLAN filtering before the OpenFlow pipeline starts, and after the pipeline ends.

The FDB table entries match VLAN and Ethernet Destination, and can either drop, output, or direct packets to a select group, to load balance across multiple ports. On a table miss, packets are sent to the **NORMAL** pipeline (§2.2.8.4) to be flooded according to the packets' VLAN tag.

The Router table has a longest prefix match on IPv4 or IPv6 destination, and can either drop the packet; or set the Ethernet destination, decrement the TTL, and either output the packet directly or send it to a select group. Notably, the router table cannot set the Ethernet Source address or rewrite a VLAN, two actions usually associated with Layer 3 routing. Rewriting the source MAC address before packets reach the routing table is practical, but this is dissimilar from other hardware routing tables in a way that makes mapping

difficult. However, not being able to rewrite VLANs makes the routing table very difficult to use, as the new VLAN depends on the matched route, so the table effectively would need to be recreated in the ACL tables with VLAN actions.

The Nvidia OpenFlow implementation has a single 12-bit metadata field, and can recirculate packets by setting an interface to recirculation mode.

4.3 Cisco

The OpenFlow implementation in Cisco IOS for Catalyst devices consists of a pipeline of up to 16 flexible tables [25]. The tables are configured with `Table-Features` messages, to define the match fields, masks, actions and next tables each table can use. As such the Cisco cannot use `lpm` matches.

The Cisco OpenFlow implementation can only set the VLAN VID, and Ethernet Source and Destination fields. It only supports `indirect` and `all` groups, and does not support metadata. Cisco Catalyst devices support recirculation natively—they can buffer packets and reprocess the headers from the start of the pipeline.

4.4 Aruba

HPe ArubaOS supports OpenFlow on a variety of switches (2920, 2930F, 2930M, 3810, and 5400R zl2 series switches) [43]. Aruba uses a pipeline of up to 12 flexible tables, configured with `Table-Features` messages. Like the Cisco, the Aruba pipeline can only use `exact` or `ternary` matches.

The Aruba allows setting any field, however it cannot set the Ethernet source

address, or Layer 4 fields for IPv6 packets. The Aruba also cannot decrement TTL fields. The Aruba does not allow setting metadata and cannot perform recirculation.

4.5 SAI

The SAI is supported by a variety of vendors with a variety of ASICs, including Broadcom, Nvidia and Intel [77]. The SAI pipeline therefore is a valid target for a mapping algorithm. This analysis is based on the Behavioural Model (a P4 architecture for a software switch) implementation, as it is clearly specified in P4 [94].

The Behavioural Model implementation of SAI has 34 tables, but does not include ACL tables. This thesis assumes that all implementations of SAI can support a simplified ACL table based on the cross section of capabilities from the Nvidia and Broadcom ACL tables, that can be applied wherever an ACL table can be bound.

The SAI pipeline makes extensive use of metadata, and many tables simply set metadata fields. For instance, the table *Ingress LAG* simply matches the ingress port and sets two metadata fields: the `ingress_metadata.is_lag` flag and the `ingress_metadata.l2_if` field, indicating whether the packet arrived on a LAG port and the L2 logical interface, respectively. Other metadata fields include the VRF and Spanning Tree Protocol (STP) ID.

The pipeline is strictly defined, but there are multiple paths through the pipeline that packets can take. For instance, routed packets can be output to a port directly, or forwarded to the VLAN bridge section of pipeline, to perform Ethernet switching.

4.6 Other OpenFlow Implementations

Noviflow supports OpenFlow with NoviWare, an operating system for whitebox switches using Intel Tofino and Nvidia NP5 ASICs [76]. The programmability of these devices allows Noveware to offer close to a complete implementation of OpenFlow, however these devices are targeted to high-speed data centre networks, and would be unsuitable for most enterprise use cases.

Corsa created high throughput hardware that supports OpenFlow, however the details of their support was not published publicly. These devices were also large and expensive and therefore unsuitable for most enterprise use cases.

A number of other vendors provide single table OpenFlow support, including Nokia [75], and Pica8 [93]. Extreme [30] support a multiple table pipeline, but only one table can match fields other than MPLS label. H3C [70] use a 4 table pipeline that provides a subset of the functionality of the OF-DPA pipeline.

4.7 Discussion

Based on this analysis, this thesis targets Aruba, Broadcom, Cisco, and Nvidia hardware, as well as any hardware implementing the SAI. The alternative hardware is either too large and expensive for use in enterprise networks, or is too limited in its functionality.

4.7.1 Features

Table 4.1 shows how the target hardware supports various features that are valuable when mapping pipelines.

All of the hardware can use recirculation, with the exception of the Aruba. Port

Vendor	Configurable Tables	Recirculation	Metadata
Aruba	12	No	No
Broadcom	No	Port-Based	VRF
Cisco	9	Native	No
Nvidia	250	Port-Based	12 bits
SAI	No	N/A	Various fields

Table 4.1: Support for various pipeline features in the target hardware. Flexible Tables refers to devices with tables that can be configured to support different matches and actions. Recirculation indicates how the hardware can recirculate packets. Metadata indicates what metadata fields are available.

based recirculation means that the bandwidth on recirculated packets will be limited to the bandwidth of the port. However, this could be mitigated by using multiple ports to recirculate packets. The impact of packet recirculation is discussed further in Section 6.2.

Metadata was poorly supported by the target hardware. This may be in part due to the difficulty of implementing it exactly as defined in the OpenFlow specification. As it is not practical to support metadata exactly as defined, vendors must come up with a compromise solution, but without a specification defining what that should be, it is possible that some vendors chose not to commit to a solution that might not suit users' needs [44].

The pipelines of the target hardware fits into two categories:

Configurable Pipelines: The Aruba, Cisco and Mellanox pipelines have tables that can be configured to match any field and use any actions. In all three cases the tables are identical and can be arranged using `Goto-Table` instructions. This greatly simplifies mapping, as, provided the hardware can support every table in the virtual pipeline, the mapping will always be possible.

Fixed Pipelines: The SAI and Broadcom pipelines have tables designed for specific functions that define exactly what matches and actions can be

Vendor	Ethernet Switching	Routing	5-Tuple
Aruba	Two-Table	No	exact
Broadcom	One-Table	Yes	ternary
Cisco	Two-Table	No	exact
Nvidia	No*	No*	ternary
SAI	Two-Table	Yes	ternary

Table 4.2: Support for different common table types in the target hardware. *The Nvidia hardware also supports SAI, and is therefore capable of supporting these tables, but the OpenFlow implementations are inadequate.

used. Occasionally these tables have optional matches or actions, but usually no more than one or two per table.

The Nvidia pipeline also features two fixed-function tables, but neither is practical for its intended purpose. Importantly, both come at the end of the pipeline, so do not interfere with how the configurable tables are used.

4.7.2 Common Table Structures

Table 4.2 shows the support for common table structures by the various pipelines. All the pipelines support some form of Ethernet switching, with the Broadcom using a single table implementation, and the other hardware having multiple tables.

Routing is not well supported by the OpenFlow implementations:

- the Aruba cannot decrement TTL fields, or configure 1pm matches;
- the Nvidia cannot update VLAN IDs or Ethernet source addresses; and
- the Cisco cannot configure 1pm matches.

However, all of this hardware does support legacy routing, so it is likely this is a limitation of the OpenFlow implementations, rather than the devices.

5-tuple matching is a common use case in SDN, however, most devices are only

able to support it with **ternary** matches. This is possibly a matter of how the OpenFlow pipelines are defined rather than a limitation of the devices.

Chapter 5

Target Controllers

This chapter investigates existing SDN controllers, to inform the design of the mapping algorithm. Mapping pipelines requires potentially searching a combinatorial space, and no previous demonstration of a mapping algorithm has mapped a pipeline with more than five tables. This chapter identifies common features of controllers that may simplify the process of mapping pipelines, and enable an algorithm to scale to support real world production pipelines. To do this, this chapter examines data collected from 6 SDN production networks, and investigates 50 projects from SDN research literature.

5.1 Features of Interest

The investigation in this chapter focusses on the following features:

Actions: This chapter looks at how frequently controllers use each type of action and action instruction, how often they use groups, and how often actions or instructions affect how an algorithm can re-order tables.

Matches: This chapter looks at how frequently controllers match each field,

and with what kinds of matches. If controllers frequently use matches that cannot be supported by the target hardware, that suggests that either those controllers should be out of scope for this work, or that the target hardware is inadequate for the purposes of this work.

Common Table Structures: This chapter identifies commonly used table structures. When the same table is used by many controllers, that could simplify a mapping algorithm, by enabling shortcuts for such tables.

Table Size and Update Rates: Arbitrarily merging tables multiplies the number of entries, harming update rates and memory usage. When tables have few, mostly static entries, merging them may not have a significant impact. This chapter identifies how frequently controllers use such tables.

Table Order: This chapter identifies scenarios where tables cannot be reordered. Aside from the scenarios relating to the actions used (mentioned above), tables generally cannot be re-ordered when one table determines what packets are applied to a subsequent table.

Common Transactions: Finally, this chapter examines common transactions in production networks, and whether these indicate inefficiencies in the standards these networks use.

5.2 Production Deployments

Analysing production deployments provides functionality that is both useful to operators, and supportable by current hardware. It also provides a real-time view of how the controller and the datapath interact.

This chapter analyses data collected from 5 Faucet deployments [8], and from the Cardigan deployment between the Research and Education Advanced Net-

work New Zealand (REANNZ) office and the Wellington Internet Exchange (WIX) [103]. While this is a small sample, lacking in diversity, unfortunately obtaining data from production deployments is difficult due to security and privacy concerns.

5.2.1 Data Collected

The data collected consists of:

1. periodic snapshots of datapath state (as JavaScript Object Notation (JSON) representations of OpenFlow Flow Stats messages), demonstrating the packet processing behaviour of the datapath;
2. the OpenFlow control channel (in Packet Capture (PCAP) format), showing the interaction between controllers and datapaths, and how the state changes over time.

From the raw data, we identified classes of entries that are suitable to be supported by a single P4 match–action table or conditional statement, and recorded the following details for each:

Table id: the OpenFlow table id

Priority offset: the lowest priority value

Priority range: the range of priority values

Match fields: the fields matched

Action sets: the combinations of actions used

Timeout: whether entries of this class timeout

Immutability: whether the entries are ever updated

Match kind: the match kinds used

Max entries: the maximum number of entries

Total updates: the total number of times entries of this class are added, modified or deleted

Max update rate: the maximum number of updates in a one second period

Related classes: other entry classes that are frequently updated in the same transaction

5.2.2 Identifying Entry Classes and Match Kinds

An *entry class* is a set of OpenFlow table entries that can be supported with a single P4 component (a conditional or a table). OpenFlow controllers often combine multiple, unrelated tasks into a single table, to work around the restrictions placed on tables by hardware implementations. Mapping entry classes is more flexible than mapping OpenFlow tables directly, and provides similar benefits to the table compression used by Sanger, Luckie, and Nelson (§3.2.3).

This analysis classifies entries based on the fields matched and priorities. Entries in the same table with the same match fields, or with overlapping priority ranges, are considered the same entry class.

Exact match entry classes are the simplest entry classes to identify. When a table has multiple unmasked entries matching the same fields, with the same priority, then those entries can be supported in a P4 table with `exact` match kinds.

`1pm` matches all match the same fields, and only one field is masked. The mask

is always a prefix, and for any pair of entries that match an overlapping set of packets, the one with the longer prefix must have a higher priority. It is not necessarily the case that `lpm` entries with a longer prefix must always have a higher priority than entries with shorter prefixes, provided the entries do not overlap.

Tables with `ternary` matches can be divided into individual tables in a variety of ways. For the purpose of identifying entry classes, we consider any entries in the same table matching the same fields to be the same entry class, and any entry classes with overlapping priorities with another entry class are considered the same entry class.

5.2.3 Identifying Transactions

When a controller updates the state of a datapath, it often requires sending multiple messages between to accomplish a single logical goal. For instance, when a datapath notifies the controller that a port has gone down, the controller may need to update multiple tunnels to direct traffic to a different port. When a single transaction requires updating a lot of entries, that may be an indicator that the design of OpenFlow or its implementation is forcing the controller to use an inefficient table structure.

The algorithm for identifying transactions reads a PCAP file of the control channel, and uses heuristics to award points to sequences of messages that make up potential transactions. The messages that make up a transaction must be contiguous, and must not be separated by significant delays. When a transaction is clearly identifiable, the algorithm finalises it, removes those messages from the pool of unconfirmed messages, and then repeats.

The heuristics include:

- whether messages reference similar field values;
- whether multiple potential transactions consist of the same messages;
- whether a transaction is similar to a previously confirmed transaction;
- whether the transaction contains redundant messages;
- whether a transaction consists of a batch of deletes followed by a batch of adds, or vice versa;
- when `Barrier` messages are used (§2.2.8.2); and
- whether messages are already favoured for a different transaction that has not been finalised yet.

5.2.4 Production Deployments

The production deployments this chapter investigated were:

Redcables: The WAND Redcables network is a Faucet deployment at the University of Waikato, providing Internet connectivity to the labs and offices used by the WAND research group. It consists of 18 datapaths in total, 5 Allied Telesis tq4600 wireless access points; 4 Allied Telesis x230 and 4 Allied Telesis x510 switches providing access; an Aruba 2930, a Cisco 9300 and two Allied Telesis x930 switches for aggregation; and a server running Open vSwitch providing routing. The datapaths are controlled by two instances of Faucet, one controlling layer 2 and one controlling layer 3. The PCAPs and flow table dumps cover a 15 hour period in November 2017.

REANNZ Office: The REANNZ office deployment consists of a single Allied Telesis x930 controlled by Faucet. It provides layer 2 connectivity to the

REANNZ main office in Wellington. The data covers a week in October 2017.

eResearch and NZNOG: The conference networks for eResearch 2018 and NZNOG 2018 and 2019 used Faucet controlled datapaths to provide layer 2 connectivity. eResearch and NZNOG 2018 used an Allied Telesis x930 and two Aruba 2930 switches, NZNOG 2019 used an Allied Telesis x930 and a server running Open vSwitch. eResearch ran for 3 days, the NZNOG conferences ran for 5 days.

Scinet: The network for the 2018 Supercomputing conference included a Faucet deployment providing Internet access to exhibition booths. The deployment consisted of a Noviflow 32x100 switch with a Tofino ASIC performing routing; and two Cisco 9500 switches, an Allied Telesis x950, an Allied Telesis x908, and a Noviflow 2122 for aggregation. The data covers 8 days including set-up and testing as well as the main conference.

Cardigan: Cardigan was a deployment of Routeflow, an OpenFlow based router, connecting the REANNZ office to the WIX in 2014. Cardigan consisted of a Pronto 3290 and a Pronto 3780 deployed at the REANNZ office and at the WIX, providing layer 3 connectivity between the two sites. Openflow channel data was not available from this deployment, this analysis only uses flow table data.

5.2.5 Updates

How frequently controllers update table entries affects how the tables can be mapped to hardware. If a table is updated infrequently, then it is less important to ensure the update rate is not affected by the mapping. For instance, entries could be split so that fields are matched across multiple tables, or they could

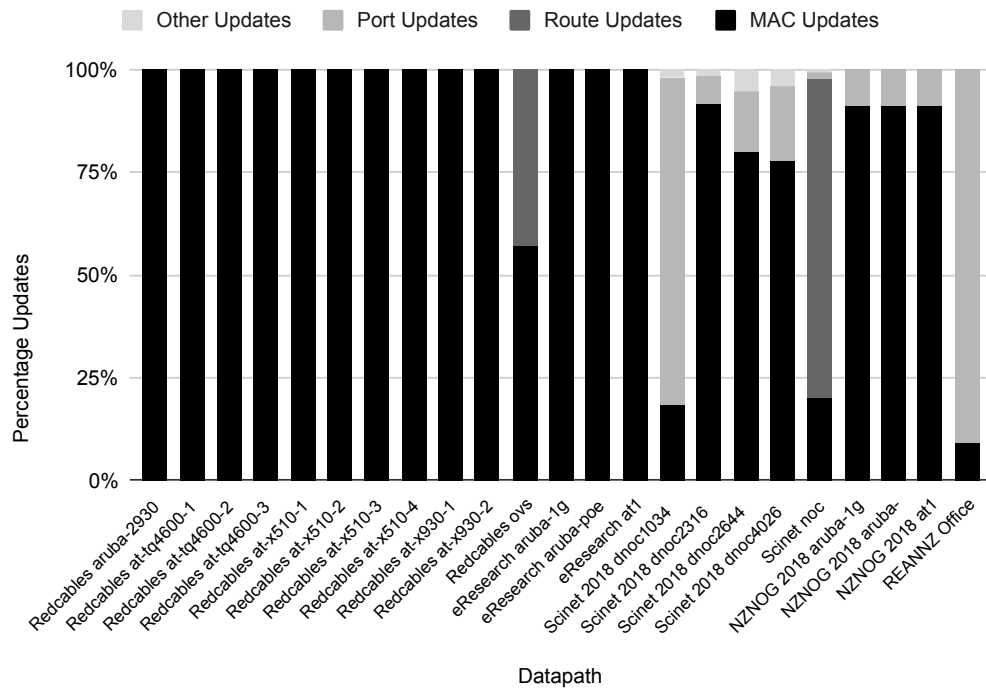


Figure 5.1: The percentage of updates to datapaths for each type of entry class.

be mapped to tables that use slower match kinds.

Figure 5.1 shows the percentage of updates relating to each type of entry class. The overwhelming majority of updates occur in entry classes relating to Ethernet switching, routing, or ports and VLANs. Many datapaths only received updates relating to Ethernet switching. There were only a handful of other entry classes that received more than 1% of the total updates. The Scinet deployment included a deployment of Poseidon [45], a network security application that could automatically install ACLs in response to anomalies it detects in the network. ACL updates accounted for up to 4.1% of updates to Scinet datapaths.

Port and VLAN updates accounted for up to 18.25% of updates, with the exception of two outliers: one of the Scinet aggregation datapaths, dnoc1034, had 85.6% of its updates for port state changes, and the REANNZ office net-

work had 90.9% of its updates for port state changes. Scinet dnoc1034 had the fewest connected hosts of any of the Scinet devices, and those hosts were stable servers where the MAC addresses did not need to be relearned often. Most of the port state changes are from the set up and testing stage rather than the conference itself.

The high rate of messages relating to port state changes in the REANNZ office was caused by inefficient handling of VLAN flooding, to accommodate incorrect OpenFlow hardware implementations. The OpenFlow specification requires that a device should not forward packets to the packet's ingress port, unless the action specifies virtual port `IN_PORT` as the egress. Due to hardware implementations that did not correctly implement this behaviour, Faucet created separate flooding rules for each VLAN on each port. When a port changed state, Faucet would update every rule associated with every VLAN on that port. The REANNZ office had many VLANs, and provides wired connections to staff laptops, which were frequently connected to, and disconnected from the network. This resulted in a peak rate of 295 messages relating to port updates in a second. Faucet has since been modified to handle VLAN flooding more efficiently.

5.2.5.1 Table Size

The size of tables impacts table mappings in a number of ways. First and foremost, physical tables must be large enough to hold as many entries as the virtual table requires. The size also impacts how practical it is for tables to be aggregated with other tables, aggregating large tables may exhaust the number of entries, and small tables may be small enough to be combined with entries from another table, provided the update rate is also low.

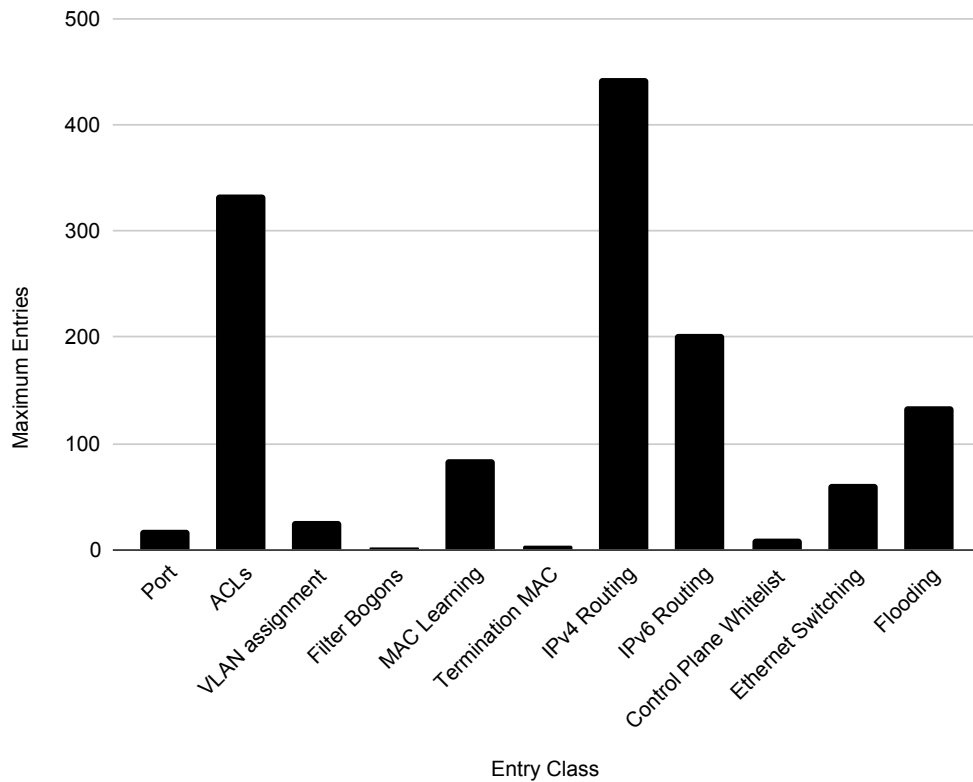


Figure 5.2: The maximum number of entries for each entry class in the Redcables OVS datapath

The entry classes with the most entries, for the most part, were those that were updated the most frequently. The exception was ACL entry classes, which could be moderately large, but received few updates. Figure 5.2 shows the maximum number of entries for each entry class in the Redcables OVS datapath, which is typical of all datapaths.

As the smaller tables often had single-digit numbers of entries and were rarely updated, mapping software could easily aggregate them into other tables, and physical hardware could support them with a flexible table such as an ACL table.

