POLITECNICO DI MILANO

STATEFUL DATA PLANE ABSTRACTIONS FOR SOFTWARE-DEFINED NETWORKS AND THEIR APPLICATIONS

CARMELO CASCONE
DIPARTIMENTO DI ELETTRONICA, INFORMAZIONE E BIOINGEGNERIA

Research director: Prof. Antonio Capone
Research co-director: Prof. Brunilde Sansò
Tutor: Prof. Gian Guido Gentili
PhD coordinator: Prof. Andrea Bonarini

DISSERTATION SUBMITTED TO OBTAIN
THE DIPLOMA OF PHILOSOPHIÆ DOCTOR
(INFORMATION TECHNOLOGY, CYCLE XXIX)

POLITECNICO DI MILANO

This thesis titled:

STATEFUL DATA PLANE ABSTRACTIONS FOR SOFTWARE-DEFINED NETWORKS AND THEIR APPLICATIONS

presented by: CASCONE Carmelo
to obtain the diploma of: Philosophiæ Doctor
has been accepted by the examination board consisting of:

CARDINAL Christian, Prof., president
CAPONE Antonio, Prof., member and research director
SANSÒ Brunilde, Prof., member and research co-director
SCHNEIDER Fabian, Dr., member
GIACCONE Paolo, Prof., external member

DEDICATION

To my family.

ACKNOWLEDGEMENTS

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

First and foremost, I wish to thank my two advisors Antonio Capone and Brunilde Sansò, without them, the journey to the point where I am writing this would not have been nearly possible. Apart from sharing with me their deep technical expertise, I thank them for seeing something in me and convincing me to start this Ph.D. adventure four years ago. They have been a role model, providing invaluable support and advice on shaping my research, presenting my work to the scientific community, advising students, finding jobs, and on life in general. Thank you!

I would also like to thank my thesis committee members, Paolo Giaccone and Fabian Schneider, who dedicated part of their precious time to read this work. Their comments on draft versions were helpful and improved the document considerably.

Since the very beginning of my Ph.D., I was fortunate to have many collaborators with whom I worked on projects which led to this dissertation. I would like to thank Giuseppe Bianchi and Marco Bonola for having me involved in this great idea of making OpenFlow stateful. Without them, the topic of this thesis would have been different. Thank you for sharing with me your ideas and enthusiasm for research. I am also grateful to Roberto Bifulco, Salvatore Pontarelli, and Nicola Bonelli. Roberto's help was fundamental in shaping my research style, he taught me the importance of being honest when writing papers, and that of always keeping an eye on the "broad picture"; Salvatore taught me everything I know about hardware, and Nicola everything I know about high-performance packet processing in software. Along with them, I would like to thank all the people involved in the BEBA EU project, for sharing their expertise and for the fun moments. It has been great working with all of you!

This work would not have been possible without the generous help of Davide Sanvito, Luca Pollini, Alessandro Q.T. Nguyen, and Luca Bianchi, whom I had the pleasure of tutoring while working on their master thesis, and which help significantly contributed to this thesis.

I would like to thank also my lab mates Davide Sanvito (again!), Luca Gianoli, Filippo Malandra, Silvia Boiardi, and all the people of the AntLab and the ISGP group at Politecnico di Milano. Thank you for sharing your knowledge with me and for all the lunches and coffee breaks. Special mention goes to Luca Gianoli: I received my first taste of research while working under his supervision for my master thesis. He helped me write my very first paper,

an experience that motivated me to apply for the Ph.D.

A big thank goes to the administrative personnel that kept these four years running smoothly. Thanks to Francesca Clemenza and Nathalie Levesque for helping me with all the bureaucratic requirements. Thanks to Roberto Resmini, Edoh Logo, and Pierre Girard for providing the necessary technical support and for allowing me to perform crazy experiments on the campus network.

Many friends outside school provided the necessary encouragement and distraction that made this ride more enjoyable and most importantly kept me up in tough moments. Special mention goes to Gianmaria, for the culinary experiences and for making my time in Milan more pleasant; to Fabrizio, for allowing me to cultivate my other big passion for music; to Gabriel, Hugo, Lorenzo, Oren, Paul, Sarah, and Sophy for making me feel so welcome during my time in Montreal.

I am grateful to my family, particularly to my parents, for their continued support of my plans, and their encouragement both before and during my studies. Without them, I probably would not have moved to Milan to study telecommunications engineering, and I would not have written this.

Finally, thank you is not enough to express how grateful I am to my partner Viviana. It's only with her endless love and support that this dissertation was possible. She was always there during tougher times. How many times did I tell you that I wanted to drop out? She made sure that I did not, and that I did not forget the lighter side of things.

ABSTRACT

Software-Defined Networking (SDN) enables programmability in the network. Unfortunately, current SDN limits programmability only to the control plane. Operators cannot program data plane algorithms such as load balancing, congestion control, failure detection, etc. These capabilities are usually baked in the switch via dedicated hardware, as they need to run at line rate, i.e. 10-100 Gbit/s on 10-100 ports.

In this work, we present two data plane abstractions for stateful packet processing, namely OpenState and OPP. These abstractions allow operators to program data plane tasks that involve stateful processing. OpenState is an extension to OpenFlow that permits the definition of forwarding rules as finite state machines. OPP is a more flexible abstraction that generalizes OpenState by adding computational capabilities, opening for the programming of more advanced data plane algorithms. Both OpenState and OPP are amenable for high-performance hardware implementations by using commodity hardware switch components. However, both abstractions are based on a problematic design choice: to use a feedback-loop in the processing pipeline. This loop, if not adequately controlled, can represent a harm for the consistency of the state operations. Memory locking approaches can be used to prevent inconsistencies, at the expense of throughput. We present simulation results on real traffic traces showing that feedback-loops of several clock cycles can be supported with little or no performance degradation, even with near-worst case traffic workloads.

To further prove the benefits of a stateful programmable data plane, we present two *novel* applications: Spider and FDPA. Spider permits to detect and react to network failures at data plane timescales, i.e. micro/nanoseconds, also in the case of distant failures. By using OpenState, Spider provides functionalities equivalent to legacy control plane protocols such as BFD and MPLS Fast Reroute, but without the need of a control plane. That is, both detection and rerouting happen entirely in the data plane.

FDPA allows a switch to enforce approximate fair bandwidth sharing among many TCP-like senders. Most of the mechanisms to solve this problem are based on complex scheduling algorithms, whose feasibility becomes very expensive with today's line rate requirements. FDPA, which is based on OPP, trades scheduling complexity with per-user state. FDPA works by dynamically assigning users to few (3-4) priority queues, where the priority is chosen based on the sending rate history of a user. Experimental results on a 10 Gbit/s testbed show that FDPA is able to provide fairness and throughput comparable to scheduling-based approaches.

SOMMARIO

Software-Defined Networking (SDN) permette la programmabilità delle reti. Purtroppo, le attuali tecnologie SDN limitano la programmabilità solo al piano di controllo. Gli operatori non possono programmare algoritmi del piano dati come il bilanciamento del carico, il controllo di congestione, il rilevamento di guasti, ecc. Queste funzionalità sono generalmente implementate nei dispositivi di interconnessione tramite hardware dedicato, poiché devono essere eseguite a velocità di linea, ovvero 10-100 Gbit/s su 10-100 porte.