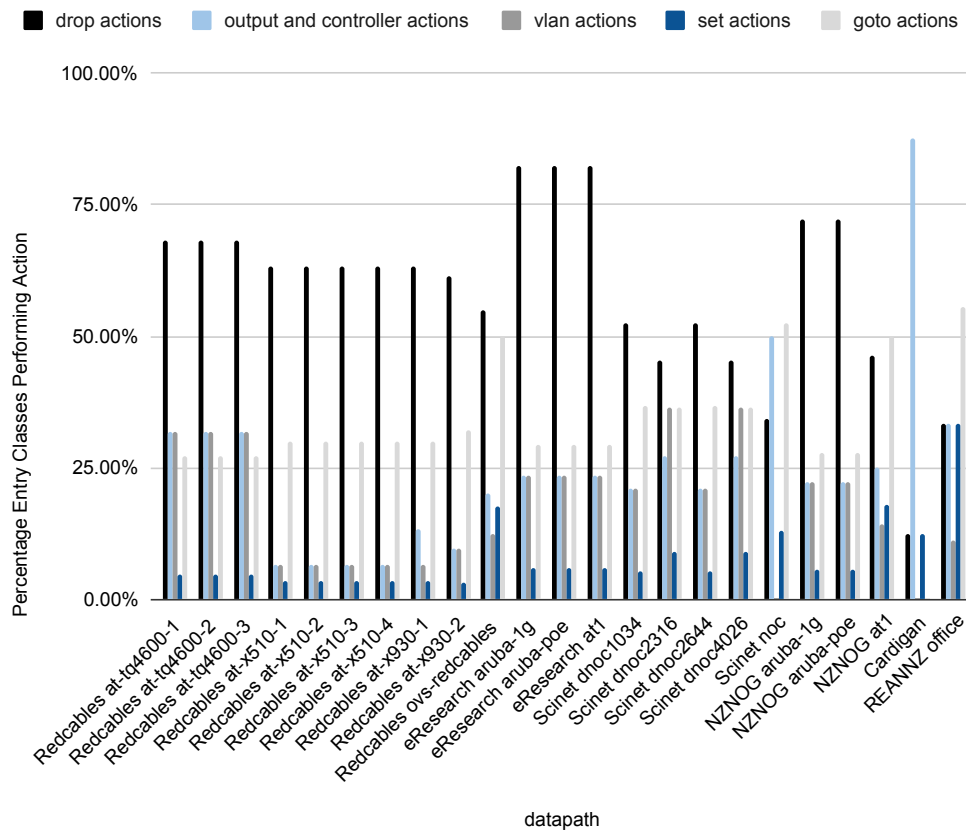


Figure 5.3: The percentage of entry classes that perform a particular action. Note that entry classes may have multiple possible action sets and that sets may include multiple types of actions.

5.2.5.2 Actions

Actions can affect how the tables in a pipeline can be re-ordered. Certain actions may need to be applied to packets in a specific order, for instance, if two tables set the same field the second table must supersede the first. Furthermore, actions can affect what entries are applied in subsequent tables.

Figure 5.3 shows the percentage of entry types that perform each action. The most common action for an entry class to apply to a packet is to drop it. This is beneficial when mapping a virtual pipeline to a physical pipeline as, for the most part, it does not matter where in the pipeline a packet is dropped. The

only actions that are impacted by a drop being applied later in the pipeline is a clone action, a count action, or an action that modifies the packet so that it will no longer match the drop entry class.

Modifying a packet and then subsequently matching on the modified field is a difficult operation for hardware to support. All of the Faucet datapaths set VLAN ID fields and then match those fields in later tables. Only Scinet noc and Redcables ovs do this with another field: both datapaths set the Ethernet destination field when routing packets, then matched that field to determine the correct output port for the next hop.

5.2.5.3 Transactions

We identified three main types of transaction that require multiple table updates. The first, and by far the most frequent—making up 100% of all messages to some datapaths—were Ethernet switching updates. In the OpenFlow model, Ethernet switching is typically handled in two tables; the first table is responsible for learning the port association for each MAC address, and the second is responsible for forwarding packets based on their Ethernet source address.

The second transaction updated flooding rules, as described in Subsection 5.2.5.

The third transaction, and the least common, is when a controller resolves a next-hop. Typically this results in multiple routes and the Ethernet switching table being updated. It is possible to efficiently update next-hops in OpenFlow with Metadata or with Group tables. Faucet does not do this, however, because of inconsistent hardware support.

Overall, this analysis suggests that the OpenFlow protocol is reasonably efficient at updating tables, with the only obvious area where improvement could

be made is with layer 2 switching and flooding.

5.3 Research Projects

Research projects provide insight into potential uses of the P4 architectures. Research projects are not necessarily limited by support from current hardware, they are often evaluated on software switches that have no limitations in terms of supported features. Investigating research projects also allows covering a much larger number of applications, as the details are publicly available, unlike production deployments, where details are very difficult to obtain. However, research projects focus on novel concepts, and therefore can overlook basic networking functions that are vital for production deployments.

Research projects are proposals that may not be fully developed, or may ultimately be impractical for production deployment. However, even if a project is ostensibly of low value to network operators, enabling support for it in our system allows researchers to experiment with physical hardware, improving their ability to find ways to make the project more useful.

Research projects often are focussed on a specific feature, rather than a complete networking solution, and their design often reflects that. For instance, many research projects use reactive forwarding [29, 33, 38, 46, 61, 64, 65, 68, 72, 73, 95, 98, 107], where every flow that arrives on the datapath is initially forwarded to the controller, which then installs rules on the datapath to handle how that flow should be handled. This is often suitable to demonstrate a concept, but would not be practical in many production networks [35].

To mitigate these issues it is important to look at a large number of research projects, that cover a variety of network types and applications. As OpenFlow

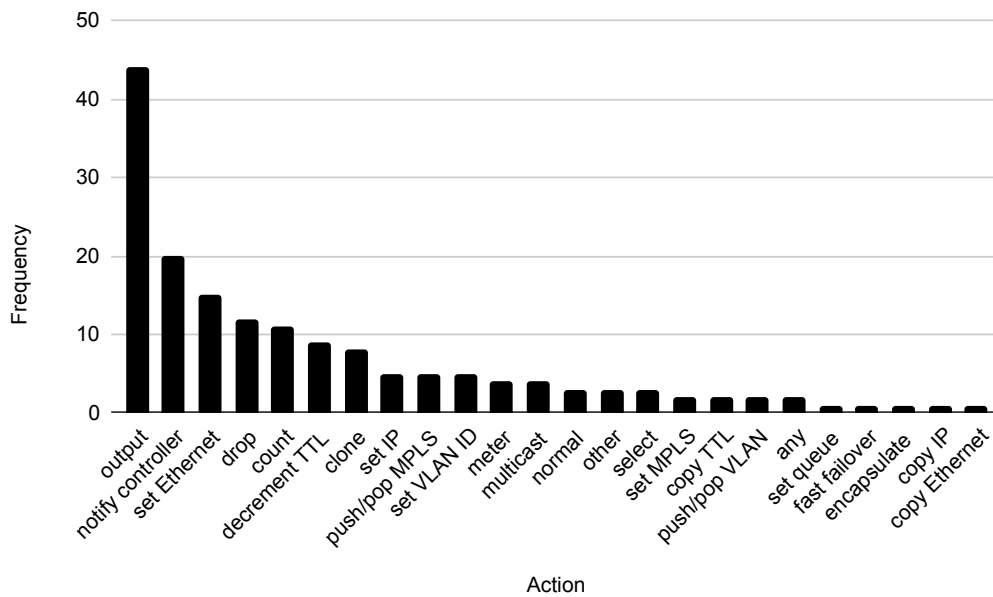


Figure 5.4: The number of controllers using each type of action

is supported by all the target hardware, and the target hardware cannot support arbitrary parsers, this analysis targets controllers that only match against header fields defined in OpenFlow. The projects must also provide enough detail about its packet handling requirements that this analysis can create a reasonable estimate of a virtual pipeline. The chosen research projects are listed in Appendix A.

Often research projects only explain the details that directly relate to the specific feature they are experimenting with. Consequently, this analysis infers certain aspects of the pipeline. For instance, many layer 3 projects omit any mention of layer 2 termination. In such cases, this analysis assumes that the controller runs ARP and/or IPv6 Neighbour Discovery, and has a termination MAC table to control access to the layer 3 tables.

5.3.1 Actions

Figure 5.4 shows the number of controllers that used each action. Output was by far the most frequent, but some controllers did not output packets. For instance, the OpenFlow discovery protocol [7] was only used to find links between datapaths, and, while in practice, it would be used alongside other controllers that would forward dataplane traffic, by itself it does not need to. Many actions are likely to be much more common in production networks rather than in research. For instance, VLAN actions are often unnecessary in a testbed, but might be necessary to partition a live network that carries traffic for multiple purposes.

Different vendors support different types of set fields. Only one controller modified Transport layer fields [1], and otherwise no controller modified fields other than source and destination addresses in Ethernet and IP headers, MPLS Labels, or VLAN VLAN ID (VID)s. Two controllers used copy actions:

- Afek, Bremler-Barr, and Shafir [1] reversed the IP and TCP source and destination, and set the sequence and acknowledgement numbers, to force remote hosts to resend TCP syn packets; and
- Bruyere et al. [21] used a shift operation of Ethernet addresses, to pop labels encoded in the address fields.

Copy actions were introduced in OpenFlow version 1.5 [84] and are not supported by any of the target hardware.

Faucet [8] was the only controller to use `Apply-Actions` to set fields and output packets mid-pipeline. Consequently, it was the only controller where table order was relevant for packet outputs. Faucet was also the only controller to have multiple tables rewrite the same field, and no controller required using

Write-Actions.

OpenFlow keeps counts of matching packets for every rule, however only 13 controllers explicitly required this [8, 14, 32, 38, 39, 51, 58, 61, 65, 72, 98, 108, 111]. Removing this requirement simplifies hardware support, but can also enable re-ordering of tables. For instance, an OpenFlow pipeline that has multiple tables that drop packets must apply those tables in the specified order, to ensure the packet counts are updated correctly. If these tables do not need to count packets, then the table ordering is irrelevant.

Controllers used Group tables infrequently. Only three applications used select groups [31, 96, 107] and one used fast-failover [7]. It is likely that for many of these controllers, group tables would be valuable in a production deployment to allow controllers to update actions more efficiently, but they were not directly relevant to the published work.

5.3.2 Match Fields

Figure 5.5 shows how many controllers use tables matching each field, omitting fields that determine protocols, such as Ethernet Type or IP Protocol, as these are compulsory matches when matching higher layer protocol fields.

16 of the controllers we investigated used some form of flow matching. Some specified matching all possible fields, some specified matching 5-tuples, and others did not specify the fields used to match flows. For all cases, this thesis assumes that 5-tuple matching is suitable, unless there is a specific reason to match other fields. Fayazbakhsh et al. [33], for instance, match the flow and the IPv6 Flow Label, where they encode data about the packets progress through the network.

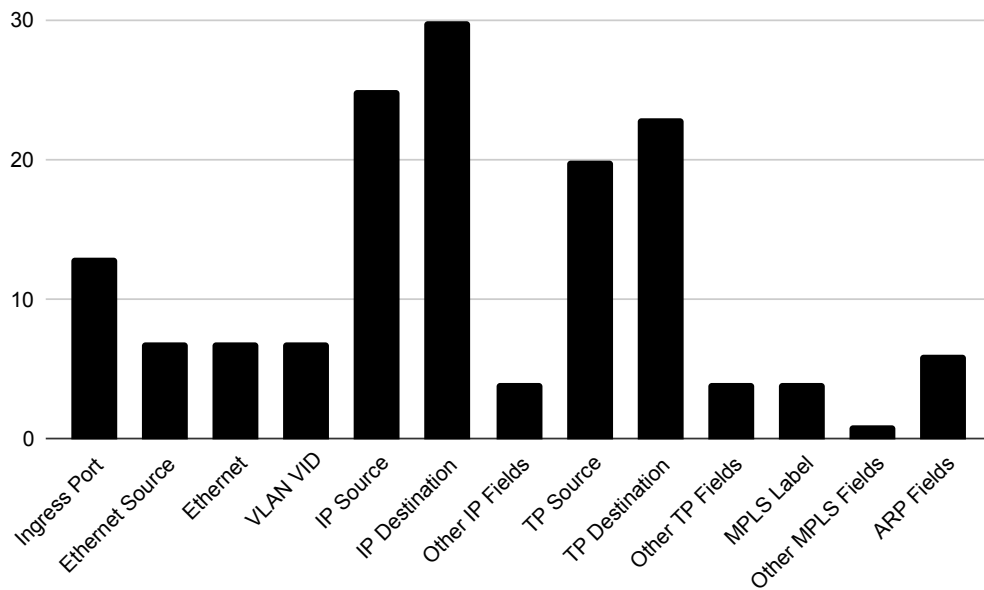


Figure 5.5: The number of controllers that use rules matching each field.

No controllers we looked at used metadata. However, metadata can be used to support features used in legacy networks, such as VRF.

46 controllers used `exact` match kinds, 18 used `ternary` match kinds, and only 4 used `lpm` match kinds.

5.3.3 Tables

The most significant factor impacting how easily an algorithm can map pipelines is how many tables they use, and how they are arranged.

Most controllers required only one or two tables. Figure 5.6 shows how many controllers used each number of tables. Only 9 applications used more than 2 tables. It is likely that when these projects are deployed in production the number of tables required will increase, to handle diverse real-world traffic. Faucet [8], a controller designed for production deployment, used the most tables by far, with 14.

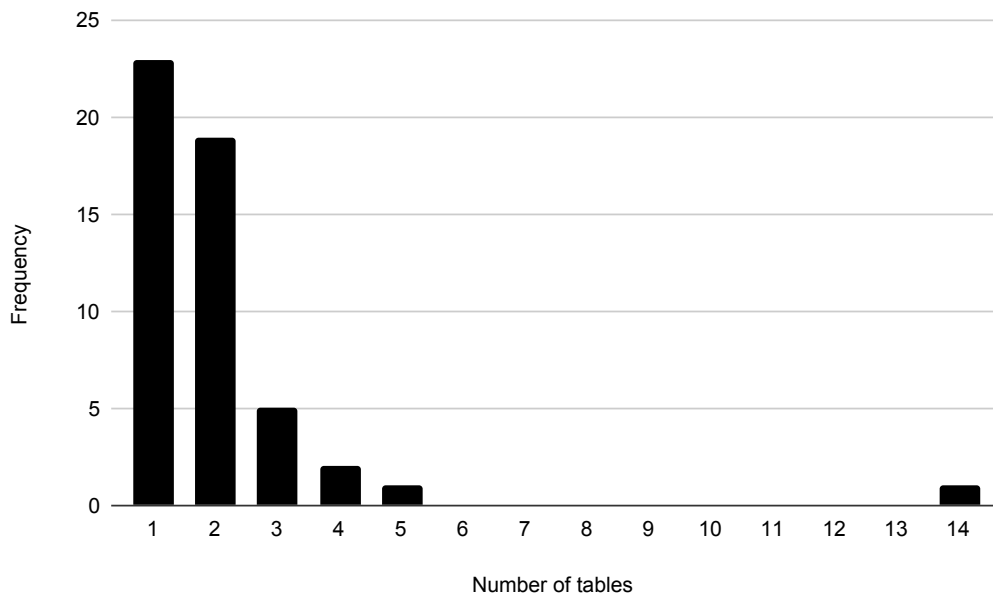


Figure 5.6: The number of controllers using the specified number of tables.

The logic to determine whether a table applies to a packet was extremely simple. Very few tables required more than a single conditional to ensure a table was applied to packets of a specific protocol (eg. all IPv6 traffic). Tables used to control access to another table belonged to three categories:

1. a filtering table, where specific traffic was dropped to prevent it being handled by another table;
2. a cache, where packets belonging to known flows are handled directly, and unknown packets continue through the pipeline;
3. a termination MAC table, matching Ethernet destination, controlling access to tables performing layer 3 functions.

Germann et al. was the only controller to require more than one table to govern access to a subsequent table, it used a two layer cache, before packets would pass through the pipeline. The only other controller with notable logic was Faucet [8], which had a single termination MAC table controlling access to

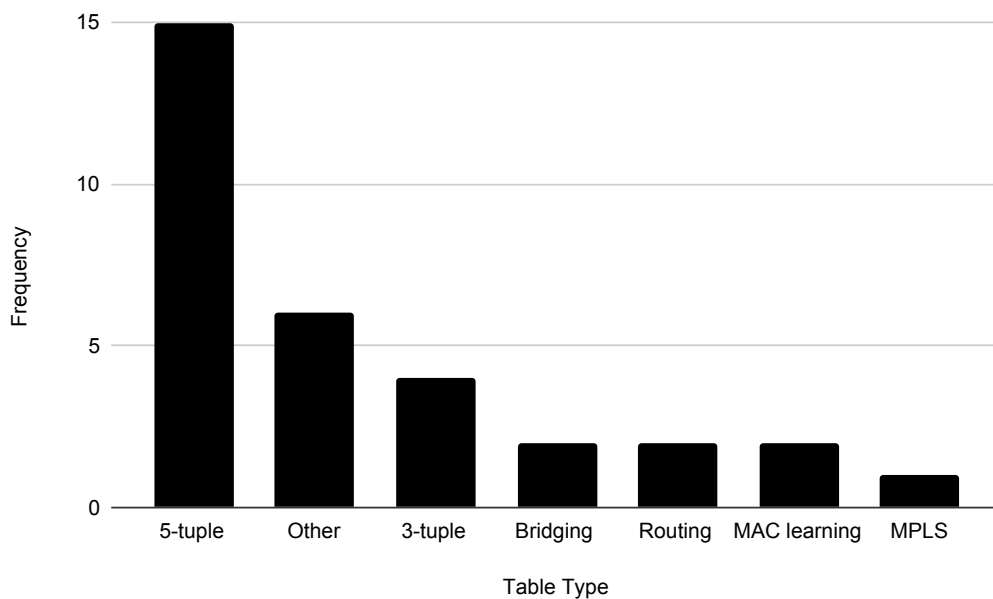


Figure 5.7: The number of tables with high update rates, by table type.

both the routing tables and a table that filters traffic destined for the controller.

5.3.4 Table Entries and Updates

Tables with frequent updates and large numbers of entries are the most difficult to map, as they cannot be aggregated, and must be supported with equivalent match kinds.

Figure 5.7 shows the number of tables that are frequently updated, for each type of table. By far the most common type of table that receives frequent updates are 5-tuple matching tables. The tables classified as *other* were either:

- other forms of flow matching—Van Adrichem, Doerr, and Kuipers used 12-tuple matches [108], and Fayazbakhsh et al. matched IPv6 5-tuples as well as the IPv6 Flow Label, to encode the progress through the network; or
- tables for DDoS mitigation [1, 37] (Germann et al. used multiple tables

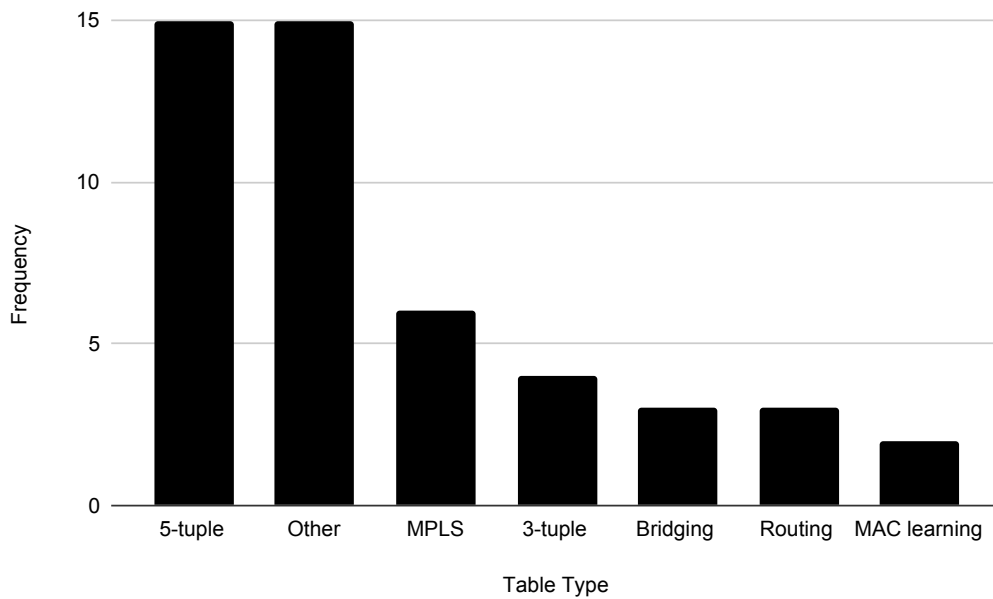


Figure 5.8: The number of tables with high numbers of entries, by table type.

with high update rates).

Tables with large numbers of entries were more diverse, as shown in figure 5.8. However, over two thirds (68%) of these tables were either matching flows or performing standard networking functions.

5.4 Summary

This chapter investigated a variety of SDN controllers from research and production deployments, and found that the design of controllers is promising for the design of a system for algorithmically mapping pipelines in a manner suitable for real-time translation.

Scenarios that would prevent tables being re-ordered or require recirculation were very rare. The access control for tables was very simple, only one table required more than one other table in its path to determine whether that table

should be applied. Likewise, actions were generally not used in ways that force a specific ordering of tables.

The tables that received the most updates generally also used the most entries. For the most part, these tables either matched common legacy networking functions such as layer 2 switching or layer 3 routing, or were used for flow matching. Most other tables had few updates, suggesting they could easily be merged or split without affecting performance.

The main source of inefficiency with existing standards is requiring layer 2 switching to be split into two separate tables.

Chapter 6

Shoehorn Overview

Shoehorn is a proof-of-concept system that demonstrates the practicality of mapping a virtual SDN pipeline to the physical pipeline of a switch in a manner that is suitable for translating control channel messages in real time. Shoehorn consists of:

Shoehorn Physical Architecture (SPA): A P4 Architecture for describing pipelines used by physical devices.