In questo lavoro di ricerca, presentiamo due astrazioni del piano dati per l'elaborazione di pacchetti in maniera stateful, ovvero OpenState e OPP. Queste astrazioni permettono agli operatori di programmare algoritmi del piano dati che comportano un'elaborazione basata su stato. OpenState è un'estensione di OpenFlow che consente la definizione di regole di flusso nella forma di macchine a stati finiti. OPP è un'astrazione più flessibile che generalizza OpenState aggiungendo funzionalità computazionali, permettendo la definizione di algoritmi più avanzati. Sia OpenState che OPP permettono un'implementazione ad alte prestazioni utilizzando componenti hardware commodity. Tuttavia, entrambe le astrazioni si basano su una scelta di progettazione problematica, cioè quella di utilizzare un feedback-loop nella pipeline di elaborazione dei pacchetti. Questo feedback-loop, se non adeguatamente controllato, può rappresentare un problema per la coerenza delle operazioni di lettura e scrittura dello stato. Approcci basati su blocco della memoria possono essere utilizzati per prevenire incongruenze, a scapito del throughput. In questa ricerca, presentiamo dei risultati di simulazione su tracce di traffico reali, che dimostrano che feedback-loop lunghi diversi cicli di clock possono essere supportati con poca o nessuna degradazione del throughput, anche in situazioni di traffico vicine al caso peggiore.

Per dimostrare ulteriormente i vantaggi di un piano di dati programmabile e stateful, presentiamo due nuove applicazioni: Spider e FDPA. Spider consente di rilevare e reagire a guasti di rete su scale temporali tipiche del piano dati, cioè micro/nanosecondi, anche in caso di guasti distanti. Utilizzando OpenState, Spider fornisce funzionalità equivalenti a protocolli noti del piano di controlli come BFD e MPLS Fast Reroute, ma senza bisogno di un piano di controllo. Cioè, sia il rilevamento che il re-instradamento vengono effettuati interamente nel piano dei dati.

FDPA consente ad un dispositivo di interconnessione di applicare una condivisione di banda equa tra diverse sorgenti di traffico TCP. La maggior parte dei meccanismi esistenti per risolvere questo problema si basano su complessi algoritmi di scheduling, la cui implementabilità si

rivela molto costosa con le richieste di throughput di oggi. FDPA, implementato utilizzando OPP, permette di sostituire alla complessità degli algoritmi di scheduling, la complessità del mantenere uno state per utente. FDPA si base sul principio che gli utenti possono essere assegnati dinamicamente ad una coda prioritaria, dove la priorità viene scelta in base alla frequenza di invio di pacchetti di un utente. Abbiamo realizzato un testbed a 10 Gbit/s, i risultati mostrano che FDPA è in grado di garantire una suddivisione della banda equa e throughput comparabile ad approcci basati su scheduling.

PREVIOUSLY PUBLISHED MATERIAL

Section 2.1 revises and combines material from two previous publications [1][24]:

- G. Bianchi, M. Bonola, A. Capone, and C. Cascone. 2014. OpenState: programming platform-independent stateful openflow applications inside the switch. SIGCOMM Comput. Commun. Rev., vol. 44, no. 2, pp. 44–51, April 2014.
- S. Pontarelli, M. Bonola, G. Bianchi, A. Capone, and C. Cascone. Stateful OpenFlow: Hardware proof of concept. In Proceedings of 16th International Conference on High Performance Switching and Routing (HPSR), July 2015, pp. 1–8.

Chapter 3 revises and combines material from two previous publications [2, 3]:

- C. Cascone, L. Pollini, D. Sanvito, A. Capone, and B. Sansò. Spider: Fault resilient SDN pipeline with recovery delay guarantees. In Proceedings of NetSoft Conference and Workshops (NetSoft), June 2016, pp. 296–302.
- C. Cascone, D. Sanvito, L. Pollini, A. Capone, and B. Sansò. Fast failure detection and recovery in SDN with stateful data plane. International Journal of Network Management (IJNM), vol. 27, no. 2, November 2016.

Chapter 4 revises material from a previous publication [4]:

- C. Cascone, N. Bonelli, L. Bianchi, A. Capone, and B. Sansò. Towards approximate fair bandwidth sharing via dynamic priority queuing. In Proceedings of IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN), June 2017, pp. 1–6.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS

ACL	Access Control List
ALU	Arithmetic Logic Unit
API	Application Programming Interface
AQM	Active Queue Management
ASIC	Application-Specific Integrated Circuit
BDP	Bandwidth-Delay Product
BFD	Bidirectional Forwarding Detection (protocol)
BGP	Border Gateway Protocol
CAM	Content-Addressable Memory
CPU	Central Processing Unit
DPI	Deep Packet Inspection
DRAM	Dynamic Random-Access Memory
DRR	Deficit Round Robin
DSCP	Differentiated Services Code Point
DSP	Digital Signal Processor
ECMP	Equal-Cost Multi Path
EFSM	Extended Finite-State Machine
FDPA	Fair Dynamic Priority Assignment
FF	Fast-Failover (OpenFlow)
FIFO	First In First Out
FPGA	Field-Programmable Gate Array
FQ	Fair Queuing
FRR	Fast-Reroute (MPLS)
FSM	Finite-State Machine
GPU	Graphic Processing Unit
IETF	Internet Engineering Task Force
IP	Internet Protocol
IR	Intermediate Representation
ISP	Internet Service Provider
LPM	Longest-Prefix Match
LSP	Label-Switched Path (MPLS)
LSR	Label Switch Router (MPLS)
LUT	LookUp Table (FPGA)

MAT	Match+Action Table
MPLS	MultiProtocol Label Switching
MTU	Maximum Transmission Unit
NAT	Network Address Translation
NFV	network Function Virtualization
NIC	Network Interface Card
NPU	Network Processing Unit
OPP	Open Packet Processor
OSPF	Open Shortest Path First
RAM	Random-Access Memory
RFC	Request For Comment (IETF)
RISC	Reduced Instruction Set Computer
RMT	Reconfigurable Match Table
RSVP	Resource Reservation Protocol
RTT	Round Trip Time
SDN	Software-Defined Networking
SFQ	Stochastic Fair Queuing
SRAM	Static Random-Access Memory
SSH	Secure Shell
TCAM	Ternary Content-Addressable Memory
TCP	Transmission Control Protocol
TPU	Tensor Processing Unit
UDP	User Datagram Protocol
VNF	Virtual Network Function

CHAPTER 1 INTRODUCTION

Networking technologies have undergone a major revolution in recent years. Both academia and industry have worked to make networks more open and programmable. Open, as opposed to the traditional closed paradigm, where network equipment vendors develop both proprietary hardware and software, bundled together in the same black box, hard to change once deployed. Programmable, to give system owners the possibility to code their needs, instead of relying on the slow development cycles adopted by vendors to add support for new protocols and services.

Software-Defined Networking (SDN) has been the epicenter of this revolution. SDN is based on the simple idea of decoupling the control plane from the forwarding devices' data plane. Abstracting networks into separate planes is a convenient way of reasoning about network programmability. The data plane is the one responsible for the forwarding of packets, i.e. the circuitry necessary to move (or drop) packets between ports of the same device. The control plane is where the network-level intelligence is implemented, i.e. decide which devices and ports a packet should be sent through to implement a given network policy, such as routing or security.

The two planes have different operative requirements that significantly affect how programmability is enabled. The data plane needs to process packets as fast as possible. Today's line rate requirements are 10–100 Gbit/s on 10–100 ports. For this reason, the data plane is also called the fast-path. Such high-performance requirements call for implementations based on dedicated hardware, such as ASIC, network processors (NPU), or FPGAs . In contrast, the control plane can operate at lower speeds, but it needs to perform relatively complex computations, e.g. find the best path in BGP. As a consequence, implementations favor general purpose CPUs and high-level languages such as C, Java or Python.

SDN was first introduced to enable programmability of the control plane. Indeed, SDN abstracts the data plane by means of an Application Programming Interface (API). Software running on the control plane can use such API to instruct the devices on how to forward packets. SDN's primary challenge was that of convincing network equipment vendors to open their boxes and provide such APIs to operators, to let them implement their own control software, i.e. convince vendors to abandon a generous stream of revenues. Clearly, a daunting task. However, challenges in the data plane are of a different nature, as they relate to the inherent technical complexity of making high-performance architectures programmable. This thesis is about mechanisms to enable programmability in the data plane. However,

before deepening into this topic, it is useful to discuss how control planes have been made programmable.

1.1 Control plane programmability

SDN builds upon the observation that most protocols, e.g. OSPF, BGP, etc. all define similar mechanisms to distribute state such as the network topology and other metadata (e.g. link utilization). Decisions on where to forward packets are then taken independently from each device, according to a given routing logic that looks at the distributed network state. The result is a plethora of standard protocols, such as IETF's RFCs (more than 8000 at the time of writing). RFCs, apart from describing ideas on how to improve networks, specify in plain English language (sometimes ambiguously) the expected behavior that the control plane of a device should implement to support a given protocol. It comes that most protocol designers had to repeat the same effort over and over to develop and debug complex state distribution mechanism for each protocol. Similarly, software engineers from different vendors had to produce many similar implementations of these specifications for the control plane of many different network devices.

In SDN, the control plane resides in a logically centralized software external to the switches, usually running on commodity servers called "controllers". A controller communicates with *many* devices' data plane using a *forwarding* API. OpenFlow [5] is a prominent instance of such an API. It is a standardized open protocol that abstracts the data plane as match+action tables (MAT). Controllers can instruct devices on where to forward packets by installing entries in the devices' MAT. Each MAT entry specifies a *match*, used to identify a slice of traffic, and an *action* that specifies processing to apply to matching packets, such dropping or forwarding to one or more ports, placing packets in output queues, and modifications to header fields.

The motivation of creating OpenFlow as an open protocol is that switches can be developed to be vendor-agnostic, greatly simplifying the work of control plane writers who can now reuse principles common in software engineering. Indeed, given the initial SDN observation and the assumption of a data plane API common to all vendors, most software engineers would end up with the same conclusion: (i) write a distributed data store to maintain an updated view of the network topology and other metadata; (ii) make it fault tolerant and scalable by replicating the data onto multiple machines; (iii) write it once and reuse it over and over for many protocols. Now, protocol developers do not have to worry about writing each time a new state distribution mechanism or a standard binary protocol. This is the purpose of today's network operating systems (OS) such as ONOS [6], OpenDaylight [7], RYU [8], and

many others. They provide high-level APIs to application developers, such as an updated topology view, monitoring primitives, hooks for link/node failure events, and other services that make far easier the implementation of networking applications, such as routing, security or virtualization. That is, operators of an SDN network can incrementally add support for new services or optimize existing ones by writing high-level software operating on a data structure, running on top of a network OS, instead of designing and standardizing a complex distributed protocol, finally waiting for equipment vendors to adopt it.

The aftermath of SDN and OpenFlow is a vendor-agnostic architecture that has profoundly changed the market. On one side, switching silicon suppliers are let free to focus on improving performance and resources of their chips, i.e. more throughput and memory. On the other end, third-party companies, or even network owner themselves, who develop the software needed to control these chips. Today one can buy so called "white-box" off-the-shelf switches at a cheaper price compared to that of traditional vendors. These switches are equipped with generic hardware and are shipped without any protocol implementation, one can then decide to run it using a free open source network OS.

1.2 Data plane programmability

Even if some ideas on programmable networking have been researched for nearly two decades [9], it is fair to say that OpenFlow is the technology which brought SDN to the real world. Quoting [9], "*Before OpenFlow, the ideas underlying SDN faced a tension between the vision of fully programmable networks and pragmatism that would enable real-world deployment. OpenFlow struck a balance between these two goals by enabling more functions than earlier route controllers and building on existing switch hardware, through the increasing use of merchant-silicon chipsets in commodity switches*".

The OpenFlow's MAT abstraction proved to be valuable in describing *commonality* between many existing packet forwarding technologies. Switching chips were already designed using dedicated *tables* for tasks such as L2 learning, L3 longest-prefix match (LPM) routing, or access control lists (ACL). As a result, vendors were able to implement an OpenFlow interface on top of their existing switching chips, making their devices marketable for the new SDN market.

However, network switches perform many tasks in addition to standard packet forwarding, such as load balancing, to make a more efficient use of network paths; rate control and active queue management (AQM), to reduce network congestion; scheduling, to provide QoS and fairness; fine-grained measurements, to let network OSs take better decisions on where

to route packets; failure detection, to promptly reroute premium customers' traffic, etc. Differently from packet forwarding, these tasks read and write state maintained at the switch as part of the packet processing that happens in the fast-path. For this reason, we can refer to these tasks as *stateful data plane algorithms*. Today these algorithms are implemented in switching chips using dedicated hardware and cannot be changed by network operators.

Making stateful data plane algorithms programmable in switching chips would bring a number of benefits.

1. **Optimize existing algorithms, research new ones.** The lesson about SDN and OpenFlow teaches us that operators like to have finer control of their network, instead of relying on equipment vendors to listen to their requests to modify existing functions or implement new ones. Moreover, programmable switching chips would allow researchers to experiment new ideas on high-performance hardware, instead of relying on expensive ASIC design, time-consuming FPGA-based implementations, inaccurate and long simulations, or low-performance software-based emulation.

2. **Lower the control burden on the SDN controller.** OpenFlow forces a "two-tiered" programming model: any stateful processing intelligence is delegated to the controller, whereas switches limit to install and enforce stateless packet forwarding rules delivered by the controller. Centralization of the network applications' intelligence turns out to be an advantage for all those applications where changes in the forwarding state do not have strict real time requirements, and depend upon *global network state*. But for applications which rely only on *local* flow or port states, the latency toll imposed by the reliance on an external controller cannot meet the requirements of high-performance networks [10]. Example of such applications are those that forward packets based on the status of a port, e.g. rerouting upon detection of a link that is down or congested, or applications that forward packets differently based on the state of a TCP connection, e.g. stateful ACLs that allow outbound connections, but not inbound. Adding support for programmable states in OpenFlow would allow for a more efficient implementation of these applications, without being affected by the latency and overhead imposed by the controller intervention. Interestingly, data center operators have already reported on the benefits of adopting a MAT-based programming model capable of using connection state as a base primitive, rather than just packets [11].