Shoehorn Virtual Architecture (SVA): A P4 Architecture for describing virtual pipelines used by controllers.

Shoehorn Mapping Algorithm: An algorithm for finding mappings between a virtual pipeline defined in the SVA and a physical pipeline defined in the SPA.

Figure 6.1 illustrates the interaction between the components of Shoehorn, and between Shoehorn and the components of the architecture shown in Figure 3.1.

This chapter gives a high-level description of the design of Shoehorn, and describes the motivation behind key Shoehorn features. First, section 6.1 gives

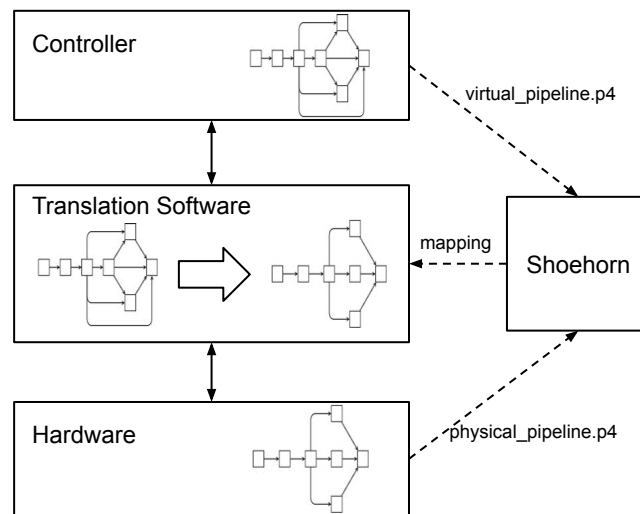


Figure 6.1: An illustration of how Shoehorn interacts with the components of the architecture shown in Figure 3.1. Shoehorn receives definitions of the pipelines from the control-plane software and the datapath, and uses those to find a mapping that the translation software can use. Shoehorn does not perform the translation, as that requires knowledge of the communication protocol, to which Shoehorn is completely agnostic.

a high-level description of the approach Shoehorn uses to map tables. Then section 6.2 describes packet recirculation, discusses the implications of using recirculation, in terms of performance and features, and argues that recirculation is practical for Enterprise networks. Section 6.3 describes in detail the implications of different scenarios where Shoehorn will aggregate multiple virtual tables into a single physical table. Section 6.4 describes how the Shoehorn architectures enable Shoehorn to change the order of virtual tables. Section 6.5 provides a brief explanation of how Shoehorn supports MAC learning. Finally, Section 6.6 provides a brief summary of this chapter.

The P4 architectures are described in greater detail in chapter 7, and the mapping algorithm is described in chapter 8.

6.1 Mapping

A practical mapping from a virtual pipeline to a physical pipeline must meet the following criteria:

- for every action the virtual pipeline applies, the physical pipeline must apply an equivalent action to the same set of packets;
- the physical pipeline must apply actions in an equivalent order to the virtual pipeline;
- when a controller updates a table in the virtual pipeline, the physical pipeline must be able to complete the update in an equivalent amount of time.

The simplest method to create a mapping that meets these criteria is to map every possible virtual table entry directly to a table in the physical pipeline. This method may exclude potential solutions, but the analysis of SDN controllers and hardware implementations in chapters 4 and 5 found few cases where virtual tables could not be easily rearranged to fit physical pipelines. Furthermore, the lack of metadata support by the physical pipelines limits the potential for splitting virtual table entries into multiple physical tables. Consequently, Shoehorn only attempts to find mappings where virtual tables are mapped directly to physical tables.

Shoehorn may re-order tables and other components, such as conditional statements, provided doing so does not affect the matches or actions of other tables. Shoehorn may also aggregate multiple virtual components together so that they are supported by a single physical component, provided this does not impact the rate at which entries can be updated. It would also be possible for Shoehorn to split entries from small, rarely updated virtual tables, so that the

match fields are divided across multiple physical tables. However, this greatly increases the complexity of finding mappings for little benefit, so Shoehorn does not use this approach.

This thesis considers this approach justified because factors that prevent tables from being re-ordered are rare. In the investigated controllers:

- the logic determining whether to apply a table to a given packet is universally straightforward (§5.3.3);
- only a single controller uses **Apply-Actions** instructions to set fields and output packets, or set the same field in multiple tables, and no controller requires the use of **Write-Actions** instructions; and
- only one controller modified fields with **Apply-Actions** instructions and then matched the modified fields in later tables.

Support for common table structures by the target hardware is mixed, but this has little bearing on whether Shoehorn will find suitable mappings. For instance, the fact that the Aruba hardware is unable to decrement a TTL field makes it unable to support routing no matter how tables are mapped.

6.2 Packet Recirculation

Packet recirculation enables Shoehorn to find mappings that apply physical tables in an order other than the order they appear in the physical pipeline. This allows Shoehorn to enforce priority between actions, and enables virtual pipelines to modify packets mid-pipeline, without excluding physical hardware that can only rewrite packets at the end of the pipeline.

6.2.1 Hardware Support

Packet recirculation is widely supported by the target hardware—the only vendor unable to support recirculation is Aruba. The Aruba has a configurable pipeline, and can perform `Apply-Actions` instructions at any table, and therefore is unlikely to need to use recirculation.

Native recirculation, where a packet is buffered, and the parsed headers and metadata is recirculated to be processed again, is only supported by Cisco hardware. The Nvidia and Broadcom hardware use port-based recirculation, where packets output to a specific port are redirected to arrive back at the ASIC from that port. Other devices that do not support this behaviour directly could recreate it using a loop-back transceiver, or a similar method.

6.2.2 Metadata

A limitation of using port-based recirculation is retaining metadata when the packet is recirculated. There are two pieces of metadata that devices must ensure are carried with packets when they are recirculated: the original ingress port, to prevent loops; and how many times the packet has been recirculated, to keep track of where in the pipeline the packet has reached.

Carrying this metadata with the packet may be practical with a single field, as both metadata fields have low cardinality, particularly the recirculation count. However, this requires that the information can be carried in a way that can be extracted by either the parser or a table. For instance, in the OF-DPA, packets can be output to a data centre overlay tunnel. These packets could be recirculated via the tunnel, and the tunnel-id could represent the recirculation and original ingress port. However, data centre overlay tunnels must be sent to

the bridging table. In order to direct packets to a routing table, the datapath would need to use a layer 3 tunnel. The Broadcom chips support layer 3 tunnels, and the OF-DPA supports decapsulating packets from layer 3 tunnels, but not encapsulating packets¹. OpenNSL does support layer 3 tunnels.

Because of these challenges, the SVA does not support user-defined metadata.

6.2.3 Throughput

Having to process packets multiple times reduces the overall throughput of the device. This thesis argues that this a reasonable trade off for low cost hardware, as many enterprise networks are constrained by factors other than device throughput, such as uplink bandwidth or firewall throughput. As an example, a Broadcom Wolfhound BCM5334x series chip provides 64GB/s of packet switching with minimum sized Ethernet frames (64B) [15]. If the bottleneck for a network using such a switch is incoming traffic on a 10GB/s uplink, then recirculating 3 times (meaning each packet is processed 4 times) requires 40GB/s of device throughput, leaving capacity to support 6GB/s of other traffic. Further, Internet traffic is usually considerably larger than the minimum size for Ethernet Frames [66].

Viegas et al. investigated the impact of recirculation on a P4 capable Netronome SmartNIC [109]. They found that one recirculation had no impact on the number of packets per second the Network Interface Card (NIC) could process, and 4 recirculations had no impact on throughput with packets of 256B or larger. Port-based recirculation is limited by the throughput of the port, but this can be scaled up by dedicating more ports to recirculation.

¹The OF-DPA specification references MPLS L3 VPN Label groups, but does not define these groups. It is unclear whether or not they can be used within the OF-DPA.

6.3 Aggregating Components

When multiple components in the virtual pipeline can be supported by a single table in the physical pipeline, Shoehorn may be able to aggregate these components together. However, it is important that when aggregating components, Shoehorn does not reduce the update rate of the virtual tables. Controllers often use tables that have static entries, and do not require real-time updates. Shoehorn allows virtual pipelines to flag such tables, indicating to Shoehorn that it can map them in a way that will reduce the update rate.

To be aggregated together without affecting the update rate, virtual tables must use the same types of matches. For instance, if one virtual table uses **exact** matches, and another uses **ternary** tables, then the physical table would need to use **ternary** tables to support the second table, reducing the update rate of the first table. The overall update rate of a table is dependant on the slowest match kind, so if a table uses **ternary** and **exact** match kinds, then aggregating it with a table that only uses **ternary** matches does not affect the update rate.

The remainder of this section describes scenarios where Shoehorn may aggregate components, and discusses the impact of each on the overall number of table entries.

6.3.1 Cartesian Product Aggregation

When tables are aggregated, a packet may only match one entry in the aggregated table each recirculation. Therefore, a mapping that aggregates two tables that could both be matched by the same packet requires an entry in the physical table for every combination of entries in the virtual tables. There are

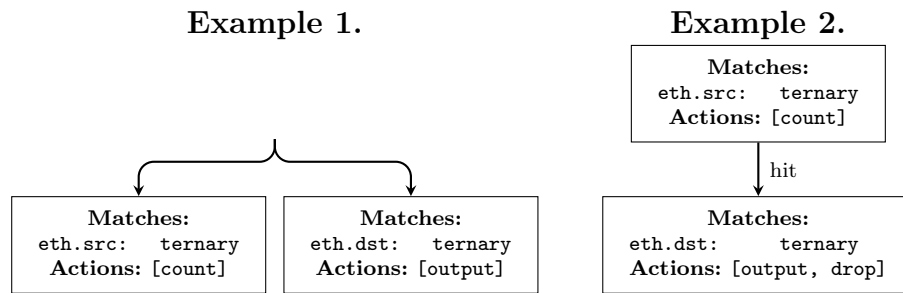


Figure 6.2: Two examples of sections of pipelines that cannot be aggregated without requiring a Cartesian product of table entries in the physical table. The key for this and other diagrams in this chapter is shown in Figure 6.3.

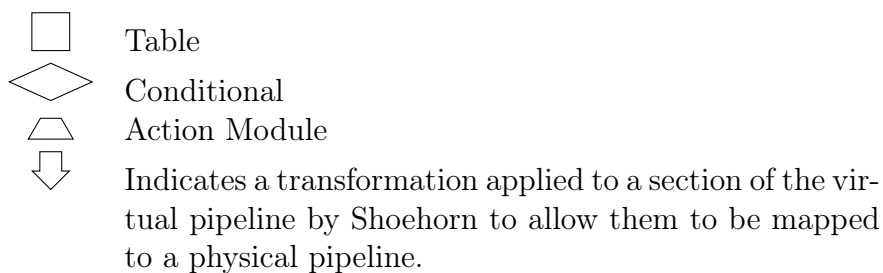


Figure 6.3: The key for the diagrams used in this chapter and in chapter 8. Arrows between components indicate the components applied following the associated evaluation of the conditional or the `hit` field of the table's apply result (§2.3.4). Branching arrows indicate that the pipeline applies multiple components following the associated result.

two scenarios where this may occur:

1. When a pipeline applies multiple tables to an overlapping set of packets, shown in Figure 6.2, Example 1.
2. When a pipeline applies one table to a subset of packets that match another table, shown in Figure 6.2, Example 2.

These situations only apply if the tables use `ternary` matches, or if the tables use different matches. For instance, taking the Cartesian product of entries from two tables that both only perform `exact` matching on Ethernet destination, results in a physical table that matches the total number of Ethernet destinations matched in the two virtual tables. However, aggregating a table that has a `1pm` match on IPv4 destination and an `exact` match on IPv4 source,

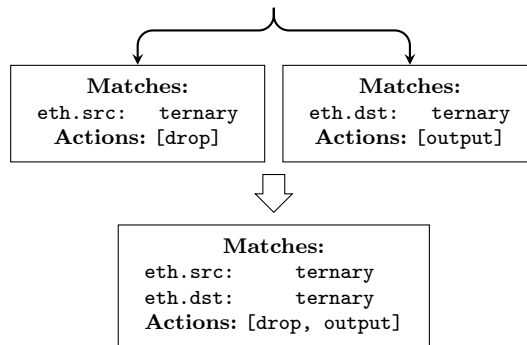


Figure 6.4: A mapping can aggregate a table that drops every packet it matches with other tables without increasing the total number of table entries, even if a packet can match entries in both tables.

with a table that has a `lpm` match on IPv4 source and an `exact` match on IPv4 destination, can still result in a Cartesian product of the two tables.

One other relevant exception to this is when a virtual pipeline has two tables applied to overlapping sets of packets, and one of the tables only drops packets. If the other table only uses actions that are no-ops when applied to packets that are subsequently dropped (for instance, setting an output port or modifying a header field), then the mapping can aggregate the two tables without increasing the total number of entries. The mapping must prioritise entries from the drop table over those from the other table, and as all actions in the other table are no-ops when combined with a drop, the mapping does not need to combine entries. An example of this kind of aggregation is shown in Figure 6.4.

6.3.2 Aggregating Conditionals

Mappings can aggregate conditionals with other components without increasing the number of table entries in the following cases:

- mappings can always aggregate a conditional with another conditional;
- mappings can always aggregate a conditional with a table that uses

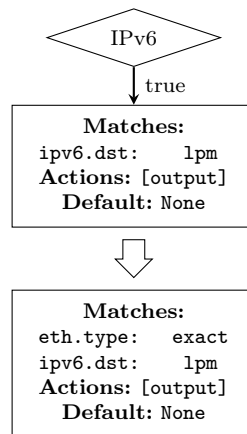


Figure 6.5: A mapping can aggregate a conditional with a table without increasing the number of table entries required in the physical pipeline.

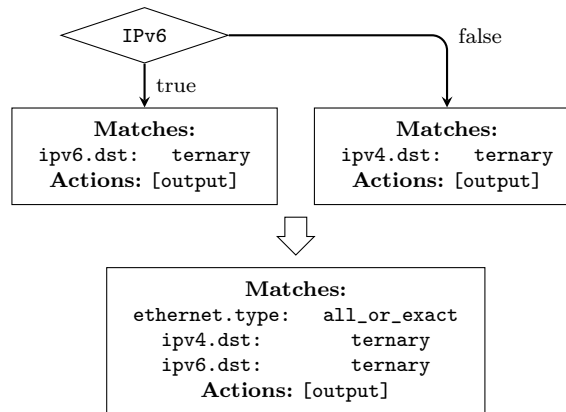


Figure 6.6: Mappings can aggregate mutually exclusive tables without increasing the overall number of table entries.

ternary match kinds; and

- mappings can aggregate a conditional with a table using `lpm` or `exact` match kinds, provided the conditional fields are unmasked, the table is applied when the packet matches the conditional fields, and the table has no default action.

An illustration of aggregating a conditional into a table is shown in Figure 6.5.

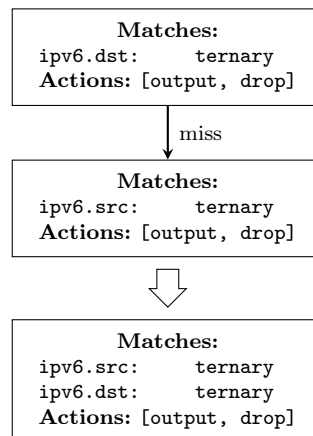


Figure 6.7: Mappings can concatenate ternary tables when the second table is accessed on a miss in the first table without increasing the overall number of table entries.

6.3.3 Aggregating Mutually Exclusive Tables

Mappings can straightforwardly aggregate tables where packets cannot match both tables without increasing the number of entries. Figure 6.6 demonstrates merging two components when the sets of packets to which they are applied are mutually exclusive.

6.3.4 Concatenating Ternary Tables

When a `ternary` table is applied following a miss on another ternary table, a mapping can aggregate these tables by concatenating the two tables, without increasing the overall number of entries. The entries from the first table are given higher priorities and the entries from the second table must apply the default action from the first table as well as their usual actions. This is shown in Figure 6.7.

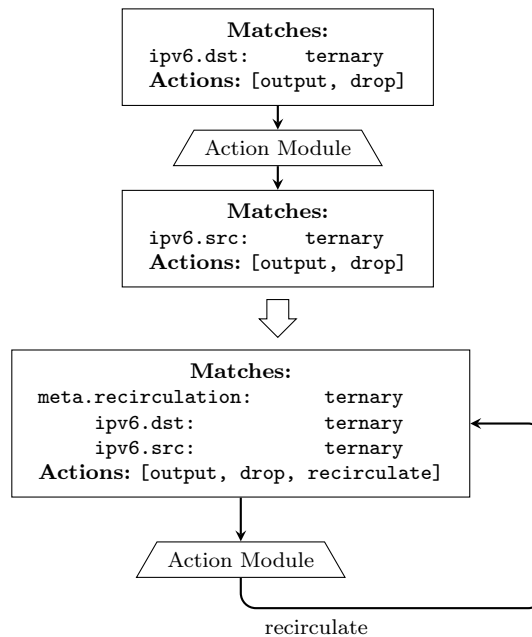


Figure 6.8: Tables separated by a recirculation can always be aggregated without increasing the overall number of table entries.

6.3.5 Aggregation after Recirculation

Figure 6.8 shows an example of aggregating tables after a recirculation. Mappings can aggregate tables after a recirculation, without increasing the overall number of table entries, provided the physical table is able to match recirculation metadata.

6.4 Table Reordering

OpenFlow requires controllers to strictly define the order of tables, but for many controllers, this ordering is arbitrary. However, it is possible that there is no way to define a strict order of OpenFlow tables without creating scenarios that prevent Shoehorn from reordering tables, even when the order between two tables is irrelevant.

For instance, Figure 6.9 shows two tables from a virtual pipeline. The first table

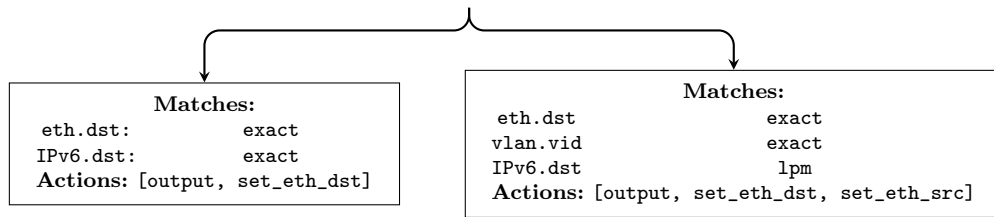


Figure 6.9: An example of two tables that cannot be easily reordered without prior knowledge that the two tables do not match overlapping packets.

directs packets to certain services, provided by hosts directly connected to the datapath. The second table provides general routing. The services are only reachable from hosts in the local LAN, so the controller places a no-op route for the services' subnet in the routing table. As no packet will ever meaningfully match both tables, the ordering between the two tables is irrelevant. Shoehorn, however, does not know that both tables do not apply to the same packets, but knows that matching in one table affects whether a packet can match the second table. Therefore, if the tables are given an explicit order, Shoehorn cannot reorder them.

To mitigate this, Shoehorn includes an extern called an *Action Module*, that allows controllers and hardware to define where the pipeline applies actions. Shoehorn does not include an equivalent to the OpenFlow `Apply-Actions` instruction, and when two tables write contradictory actions for the same Action Module, the outcome is explicitly undefined. Controller developers can choose to have two tables write the same actions for a single Action Module when they know that the actions will not be contradictory for the same packets. If a controller requires an explicit ordering, the developer can separate the tables with an Action Module.

As a consequence of this design, the order of tables in the virtual pipeline is only relevant to Shoehorn when one table provides access control for another table, or when the tables are separated by Action Modules.

6.5 MAC Learning

MAC learning tables are difficult to support in OpenFlow, and this is reflected by poor support for such tables amongst the target hardware. Supporting an exact-match MAC learning table effectively in OpenFlow requires three tables:

1. a table with an **exact** match on VLAN VID, Ethernet source address, and ingress port, that notifies the controller whenever a new Ethernet source address is seen on a VLAN and port combination;
2. a table with an **exact** match on Ethernet destination and VLAN VID, that outputs matching packets to the correct port; and
3. a table matching VLAN VID, that floods packets to every port associated with that VLAN (but not the ingress port).

The separate table for VLAN VID is needed as an **exact** match table cannot differentiate by VLAN following a miss in OpenFlow.

This three table system creates a synchronisation problem for controllers, as they must ensure that the first two tables have consistent entries. This also requires redundant messages, as the controller has to update two tables instead of one. The OF-DPA bridging table avoids this issue, but, as it is a bespoke vendor extension, it is a barrier to portability. Shoehorn provides a generic adaptation of the OF-DPA bridging table, that can be supported by all the target hardware, except Nvidia. Shoehorn includes two new externs, **EthernetLearning** and **EthernetSwitching**. These are two tables that have synchronised entries: the **EthernetLearning** learns MAC address–port associations, and the **EthernetSwitching** table forwards packets to the associated port. The implementation details of these tables is described in more detail in Section 7.10.

6.6 Summary

Shoehorn is a system for finding mappings from virtual pipelines to physical pipelines that are practical for real-time translation. Shoehorn features two P4 architectures, one for virtual pipelines and one for physical pipelines.

Shoehorn ensures that the mappings are practical for real-time translation by finding the mappings offline, and ensuring that updating a table entry in the virtual pipeline requires only modifying a single entry in the physical pipeline (except when explicitly directed otherwise).

Shoehorn maps tables directly, each virtual table is supported by a single physical table. Physical tables can support multiple virtual tables, but only when this does not impact the rate that entries can be updated. While this reduces the likelihood of finding a mapping in the general case, the analysis of hardware and controllers in chapters 4 and 5 suggests that this should have little impact in practice.

The Shoehorn architectures are designed to minimise scenarios where tables must be applied in a specific order. The architectures use an extern called an Action Module to indicate where actions take place, so actions will not affect the result of a look-up in another table in between Action Modules; and when two tables write contradictory actions, the result is explicitly undefined, so the order actions are written is not relevant. The only scenario where table order is relevant between Action Modules is when the result of a look-up in one table determines access to another.

Where these features are insufficient to find a valid mapping, Shoehorn uses packet recirculation, processing packets an additional time. This enables virtual pipelines to modify packets, before matching the modified fields in subse-

quent tables.