3. **Offload of middlebox processing functions to the network.** Following the programmability trend, stateful network functions commonly implemented using hardware middleboxes, are being transformed in virtualized software appliances that run on com-

modity servers [12]. Network operators are supporting this trend [13], usually called *Network Function Virtualization* (NFV) [14]. Virtual network functions (VNFs) have a number of advantages when compared to legacy hardware ones. They can be dynamically created, updated, migrated and run on commodity servers. However, developing VNFs on general purpose CPUs is a hard task [15]. Indeed, a commodity server is usually equipped with a couple of 10 Gbit/s network interfaces, and 40 Gbit/s interfaces are becoming common. Unfortunately, current general purpose CPUs speed is not growing as fast as the network interfaces speed [16]. Therefore, a switching chip with support for the programming of custom stateful processing tasks, would permit the offloading of some or all the processing to the server NICs [11] or to the network nodes, thus permitting VNFs to support larger traffic volumes, at a lower cost.

There is no reason why programmability should not be extended to the full spectrum of tasks performed by the data plane, even those requiring to access and modify state at each packet. A major concern is the feasibility of such a programmable *stateful* data plane. Indeed, state-of-the-art fixed-function switching chips are required to operate at line rate, i.e. 10–100 Gbit/s on 10–100 ports, likely more in the future. Programmability should not be introduced at the expenses of performance.

Luckily, silicon manufacturers are already producing programmable switching chips [17, 18, 19] competitive in performance with state-of-the-art fixed-function chips. These chips provide low-level primitives that can be configured with software to add support for arbitrary protocol headers and forwarding actions, i.e. a programmer is not restricted to a subset of L2-L4 headers as in OpenFlow, but she can program the switch to parse new, non-standard headers with arbitrary length and structure; he can also define custom actions to modify the parsed headers.

While these programmable chips greatly improve the flexibility of the forwarding tasks, they ignore or address only marginally the problem of programming stateful data plane algorithms. Some [20, 18] limits *statefulness* to dedicated, i.e. not programmable, functions, such as counters or metering. Others [17, 19], in order to guarantee the forwarding of packets at line rate, offer limited support [11]: have limited memory for state, support limited types of operations, and limit the type of per-packet computation.

The design of a switching chip is usually optimized to contain its area and power consumption. Seminal work on Reconfigurable Match Tables (RMT) [20], while addressing the case of a *stateless* programmable chip, proved an important result: the improved flexibility of the forwarding circuitry comes at an additional cost of less than 15% in chip area and power consumption when compared to a similar fixed-function chip. 80% of chip area is due to

memory (TCAMs and the IO/buffer/queue subsystem), and less than 20% area is due to logic. Hence, as a rule of thumb, if we would like to add more programmable processing logic to execute stateful tasks, that would be unlikely constrained by chip area or power consumption.

Despite the recent advances in programmable switching chips and the promising RMT result, the feasibility of a programmable data plane that is optimized for the programming of stateful data plane algorithms, is still an open question.

1.3 Research questions

In this thesis we address the following research questions:

1. *Can we devise an abstraction for a programmable data plane that is specifically targeted for stateful packet processing?*
2. *Would this abstraction be feasible at line rate?*
3. *Is there a tradeoff between programmability and performance in packet processing? If yes, is there a sweet spot that maximizes both?*
4. *Which new applications would be enabled by such programmable data plane?*

1.4 Summary of contributions and organization of thesis

The remainder of this thesis is organized as follows. In Chapter 2, we present two abstractions to program stateful data plane algorithms: OpenState and Open Packet Processor (OPP). OpenState has been designed as an extension to OpenFlow and allows the description of forwarding rules as finite state machines (FSM) operating on *per-flow* state, maintained by the switch and updated as a consequence of packet-level and timeout events. We show how OpenState can be supported in hardware with minimal extension to an ordinary OpenFlow, TCAM-based, hardware pipeline. However, OpenState allows to describe a very limited set of applications. To solve this problem, we propose OPP. Building upon OpenState, OPP enables more programmability by permitting the description of packet processing tasks using Extended Finite State Machines (EFSM). Relative to OpenState, OPP adds support for the evaluation of conditions on state, arbitrary custom registers, and support for arithmetic and logic operations. We then discuss the hardware feasibility of OPP, which similarly to OpenState, can be realized with commodity switch hardware components.

The design of both OpenState and OPP is characterized by a feedback-loop architecture that might generate state inconsistencies when processing traffic at high-rates. Preventing such inconsistencies via memory locking introduces a tradeoff between programmability and performance. We analyze this tradeoff by running simulations on real traffic traces from both carrier and data center networks. We find that, in most of the cases, the risk of incurring in such inconsistencies is low. We design a scheme to perform memory locking in a packet forwarding pipeline. We show that even long complex operations (up to 30 clock cycles per packet) affect throughput only marginally, at the cost of extra processing latency in the order of hundreds of nanoseconds.

The rest of the thesis describes two *novel* applications of programmable stateful data planes.

In Chapter 3, we present Spider, a data plane scheme that provides (i) detection of failures based on switches' periodic link probing and (ii) fast reroute of traffic flows even in the case of distant failures. Spider is inspired by legacy control plane protocols such as Bidirectional Forwarding Detection (BFD) and MPLS Fast Reroute. Differently from these protocols, Spider performs detection and rerouting entirely in the fast-path. It offers guaranteed, short (few microseconds or less) failure detection and recovery delays, with a configurable tradeoff between probing overhead and failover responsiveness. We provide an analysis on flow tables' memory impact, and experimental results on its performance in terms of recovery latency and packet loss.

In Chapter 4, we present FDPA, a data plane scheme to enforce fair bandwidth sharing of the same link among many TCP-like senders. Most of the mechanisms to solve this problem are based on complex scheduling algorithms, whose feasibility becomes very expensive with today high-throughput requirements. We propose a scheme that does not modify the scheduler. FDPA enforce fairness by dynamically assigning traffic flows to an existing strict priority scheduler, based on the users' rate history, where rate estimation is performed using OPP capabilities. We conducted experiments on a physical 10 Gbit/s testbed with real TCP traffic. Results show that FDPA produces fairness comparable to approaches based on scheduling.

CHAPTER 2 ABSTRACTIONS FOR STATEFUL DATA PLANES

2.1 OpenState

OpenFlow is a *stateless* data plane abstraction. Changes of the forwarding rules do *not* depend on the history of packets seen by the switch, rather, forwarding rules can be update only via the explicit involvement of an external controller. The latter implements stateful processing intelligence, i.e. decides how rules should evolve in time as a consequence of events, either generated at the data plane or external. The explicit involvement of the controller for *any* stateful task and for *any* update of the match+action rules, is problematic as it imposes excessive overheads [10]. Centralization of the stateful intelligence is an advantage for all those applications where changes do not have strict real time requirements, and depend upon global network states. But for applications relying only on local flow or port states, the reliance on an external controller for any update seems a very inefficient design choice. In the worst case, the slow control plane operations a priori prevents the support of network control algorithms which require prompt, real time reconfiguration of the data plane.

To solve this problem, we propose OpenState, an extension to OpenFlow that adds support for *stateful* forwarding behaviors. OpenState is based on the observation that many networking applications can be expressed as switch-local state machines operating on *flow-state*, i.e. state that is unique to each flow, where the definition of a flow is left to the programmer's choice, e.g. all packets belonging to the same IP source-destination address, or the same TCP/UDP 5-tuple. OpenState allows to program in the switch any networking tasks that can be modeled using a formal finite state machine abstraction called "Mealy Machine". While OpenState does not permit to describe *any* stateful data plane algorithm, it is important because it sets the basic requirements for a programmable packet processing architecture to support said stateful behaviors.