Chapter 7

Architectures

This chapter describes the Shoehorn Virtual Architecture (SVA) and the Shoehorn Physical Architecture (SPA). The SVA is used by controller developers to define the pipeline that their software requires, while the SPA is used by hardware vendors to define the pipeline used by their hardware.

The architectures are as similar as possible, only differing when a concept is intrinsic to a physical or virtual pipeline. For instance, configurable tables are only supported in the SPA, as pipelines in the SVA are reconfigured by replacing the P4 definition of the pipeline.

7.1 Packet Paths

The Shoehorn architectures both specify multiple packages (§2.3.1) to accommodate packet processing paths of various lengths. The packet processing paths both start with a fixed parser, which is followed by 1 to 16 pairs of programmable control blocks followed by an Action Module, and then finally a target dependant Buffer Queueing Engine, based on the extern used in the

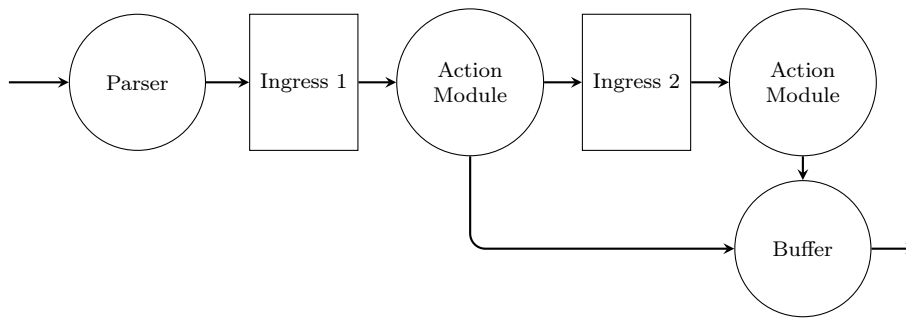


Figure 7.1: An example SVA packet path with two action modules. Square components are programmable control blocks, circular components are not programmable.

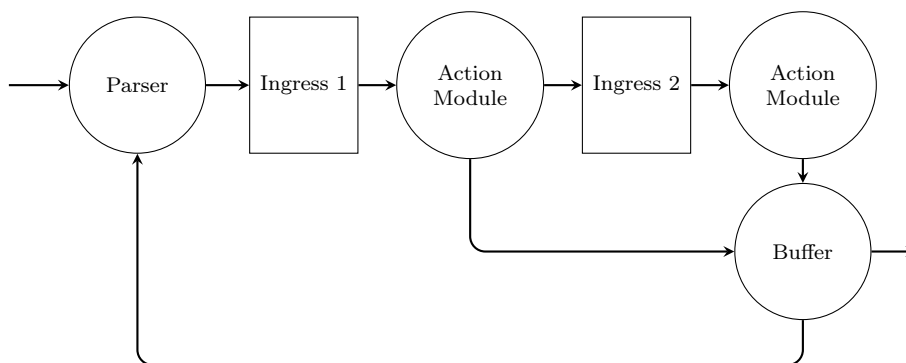


Figure 7.2: An example SPA packet path with two action modules.

PSA [88]. The number of control blocks is limited to 16, as this is the number of tables in the Cisco pipeline, the second highest number of tables from the target hardware. The Nvidia pipeline ostensibly supports 256 stages, but, as Shoehorn aims for portability, it limits the stages to 16. Figures 7.1 and 7.2 illustrate packet paths with two Action Modules in the virtual and physical architectures respectively.

The SPA supports packet recirculation, as discussed in chapter 6.1, but this is not supported in the SVA. When Shoehorn maps a virtual pipeline to the physical pipeline, it will automatically recirculate packets as needed, so it is unnecessary for the SVA to support recirculation directly.

The Shoehorn architectures do not include an egress pipeline. Investigating how the target hardware can support an egress pipeline remains future work.

However, no controllers investigated in chapter 5 used an egress pipeline.

7.2 Parser

The Shoehorn architectures do not allow controllers to define a parser. Fixed-function ASICs are not capable of supporting arbitrary parsers. When a controller requires bespoke headers, more flexible hardware should be used. The parser is modelled on the OpenFlow specification version 1.3 [82] as it is the most widely supported OpenFlow version among the target hardware.

Table 7.1: Header fields set by the Shoehorn parser

Ethernet Destination	Ethernet Source	Ethernet Type
VLAN VID	VLAN PCP	MPLS Label
MPLS BOS	IPv4 DSCP	IPv4 ECN
IPv4 Protocol	IPv4 Source	IPv4 Destination
IPv6 DSCP	IPv6 ECN	IPv6 Next Header
IPv6 Source	IPv6 Destination	IPv6 Flow Label
TCP Source	TCP Destination	UDP Source
UDP Destination	ARP Operation	ARP SPA
ARP TPA	ARP SHA	ARP THA

The parser extracts the header fields shown in Table 7.1 and sets the input metadata described below in section 7.3. The header fields are those used by controllers in chapter 5. Datapaths do not need to be able to set all of these fields to support Shoehorn, as Shoehorn does not guarantee that it will find mappings. For instance, the OF-DPA is the only target capable of matching MPLS fields, but cannot match ARP SPA. When a controller requires the parser to extract a field that the datapath cannot match, the Shoehorn mapping algorithm will fail, because the datapath is unsuitable for the controller.

7.3 Metadata

Table 7.2: Shoehorn control block input metadata

Field	SVA/SPA	Notes
<code>ingress_port</code>	Both	The physical port where the packet arrived
<code>is_tagged</code>	Both	Indicates the presence of a VLAN tag
<code>l3_remote_address</code>	Both	A union of the IPv4 Destination and the ARP TPA fields
<code>l3_local_address</code>	Both	A union of the IPv4 Source and the ARP SPA fields
<code>l4_protocol</code>	Both	A union of the IPv4 Protocol and the IPv6 Next Header fields
<code>l4_source</code>	Both	A union of the TCP Source and UDP Source fields
<code>l4_destination</code>	Both	A union of the TCP Destination and UDP Destination fields
<code>recirculation</code>	SPA only	The number of times the packet has been recirculated

The input metadata for Shoehorn control blocks are shown in Table 7.2. All fields are initialised by the Parser. `ingress_port` and `is_tagged` are frequently used fields, and `recirculation` is needed in the SPA to direct the packet to the correct tables. The union fields enable tables to match the same values from different headers. For instance, `exact` five-tuple matching cannot otherwise be supported in a single table for both TCP and UDP packets.

Table 7.3 shows the intrinsic action metadata for Shoehorn control blocks. The direction of the metadata is `out` in the SVA, but is `inout` in the SPA. This is to support configurable pipelines having greater flexibility over when they apply actions, whereas in the SVA, using `inout` metadata would require the metadata to be carried with packets when they recirculate.

Pipelines initialise all fields to 0, and they cannot be set directly—instead, they can only be set by action primitive externs. Requiring the use of action primitive externs ensures that the fields are set in a consistent manner, sim-

Table 7.3: Shoehorn intrinsic control block action metadata

Field	SVA/SPA	Notes
drop	Both	Instructs the Action Module to drop the packet
pop_vlan	Both	Instructs the Action Module to pop a VLAN tag
push_vlan	Both	Instructs the Action Module to push a VLAN tag
pop_mpls	Both	Instructs the Action Module to pop a MPLS tag
push_mpls	Both	Instructs the Action Module to push a MPLS tag
decrement_ttl	Both	Instructs the Action Module to decrement the IP or MPLS TTL
notify	Both	Instructs the Action Module to notify the controller
clone_spec	Both	Instructs the Action Module to clone the packet, and how the clone should be output
egress_spec	Both	Instructs the Action Module how to output the packet, including multicast
set_<FIELD>_spec	Both	A series of metadata fields, one for each writeable header field, that instruct the Action Module how to rewrite each field
recirculate	SPA Only	Instructs the Action Module to recirculate the packet
goto	SPA Only	The next table

plifying the process of determining whether updates to metadata fields in a virtual pipeline are equivalent to those in a physical pipeline. The writeable fields are Ethernet source and destination, VLAN VID, MPLS Label, and IPv4 Source and Destination.

7.4 Action Modules

In the SVA and SPA, pipelines apply actions to packets in a new extern called an *Action Module*. Whenever control blocks are instantiated, they are followed

by an Action Module. Action Modules recreate, in P4, the different ways to apply actions in OpenFlow: `Write-Actions` instructions, `Apply-Actions` instructions, and group tables.

Action Modules process packets based on the Shoehorn control block action metadata. Controllers configure Action Modules to associate control block metadata with *Action Module Actions (AMAs)*, in a similar manner to an OpenFlow group table. AMAs functions similarly to the equivalent OpenFlow action, when applied to a packet.

Controllers configure Action Modules similarly to OpenFlow group tables. There are separate tables for cloning, egress, and for each writable header field. Action Modules look up each table with the associated metadata specification field, and apply the AMAs found in the table. The egress actions include unicast actions, multicast actions and flooding, and can include modifying header fields in the packet, to allow actions such as popping VLANs when flooding to access ports, for instance. Action Modules apply AMAs in the following order, based on the order used by OpenFlow (the order differs to accommodate differences in the actions used by Shoehorn and OpenFlow):

1. packets are cloned, and the cloned packet is output to a port associated with the metadata value;
2. packets are dropped;
3. the TTL field is copied inwards;
4. MPLS tags are popped;
5. VLAN tags are popped;
6. MPLS tags are pushed;
7. VLAN tags are pushed;

8. the TTL field is decremented;
9. header fields are written, to values associated with the metadata value;
10. the controller is notified;
11. egress AMAs are applied; and
12. packets are recirculated (SPA only).

Datapaths do not need to be able to support all AMAs to support Shoehorn, as Shoehorn does not guarantee that it will successfully find mappings.

Dropping and outputting packets (with an egress specification) are terminating actions: the packet does not continue through the pipeline to the next control block. Output packets, instead, are sent to the Buffer Queueing Engine immediately.

In the SPA, action metadata can be passed through to the next control block. This is because the configurable pipelines have a single table in each control block. Allowing actions to pass through the Action Module means that configurable pipelines can wait for packets to reach the virtual Action Module before applying actions. Pipelines define which actions each Action Modules applies, and which actions are passed through to the following control block. Whenever an Action Module applies an action, it sets the corresponding metadata specification field to 0. In the SVA, however, Action Modules must apply all actions.

7.5 Counters

Shoehorn implements table counters using `DirectCounter` externs (§2.3.6.3), taken from the PSA.

The SVA does not guarantee that counters will be consistent when a controller is run on diverse hardware. When Shoehorn reorders tables, it may result in tables not applying to packets that are dropped in other tables. Consequently, the update counts may be inconsistent depending on how Shoehorn reorders the tables. A potential solution for controllers that require accurate counters is to have two types of counters—the `DirectCounter`, and a `StrictDirectCounter` which always produces consistent packet counts. However, this remains future work. Counters are always consistent in the SPA, as Shoehorn cannot alter the order that physical tables are applied.

7.6 Actions

Table 7.4: Shoehorn primitive action externs

<code>notify</code>	<code>clone</code>	<code>drop</code>	<code>dec_ttl</code>
<code>copy_ttl_in</code>	<code>push_vlan</code>	<code>pop_vlan</code>	<code>push_mpls</code>
<code>pop_mpls</code>	<code>set_eth_src</code>	<code>set_eth_dst</code>	<code>set_vid</code>
<code>set_mpls_label</code>	<code>set_ipv4_src</code>	<code>set_ipv4_dst</code>	<code>output</code>
<code>multicast</code>	<code>goto</code>	<code>recirculate</code>	

Actions in the SPA and SVA may not use logical statements. Instead, they must only call primitive action externs (shown in Table 7.4) and, optionally, the `count` method of a `DirectCounter`. By preventing logic in actions, Shoehorn ensures that it can identify equivalent actions in the virtual and physical pipelines when mapping. The Shoehorn architectures could allow for more efficient code re-use by allowing actions to call other actions, but this remains future work.

With the exception of `count`, the action primitives simply write to the corresponding field in the control block output metadata. Writing to a field that has already been written either fails, or overwrites the existing value, but which

occurs is explicitly undefined. This gives Shoehorn flexibility when re-ordering tables, as there is no explicit order the metadata needs to be written. For instance, if a controller developer knows that no packet will ever match entries in two tables, even if that is not explicitly prevented in the P4 code, they can define both tables in the same control block without having to explicitly define the order they are applied. If the controller developer requires tables to write to the same field with explicit priority, then they should define the tables in separate control blocks.

7.7 Conditionals

The Shoehorn architectures require that conditional statements must only be used in the following ways:

- to compare header or input metadata fields with constant values, or
- to evaluate the `hit` field of a table apply result (§2.3.4).

The only operators that can be used in conditional statements are: *equality* (`==`), *inequality* (`!=`), *boolean and* (`&&`), *boolean or* (`||`), and *mask* (`&&&`). Limiting conditionals to comparisons between fields and constant values, using only these operations, ensures that the conditional statements can be supported using an OpenFlow table, and also simplifies mapping between virtual and physical pipelines. `goto` metadata can only be used with the equality operator.

7.8 Match Kinds

The SVA and SPA both use the match kinds defined in the P4 core library [13]: `exact`, `lpm`, and `ternary`, as well as an `all_or_exact` match kind, taken from the OpenFlow Table-Type Patterns Specification [85]. The `all_or_exact` match kind is included as it is supported by the OF-DPA.

The configurable tables used by the Cisco and Aruba pipelines are able to match any field with `exact` match kinds, and either `all_or_exact`, or `ternary` match kinds, depending on the field. To support such tables, the SPA uses *configured* match kinds: for each of the match kinds mentioned above, the SPA adds a configured counterpart, as well as a `configured_any` match kind. The configured match kinds indicate that the table can be configured to match those fields with the associated match kind, or the fields can be omitted. The `configured_any` match kind indicates that a field can use any match kind.

The Cisco and Aruba OpenFlow implementations do not support `lpm` match kinds. Both vendors support legacy routing with the target hardware, so this thesis assumes that `lpm` matching IP destination is supportable in the configurable tables. Consequently, the `configured_any` match kind indicates that physical tables can support `lpm` match kinds with IPv4 and IPv6 destination fields.

7.9 Annotations

There are two annotations in the SVA that can be added to tables to indicate the tables are able to be mapped in a manner that may reduce the update rate: `flexible_match_kinds` and `flexible_mapping`. `flexible_match_kinds` indicates that Shoehorn can map virtual tables with `exact` and `lpm` matches

to physical tables with `ternary` matches. `flexible_mapping` indicates that Shoehorn may merge two tables with this annotation, even if doing so will result in an increase in the overall number of table entries.

7.10 MAC Learning Externs

The Shoehorn architectures include two externs to implement MAC learning more efficiently than OpenFlow or P4 tables (§6.5). The `EthernetLearning` extern and the `EthernetSwitching` extern can be used to perform Ethernet learning and switching. Both externs are instantiated and applied like tables.

The `EthernetSwitching` extern takes two types of entries. The first matches VLAN VID and Ethernet Destination, and sets the egress specification (indicating unicast to a layer 2 interface) and updates an associated `DirectCounter`. The second entry type has a lower priority, matches VLAN VID, and sets the egress spec (indicating multicast to all layer 2 interfaces associated with the VLAN) and updates an associated `DirectCounter`. The controller can add entries to the `EthernetSwitching` table in a similar manner to a table.

Entries in the `EthernetLearning` extern are automatically populated when entries are added to the `EthernetSwitching` extern. `EthernetLearning` entries match Ethernet Source, VLAN VID, and Ingress Port. The fields are drawn from the Ethernet Destination and VLAN VID of the `EthernetSwitching` entry, and the AMA associated with the egress spec the entry sets.

These externs can be supported either with multiple look-ups in a single table, or with multiple tables, depending on the nature of the underlying ASIC.

Chapter 8

Shoehorn Mapping Algorithm

This chapter describes the Shoehorn Mapping Algorithm in detail.

Shoehorn finds mappings from a virtual to a physical pipeline that maintain the update rate of the virtual pipeline. Shoehorn ensures that updating a single entry in the virtual pipeline requires only updating a single entry in the physical pipeline, except when explicitly directed by a `flexible_mapping` annotation.

Shoehorn's mapping algorithm takes a compiled representation of the physical and virtual pipelines. This consists of a list of control blocks, each containing a list of component trees. The Shoehorn architectures ensure that the order that pipelines apply tables in different component trees is irrelevant. Shoehorn maintains the order of tables within component trees, however, because components in the tree control access to their descendant components.

Tables in the physical pipeline can support multiple virtual components (tables and conditional statements), provided doing so does not impact the update rate of the virtual tables. Shoehorn will only map virtual tables (without the `flexible_mapping` annotation) to a single physical table.

Conditionals cannot be updated and never require more than a single table entry to support, so mapping a conditional to multiple physical components will never affect the update rate, and will have only a trivial impact on the memory usage. Consequently, Shoehorn allows mappings where virtual conditionals are mapped to multiple physical components.

The procedure for finding mappings occurs in three stages:

1. Shoehorn identifies all potential component mappings for each component in the virtual pipeline without consideration of the layout of the pipelines. This is described in Section 8.1.
2. Shoehorn finds mappings that ensure that each table is applied to the correct set of packets. This is described in Section 8.2.
3. Shoehorn rearranges tables in configurable pipelines to ensure that components from multiple component trees are not interleaved. This is described in Section 8.3.

Section 8.4 describes how translation software can use mappings found by Shoehorn to populate tables in the physical pipeline. Finally, this chapter concludes with a brief discussion of the limitations of the Shoehorn mapping algorithm.

8.1 Stage 1: Identifying Supporting Components

In stage 1, Shoehorn identifies every component in the physical pipeline that could be used to support each component in the virtual pipeline. At this stage, Shoehorn ignores the layout of the pipeline, and only looks at whether the physical component can support the same entries as the virtual component.

To support a virtual table, a physical table needs to support all the actions

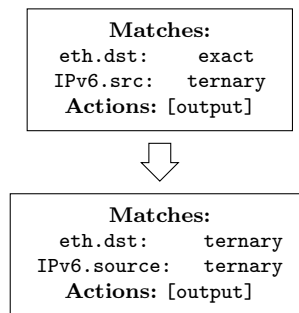


Figure 8.1: The update rate of tables is limited by the slowest match kind used in that table. Because the virtual table uses a **ternary** match for IPv6 source, then it can be supported by a physical table that uses **ternary** match kinds for Ethernet destination, despite the virtual table using an **exact** match. The key for this and other diagrams in this chapter is shown in Figure 6.3.

and matches used by the virtual table. Virtual tables have multiple entries that are updated at run-time, and so cannot be supported by physical conditionals, but virtual conditionals can be supported by either physical conditionals or tables. To support a virtual conditional, a physical conditional must apply the same operators, to the same fields and constant values. A physical table can support a virtual conditional if it can match the same field with an equivalent mask.

Shoehorn ensures that the match kinds used by virtual matches are supported by physical matches with equivalent match kinds. In P4, tables define match kinds on a field by field basis. However, the speed at which a controller can update entries in the table is limited by the slowest match kind the table uses. Therefore, if a virtual table uses any **ternary** matches, then Shoehorn will allow other **exact** or **lpm** matches used by that table to be supported by physical matches with **ternary** match kinds. This is illustrated in Figure 8.1.

At this stage, Shoehorn accepts physical tables as supporting virtual tables even if the physical table requires match fields that are not matched by the virtual table. It is possible that Shoehorn will be able to fill that field with

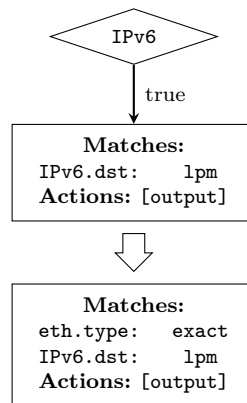


Figure 8.2: During stage 1 Shoehorn disregards excess matches in the physical table. By accepting this physical table as potential mapping for the virtual table, Shoehorn will be able to aggregate the virtual conditional and table together in stage 2.

a value from another component by aggregating components together. For instance, the physical table in Figure 8.2 requires an **exact** match on Ethernet type, but Shoehorn considers that this table is able to support the virtual table. This will allow Shoehorn to successfully map this table by aggregating the conditional with the virtual table in stage 2.

8.2 Stage 2: Finding Mappings

In stage 2, Shoehorn finds a combination of virtual to physical component mappings that ensures that the aggregation of virtual components is valid and that the mapped pipeline applies each packet to an equivalent set of components to the virtual pipeline.

8.2.1 Mapping Pipelines

Algorithm 1 shows `map_pipelines`, the method Shoehorn uses to map a virtual pipeline to a physical pipeline. `map_pipelines` iterates through the virtual

Algorithm 1 Stage 2 of the Shoehorn mapping algorithm.

```

1: function MAP_PIPELINES(vcb_list, pcb_list, config) ▷ §8.2.1.1
▷ vcb_list: virtual control blocks
▷ pcb_list: physical control blocks
▷ config: configuration data

2:   recirculations  $\leftarrow$  0;
3:   candidates_list  $\leftarrow$   $\emptyset$ ;
4:   unmapped_pcb_list  $\leftarrow$  copy(pcb_list);
5:   for each control block vcb in vcb_list do
6:     complete_list  $\leftarrow$   $\emptyset$ ;
7:     while recirculations  $\leq$  config.max_recirculations do
8:       while unmapped_pcb_list is not  $\emptyset$  and complete_list is  $\emptyset$  do
9:         pcb  $\leftarrow$  unmapped_pcb_list.pop();
10:        candidates_list = map_cb(candidates_list, vcb, pcb, config); ▷ §8.2.2
11:        complete_list  $\leftarrow$  get_complete(vcb, candidates_list) ▷ §8.2.1.2
12:        if complete_list is not  $\emptyset$  then
13:          break;
14:        else ▷ §8.2.1.3
15:          remove_not_updated(candidates_list);
16:          if candidates_list is  $\emptyset$  then
17:            break;
18:          unmapped_pcb_list  $\leftarrow$  copy(pcb_list);
19:          recirculations++;
20:          candidates_list  $\leftarrow$  complete_list; ▷ §8.2.1.4
21:          if candidates_list is  $\emptyset$  then break;
   return candidates_list;

```

control blocks, mapping components to each physical control block in turn. If `map_pipelines` reaches the end of the physical pipeline without mapping all components from a virtual control block, it recirculates and continues mapping from the first physical control block. This process continues until:

- all components in the virtual control block are mapped, in which case `map_pipelines` moves onto the next virtual control block;
- the number of recirculations exceeds the maximum, in which case the mapping fails; or
- a complete recirculation fails to map any components, in which case the mapping fails.