OpenState provides a new forwarding API for SDN. The design of such API builds upon the observation that the *very same* OpenFlow MAT primitives can be reused for a different goal and with a broadened semantic. In a nutshell, OpenState permits to i) perform matches on packet header fields plus a flow-state label maintained by the switch, and ii) associate to such match both a forwarding action (or set of actions) and a *state transition*, which determines the flow-state that will be visible to the next packet of the same flow. The immediate result is a forwarding behavior that depends on the history of packets seen by the switch.

The design of OpenState is based on two fundamental principles that follow the design of

OpenFlow itself:

- The abstraction must be amenable to high speed implementation, i.e. 10-100 Gbit/s on 10-100 ports.
- The abstraction must not violate the vendor-agnostic principle which has driven the OpenFlow invention, and that has fostered SDN, i.e. it must not emerge as a low-level technical approach, rather it should leave space to switch vendors to optimize and differentiate their implementations.

2.1.1 Abstraction

An illustrative example: port knocking

A very descriptive example of an application that would benefit of a stateful data plane abstraction is that of a *port knocking* firewall, a well known method for opening a port on a server otherwise inaccessible. A host that wants to establish a connection (say an SSH session, i.e., port 22) delivers a sequence of packets addressed to an ordered list of pre-specified closed ports, say ports 5123, 6234, 7345 and 8456. Once the exact sequence of packets is received, the firewall opens port 22 for the considered host. Before this event, all packets including the knocking ones are dropped.

As any other stateful application, such an operation cannot be configured *inside* an OpenFlow switch, but must be implemented in an external controller. The price to pay is that a potentially large amount of signaling information (in principle up to all packets) must be conveyed to the controller. Moreover, a timely flow-mod command[1] from the controller is needed to open port 22 after a correct knocking sequence, to avoid that the first legitimate SSH packet finds port 22 still closed. Implementing this application in the logically centralized controller brings no gain: it does not benefit from network-wide knowledge or high level security policies, but uses just local states associated to specific flows on a single specific device. It is then reasonable to devise an abstraction to program the port knocking algorithm in the switch itself.

The port knocking behavior can be modeled as follows: *each host* is associated with the FSM illustrated in Figure 2.1. Starting from a `DEFAULT` state, each correctly knocked port will cause a transition to a series of three intermediate states, until a final `OPEN` state is reached. Any knock on a port different from the expected one will bring back to the `DEFAULT` state. When in the `OPEN` state, packets addressed to port 22 (and only to this port) will be

[1] A flow-mod is message in the OpenFlow protocol to add, modify or remove MAT entries.

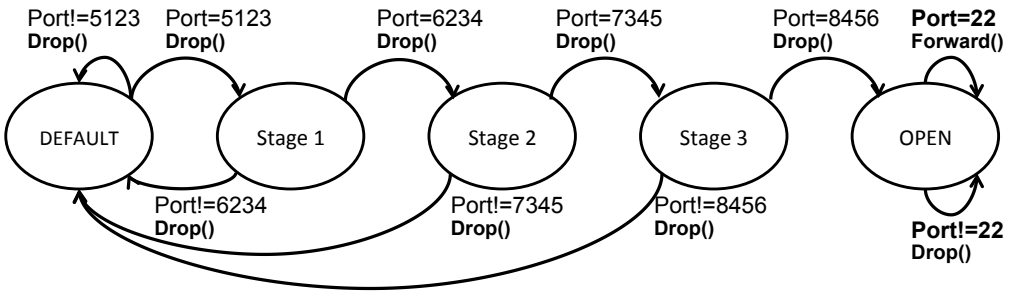


Figure 2.1 FSM of the port knocking example.

forwarded, whereas all remaining packets will be dropped, but without resetting the state to `DEFAULT`.

A closer look at Figure 2.1 reveals that each state transition is caused by an *event*, which specifically consists in a packet *matching* a given port number. Moreover, each state transition caused by a match event, is associated to a forwarding *action* (in the example, drop or forward). A state transition thus reminds very closely an OpenFlow match+action rule.

Mealy Machines

The match which specifies an event not only depends on packet header information, but also depends on the state; using the above port knocking example, a packet with destination port 22 is associated to a forward action when in the `OPEN` state, but to a drop action when in any other state. Moreover, the event not only causes an action, but also a transition to a next state (including self-transitions from a state to itself).

All this can be modeled in an *abstract* form by means of Mealy finite-state Machine, or simply Mealy Machine[2] [21]. Formally, a Mealy Machine is an abstract model comprising a 4-tuple $< S, I, O, T >$ where:

- S is a finite set of states;
- I is a finite set of input symbols (events);
- O is a finite set of output symbols (actions); and
- $T : S \times I \rightarrow S \times O$ is a transition function which maps <state, event> pairs into <next-state, action> pairs.

[2]Classical theory on state machines makes a distinction between a Mealy Machine and a Moore Machine. A Mealy Machine associates outputs with transitions, while a Moore Machine associates outputs with states.

Such an abstract model can be made concrete by restricting the set O of actions to those available in current OpenFlow devices, and by restricting the set I of events to OpenFlow matches on header fields and metadata easily implementable in hardware platforms. The finite set of states S (concretely, state labels, i.e., bit strings), and the relevant state transitions, in essence the "behavior" of a stateful function, are left to the programmer's choice.

In the following we will loosely refer to a Mealy Machine simply as a FSM.

2.1.2 From state machines to pipelines

State management

Matches in OpenFlow are generically collected in flow tables. The discussion carried out so far recommends to *clearly* separate the matches which define *events* (matching on the destination port in the port knocking example) from those which define *flows*, meant as entities which are attributed a state (each host IP address in the port knocking example). Two distinct tables, **State Table** and **FSM table**, and three logical steps thus naturally emerge for handling a packet (Figure 2.2):

1. **State lookup**: It consists in querying the state table using as key the packet header field(s) which identifies the flow, for instance the source IP address; if a state is not found for a queried flow, we can assume that a `DEFAULT` state is returned;
2. **FSM execution**: The retrieved state label, added as metadata to the packet, is used along with the header fields involved in the event matching (e.g., port number), to perform a match on the FSM table, which returns i) the associated actions, and ii) the label of the next state;
3. **State update**: It consists in rewriting or adding a new entry to the state table using the provided next state label.