The steps of `map_pipelines` are described in detail below.

8.2.1.1 Initialisation and Main Loop (Algorithm 1. L1–9)

`map_pipelines` takes as arguments:

- `vcb_list`, a list of the control blocks used in the virtual pipeline;
- `pcb_list`, a list of the control blocks used in the physical pipeline; and
- `config`, a struct holding the configuration;

and initialises the following fields:

- `recirculations`, a count of the times the pipeline recirculates;
- `candidates_list`, an initially empty list of partially complete candidate mappings; and
- `unmapped_pcb_list`, a copy of `pcb_list`, to allow `map_pipelines` to track its progress through the physical pipeline by popping control blocks as they are mapped.

`map_pipelines` then iterates through the virtual control blocks. When it reaches a new virtual control block (`vcb`) it initialises a list, `complete_list`, that holds any candidates that map all components in `vcb`. When `complete_list` is not empty, `map_pipelines` stops mapping `vcb`, and continues with the next virtual control block.

To map `vcb` (provided the maximum number of recirculations has not been exceeded), `map_pipelines` pops the next physical control block (`pcb`) from `unmapped_pcb_list`, and attempts to map `vcb` to it. `map_cb` is detailed in subsection 8.2.2.

Each virtual action module applies all actions before any tables in subsequent control blocks can be applied, so each physical control block is only able to support components from one virtual control block each recirculation. However,

provided the actions are not contradictory, multiple physical control blocks can be used to support a single virtual control block. There is a potential scenario where a physical control block could support actions from multiple virtual control blocks in the same recirculation, described in Subsection 8.5.1. This remains future work, however, and is not supported by Shoehorn.

8.2.1.2 *get_complete* (Algorithm 1. L11–13)

`get_complete` returns all candidate mappings that have successfully mapped all components in the virtual control block, or an empty list if no complete candidates are found. Once it finds a complete candidate mapping, `map_pipelines` stops iterating through the physical control blocks, but does not reset the `unmapped_pcb_list`. This allows `map_pipelines` to begin mapping the next virtual control block from the next control block in the physical pipeline.

8.2.1.3 *Recirculating* (Algorithm 1. L14–19)

If `map_pipelines` reaches the end of the physical pipeline without having successfully mapped every component in a virtual control block, then it will recirculate. First, `map_pipelines` eliminates any candidate from `candidates_list` that has not been updated since the last recirculation. If there are no candidates remaining, then the mapping has failed. Otherwise, `unmapped_pcb_list` is reset, `recirculations` is incremented, and `map_pipelines` continues mapping the components of the virtual control block, starting from the first physical component in `unmapped_pcb_list`.

Algorithm 2 Map components from a virtual control block to a physical control block

```

1: function MAP_CB(candidates_list, vcb, pcb, config) ▷ §8.2.2.1
   ▷ candidates_list: partially complete candidate mappings
   ▷ vcb: virtual control block
   ▷ pcb: physical control block
   ▷ config: a configuration object
2:   for each component pc in pcb do ▷ §8.2.2.2
3:     for each component vc in pc.supports() do
4:       for each candidate in candidates_list do
5:         if vc.type is Table and is_mapped(candidate, vc) then
6:           continue;
7:         new ← candidate.clone();
8:         map_component(new, vc, pc); ▷ §8.2.2.3
9:         check_access_set(new) ▷ §8.2.2.4
10:        if new is valid then
11:          candidates_list.add(new);
12:        remove_unreachable_children(candidates_list, pc) ▷ §8.2.2.5
13:        remove_invalid_match_kinds(candidates_list, pc); ▷ §8.2.2.6
14:        prune_strictly_outclassed(candidates_list); ▷ §8.2.2.7
15:        remove_incomplete_trees(candidates_list); ▷ §8.2.2.8
16:        prune_partially_outclassed(candidates_list); ▷ §8.2.2.9
17:        prune(candidates_list, config.limit); ▷ §8.2.2.10
   return candidates_list;

```

8.2.1.4 Updating Candidates List and Returning (Algorithm 1. L20–21)

After `map_pipelines` finds a complete mapping for a virtual control block, it replaces `candidates_list` with `complete_list`, and repeats the process for the next virtual control block. `unmapped_pcb_list` is not updated at this point, so the mapping will continue with the next physical control block.

Once `map_pipelines` finds a successful mapping for all virtual control blocks, it returns the successful candidates. If `map_pipelines` is unable to find a successful mapping for a virtual control block, it returns an empty list.

8.2.2 Mapping Control Blocks

Algorithm 2 shows `map_cb`, the function used by Shoehorn to map the components of a virtual control block to a physical control block.

The algorithm iterates through the components in the physical control block, and for each physical component, finds all combinations of virtual components that can be mapped to the physical component. For each new combination that it finds, and each pre-existing candidate mapping, it generates a new candidate mapping by combining the two. The algorithm validates the new candidates and then uses heuristics to discard all but the most promising candidates. The steps of the algorithm are described in detail below.

8.2.2.1 Method Call (Algorithm 2, L1)

`map_cb` takes as arguments:

- `candidates_list`, a list of partially complete candidate mappings;
- `vcb`, an object representing the virtual control block;
- `pcb`, an object representing the physical control block; and
- `config`, a configuration object.

The control block objects contain a list of the component trees in that control block, and the configuration object contains the maximum size of the list of candidates.

8.2.2.2 Loop (Algorithm 2, L2–7)

`map_cb` iterates through each component in the physical control block (`pc`), and finds every virtual component, `vc`, that `pc` can support. For each pair of components, it iterates through every candidate mapping (`candidate`) and clones it, to create a new candidate, `new`, to attempt to map the two components.

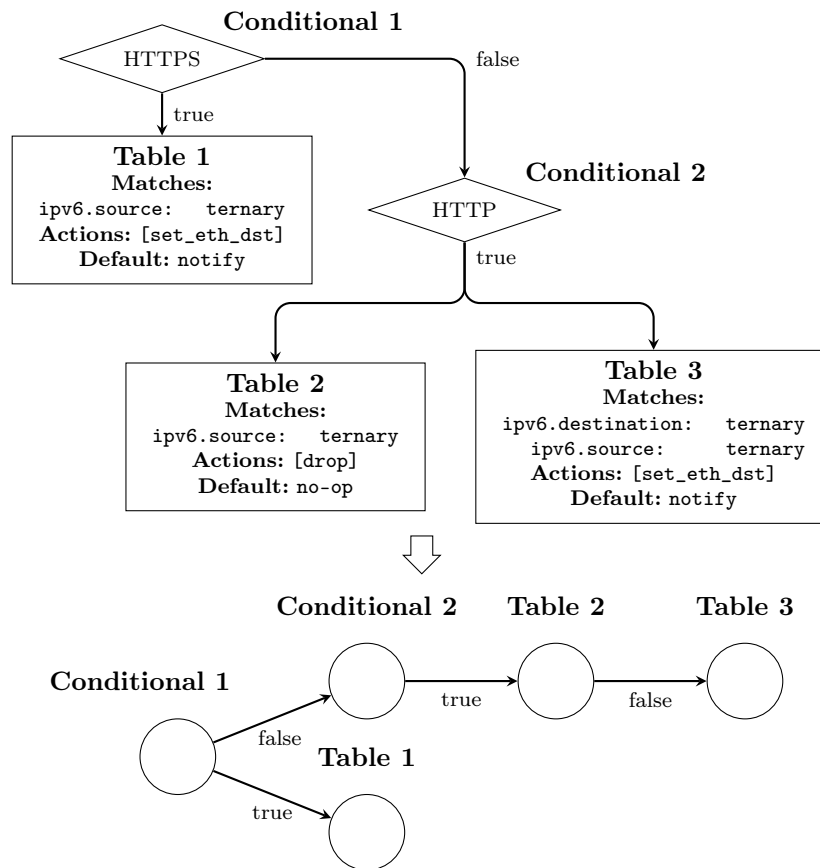


Figure 8.3: A section of a virtual pipeline that is aggregated into a single physical table, and the corresponding entry tree. Nodes in the entry tree can only have one child on a true (or false) evaluation. However, as Table 2 drops all packets it matches, it can still be merged with Table 3. Table 2 becomes the immediate child of Conditional 2, and Table 3 becomes the child of Table 2.

8.2.2.3 *map_component* (Algorithm 2, L8)

To ensure mappings aggregate tables in a manner that does not negatively affect the rate entries can be updated, mappings add the virtual components mapped to each physical component in a tree, referred to as an *entry tree*. Entry trees have rules that ensure that the number of entries in the physical table is no greater than the sum of entries in the aggregated virtual tables, unless explicitly allowed. Each node in a valid entry tree represents a virtual component and can have up to two child nodes, representing the virtual components applied

following a true or false evaluation of the parent (for tables the evaluation represents the `hit` field of the apply result).

Virtual pipelines can have multiple components accessed following a true or false evaluation of a single component, but aggregating them together results in a physical table with a Cartesian product of the tables entries. While the `flexible_mapping` annotation can indicate that Shoehorn can aggregate such tables, how best to support that remains future work. Shoehorn only allows aggregating these tables when one of the tables always drops packets on a match, and the other table does not clone or count packets, which does not increase the number of table entries in the physical pipeline. In such cases, `map_component` inserts the table that drops packets as the immediate child of the parent, and the other table is added as its child on a false evaluation.

If a node representing a table has a child for a true evaluation, then the physical table will contain a Cartesian product of the entries of the two tables. In such cases, the entry tree is only valid if both tables have the `flexible_mapping` annotation.

Figure 8.3 shows a section of a virtual pipeline, and the entry tree stored in the candidate mapping when that section of pipeline is mapped to a single physical component.

`map_component` adds `vc` to the entry tree associated with `pc` in `new`. If the entry tree associated with `pc` is invalid, then `map_component` sets `new` as invalid.

8.2.2.4 `check_access_set` (Algorithm 2, L9–12)

Once `map_modules` finds a potential mapping for a table, it verifies the mapping applies all the virtual components to the correct set of packets, with the function `check_access_set`.

The set of packets to which a component in the SVA is controlled in two ways:

1. by conditionals comparing a masked header value with a specified constant value, or
2. by the `hit` value of a table's apply result (§2.3.4).

Therefore, the set of packets to which components apply is defined by a set of masked field values the packet must match, a set of masked field values the packet must not match, a set of tables the packet must match, and a set of tables the packet must not match. Collectively, these sets make up the *access set* for a given component. Access sets are only relevant for actions, and therefore conditionals do not require correct access sets, except when that affects the access set of a table.

If the packets to which `pc` is applied is determined by `goto` metadata, then `check_access_set` only needs to verify that `new` has successfully mapped all components in the access set set for `vc`. As all hardware that uses `goto` metadata uses `goto` metadata for every table, and the components in the physical pipeline are mapped in order, then the components in the access set can always be connected with `goto` action primitives.

If the hardware has a fixed pipeline, then `check_access_set` finds the physical components in the access set of `pc`. Then `check_access_set` iterates through the entry trees mapped to those physical components, and finds all virtual components that must be matched, and not matched, to reach `pc`. If that set of components is equivalent to the access set of `vc`, then the access set is correct.

If the entry tree is valid and the access set is correct, `map_cb` adds `new` to `candidates_list`, and continues with the next virtual component.

8.2.2.5 *remove_unreachable_children* (Algorithm 2, L13)

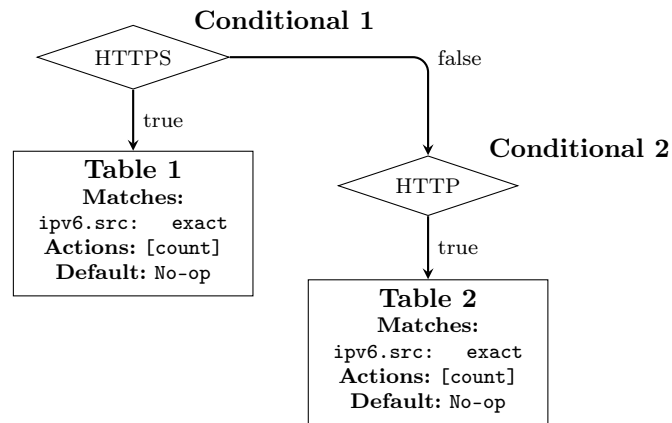
Once `map_cb` has found all candidate mappings for `pc`, it calls the function `remove_unreachable_children`, to eliminate mappings with unmapped virtual tables that cannot be mapped with a correct access set.

If `pc` does not support `goto` actions, then any physical tables reached by the result of a look-up in `pc` will be accessed depending on whether any virtual component mapped to `pc` is hit. Consequently, any unmapped child of a virtual component mapped to `pc` will have an incorrect access set if another virtual component mapped to `pc` can be hit by packets that do not also hit the first virtual component.

If `pc` does not support `goto` actions, `remove_unreachable_children` iterates through each candidate in `candidates_list`, checking whether any of the virtual components mapped to `pc` control access to an unmapped table. If so, then the candidate is invalid if any ancestor of the virtual component in the entry tree has a child on a false evaluation.

`remove_unreachable_children` will attempt to correct this by removing nodes representing virtual conditionals from `pc`'s entry tree, but any node representing a virtual table cannot be removed. `map_cb` waits until after all components have been mapped to the physical table to perform this check, as an invalid aggregation may be resolved by aggregating the child table into `pc`.

If `pc` supports `goto` action primitives, `remove_unreachable_children` returns `candidates_list` unmodified.



Partially Aggregated Table

IPv6 Source:	TCP Destination	Actions:
2001:db8::1	443	count
2001:db8::2	443	count
::/0	80	goto Table 2

Fully Aggregated Table

IPv6 Source:	TCP Destination	Actions:
2001:db8::1	443	count
2001:db8::2	443	count
2001:db8::1	80	count
2001:db8::2	80	count

Figure 8.4: A section of pipeline where the components can be aggregated into a single table, but adding components incrementally results in invalid match kinds until the final component is added. The partially aggregated table has to use an `all_or_exact` match kind on IPv6 Source, whereas the fully aggregated table can use `exact`.

8.2.2.6 `remove_invalid_match_kinds` (Algorithm 2, L10)

`map_cb` then checks that the match kinds for each virtual component that is aggregated into `pc` do not conflict with one another, using the function `remove_invalid_match_kinds`. For instance, if a table with `ternary` match kinds is aggregated with a table with only `exact` match kinds, then the physical table will need to use `ternary` match kinds, reducing the update rate of the second table. If the second table does not have the `flexible_match_kinds` annotation, then this would be an invalid mapping.

`map_cb` only does this after all virtual components have been mapped to the

physical table, as it is possible that a conflicting match kind can be resolved by aggregating a new component. For example, the pipeline segment shown in Figure 8.4 monitors the HTTP and HTTPS traffic from target hosts. Because the tables require `exact` match kinds, they cannot be merged with both conditionals without the presence of the other table. For instance, aggregating Table 1 with the two conditionals, but not Table 2, requires an entry for a true evaluation of Conditional 2, matching Ethernet Type and masking IPv6 source, that uses a `goto` action to direct packets to Table 2. Because this entry must match all IPv6 source addresses, the table must use an `all_or_exact` match for IPv6 source, conflicting with the `exact` match on IPv6 source used by Table 1. Once Table 2 is aggregated with the other components then all entries match Ethernet Type and IPv6 source and no masking is required in the physical table.

`remove_invalid_match_kinds` iterates through `candidates_list` and removes any candidates where multiple virtual components with conflicting match kinds are mapped to `pc`.

8.2.2.7 `prune_strictly_outclassed_candidates` (Algorithm 2, L14)

Before moving on to the next component in the physical control block, `map_cb` does a first pass of pruning the number of candidate mappings. `map_cb` finds every valid combination of mappings for each physical table; pruning candidates prevents the scale from becoming unmanageable.

`prune_strictly_outclassed_candidates` removes from the candidates list all candidates that:

1. map a subset of the virtual components of another candidate, and
2. map every virtual component that is used to control access to an un-

mapped virtual table to the same physical components as that other candidate.

As `pc` is fully mapped at this point, any virtual table mapped to it will not affect how any non-descendant virtual table is mapped. So if a candidate maps a subset of the virtual components of a second candidate, and the components mapped in the second candidate do not control access to unmapped tables, then any complete mapping found from the first candidate will map all remaining tables in a manner that is also compatible with the second candidate. Therefore, the first candidate mapping is redundant, and can be removed.

8.2.2.8 *remove_incomplete_trees* (Algorithm 2, L15)

`map_cb` continues mapping components in this manner, until it has mapped the last component in the physical control block. At that point, any candidate that maps a virtual table with unmapped children is invalid, unless the virtual table uses `goto` actions or has the `flexible_mapping` annotation. This is because any unmapped components will be mapped to subsequent physical control blocks. As access sets cannot include components from multiple control blocks (except with `goto` actions), the children will be unreachable. `remove_incomplete_trees` removes these candidates from `candidates_list`. Tables with the `flexible_mapping` annotation can be mapped twice, so the fact that they have been mapped does not prevent them from being mapped a second time, to provide access to their children.

`remove_incomplete_trees` also checks that the actions applied by the mapped virtual tables will not modify fields matched in unmapped virtual components. If the mapped virtual tables apply such actions, then the candidate is invalid, and `remove_incomplete_trees` removes it from `candidates_list`.

8.2.2.9 *prune_partially_outclassed_candidates* (Algorithm 2, L16)

Valid mappings of a virtual component to a component in one physical control block cannot prevent virtual components from being mapped to subsequent control blocks. Therefore, once the physical control block is completely mapped, any remaining candidate that maps a subset of the virtual tables of another candidate is redundant.

`prune_partially_outclassed_candidates` removes any such candidate from `candidates_list`.

8.2.2.10 *Final Pruning* (Algorithm 2, L17)

At this point, fixed-function pipelines typically have very few remaining candidate mappings. However, configurable pipelines provide much more flexibility in how components are mapped, and may still have a large number of mappings. In order to keep the scale of the workload to a manageable level, `prune` reduces the candidates list to a set of configurable size, by selecting randomly from the candidates that map the largest number of components. Because the configurable pipelines consist of identical tables, this is very unlikely to impact the likelihood of finding a correct result, unless the limit is small enough to discard candidates from fixed-function pipelines.

Finally, `map_cb` returns the remaining candidates in `candidates_list`.

8.3 Stage 3: Resolving Goto Actions

Once Shoehorn has found a valid mapping for every virtual component, it removes any redundant conditionals. Because virtual conditionals can be mapped

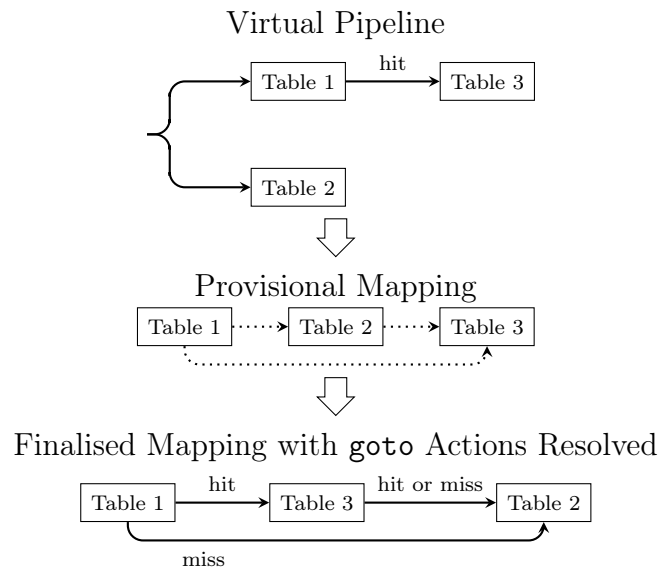


Figure 8.5: Resolving goto actions when multiple components have equivalent access sets

multiple times, and Shoehorn uses a greedy approach to mapping components, Shoehorn will often map conditionals to physical components where they are unneeded to reach any table. Shoehorn removes any virtual conditional mapping that is not part of the path to a table.


Then Shoehorn rearranges tables accessed by goto actions. When components have equivalent access sets, the provisional mapping will arrange them in an arbitrary order. This can result in impossible mappings when those tables are used to control access to child tables. Shoehorn rearranges tables so that when two tables have equivalent access sets, all descendants of the first table are moved before the second table. The tables can always be rearranged because all tables that use goto actions are identical. This process is shown in Figure 8.5.

Finally, Shoehorn resolves the goto action primitives to target the correct components. When Shoehorn maps two virtual tables that have equivalent access sets to different physical components, it must ensure that those physical

Table 1				
IP Source:		Actions:		Priority:
2001:db8:1::/48		set_eth_dst		100
2001:db8:2::/48		set_eth_dst		100
::/0		notify		0

Table 2				
IP Source:		Actions:		Priority:
2001:db8:1::2		drop		100
::/0		no-op		0

Table 3				
IP Source:	IP Destination:		Actions:	Priority:
2001:db8:1::/48	2001:db8:3::1		set_eth_dst	100
::/0	::/0		notify	0



Mapped Table				
IP Source:	IP Destination:	Port:	Actions:	Priority:
2001:db8:1::/48	::/0	443	set_eth_dst	500
2001:db8:2::/48	::/0	443	set_eth_dst	500
::/0	::/0	443	notify	400
2001:db8:1::2	::/0	80	drop	300
2001:db8:1::/48	2001:db8:3::1	80	set_eth_dst	200
::/0	::/0	80	notify	100
::/0	::/0	ANY	no-op	0

Figure 8.6: A demonstration of translating table entries. This shows hypothetical table entries for the section of virtual pipeline shown in Figure 8.3 and the corresponding entries for the mapped table. Entry priority in the mapped table is determined by the path through the entry tree. The highest priority entries belong to Table 1, followed by the Table 1 default entry. The next highest priority entries are from Table 2, followed by its default (in this case a no-op) combined with the entries from Table 3, and finally a no-op entry for packets that return False when applied to Conditional 2.

components are both applied to the same set of packets. For all descendants of the first physical table, Shoehorn adds a `goto` action primitive (directing packets to the second table) to all actions that do not already apply a `goto` action primitive.