The example in Figure 2.2 shows how the port knocking example is supported by the proposed approach. The 7 entries in the FSM table implement the port knocking state machine. Assuming the arrival of a packet from host 1.2.3.4; the state lookup (top figure) permits to retrieve the current state, STAGE-3. Via the FSM table (middle figure), we determine that this state, along with the knocked port 8456, triggers a drop action and a state transition to `OPEN` (middle figure). The new state is written (bottom figure) back in the state table for the host entry. In the FSM table, we assume an ordered matching priority, with the last row having the lowest priority. As a result, all the four transitions to the default state for

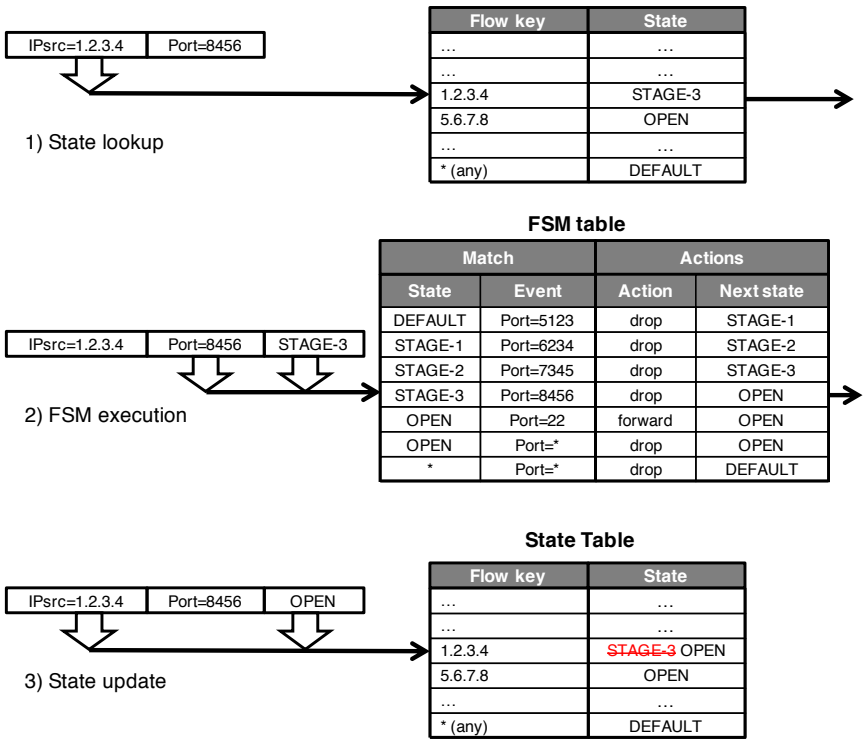


Figure 2.2 State Table, FSM table, and packet handling for the port knocking example.

packets not matching the expected knocked port are coalesced in the last entry. A notable characteristic of the proposed solution is that the length of the tables is proportional to the number of flows (state table) and number of states (FSM table), but *not* to their product.

The above described abstraction still misses a fundamental further step: how to extract the flow key from a packet? To this purpose, we have to conceptually separate the *identity* of the flow to which a state is associated, from the actual *position* in the header field from which such an identity is retrieved. Moreover, as further motivated in Section 2.1.2, we find useful to distinguish between two keys: lookup key, and update key, used respectively to read and write values in the state table. Thus, we need to provide the programmer with the ability to use eventually different header fields in these two accesses to the state table. We define as *lookup-scope* and *update-scope* the ordered sequence of header fields that shall be used to produce the key used to access the state table. In the port knocking example `lookup_scope = update_scope = [ipsrc]`.

To summarize, the basic data plane abstraction introduced so far comprises the specification of:

1. an FSM table comprising four columns: i) a **state**, ii) an **event** expressed as an Open-

Flow ternary match on packet headers, iii) a list of OpenFlow **actions** to apply to the matching packet, and iv) a **next-state** label;

2. a state table comprising two columns: i) a **flow key** and the associated ii) **state** label.

3. the **lookup-scope** and **update-scope** used to extract the flow key to access the state table for, respectively, read and write operations. For simplicity we can refer to a *key scope* as any ordered sequence of pointers to header fields used to extract a key.

OpenState API

In this section we describe the API necessary to program an OpenState switch. We describe also two important additional capabilities that for simplicity we have not mentioned so far: cross-flow state handling, and time-based events.

The API is defined as an extension to OpenFlow, meaning that we can assume all primitives and capabilities assumed by OpenFlow *plus* the OpenState's primitives. The OpenFlow specification refers to a MAT simply as a flow table, and to MAT entries as flow entries. OpenFlow version 1.0 is based on a single flow table, while starting from version 1.1 the switch can support multiple flow tables pipelined. The protocol message used to add, modify and remove flow entries is the flow-mod message. Unless explicitly changed by the remote controller through flow-mod messages, flow tables are *static*, i.e., all packets matched by a given flow entry experience the same forwarding behavior, for this reason we can refer to an OpenFlow flow table as a *stateless* processing block. OpenState introduces the notion of a *stateful* block, as an extension of a *single* OpenFlow table. Stateful blocks can be pipelined with other stateful blocks as well as stateless blocks. In the following we describe the API necessary to program a single stateful block.

Figure 2.3 depicts the packet processing architecture of an OpenState's stateful block.

The FSM table introduced before is implemented using an ordinary OpenFlow flow table, preceded by a state table. Extraction of the key to access the state table is performed by two dedicated key extractors, configured respectively with the lookup-scope and update-scope. The key is extracted concatenating the header fields in the same order as specified in the key scopes. As further explained in Section 2.1.2, the order of the fields is important because it permits to handle the case of bidirectional flows, i.e. in a client/server paradigm, update the state of the flow carrying the reply as a consequence of the behavior of the flow carrying the request. The state table maps keys to state labels, an entry of such table is said state entry. When processing a packet, if a given key is not found in the state table a *default* state label, said also state 0, is returned. The state table has an additional column that

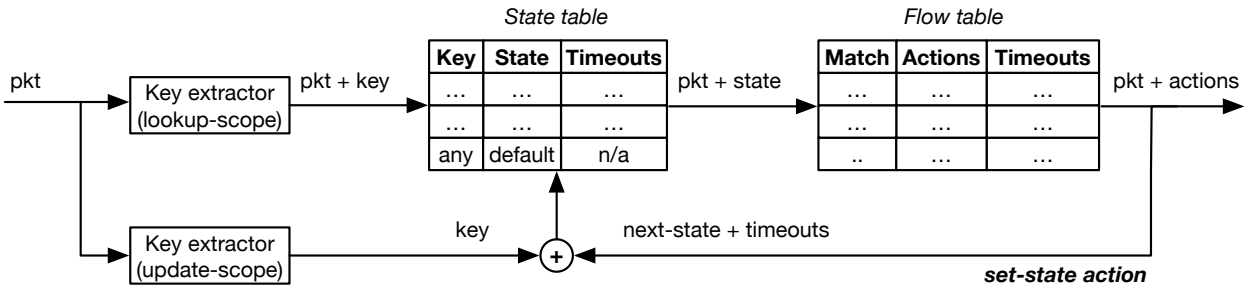


Figure 2.3 Abstract pipeline model of a stateful processing block in OpenState.

defines the timeouts associated with a given state entry. Similarly to flow entries timeout in OpenFlow, state timeouts allow a given state entry to expire after a certain amount of time, moreover they can be used in OpenState to define time-based events to force state transitions independently from packets received. Details of the state timeout mechanism are provided in Section 2.1.2.

After a state label is returned by the state table, the packet is processed by the flow table. Here flow entries can be defined to match both the packet headers (we assume the same headers as in OpenFlow) and on the state label, virtually added to the packet headers as a metadata. As in OpenFlow, the flow table returns the packet and a list of actions, to be applied either immediately or at the end of the pipeline.

The next-state column of the FSM table is embodied by a special action *set-state*. When adding an entry to the flow table, along with the ordinary OpenFlow actions, a programmer can add a set-state action to instruct the switch on how to update the state table when that entry is matched. The set-state action takes as parameters the new state label to write in the state table, and the timeouts associated with it. Differently from other OpenFlow actions, the set-state action is not appended to the action list, but instead it is executed immediately to update the state table, creating a *feedback-loop*.