8.4 Populating Tables

Shoehorn finds mappings—it is not responsible for the translation of entries. However, the translation software must ensure the entries in aggregated tables

are translated correctly.

Every virtual table in an entry tree without two children corresponds to an entry type in the aggregated table. Entries match the combined fields of all virtual components that evaluate true in the path to the node from the root of the tree. When a table with the `flexible_mapping` annotation has a child on a true evaluation, this creates a Cartesian product of its entries with its child.

Entries must also apply all the actions of the virtual tables from the path from the root, for virtual tables without the `flexible_mapping` annotation this is the default actions for each virtual table in the path.

The priority offset for entries is determined by the path through the tree: all entries from nodes descending from a true evaluation of given node have higher priorities than all entries from nodes descending from a false evaluation.

An example of translating table entries is shown in Figure 8.6.

8.5 Discussion

8.5.1 Splitting Physical Control Blocks

Shoehorn does not allow two virtual control blocks to be combined with a single physical control block. The rationale for this is that every table (A) in the second control block, must come after the actions of some table in the previous control block (B) are applied. However, if the first control block is mapped to two physical control blocks, then it is possible that table B could have been mapped to the first physical control block, meaning table A could be mapped to the same physical control block as the remaining tables in the first virtual control block. Shoehorn does not support this currently, but, to

do so it would need to be able to identify the tables in different control blocks that could, or could not, be mapped to the same physical control block.

8.5.2 Wide Mappings

Shoehorn maps tables greedily, and prunes candidate mappings according to how many tables are mapped, meaning that physical tables tend to support as many virtual tables as possible. This can result in physical tables having to map a large number of fields. In practice, it is unlikely that the hardware can support so many fields in a single table. Shoehorn could include a `maximum_width` annotation, to indicate how many fields each table can match, but this remains future work at this stage.

Chapter 9

Evaluation

This chapter demonstrates the practicality of supporting SDN controllers with diverse, low-cost hardware, by using Shoehorn to find mappings from virtual pipelines to physical pipelines. The evaluation maps a variety of virtual pipelines, based on real world SDN controllers, to diverse physical pipelines, based on the target hardware chosen in chapter 4. This chapter describes the implementation of the mapping algorithm, and gives detailed descriptions of the physical and virtual pipelines, before discussing the results.

9.1 Mapping Application

The evaluation uses an implementation of the Shoehorn mapping algorithm, created for this evaluation. The mapping application takes a YAML representation of the virtual and physical pipelines and finds a mapping between them.

The mapping application has a few minor limitations that should be noted. The implementation of annotations is faulty, and would not change how tables were

mapped. Instead, when a virtual table had the `flexible_match_kinds` annotation, the evaluation ran the mapping application multiple times, with `exact` and `all_or_exact` match kinds for each match in the table. The evaluation supported the `flexible_mapping` annotation, when necessary, by aggregating tables by hand.

The mapping application does not determine whether conditionals are equivalent, instead it requires that the conditionals in the access set for a virtual table are all mapped to components controlling access to the physical table. This is most relevant with tables that apply to specific protocols. For instance, if a virtual table has an access set with conditionals that first check whether a packet is an IPv4 packet, and if not, check whether the packet is an IPv6 packet, then the physical pipeline must also verify that the packet is not an IPv4 packet, even if the pipeline has already established that the packet is IPv6, otherwise the mapping application considers this an incorrect access set. The evaluation resolved this by always defining pipelines with conditionals for protocols applied in the same order.

The mapping application also does not perform Stage 3 of the mapping algorithm. Stage 3 does not affect the success or failure of Shoehorn, so was not necessary for the evaluation.

The mapping application is available under a BSD license [24].

9.2 Physical Pipelines

The evaluation uses physical pipelines based on the target hardware identified in chapter 4. The pipelines recreate the functionality of the hardware's support for SDN standards in the SPA as faithfully as possible. However, without ac-

cess to the full specification of each vendor's ASIC, the evaluation must make assumptions about how best to define the SPA pipeline. While these assumptions mean that this is not a perfect demonstration of Shoehorn's suitability for mapping to real world hardware, the physical pipelines do represent a diverse range of configurable and fixed-function pipelines.

Most significantly, this evaluation assumes that hardware is capable of recirculating packets, while retaining the ingress port metadata, and keeping track of the number of times the packet has been recirculated. Recirculation can always be achieved through the use of loop-back transceivers or similar methods, but retaining the metadata may be more challenging. This evaluation assumes any table that matches ingress port or tunnel ID is capable of inferring the recirculation metadata, and that if such a table can set a metadata field (for instance the VRF field in the OF-DPA), then any subsequent table that matches that metadata field can also infer the recirculated metadata.

This section describes each physical implementation, and notes any assumptions that the evaluation made.

9.2.1 Aruba

The Aruba uses a configurable pipeline, and consequently is very easy to define in the SPA. There are two major assumptions that this evaluation has made about the Aruba pipelines.

The first is that the pipeline is capable of performing `lpm` matches on IP source and destination fields. Both the Cisco and the Aruba use OpenFlow `Table-Features` messages to configure their pipelines, but `Table-Features` messages cannot indicate a `lpm` match. This evaluation assumes that this hardware can support these matches as `lpm` matches are a fundamental feature of

Layer 3 switches. This assumption had little impact on the outcome of the evaluation for the Aruba pipeline, however, as it does not support decrementing TTL fields, which every virtual pipeline that used `lpm` matches also required.

The second assumption, is that actions can be deferred arbitrarily throughout the pipeline. The Aruba OpenFlow implementation supports both `Apply-Actions` and `Write-Actions` instructions, but in the Shoehorn architecture actions are applied by Action Modules at arbitrary points throughout the pipeline. An OpenFlow implementation could support this by writing to metadata and then, in a subsequent table, match the metadata, and apply the appropriate actions, but the Aruba OpenFlow implementation does not support metadata. This evaluation assumes that the Aruba pipeline can achieve this result using a similar method.

The Aruba OpenFlow implementation is unable to recirculate packets. It has a flexible pipeline, however, so it did not require recirculation to map any of the virtual pipelines in this evaluation.

9.2.2 Cisco

The Cisco OpenFlow implementation uses `Table-Features` messages to configure its pipeline, similar to the Aruba implementation. Consequently this evaluation makes the same assumptions as with the Aruba pipeline. Unlike the Aruba pipeline, the Cisco pipeline can decrement TTL fields, so assuming that it can perform `lpm` matches is more impactful.

Discussion with engineers at Cisco has indicated that the Cisco ASIC is capable of supporting metadata in the OpenFlow pipeline, although the OpenFlow implementation does not currently support this [44]. Therefore, the method proposed above to support Action Modules is possible with the Cisco ASIC.

Furthermore, the Cisco ASIC can recirculate packets natively, and therefore could achieve the same result using packet recirculation.

9.2.3 OF-DPA

The OF-DPA pipeline is well specified, and requires few modifications to be expressed in the SPA. The OF-DPA has a Policy ACL table, which performs ternary matches on a variety of fields, and can drop packets or send them to group tables. The OF-DPA applies the Policy ACL table near the end of the pipeline, to allow its actions to overwrite the actions of previous tables, such as the Routing table. In the Shoehorn architectures, however, overwriting actions is not possible. Instead, our evaluation places the Policy ACL table in a control block before the Routing table, so that if the Policy ACL table applies actions to a packet, it will not reach the subsequent tables. This ensures the packet handling is correct, but may cause inaccurate counters in some tables.

9.2.4 SAI

The SAI has an existing P4 definition for the behavioural model architecture, making translation into Shoehorn simple. The P4 definition does not support ACL tables, however. This evaluation assumes that the ACL tables can support any entry supportable by both the Nvidia and OF-DPA pipelines.

The SAI pipeline makes extensive use of metadata, which is largely unutilised by Shoehorn. Consequently, the evaluation implementation does not expose the full functionality of the SAI pipeline, such as STP tables.

9.2.5 Nvidia

The Nvidia OpenFlow pipeline frequently redirects traffic to the legacy pipeline for additional processing. As the legacy pipeline is not clearly defined, this evaluation omits that functionality. The only other assumption the evaluation implementation required was that the pipeline is capable of applying actions throughout the pipeline, in a similar manner to the Cisco and Aruba. The Nvidia OpenFlow implementation does include metadata, so the only assumption is that the metadata can be written to combine actions accurately.

9.3 Virtual Pipelines

This evaluation used virtual pipelines based on 25 controllers chosen from those described in chapter 5. The virtual pipelines were defined in the SVA, and were a best-effort interpretation based on the published details.

This evaluation selected controllers representing a variety of applications and network types, favouring those that have been used in production deployments, and those with multi-table pipelines. The evaluation eliminated controllers that use features not supported in OpenFlow version 1.3.

When controllers use the `NORMAL` port to redirect packets to the legacy pipeline of the device, this evaluation replaced the legacy pipeline with the pipeline used by Faucet [8], as Faucet is an OpenFlow implementation of legacy switching and routing.

9.3.1 Implementation Details

The controllers used in this evaluation, and the details of their implementations are described below.

9.3.1.1 *AuthFlow*

AuthFlow is a host authentication mechanism for enterprise networks [59]. The evaluation implementation uses three tables and two control blocks. The first control block has two tables, sending Link Layer Discovery Protocol (LLDP) and Extensible Authentication Protocol (EAP) packets to the controller. The second control block has an authorisation table, matching ingress port, Ethernet source and Ethernet type, allowing authorised traffic to pass through the pipeline, and dropping all other traffic. *AuthFlow* does not define a mechanism for how the pipeline should handle packets once they are authorised, so our evaluation implementation does not include this functionality.

This design highlights a weakness of the Shoehorn mapping algorithm. All the tables could trivially be merged into a single table, but, because Shoehorn does not merge tables in different control blocks this is impossible. Defining the tables in a single control block, however, could result in the authorisation table dropping protocol traffic that needs to be sent to the controller. Defining the first three virtual tables as a single table, on the other hand, means that physical pipelines are unable to use different tables to support each virtual table.

9.3.1.2 Castor

Castor is a SDN Internet exchange interconnect controller that provides telemetry and ARP hygiene [51]. The implementation for this evaluation uses two control blocks, the first has three tables, two tables for defining intents (one for IPv6 and one for IPv4), that let IX members control how packets are forwarded to them, and one table to unicast ARP packets to the correct IX member. The second control block has a switching table that forwards general traffic. As the MAC addresses on the IX are static, Castor does not require a table for MAC learning.

9.3.1.3 Faucet

Faucet is a widely deployed production enterprise controller for basic switching and routing [8]. This evaluation uses an implementation with 4 control blocks and 20 tables. This is more tables than are used in the OpenFlow implementation of Faucet, but this is the result of the OpenFlow implementation being forced to combine multiple functions into some tables. Shoehorn eliminates this need, as it can aggregate tables as needed.

The first control block applies the following tables:

- tables filtering unwanted packets such as unwanted protocols, packets with illegal headers (eg. a broadcast Ethernet Source address), packets with incorrect VLAN tags, and packets arriving on disabled ports;
- tables adding VLAN tags to packets arriving on access ports;
- tables that send Link Aggregation Control Protocol (LACP) packets to the controller; and
- Port ACL tables.

The second control block applies VLAN ACL tables.

The third control block has the following tables:

- an `EthernetLearning` extern;
- routing tables, accessed by packets that match a termination MAC table;
and
- tables filtering traffic directed to the control plane.

The control-plane filtering tables differ from those used by Faucet, as the `flexible_mapping` extern was unusable in this test. The control-plane filtering tables were aggregated by hand with the termination MAC table, as this is supportable by more of the target hardware.

Finally, the fourth control block has an `EthernetSwitching` extern.

9.3.1.4 Hierarchical SDN

Hierarchical SDN (HSDN) [31] is an architecture for data centre networks for scalable TE. The implementation used in this evaluation has two control blocks and two tables. The table in the first control block matches ingress port and MPLS label, and on a miss, forwards packets to a switch above the current switch in the network hierarchy (including rewriting Ethernet headers and decrementing TTL). On a hit, the table pops the MPLS label, and allows it to pass to the next control block.

The second control block contains one table, matching MPLS label, and forwards the packet to a device lower in the network hierarchy than the current switch.

9.3.1.5 In-Packet Bloom Filters

In-Packet Bloom Filters [57] is an architecture for data centres using bloom filters encoded in Ethernet addresses to load-balance traffic. The evaluation implementation uses three tables in one control block. Two tables encode bloom filters for IPv4 and IPv6 packets, and forward the packet, the other table is for packets that already have filters encoded, and forwards according to the Ethernet Source and Destination.

9.3.1.6 iTelescope

iTelescope [38] is a bump-in-the-wire system for identifying video traffic and providing telemetry. The evaluation implementation has one control block with three tables. There are two levels of cache, the first matches 5-tuples of elephant flows, the second clones all TCP and UDP traffic to an inspection device, and the final table forwards all remaining traffic to the correct egress port.

9.3.1.7 Magneto

Magneto [48] is a system for providing fine-grained path control in a hybrid legacy–OpenFlow enterprise network, by using `Packet-Out` messages to manipulate MAC learning on the legacy devices. The evaluation implementation has one control block with three tables. Two tables have `1pm` matches on IPv4 and IPv6 destination, and rewrite Ethernet Destination addresses to direct packets through the predefined paths. The final table is used to handle STP BPDUs.

9.3.1.8 NetPaxos

NetPaxos [27] is a system for accelerating consensus Paxos protocols in data centres by creating a canonical ordering of messages within the network. The evaluation implementation uses a single table, matching ingress port and IPv6 destination, and multicasts packets (now in a canonical ordering) to the other switches in the network.

9.3.1.9 NFShunt

NFShunt [65] is a system for accelerating Netfilter with network hardware for use in a Science DMZ. The evaluation implementation has one control block with two tables, with 5-tuple matches for flows that have been accepted by Netfilter, and a default rule forwarding all other traffic to the Netfilter host.

9.3.1.10 OF-Like PBR

OF-Like PBR [64] is a system for applying policy in a hybrid legacy–OpenFlow network. It uses OpenFlow switches to rewrite IP addresses to control how the legacy devices forward traffic. The evaluation implementation uses a single table, that matches ingress port and 5-tuple, and rewrites IPv4 source and destination before forwarding the packet.

9.3.1.11 OFLoad

OFLoad [107] is a load balancing system for data centres that generates tunnels for elephant flows. The evaluation implementation uses two tables in two control blocks. The first has 5-tuple matches for elephant flows, and the second matches IPv6 destination for mice flows.

9.3.1.12 OFTDP

OFTDP [7] is a topology discovery technique for SDN. OFTDP merely learns the topology of a network, so would need to be used with a system for forwarding host traffic, but the evaluation implementation just implements the topology discovery. It consists of one table, matching LLDP packets.

9.3.1.13 OLIMPS

OLiMPS [71] is a system for load balancing across point-to-point VLANs in research networks. The evaluation implementation consists of one table, forwarding packets according to their ingress port and VLAN VID.

9.3.1.14 OpenNetMon

OpenNetMon [108] is loss monitoring system in reactive OpenFlow networks. The evaluation implementation consists of a single table, matching flows with 10 fields.

9.3.1.15 Precision Medicine

Precision Medicine [92] is a Science DMZ for campus networks for medical applications. The evaluation implementation uses two tables in one control block: one handling ARP, and another doing 5-tuple based forwarding.

9.3.1.16 Random Host Mutation

Random Host Mutation [46] is a security system for enterprise networks that randomly rewrites IP addresses. The evaluation implementation has one table,

matching 5-tuple, that sets IPv4 source and destination, and outputs packets.

9.3.1.17 RouteFlow

RouteFlow [69] is a SDN router for OpenFlow v1.0, that has been deployed in production [103]. The evaluation implementation uses `lpm` match kinds to improve scalability, rather than the original single table design. It consists of two control blocks, the first has a table filtering control plane traffic, the second has a termination MAC table and IPv4 and IPv6 routing tables.

9.3.1.18 SciPass

SciPass [9] is a Science DMZ for campus networks. The evaluation implementation uses one control block with two tables. The first table directly outputs pre-configured flows based on 5-tuple matching, and the remaining traffic is forwarded to a second table which matches ingress port and clones packets to an IDS as well as forwarding them.

9.3.1.19 SDProber

SDProber [96] is a system for monitoring link latency in SDNs. The evaluation implementation uses a single table matching Ethernet Destination, either forwarding probes back to the source, or cloning the packet back to its source, as well as forwarding it on through the network.

9.3.1.20 SIMPLE

SIMPLE [95] is a system for enforcing middle box policies in enterprise networks by tunnelling packets through the network using their Ethernet ad-

addresses and VLAN VIDs. The evaluation implementation consists of two tables, in two control blocks. The table in the first control block matches tunnelled packets, forwarding them through the network. The table in the second control block matches 5-tuples, and sets the Ethernet Destination and VLAN VID to add packets to tunnels.

9.3.1.21 *Tennison*

Tennison [32] is a SDN framework for security monitoring and attack mitigation. Tennison monitors traffic, either in the data plane or by mirroring traffic to DPI servers, and allows applying mitigations to malicious flows once identified. The evaluation implementation consists of three control blocks. The first two control blocks contain one table each, for forwarding tunnelled traffic to DPI servers, and for decapsulating tunnelled traffic, respectively. The third control block performs attack mitigation and IPFIX monitoring, using two sets of two tables, for IPv4 traffic and IPv6 traffic. Tennison can mitigate attacks by dropping flows, or by rate-limiting flows. How to support rate-limiting with Shoehorn remains future work at this point, so the evaluation implementation of Tennison only drops packets.

9.3.1.22 *TouSIX*

TouSIX [52] is a production deployment at an Internet exchange in Toulouse. The TouSIX deployment provides layer 2 security and fine-grained monitoring. The evaluation implementation has one control block, with two tables, one handling ARP traffic, and one handling switching. Like Castor, TouSIX preconfigures the paths to all devices on the network, so does not need to learn MAC addresses.

9.3.1.23 VIP Lanes

VIP Lanes [39] is an architecture for campus networks that allows creation of on-demand Science DMZs. VIP Lanes uses the `NORMAL` port, so the implementation replaces the Faucet VLAN ACL table, with a table performing 5-tuple matching with the `flexible_match_kinds` annotation for DMZ traffic.

9.3.1.24 VPNs

VPNs [55] is a system for simplifying the configuration of VPNs in SDN data centres. The paper describes two different types of devices, a P node and a PE node.

The implementation of the P node has two tables in one control block, a termination MAC table and a table matching MPLS label, that forwards packets, and pops MPLS label when the switch is the penultimate hop.

The PE node implementation has 3 tables in two control blocks. The first control block handles MPLS packets egressing the network, and has a termination MAC table, and a table matching MPLS labels. The MPLS table pops MPLS labels, and outputs the packets to the customer network. The second control block handles traffic arriving from the customer network, matching ingress port, and source and destination IP subnets. Entries either output the packet to another customer site, or encapsulates the packet in MPLS and forwards it through the network. The PE node pipeline simulates `flexible_mapping` by combining a termination MAC table with the routing table in the second control block. The routing table may have a large number of entries, but as the routing table already matches the IP source subnet, and termination MAC addresses have a one to one mapping with subnets, this would not result in an increase in the overall number of entries.

Table 9.1: Overview of the target controllers.

Controller	Network Type	Control Blocks	Tables
AuthFlow	Enterprise	2	3
Castor	Internet Exchange	2	4
Faucet	Enterprise	4	20
HSDN	Data Centre	2	2
In-Packet Bloom Filters	Data Centre	1	3
iTelescope	Campus	1	3
Magneto	Hybrid	1	3
NetPaxos	Data Centre	1	1
NFShunt	Science DMZ	1	2
OF-Like PBR	Hybrid	1	1
OFLoad	Data Centre	1	2
OFTDP	Any	1	1
OLiMPS	Campus	1	1
OpenNetMon	Any	1	1
Precision Medicine	Science DMZ	1	2
Random Host Mutation	Hybrid	1	1
RouteFlow	Any	2	4
SciPass	Science DMZ	1	2
SDProber	Any	1	1
SIMPLE	Enterprise	2	2
Tennison	Enterprise	3	6
TouSIX	Internet Exchange	1	2
VIP Lanes	Campus	4	20
VPNs-P	ISP	1	2
VPNs-PE	ISP	2	3

9.3.2 Summary of Controllers

The evaluation uses a diverse set of controllers, for a variety of applications and network types. Table 9.1 summarises the controllers used in this evaluation, showing the network type, and how many control blocks and tables each uses.

9.4 Results

Table 9.2 shows the results of mapping each virtual pipeline to each physical pipeline. All but 6 virtual pipelines were supported by hardware from multiple

Table 9.2: The number of recirculations required for each physical pipeline to support each of the 25 controllers’ virtual pipelines. A dash indicates Shoehorn was unable to find a mapping.

Controller	Aruba	Cisco	Nvidia	OF-DPA	SAI
AuthFlow	0	0	-	2	2
Castor	0	0	-	-	-
Faucet	-	0	-	3	3
HSDN	-	-	-	1	-
In-Packet Bloom Filters	0	0	-	1	1
iTelescope	0	0	-	0	0
Magneto	-	0	-	0	0
NetPaxos	0	0	0	0	0
NFShunt	0	0	-	-	-
OF-Like PBR	0	-	-	-	-
OFLoad	0	-	-	-	-
OFTDP	0	0	0	0	0
OLIMPS	0	0	0	0	0
OpenNetMon	-	-	0	0	0
Precision Medicine	0	0	0	1	1
Random Host Mutation	0	-	-	-	-
RouteFlow	-	0	-	0	0
SciPass	0	0	0	0	0
SDProber	-	0	-	0	0
SIMPLE	0	0	-	1	1
Tennison	0	0	-	2	2
TouSIX	0	0	-	-	-
VIP Lanes	-	0	-	3	3
VPNs-P	-	-	-	0	-
VPNs-PE	-	-	-	0	-
Total Controllers Supported	16	18	6	19	16

vendors. The layout of the pipelines was never a cause for failure, in every case where there was a failure, it was due to a table in the virtual pipeline that could not be supported by any table in the physical pipeline. While, in some cases, it might be possible for a mapping algorithm to find mappings where virtual tables are supported by a combination of tables in the physical pipeline, doing so negatively impacts the update rate.