When a packet triggers a set-state action, the state table is updated using the flow key returned by the key extractor configured with the update-scope, which can produce a key different from the one used in the lookup phase. If a given flow key already exists in the flow table, the state label is rewritten and the new timeouts are set, otherwise, if the flow key is not found, a new state entry is instantiated. It is important for the update key to be extracted *before* the packet is processed by the flow table, otherwise the evaluated key would depend on the result of any OpenFlow header rewrite action. As explained in Section 2.3, knowing beforehand which state entry could be potentially updated, allows us to lock access

to the state table when processing packets in parallel, so to prevent state inconsistencies.

To summarize, the necessary steps to configure a stateful block in OpenState are:

1. **Instantiate the state table,** via configuration of the lookup-scope and update-scope;
2. **Define state transitions,** by inserting flow entries that match the state label and use the set-state action.

After being provisioned, the state table is initially empty. It is then populated based on the set-state actions defined in the flow table.

Figure 2.4 show an example of OpenState configuration for the port knocking example.

Cross-flow state handling

There are a number of stateful processing functions where the history of packets of a given flow is used to determine the state of another flow, this situation can be defined as cross-flow state handling. We provide here an example and relative OpenState configuration of an application based on cross-flow state handling: a L2 learning switch.

Example: L2 learning switch L2 learning is the process implemented by a legacy L2 bridge. In such devices, each time a packet is received the following operations are performed:

1. a *learning table* is update mapping the Ethernet *source* address (for simplicity ethsrc) to the switch input port where the packet was received;
2. the learning table is queried using the Ethernet *destination* address (ethdst), if an entry is found, then the packet is forwarded to the corresponding port (learned before), otherwise the packet is flooded on all ports.

Without loss of generality, we assume the learning table is indexed using only the Ethernet address. In real bridge implementations, a combination of the Ethernet address, the VLAN ID and the switch input port is used to distinguish between VLANs and to avoid loops.

While there are many different OpenFlow-based implementations of a L2 learning switch, more or less optimized, given the stateless nature of the OpenFlow data plane, they are all based on the following reactive approach. The flow table of the OpenFlow switch is initially empty, apart for the *table-miss*[3] that generates a *packet-in*. The controller implements the

[3]Table-miss is the OpenFlow wording for the lowest priority rule that matches all packets otherwise not matched by any other rule.

Port knocking

```
lookup_scope = [ipsrc]
update_scope = [ipsrc]
```

Match	Actions
state=DEFAULT, dport=5123	drop, set_state(STAGE-1)
state=STAGE-1, dport=6234	drop, set_state(STAGE-2)
state=STAGE-2, dport=7345	drop, set_state(STAGE-3)
state=STAGE-3, dport=8456	drop, set_state(OPEN)
state=OPEN, dport=22	forward
state=OPEN	drop
*	drop, set_state(DEFAULT)

Figure 2.4 Example of flow table configuration and key scopes to implement the port knocking example. Flow entries are assumed to be in priority order, where the first entry has maximum priority. "dport" is the TCP/UDP destination port. The generic action "forward" indicates that the packet can be forwarded. In a real OpenFlow-like configuration, a programmer should include routing actions to output packets on specific ports.

learning table. Indeed, for each packet-in the controller updates and queries a local version of the learning table, if an entry is found for the packet-in's ethdst, then the controller installs a flow rule in the switch to forward this and subsequent packets of the same flow over the learned port, otherwise the controller generates a packet-out[4] to flood the packet on all switch ports. Using this approach, the number of flow rules depends on the number of hosts in the network, which can be problematic in some cases, e.g. in a data center with tens of thousands of virtual machines each one with a different Ethernet address.

A more efficient way of implementing a L2 learning switch based on OpenState, that does not involve interacting with the controller, is depicted in Figure 2.5. In this case, state labels are used to codify the location (port) of a given host. The state table is queried looking at the ethdst of the packet (lookup-scope), while it is updated using the ethsrc (update-scope). The entries in the table forward packets according to the state label (flood if default) and update the state based on the packet's input port. This example on purpose assumes compatibility with the current OpenFlow specification, and the N^2+N size of the table (being N the number of switch ports) thus depends on the OpenFlow limitations, and not on our abstraction. Indeed, assuming a platform compatible with RMT [20] where action parameters can be pointers to header fields, this assumption would yield a flow table comprising of only

[4]Packet-out is the OpenFlow wording for the action of sending a data packet from the controller to a switch, encapsulated in an OpenFlow header that specifies the actions to be applied by the switch on the packet, e.g. output on a given port or flood.

L2 learning switch

```
lookup_scope = [ethdst]
update_scope = [ethsrc]
```

Match	Actions
state=DEFAULT, inport=1	flood, set_state(PORT-1)
state=PORT-1, inport=1	output(1), set_state(PORT-1)
state=PORT-2, inport=1	output(2), set_state(PORT-1)
...	
state=PORT-N, inport=1	output(N), set_state(PORT-1)
...	
...	
state=DEFAULT, inport=2	flood, set_state(PORT-2)
state=PORT-1, inport=2	output(1), set_state(PORT-2)
...	
state=PORT-N, inport=2	output(N), set_state(PORT-2)
...	
...	
state=DEFAULT, inport=N	flood, set_state(PORT-N)
state=PORT-1, inport=N	output(1), set_state(PORT-N)
state=PORT-2, inport=N	output(2), set_state(PORT-N)
...	
state=PORT-N, inport=N	output(N), set_state(PORT-N)

Figure 2.5 OpenState configuration to implement a L2 learning switch with N ports; inport indicates the switch input port of the packet, action flood replicates and forward a packet to all switch ports except inport; the state label represents the port number to which the packet shall be forwarded, i.e. the location of a given host.

two entries (Figure 2.6).

With OpenState, the size of the flow table does not depend on the number of host H but on the number of switch ports $N \ll H$ (or just 2 if OpenFlow would support pointers to header fields as action parameters). Instead the size of the state table depends on H. As we will explain in Section 2.1.3, given the nature of the state table, i.e. exact match on the key, this can be implemented in a cheap RAM-based hash table, opposed to the expensive (in terms of chip area and power consumption) TCAM-based flow table, necessary to support OpenFlow ternary matches. Such consideration makes the OpenState-based L2 learning switch implementation scale easier than the OpenFlow one based on the reactive-approach.

Finally, to provide a further example of the generality of OpenState, if one would like to implement bridging based on VLANs, it would suffice to set the key scope as `lookup_scope`

Simplified L2 learning switch

```
lookup_scope = [ethdst]
update_scope = [ethsrc]
```

Match	Actions
state=DEFAULT	flood, set_state($inport)
state=*	output($state), set_state($inport)

Figure 2.6 Simplified configuration of the example in Figure 2.5. `$inport` and `$state` are used to point at the value contained in the packet header fields, respectively the switch input port and the state label.

`= [vlanid, ethdst]`, and `update_scope = [vlanid, ethsrc]`.