9.4.1 Physical Pipelines

This section provides further details on the results for each hardware pipeline.

9.4.1.1 Aruba

The physical pipeline based on the Aruba supported 64% of the virtual pipelines overall. As the Aruba has a configurable pipeline, it did not require recirculation for any controller.

The causes for failure for controllers when mapping to the Aruba pipeline were:

- the Aruba pipeline cannot support decrementing TTL fields, this prevented it from supporting controllers that perform layer 3 routing—Faucet, Magneto, RouteFlow, and SDProber;
- the Aruba pipeline cannot perform MPLS matches and actions, preventing it from supporting either VPNs controller or HSDN; and
- the Aruba pipeline cannot match IP ECN bits, and therefore could not support OpenNetMon.

9.4.1.2 Cisco

The Cisco pipeline was the second most successful physical pipeline, supporting 72% of the virtual pipelines. Like the Aruba, the Cisco has a configurable pipeline and does not require recirculation to support any virtual pipeline.

The reasons Shoehorn failed to find mappings for the Cisco pipeline were:

- the Cisco pipeline cannot perform MPLS matches and actions, preventing it from supporting the VPNs controllers, and HSDN;
- the Cisco pipeline cannot support virtual pipelines that rewrite IP source

and Destination fields—Random Host Mutation, and OF-Like PBR; and

- the Cisco pipeline could not support the IPv4 DSCP and ECN match fields used by OFLoad and OpenNetMon.

9.4.1.3 Nvidia

The Nvidia pipeline only supported 5 controllers. This is primarily due to the Nvidia OpenFlow pipeline using entirely `ternary` match kinds—any virtual pipeline that requires `exact` match kinds could not be supported by the Nvidia pipeline. By using `configured_any` match kinds in the place of `ternary` match kinds, Shoehorn was able to map 18 of the virtual pipelines to the Nvidia pipeline. Shoehorn failed to find mappings for virtual pipelines based on MPLS controllers, controllers that rewrite IP source or destination fields, and controllers that match ARP TPA.

The Nvidia hardware does support the SAI, however, so, in practice, the hardware could support more of these controllers by using its SAI pipeline.

9.4.1.4 OF-DPA

The OF-DPA pipeline supported 76% of the virtual pipelines, the most of any pipeline. The OF-DPA is the only controller to support MPLS matches and actions, and is therefore the only controller to support HSDN, or either Virtual Private Networks (VPNs) pipeline.

The most common reason Shoehorn failed to find a mapping for the OF-DPA is because it cannot support tables that require `exact` 5-tuple matches. The OF-DPA can only match 5-tuple fields in the Policy ACL table, but that uses `ternary` match kinds. This affected iTelescope, NFShunt, and OFLoad. Some

other controllers used 5-tuple matches, but when these targeted preconfigured flows, the evaluation used the `flexible_match_kinds` annotation, as the update rate is not as relevant for preconfigured flows.

The other reasons that the OF-DPA pipeline was unable to support virtual controllers were:

- Castor and TouSIX unicast ARP packets by matching ARP target protocol address fields, which the OF-DPA pipeline does not support; and
- The OF-DPA pipeline cannot write IP source and destination fields, preventing it from supporting OF-Like PBR or Random Host Mutation.

9.4.1.5 SAI

The SAI pipeline supported 64% of the virtual pipelines.

The reasons the SAI pipeline failed to support pipelines were:

- the SAI pipeline cannot support iTelescope, NFShunt or OFLoad as it cannot perform `exact` 5-tuple matches;
- the SAI pipeline cannot support ARP target protocol matches, preventing it from supporting Castor and TouSIX;
- the SAI pipeline cannot perform MPLS matches and actions, which are required by VPNs-P, VPNs-PE, or HSDN; and
- the SAI pipeline cannot rewrite IP source and destination fields, meaning it cannot support OF-Like PBR or Random Host Mutation.

9.4.2 Recirculations

No mapping required more than three recirculations, which is the threshold identified in section 6.2.3. This section describes the reasons that recirculation was required for each controller.

The evaluation implementation of AuthFlow has two control blocks, the first handles protocol traffic, used to negotiate access, and the second control block drops all traffic from unauthorised hosts. Shoehorn used 2 recirculations to map the AuthFlow pipeline to the SAI and OF-DPA pipelines. The first recirculation is needed because the two tables in the first control block both need to be supported by the physical pipelines' ACL tables, this would not have been necessary with the `flexible_mapping` annotation. The second recirculation is necessary because the table in the second control block must only be applied to packets that do not match either table in the first control block.

Supporting the Faucet pipeline with the OF-DPA or SAI pipelines requires 3 recirculations. This is caused by tables in the Faucet pipeline that modify fields that are matched by subsequent tables. The first two cases are the Port and VLAN ACL tables, and finally the routing table sets the Ethernet Destination, which is matched by the `EthernetSwitching` extern in the final control block. VIP Lanes also requires 3 recirculations, as it uses the `NORMAL` action, and therefore the implementation for this evaluation was integrated with the Faucet pipeline.

The In-Packet Bloom Filters pipeline requires recirculation when mapped to the OF-DPA or SAI pipelines as the forwarding table in the second control block is applied to packets that are not matched by either table in the first control block. Shoehorn maps all these tables to the ACL tables of the two pipelines.

SIMPLE requires recirculation with the OF-DPA and SAI pipelines, as it re-uses the ACL tables.

The OF-DPA and SAI pipelines require a recirculation to support the Precision Medicine pipeline, as the two tables used by Precision Medicine are applied to overlapping packets, but both need to be supported by the physical pipelines' ACL tables. This could easily be resolved by redesigning the virtual pipeline, as the two tables apply to mutually exclusive sets of packets. A design where the ARP table is only applied to ARP packets, and the forwarding table applies to other packets, would allow the tables to be merged.

The OF-DPA and SAI require two recirculations to support Tennison. All of the tables used by Tennison are supported using the pipelines' ACL tables, and therefore Shoehorn recirculates after each control block, to access the ACL table multiple times.

9.4.3 Run Time

The mappings were performed on a standard PC (Intel Core i3-2120 CPU with 4GB RAM). In all cases the mapping was found, or Shoehorn failed, in less than 5 minutes. As Shoehorn runs at compile time, this is a satisfactory outcome for use in production.

9.5 Discussion

9.5.1 Likely Causes for Failure

This evaluation found that in every case where Shoehorn was unable to find a mapping, it was due to a virtual table that could not be supported by any

table in the physical pipeline. For the flexible pipelines—the Aruba, the Nvidia, and the Cisco—this is likely to always be true, as the tables can be arranged in any order. Provided they can support all of the virtual tables, and have enough tables available, these pipelines can simply recreate the virtual pipeline directly.

The fixed-function pipelines, however, might fail due to unsupportable access sets. The fixed-function pipelines rely heavily on their ACL tables to support a variety of virtual tables, and the ACL tables apply to all packets. Any virtual table that is accessed following a hit in another table cannot be mapped to the ACL table, unless it uses the `flexible_mapping` annotation. Virtual tables that are accessed following a miss in another table can more easily be supported by the ACL table, as `ternary` tables can be concatenated (§6.3.4).

9.5.2 Programming Pipelines

To ensure that Shoehorn has the best possible chance of finding a mapping, it is important when programming virtual pipelines to ensure that tables are not being restricted arbitrarily. Most importantly, programmers should use the `flexible_match_kinds` annotation whenever the update rate for a table is low. The fixed-function physical pipelines, SAI and OF-DPA both relied heavily on ACL tables to support virtual tables, often aggregating tables together to be supported by the ACL table, then recirculating to allow tables in the next control block to be mapped to the ACL table as well.

The `flexible_mapping` annotation was only simulated to merge tables with termination MAC tables—with the VIP table in the Faucet pipeline, and with the ingress routing table in the VPNs PE pipeline. The `flexible_mapping` annotation not only reduces the update rate, but also increases the memory

usage, so it is only practical for very small tables, or when the controller developer knows that two tables can be combined without increasing the overall number of entries.

In some cases the definition of the physical pipeline can reduce the complexity of finding mappings. For instance, the OF-DPA always carries internal VLAN tags, this is not included in the pipeline definition however, as a pipeline where all packets arrive and exit the datapath untagged, and internally only carry the default VLAN tag, is—from the perspective of Shoehorn—identical to one with no VLAN tags.

9.5.3 Ideal Hardware

From these results, it is easy to propose a hardware pipeline that supports all the controllers in this evaluation. Simply by adding a table that does `exact` 5-tuple matching, as well as adding ARP TPA matches and set IPv4 address actions to the Policy ACL table, the OF-DPA pipeline would support all the controllers.

Changing the Policy ACL table to use `configured_any` matches does not allow Shoehorn to find mappings to the OF-DPA for all pipelines that use `exact` 5-tuple matches. A table using `configured_any` matches, cannot support multiple tables when one of them uses `exact` matches, unless all the tables use the same matches. Instead, a new table that can perform `exact` 5-tuple matches is required. An exact match table that matches arbitrary combinations of fields would also allow supporting other controllers such as FlowTags [33].

Two controllers aggregated tables to simulate the `flexible_mapping` annotation, and the OF-DPA supported both in the Policy-ACL table. This would not have been necessary with a two stage Policy-ACL table, where the second

table is optionally accessed after a hit in the first.

This approach, of including a small number of flexible tables in a fixed-function pipeline, is similar to that proposed by PINS [86].

9.5.4 Other Algorithmic Mapping Systems

This evaluation does not compare Shoehorn directly to FlowConvertor [89] or the system proposed by Sanger, Luckie, and Nelson [100], as these previous system map populated OpenFlow flow tables, whereas Shoehorn maps pipelines defined in P4. This evaluation demonstrated mapping complicated real-world controllers to real physical pipelines, which neither previous algorithm achieved.

Unlike the previous algorithms, Shoehorn ensures that real-time translation is practical, FlowConvertor demonstrated their system is practical for real-time translation with simple pipelines, but do not guarantee the update rate will be preserved, and likely cannot achieve real-time translation with larger pipelines. Sanger, Luckie, and Nelson found their algorithm is not suitable for real-time translation. Likewise, by preserving the match kinds used by each table, Shoehorn ensures the power consumption of the pipeline is not negatively impacted.

FlowConvertor relies heavily on metadata to control the path of packets through the pipeline. Metadata is poorly supported by the target hardware, however. Shoehorn does not have this requirement.

FlowConvertor has not been demonstrated to work with flexible hardware. Sanger, Luckie, and Nelson found that their solution was unable to scale to work with flexible hardware. Shoehorn is able to support flexible hardware as

well as fixed-function pipelines.

Chapter 10

Conclusion

10.1 Summary

This thesis investigated the practicality of improving portability in SDN controllers for low cost hardware, by algorithmically mapping from a virtual pipeline to a constrained physical pipeline, in a manner suitable for real-time control channel translation. This thesis finds that such an approach is practical, provided the hardware is capable of efficiently recirculating packets, while retaining metadata identifying the ingress port, and recirculation count.

To demonstrate the practicality of this approach, this thesis presented Shoehorn, a new system for finding mappings from virtual to physical pipelines. This thesis used Shoehorn to map 25 virtual pipelines, based on real SDN controllers, to 5 physical pipelines, based on different vendors implementations of SDN standards. Provided the physical pipeline included tables capable of supporting the matches and actions of each table in the virtual pipeline, Shoehorn was always able to find a successful mapping. Overall, every virtual pipeline was supported by at least one physical pipeline, and all but 6 virtual pipelines

were supported by multiple physical pipelines.

This thesis described the mapping algorithm used by Shoehorn. The algorithm ensures that mappings are suitable for real-time control channel translation by mapping virtual tables directly to physical tables, so that the number of table entries in the physical table is never greater than the number of table entries in the virtual table (unless explicitly instructed otherwise). Consequently, updating a single entry in the virtual pipeline only requires updating a single entry in the physical pipeline.

Shoehorn augments this approach by using packet recirculation. Packet recirculation allows Shoehorn to resolve incompatible table ordering between pipelines. No mapping produced as part of the evaluation required more than 3 recirculations, which is below the threshold where recirculation has a significant impact on overall throughput.

This thesis presented the design of two new P4 architectures, for defining the virtual and physical pipelines for Shoehorn to map. P4 can describe the capabilities of pipelines precisely, including specifying the control flow more efficiently than OpenFlow. The architectures use an extern called an Action Module to specify where pipelines apply actions. This eliminates many situations where, in OpenFlow, tables are forced into an explicit ordering that is not relevant to the behaviour of the datapath.

The design of Shoehorn was informed by a detailed analysis of a variety of SDN controllers and SDN implementations in hardware. The analysis of controllers included analysis of 6 deployments of SDN in production, as well as the published details of 50 controllers from research projects. This analysis produced a dataset containing details of the pipelines used by these controllers, which this thesis has made publicly available for use with future research into the use

of SDN.

10.2 Future Work

Shoehorn is capable of mapping virtual pipelines to diverse physical pipelines, but relies on hardware recirculating packets efficiently, while retaining metadata. As hardware vendors do not make the specifications of their hardware publicly available, this thesis has made assumptions about how hardware can achieve this. With better knowledge of the capabilities of physical devices, it is possible that Shoehorn could provide better support for metadata in the SVA. Likewise, Shoehorn assumes that physical hardware will use what metadata is available to it to ensure that Shoehorn's intrinsic metadata functions correctly. Shoehorn could use that metadata to improve its ability to find mappings for virtual tables.

Shoehorn uses two annotations to indicate how virtual tables can be mapped. A richer set of annotations, similar to those used in NOSIX [114], could enable Shoehorn to find suitable mappings more often. For instance, an annotation indicating the maximum number of entries a table can have, and another indicating a maximum scale factor (indicating the maximum number of physical table entries that can be updated when a single virtual table entry is updated), would allow Shoehorn to aggregate small tables together with a Cartesian product of entries, provided the maximum number of entries was less than the scale factor of the other table. A scale factor annotation would also allow Shoehorn to use split mappings, as proposed by Sanger, Luckie, and Nelson [100].

Shoehorn maps virtual tables greedily, so the mappings often require physical tables to match a large number of fields, which hardware may find difficult to

support. The inclusion of a `maximum_width` annotation could allow Shoehorn to avoid mappings with more fields than hardware can support.

Shoehorn requires that P4 actions do not use any control logic, to simplify the process of identifying equivalent actions in different pipelines. Further research could investigate practical methods for mapping more sophisticated actions.

References

- [1] Yehuda Afek, Anat Bremler-Barr, and Lior Shafir. “Network anti-spoofing with SDN data plane”. In: *IEEE INFOCOM 2017-IEEE conference on computer communications*. IEEE. 2017, pp. 1–9.
- [2] Anurag Agrawal and Changhoon Kim. “Intel Tofino2-A 12.9 Tbps P4-Programmable Ethernet Switch”. In: *Hot Chips Symposium*. 2020, pp. 1–32.
- [3] Allied Telesis, Inc. *The OpenFlow™ Protocol Feature Overview and Configuration Guide*. Allied Telesis, Inc. 2019.
- [4] Arista Networks. *EOS 4.27.0F User Manual*. Arista Networks. 2021.
- [5] Alon Atary and Anat Bremler-Barr. “Efficient round-trip time monitoring in OpenFlow networks”. In: *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*. IEEE. 2016, pp. 1–9.
- [6] Mounir Azizi, Redouane Benaini, and Mouad Ben Mamoun. “Delay measurement in OpenFlow-enabled MPLS-TP network”. In: *Modern Applied Science* 9.3 (2015), p. 90.
- [7] Abdelhadi Azzouni, Nguyen Thi Mai Trang, Raouf Boutaba, and Guy Pujolle. “Limitations of OpenFlow topology discovery protocol”. In:

- 2017 16th annual mediterranean Ad hoc networking workshop (Med-Hoc-Net)*. IEEE. 2017, pp. 1–3.
- [8] Josh Bailey and Stephen Stuart. “Faucet: Deploying SDN in the enterprise”. In: *Communications of the ACM* 60.1 (2016), pp. 45–49.
- [9] Edward Balas and A Ragusa. “SciPass: a 100Gbps capable secure Science DMZ using OpenFlow and Bro”. In: *Supercomputing 2014 conference (SC14)*. Supercomputing 2014 conference (SC14) New Orleans, Louisiana. 2014.
- [10] Pankaj Berde et al. “ONOS: towards an open, distributed SDN OS”. In: *Proceedings of the third workshop on Hot topics in software defined networking*. 2014, pp. 1–6.
- [11] Big Switch Networks. *Open Network Linux*. <https://opennetlinux.org/>. Accessed: 2022-08-19. 2019.
- [12] Pat Bosshart et al. “Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN”. In: *ACM SIGCOMM Computer Communication Review* 43.4 (2013), pp. 99–110.
- [13] Pat Bosshart et al. “P4: Programming protocol-independent packet processors”. In: *ACM SIGCOMM Computer Communication Review* 44.3 (2014), pp. 87–95.
- [14] Rodrigo Braga, Edjard Mota, and Alexandre Passito. “Lightweight DDoS flooding attack detection using NOX/OpenFlow”. In: *IEEE Local Computer Network Conference*. IEEE. 2010, pp. 408–415.
- [15] Broadcom. *BCM53346 64 Gb/s Multilayer Switch Product Brief*. Product Brief. 2020.
- [16] Broadcom. *BCM56370 560 Gb/s Programmable Multilayer Switch*. Product Brief. 2020.

-
- [17] Broadcom. *NPL - Network Programming Language Specification v1.3*. Specification. 2019.
- [18] Broadcom. *OpenFlow™-Data Plane Abstraction (OF-DPA): Abstract Switch Specification*. 2014.
- [19] Broadcom. *OpenNSL 2.0—Library of Open Networking APIs*. Product Brief. 2015.
- [20] Broadcom. *SDKLT: Logical Table-Based Switch Development Kit*. Product Brief. 2018.
- [21] Marc Bruyere et al. “Rethinking IXPs’ architecture in the age of SDN”. In: *IEEE Journal on Selected Areas in Communications* 36.12 (2018), pp. 2667–2674.
- [22] Yanghee Choi. “Implementation of content-oriented networking architecture (CONA): a focus on DDoS countermeasure”. In: *Proceedings of European NetFPGA developers workshop*. Citeseer. 2010.
- [23] Christopher Lorier. *SDN Pipelines*. <https://github.com/wandsdn/shoehorn-sdn-pipelines>.
- [24] Christopher Lorier. *Shoehorn*. <https://github.com/wandsdn/shoehorn>.
- [25] Cisco Systems, Inc. *Programmability Configuration Guide, Cisco IOS XE Gibraltar 16.11.x*. Cisco Systems, Inc. 2021.
- [26] Michael Dalton et al. “Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization”. In: *15th USENIX symposium on networked systems design and implementation (NSDI 18)*. 2018, pp. 373–387.
- [27] Huynh Tu Dang et al. “NetPaxos: Consensus at network speed”. In: *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*. 2015, pp. 1–7.

-
- [28] Avri Doria et al. *Forwarding and control element separation (ForCES) protocol specification*. RFC 5810. RFC Editor, Mar. 2010, pp. 1–124. URL: <http://www.rfc-editor.org/rfc/rfc5810.txt>.
- [29] Hilmi E Egilmez, S Tahsin Dane, K Tolga Bagci, and A Murat Tekalp. “OpenQoS: An OpenFlow controller design for multimedia delivery with end-to-end Quality of Service over Software-Defined Networks”. In: *Proceedings of the 2012 Asia Pacific signal and information processing association annual summit and conference*. IEEE. 2012, pp. 1–8.
- [30] Extreme Networks, Inc. *ExtremeXOS OpenFlow User Guide*. Extreme Networks, Inc. 2015.
- [31] Luyuan Fang et al. “Hierarchical SDN for the hyper-scale, hyper-elastic data center and cloud”. In: *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*. 2015, pp. 1–13.
- [32] Lyndon Fawcett et al. “Tennison: A distributed SDN framework for scalable network security”. In: *IEEE Journal on Selected Areas in Communications* 36.12 (2018), pp. 2805–2818.
- [33] Seyed Kaveh Fayazbakhsh et al. “Enforcing Network-Wide Policies in the Presence of Dynamic Middlebox Actions using FlowTags”. In: *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. 2014, pp. 543–546.
- [34] Nick Feamster, Jennifer Rexford, and Ellen Zegura. “The road to SDN: an intellectual history of programmable networks”. In: *ACM SIGCOMM Computer Communication Review* 44.2 (2014), pp. 87–98.
- [35] Marcial P Fernandez. “Comparing OpenFlow controller paradigms scalability: Reactive and proactive”. In: *2013 IEEE 27th International Con-*

-
- ference on Advanced Information Networking and Applications (AINA)*.
IEEE. 2013, pp. 1009–1016.
- [36] Stefan Geissler et al. “The Power of Composition: Abstracting a Multi-Device SDN Data Path Through a Single API”. In: *IEEE Transactions on Network and Service Management* 17.2 (2020), pp. 722–735. DOI: 10.1109/TNSM.2019.2951834.
- [37] Bastian Germann, Mark Schmidt, Andreas Stockmayer, and Michael Menth. “OFFWall: A static OpenFlow-based firewall bypass”. In: *11. DFN-Forum Kommunikationstechnologien*. Gesellschaft für Informatik eV. 2018.
- [38] Hassan Habibi Gharakheili et al. “iTeleScope: Intelligent Video Telemetry and Classification in Real-Time using Software Defined Networking”. In: *arXiv preprint arXiv:1804.09914* (2018).
- [39] James Griffioen et al. “VIP Lanes: High-speed custom communication paths for authorized flows”. In: *2017 26th International Conference on Computer Communication and Networks (ICCCN)*. IEEE. 2017, pp. 1–9.
- [40] David Hancock and Jacobus Van der Merwe. “Hyper4: Using p4 to virtualize the programmable data plane”. In: *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*. 2016, pp. 35–49.
- [41] Nikhil Handigol et al. “Plug-n-Serve: Load-balancing web traffic using OpenFlow”. In: *ACM Sigcomm Demo* 4.5 (2009), p. 6.
- [42] Zijun Hang, Yongjie Wang, and Shuguang Huang. “P4 Transformer: Towards Unified Programming for the Data Plane of Software Defined

- Network”. In: *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE. 2021, pp. 544–551.
- [43] Hewlett Packard Enterprise Development LP. *Aruba OpenFlow 1.3 Administrator Guide for ArubaOS-Switch 16.07*. Hewlett Packard Enterprise. 2018.
- [44] Atri Indiresan. *Private communication*. 2019.
- [45] IQTLabs. *Poseidon*. <https://github.com/IQTLabs/poseidon>. Accessed: 2022-07-24.
- [46] Jafar Haadi Jafarian, Ehab Al-Shaer, and Qi Duan. “OpenFlow random host mutation: transparent moving target defense using software defined networking”. In: *Proceedings of the first workshop on Hot topics in software defined networks*. 2012, pp. 127–132.
- [47] Sushant Jain et al. “B4: Experience with a globally-deployed software defined WAN”. In: *ACM SIGCOMM Computer Communication Review*. Vol. 43. 4. ACM. 2013, pp. 3–14. URL: <http://dx.doi.org/10.1145/2486001.2486019>.
- [48] Cheng Jin et al. “Magneto: Unified fine-grained path control in legacy and OpenFlow hybrid networks”. In: *Proceedings of the Symposium on SDN Research*. 2017, pp. 75–87.
- [49] Nattapong Kitsuwon, David B Payne, and Marco Ruffini. “A novel protection design for OpenFlow-based networks”. In: *2014 16th International Conference on Transparent Optical Networks (ICTON)*. IEEE. 2014, pp. 1–5.
- [50] Christos Kozanitis, John Huber, Sushil Singh, and George Varghese. “Leaping multiple headers in a single bound: Wire-speed parsing using

- the Kangaroo system”. In: *2010 Proceedings IEEE INFOCOM*. IEEE. 2010, pp. 1–9.
- [51] Himal Kumar, Craig Russell, Vijay Sivaraman, and Sujata Banerjee. “A software-defined flexible inter-domain interconnect using ONOS”. In: *2016 Fifth European Workshop on Software-Defined Networks (EWSDN)*. IEEE. 2016, pp. 43–48.
- [52] Rémy Lapeyrade, Marc Bruyère, and Philippe Owezarski. “OpenFlow-based Migration and Management of the TouIX IXP”. In: *NOMS 2016-2016 IEEE/IFIP Network Operations and Management Symposium*. IEEE. 2016, pp. 1131–1136.
- [53] Dan Levin, Marco Canini, Stefan Schmid, and Anja Feldmann. “Panopticon: Reaping the benefits of partial sdn deployment in enterprise networks”. In: (2013).
- [54] Christopher Lorier, Matthew Luckie, Marinho Barcellos, and Richard Nelson. “Shoehorn: Towards Portable P4 for Low Cost Hardware”. In: *2022 IFIP Networking Conference (IFIP Networking)*. IEEE. 2022, pp. 1–9.
- [55] Gabriele Lospoto, Massimo Rimondini, Benedetto Gabriele Vignoli, and Giuseppe Di Battista. “Rethinking virtual private networks in the software-defined era”. In: *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*. IEEE. 2015, pp. 379–387.
- [56] Thomas Lukaseder et al. “An sdn-based approach for defending against reflective ddos attacks”. In: *2018 IEEE 43rd Conference on Local Computer Networks (LCN)*. IEEE. 2018, pp. 299–302.
- [57] Carlos AB Macapuna, Christian Esteve Rothenberg, and Magalhães F Mauricio. “In-packet bloom filter based data center networking with

- distributed OpenFlow controllers”. In: *2010 IEEE Globecom Workshops*. IEEE. 2010, pp. 584–588.
- [58] Sharat Chandra Madanapalli et al. “Real-time detection, isolation and monitoring of elephant flows using commodity SDN system”. In: *NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium*. IEEE. 2018, pp. 1–5.
- [59] Diogo Menezes Ferrazani Mattos and Otto Carlos Muniz Bandeira Duarte. “AuthFlow: authentication and access control mechanism for software defined networking”. In: *annals of telecommunications* 71.11 (2016), pp. 607–615.
- [60] Nick McKeown et al. “OpenFlow: enabling innovation in campus networks”. In: *ACM SIGCOMM computer communication review* 38.2 (2008), pp. 69–74.
- [61] Syed Akbar Mehdi, Junaid Khalid, and Syed Ali Khayam. “Revisiting traffic anomaly detection using software defined networking”. In: *International workshop on recent advances in intrusion detection*. Springer. 2011, pp. 161–180.
- [62] Mellanox Technologies. *Mellanox Onyx User Manual*. 5.7. Mellanox Technologies. 2019.
- [63] Michael Menth et al. “Resilient integration of distributed high-performance zones into the BelWue network using OpenFlow”. In: *IEEE Communications Magazine* 55.4 (2017), pp. 94–99.
- [64] Anshuman Mishra, Deven Bansod, and Kotakula Haribabu. “A Framework for OpenFlow-like Policy-based Routing in Hybrid Software Defined Networks.” In: *INC*. 2016, pp. 97–102.

-
- [65] Simeon Miteff and Scott Hazelhurst. “NFShunt: A Linux firewall with OpenFlow-enabled hardware bypass”. In: *2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*. IEEE. 2015, pp. 100–106.
- [66] David Murray and Terry Koziniec. “The state of enterprise network traffic in 2012”. In: *In Communications (APCC), 2012 18th Asia-Pacific Conference on*. IEEE, 2012, pp. 179–184.
- [67] Yukihiro Nakagawa, Kazuki Hyoudou, and Takeshi Shimizu. “A management method of IP multicast in overlay networks using OpenFlow”. In: *Proceedings of the first workshop on Hot topics in software defined networks*. 2012, pp. 91–96.
- [68] Chawanat Nakasan, Kohei Ichikawa, Hajimu Iida, and Putchong Uthayopas. “A simple multipath OpenFlow controller using topology-based algorithm for multipath TCP”. In: *Concurrency and Computation: Practice and Experience* 29.13 (2017), e4134.
- [69] Marcelo R Nascimento et al. “Virtual routers as a service: the route-flow approach leveraging software-defined networks”. In: *Proceedings of the 6th International Conference on Future Internet Technologies*. 2011, pp. 34–37.
- [70] New H3C Technologies Co., Ltd. *OpenFlow Configuration Guide*. New H3C Technologies Co., Ltd. 2019.
- [71] Harvey B Newman, Artur Barczyk, and Michael Bredel. *OLiMPS. open-flow link-layer multipath switching*. Tech. rep. California Institute of Technology (CalTech), Pasadena, CA (United States), 2014.
- [72] Bryan Ng, Matthew Hayes, and Winston KG Seah. “Developing a traffic classification platform for enterprise networks with SDN: Experiences

- & lessons learned”. In: *2015 IFIP Networking Conference (IFIP Networking)*. IEEE. 2015, pp. 1–9.
- [73] Mehdi Nobakht, Vijay Sivaraman, and Roksana Boreli. “A host-based intrusion detection and mitigation framework for smart home IoT using OpenFlow”. In: *2016 11th International conference on availability, reliability and security (ARES)*. IEEE. 2016, pp. 147–156.
- [74] Mehdi Nobakht, Vijay Sivaraman, and Roksana Boreli. “A host-based intrusion detection and mitigation framework for smart home IoT using OpenFlow”. In: *2016 11th International conference on availability, reliability and security (ARES)*. IEEE. 2016, pp. 147–156.
- [75] Nokia Corporation. *Router Configuration Guide R15.0.R1*. Nokia Corporation. 2017.
- [76] NoviFlow, Inc. *NoviSwitch™2128 High Performance OpenFlow Switch*. Product Datasheet. 2018.
- [77] Open Compute Project. *Switch Abstraction Interface (SAI)*. Tech. rep. Open Compute Project, 2015.
- [78] Open Networking Foundation. *Open Networking Foundation TTP repository*. https://github.com/OpenNetworkingFoundation/TTP_Repository. Accessed: 2022-24-07.
- [79] Open Networking Foundation. *OpenFlow Switch Specification Version 1.0. (Wire Protocol 0x01)*. Specification. 2009.
- [80] Open Networking Foundation. *OpenFlow Switch Specification Version 1.1. (Wire Protocol 0x02)*. Specification. 2011.
- [81] Open Networking Foundation. *OpenFlow Switch Specification Version 1.2. (Wire Protocol 0x03)*. Specification. 2011.

-
- [82] Open Networking Foundation. *OpenFlow Switch Specification Version 1.3. (Wire Protocol 0x04)*. Specification. 2012.
- [83] Open Networking Foundation. *OpenFlow Switch Specification Version 1.4. (Wire Protocol 0x05)*. Specification. 2013.
- [84] Open Networking Foundation. *OpenFlow Switch Specification Version 1.5. (Wire Protocol 0x06)*. Specification. 2014.
- [85] Open Networking Foundation. *OpenFlow Table Type Patterns*. Tech. rep. Open Networking Foundation, 2014.
- [86] Open Networking Foundation. *P4 Integrated Network Stack (PINS)*. <https://opennetworking.org/pins/>. Accessed: 2022-02-10.
- [87] P4.org API Working Group. *P4Runtime Specification*. <https://p4.org/p4-spec/p4runtime/main/P4Runtime-Spec.html>. Accessed: 2021-07-24. 2021.
- [88] P4.org Architecture Working Group. *P4₁₆ Portable Switch Architecture (PSA)*. <https://p4.org/p4-spec/docs/PSA.html>. Accessed: 2021-11-04. 2021.
- [89] Heng Pan et al. “FlowConvertor: Enabling portability of SDN applications”. In: *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*. IEEE. 2017, pp. 1–9.
- [90] Heng Pan et al. “The FlowAdapter: Enable flexible multi-table processing on legacy hardware”. In: *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. 2013, pp. 85–90.
- [91] Ben Pfaff et al. “The Design and Implementation of Open vSwitch”. In: *12th USENIX symposium on networked systems design and implementation (NSDI 15)*. 2015, pp. 117–130.

-
- [92] Nam Pho et al. “Data transfer in a science DMZ using SDN with applications for precision medicine in cloud and high-performance computing”. In: *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal. (SC15)*. 2015, pp. 1–4.
- [93] Pica8 Inc. *PICOS 4.3.0 Configuration Guide*. Pica8 Inc. 2022.
- [94] *pipeline_v6.pdf*. https://github.com/opencomputeproject/SAI/blob/master/doc/behavioral%20model/pipeline_v6.pdf. Accessed: 2022-08-08.
- [95] Zafar Ayyub Qazi et al. “SIMPLE-fying middlebox policy enforcement using SDN”. In: *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*. 2013, pp. 27–38.
- [96] Sivaramakrishnan Ramanathan, Yaron Kanza, and Balachander Krishnamurthy. “SDProber: A software defined prober for SDN”. In: *Proceedings of the Symposium on SDN Research*. 2018, pp. 1–7.
- [97] Charalampos Rotsos et al. “OFLOPS: An open framework for OpenFlow switch evaluation”. In: *Passive and Active Measurement*. Springer. 2012, pp. 85–95.
- [98] Rishikesh Sahay, Gregory Blanc, Zonghua Zhang, and Hervé Debar. “ArOMA: An SDN based autonomic DDoS mitigation framework”. In: *computers & security* 70 (2017), pp. 482–499.
- [99] Richard Sanger, Brad Cowie, Matthew Luckie, and Richard Nelson. “Characterising the limits of the OpenFlow slow-path”. In: *2018 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE. 2018, pp. 1–7.

-
- [100] Richard Sanger, Matthew Luckie, and Richard Nelson. “Towards Transforming OpenFlow Rulesets to Fit Fixed-Function Pipelines”. In: *Proceedings of the Symposium on SDN Research*. 2020, pp. 123–134.
- [101] Devavrat Shah and Pankaj Gupta. “Fast updating algorithms for TCAM”. In: *IEEE Micro* 21.1 (2001), pp. 36–47.
- [102] Sachin Sharma et al. “In-band control, queuing, and failure recovery functionalities for OpenFlow”. In: *IEEE Network* 30.1 (2016), pp. 106–112.
- [103] Jonathan Stringer et al. “Cardigan: SDN distributed routing fabric going live at an Internet exchange”. In: *2014 IEEE Symposium on Computers and Communications (ISCC)*. IEEE. 2014, pp. 1–7.
- [104] Mitsuhiro Suenaga, Makoto Otani, Hisaharu Tanaka, and Kenzi Watanabe. “Opengate on OpenFlow: system outline”. In: *2012 Fourth International Conference on Intelligent Networking and Collaborative Systems*. IEEE. 2012, pp. 491–492.
- [105] Xiaoye Sun, TS Eugene Ng, and Guohui Wang. “Software-defined flow table pipeline”. In: *2015 IEEE International Conference on Cloud Engineering*. IEEE. 2015, pp. 335–340.
- [106] The P4 Language Consortium. *The P4 Language Specification, Version 1.0.5*. <https://p4.org/p4-spec/p4-14/v1.0.5/tex/p4.pdf>. Accessed: 2021-07-24. 2018.
- [107] Ramona Trestian, Kostas Katrinis, and Gabriel-Miro Muntean. “OFLoad: An OpenFlow-based dynamic load balancing strategy for datacenter networks”. In: *IEEE Transactions on Network and Service Management* 14.4 (2017), pp. 792–803.

-
- [108] Niels LM Van Adrichem, Christian Doerr, and Fernando A Kuipers. “OpenNetMon: Network monitoring in OpenFlow software-defined networks”. In: *2014 IEEE Network Operations and Management Symposium (NOMS)*. IEEE. 2014, pp. 1–8.
- [109] Pablo B Viegas et al. “The actual cost of programmable smartnics: Diving into the existing limits”. In: *International Conference on Advanced Information Networking and Applications*. Springer. 2021, pp. 181–194.
- [110] Guohui Wang, TS Eugene Ng, and Anees Shaikh. “Programming your network at run-time for big data applications”. In: *Proceedings of the first workshop on Hot topics in software defined networks*. 2012, pp. 103–108.
- [111] Richard Wang, Dana Butnariu, and Jennifer Rexford. “{OpenFlow-Based} Server Load Balancing Gone Wild”. In: *Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE 11)*. 2011.
- [112] Xitao Wen et al. “RuleTris: Minimizing rule update latency for TCAM-based SDN switches”. In: *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2016, pp. 179–188.
- [113] SONiC wiki. *SONiC*. <https://github.com/sonic-net/SONiC/wiki>. Accessed: 2022-07-24. 2022.
- [114] Minlan Yu, Andreas Wundsam, and Muruganantham Raju. “NOSIX: A lightweight portability layer for the SDN OS”. In: *ACM SIGCOMM Computer Communication Review* 44.2 (2014), pp. 28–35.

Appendix A

Controllers

This appendix lists the controllers studied in chapter 9.1.

Publication	Network Type	Description
Dalton et al. [26]	Data centre	An OpenFlow controlled caching system for software switching between virtual hosts in Google data centres.
Mehdi, Khalid, and Khayam [61]	Enterprise	Implementations of 4 anomaly detection algorithms using OpenFlow
Afek, Bremler-Barr, and Shafir [4]	ISP	A SDN anti-spoofing scrubber for mitigating DDoS attacks
Mattos and Duarte [59]	ISP	A authentication framework for SDNs
Sahay et al. [98]	ISP	A framework for mitigating DDoS in SDNs
Menth et al. [63]	NREN	An OpenFlow controlled multi-campus Science DMZ (SDMZ)
Wang, Ng, and Shaikh [110]	Data centre	Implementations of forwarding algorithms for handling Hadoop aggregation traffic using OpenFlow.
Macapuna, Rothenberg, and Maurício [57]	Data centre	A load-balancing system based on encoding bloom filters into packet headers

Publication	Network Type	Description
Kumar et al. [51]	IX	A SDN IX controller that controls broadcast, improves telemetry, and security
Ng, Hayes, and Seah [72]	Enterprise	A SDN traffic classification platform for Enterprise networks
Choi [22]	ISP	An implementation of an ISP level content cache with DDoS mitigation
Braga, Mota, and Pasito [14]	Unknown	A DDoS detection system implemented in OpenFlow
Azizi, Benaini, and Mamoun [6]	ISP	A system for monitoring delay in OpenFlow MPLS-TP networks
Madanapalli et al. [58]	Enterprise	A system for detecting and monitoring elephant flows in SDN enterprise networks
Bailey and Stuart [8]	Enterprise	An enterprise SDN controller
Fayazbakhsh et al. [33]	Enterprise	A system for controlling how middle boxes are applied to packets in OpenFlow networks

Publication	Network Type	Description
Atary and Bremner [5]	N/A	A system for monitoring SDN link latency
Nobakht, Sivaraman, and Boreli [74]	Home	A system for monitoring traffic to IoT devices to detect and mitigate malicious intrusions
Fang et al. [31]	Data centre	An SDN data centre architecture
Sharma et al. [102]	N/A	A system for establishing and maintaining in-band control of OpenFlow networks
Gharakheili et al. [38]	Enterprise	A traffic classification and telemetry system
Wang, Butnariu, and Rexford [111]	Data centre	A system for load balancing with OpenFlow v1.0. Later versions of OpenFlow have made this system obsolete
Jin et al. [48]	Hybrid	An architecture for hybrid SDN–legacy networks
Pho et al. [92]	SDMZ	A system for creating protected paths for precision medicine flows
Nakasan et al. [68]	Enterprise	A system for ensuring MultiPath TCP (MPTCP) flows follow diverse paths in enterprise networks using OpenFlow

Publication	Network Type	Description
Nakagawa, Hyoudou, and Shimizu [67]	Data centre	A system for controlling multicast forwarding in data centres
Dang et al. [27]	Data centre	A technique for accelerating paxos protocols in the network
Miteff and Hazelhurst [65]	Campus	A controller for hardware acceleration of NetFilter firewalls
Azzouni et al. [7]	N/A	A secure approach for topology discovery in SDNs
Egilmez et al. [29]	Enterprise	A system for providing QoS by controlling the forwarding of traffic classes in OpenFlow networks
Germann et al. [37]	Enterprise	A system for mitigating TCP syn flood attacks
Trestian, Katrinis, and Muntean [107]	Data centre	A system for load balancing traffic in data centres
Jafarian, Al-Shaer, and Duan [46]	Enterprise	A controller that provides security by randomly rewriting the ip addresses of hosts within the network
Newman, Barczyk, and Bredel [71]	ISP	A controller that creates point to point tunnels using VLANs

Publication	Network Type	Description
Suenaga et al. [104]	Campus	An authentication system for campus networks
Van Adrichem, Doerr, and Kuipers [108]	Campus	A network monitoring application
Egilmez et al. [29]	Enterprise	A controller for providing Quality of Service (QoS) with OpenFlow
Levin et al. [53]	Hybrid	An architecture for hybrid SDN–legacy networks
Mishra, Bansod, and Haribabu [64]	Hybrid	An architecture for providing policy-based routing in hybrid SDN–legacy networks
Handigol et al. [41]	Enterprise	A system for providing HTTP load balancers in OpenFlow controller enterprise networks
Kitsuwan, Payne, and Ruffini [49]	ISP	A technique for creating protected MPLS circuits
Lukaseder et al. [56]	ISP	A system for defending against reflective DDoS attacks
Balas and Ragusa [9]	Campus	A secure SDMZ controller

Publication	Network Type	Description
Ramanathan, Kanza, and Krishnamurthy [96]	N/A	A mechanism for measuring link latency in software-defined networks
Qazi et al. [95]	Enterprise	An enterprise controller that tunnels packets through middleboxes
Fawcett et al. [32]	Enterprise	A system for network intrusion detection and mitigation in ONOS [10]
Lapeyrade, Bruyère, and Owezarski [52]	IX	An IX controller that reduces broadcast traffic
Bruyere et al. [21]	IX	An IX controller that encodes switching labels into Ethernet headers
Griffioen et al. [39]	Enterprise	A system for creating protected paths for authorised flows
Lospoto et al. [55]	ISP	An architecture for simplifying the configuration of 13 VPNs

Appendix B

Ethics Approval

This appendix contains the ethics approval letter to collect data from production SDN deployments.

Department of Computer Science
Faculty of Computing and Mathematical Sciences
Rorohiko me ngā Pūtaiao Pāngarau
The University of Waikato
Private Bag 3105
Hamilton 3240
New Zealand

Phone +64 7 838 4021
www.waikato.ac.nz



23 May, 2018

Christopher Lorier,
Department of Computer Science.

Dear Christopher,

As we have recently discussed, your proposed PhD research is covered by our newly formulated requirements for data security and privacy. We indicated that you needed to provide responses to the following four points:

- 1 A statement from the provider of the data about the conditions under which they have acquired the data, and the conditions under which they have passed it on, or allowed access by you. The details of such a statement will vary from situation to situation, so you would need to craft it appropriately. For example, an ISP providing access to the data might suggest that the terms and conditions which their customers sign up to acknowledges that the ISP may monitor traffic, but would not release any personal information other than to appropriate authorities, and that they are releasing it, or providing access, to you on the understanding that it will be securely stored, and that you will not divulge details to anyone else.
- 2 A statement from you relating to the security of the data. It might be accessed remotely from the client's server, or it might be downloaded onto a University computer. What steps will be taken to ensure that only those directly involved with the research will be able to gain access? What will happen to the data or the access path once the research is completed?
- 3 A non-disclosure agreement signed by each of the researchers who will have access to the data, stating that they will not divulge, and treat as strictly confidential, any personal or identifying information that they may see, discover, or uncover, while accessing and/or analysing this data. This would appear to also cover "de-identified" data, where there is a possibility that by matching with some other source, individuals could be identified.
- 4 A statement that steps would be taken to ensure that in any publication or report arising from the research, no personal or identifying information will appear.

Given that you have now provided us with appropriate responses, I am happy to confirm that there are no ethical issues raised by your planned research programme, although you may still need to apply to the FCMS Committee for approval for individual studies in due course.

I can confirm that in general terms, you are free to continue with the research.

Yours sincerely,



Mark Apperley
Professor of Computer Science
Convenor, FCMS Ethics Committee