Time-based events

OpenState defines timeouts for the entries of the state table, useful to specify time-based events and transitions in a FSM, e.g. *"move to state B after X time in state A"*, or *"set the state of the flow to `IDLE` if no packets are received for X time"*. So far, the only type of events that can trigger a state transition are packet-based events, i.e. match on an incoming packet. However, some applications would benefit from defining events based on time, regardless if a packet is received or not, i.e. in a stateful firewall, set the state of a TCP connection to `CLOSED` after no packets are received for a given amount of time. Similarly, timeouts could be used to flush unused state entries, to limit the size of the state table.

Analogue to OpenFlow's flow entries, the state table is designed such that each entry has an idle-timeout and a hard-timeout associated with it. Moreover, both timeouts are associated with an *expired state*, respectively idle-expired-state and hard-expired-state, to be applied when a state timeout expires. To summarize, a row of the state table would look like this:

key	state	idle-timeout	hard-timeout	idle-expired-state	hard-expired-state

Timeouts are set the same way state entries are added (updated) to the state table, by means of a set-state action. We can assume state timeouts are not set unless explicitly declared in the set-state action, i.e. the entry will never expire.

Hard-timeouts cause the state entry to be updated to the hard-expired-state after a given interval since the entry was last updated, regardless of how many packets it has matched. Idle-timeouts cause the state entry to be updated to the idle-expired-state when it has matched no packets in a given interval, i.e. the state entry was idle. When a state entry is expired because of a timeout, and its value updated with the expired state, it remains in that state

until a new set-state action is performed. If the expired state is the `DEFAULT` state, then the entry is removed from the state table. The reader should remember that all flows are virtually in state `DEFAULT`, even if a state entry is not present in the state table. Hence, removing an entry is equivalent to setting its state label to `DEFAULT`.

State timeouts are useful to define events at the granularity of packet time-scales, i.e. short time scales comparable to the interval between two packets of the same flow. Hence, a switch target implementing OpenState should provide a timeout granularity in the order of microseconds or less. This is easy, considering that current switch targets already expose a clock interface operating at finer frequencies, i.e. 0.1-1 GHz.

The application presented in Chapter 3 makes extensive use of state timeouts. However, it is useful to present another programming example of OpenState: flowlet-based path load balancing.

Example: flowlet-based path load balancing. Load balancing traffic over multiple paths (also known as load sharing) is an important feature that allows flexible and efficient allocation of network resources. Given multiple paths that a switch can use to forward a packet, legacy load balancing schemes such as ECMP selects an output port hashing the transport layer 5-tuple[5] modulo the number of paths, thus splitting load across multiple paths. Hashing is a convenient approach as it guarantees consistency of the forwarding decision, i.e. packets with the same 5-tuple will be forwarded to the same output port, and hashing is a stateless function that can be implemented very efficiently in hardware. Different hashing schemes exist, each one with its associated tradeoffs [22], however, a key limitation of hash-based schemes is that two or more large, long-lived flows (usually referred in the literature as "elephant" flows) can collide on their hash and end up on the same output port, creating a bottleneck.

The ideal solution would be to split traffic of the same flow at the packet level, instead of pinning the whole flow to a static path choice. However, sending two packets of the same flow over two different paths with different delays might generate packet reordering at the receiver, which can cause unnecessary throughput degradation when using TCP.

Flowlet-based switching is a technique first introduced in [23], that suggests splitting traffic at the level of bursts of packets, called flowlets. Switching bursts, instead of packets, minimizes the risk of packet reordering. The approach is based on the fact that, if the idle time between

[5] A hash function is applied to the combination of 5 header fields: IP source address, IP destination address, IP protocol, source port, and destination port. Packets of the same transport layer flow will have the same hash value.

Flowlet-based path load balancing

```
lookup_scope = [ipsrc,ipdst,ipproto,sport,dport]
update_scope = [ipsrc,ipdst,ipproto,sport,dport]
```

Flow table

Match	Actions
state=DEFAULT	group(1)
state=1	output(1)
state=2	output(2)
...	...
state=N	output(N)

Group table

Group ID	Type	Action buckets
1	SELECT	<set_state(1, idleto=D), output(1)>, <set_state(2, idleto=D), output(2)>, ... <set_state(N, idleto=D), output(N)>,

Figure 2.7 OpenState configuration to implement a flowlet-based load balancer.

two successive packets is larger than the delay difference between two parallel paths, one can route the two packets on different paths without causing reordering at the receiver. The main origin of flowlets is the burstiness of TCP, where idle time at RTT and sub-RTT scales can be observed.

By using flow states and associated timeouts, OpenState allows a programmer to program a flowlet-based load balancer (Figure 2.7). Key scopes distinguish state between different TCP flows. For each incoming packet of a new TCP connection, a `DEFAULT` state is returned by the state table, causing a group table to be invoked. OpenFlow's group table are used to randomly choose (`SELECT`[6]) an output port among N available. Here the group table picks a random action bucket, forwarding the packet on one of the N available ports, and updating the state accordingly. Subsequent packets will be forwarded using the value returned from the state table, not anymore `DEFAULT`. By setting the state idle-timeout $idleto = D$ we can define the lifetime of the forwarding decision. For example, with $D = 10$ seconds the flow-state will be maintained only if a packet from that flow is seen at least once every 10 seconds.

[6]Starting from OpenFlow 1.1, the `SELECT` group type has been introduced to support load sharing over multiple ports. Citing the latest OpenFlow specification *"Packets are processed by a single bucket in the group, based on a switch-computed selection algorithm (e.g. hash on some user-configured tuple or simple round robin). All configuration and state for the selection algorithm is external to OpenFlow."*. In our example we would need the switch to implement a selection scheme other than hashing, preferably a randomized one.

For the sake of path load balancing, we can safely assume that an idle interval of 10 seconds represents the end of an instance of a TCP flow, hence this setting would be equivalent to ECMP. By setting D to a value smaller than the average idle time between flowlets, a new forwarding decision will be taken for each flowlet, hence implementing flowlet-based load balancing.

2.1.3 Hardware feasibility

In this section we provide an overview of the feasibility of a switching chip supporting OpenState. A prototype implementation of OpenState on a FPGA-based platform was presented in [24], however, such a prototype is not a contribution of this thesis. Nevertheless, it is important to point out the details of this implementation as they prove the applicability and limitations of the design choices we devised so far. The OpenState hardware design presented here has been developed incrementally starting from a typical OpenFlow pipeline design.

Background: pipelined architectures. The implementation of a high performance packet processing chip, i.e. 10-100 Gbit/s on 10-100 ports, requires processing packets in parallel. The reason is simple. A typical requirement for a switch with terabit throughput is that of being able to process 1 packet per nanosecond, which translates to 1 billion packets per second. Assuming that the switch needs to perform only 1 operation on each packet, e.g. lookup the destination address in a learning table, and that this operation can be completed in 1 cycle of the chip clock, then building a chip with operating frequency of 1 GHz would be enough to support such line rate. However, switches today, both legacy and SDN, need to perform many operations on each packet, for example push/pop of a VLAN header, lookup up a L2 learning table, LPM lookup on the IP address, ACL, encapsulation, etc. Let us assume we need to perform 10 such operations, unfortunately it is hard and extremely expensive to build a 10 GHz processor, instead, these 10 operations can be performed in parallel using a pipelined architecture.

In such architecture, each stage of the pipeline performs one or few operations, and all the stages are executed in parallel. At each tick of the hardware's clock, packets are moved to the next pipeline's stage, the packet in the last stage exits the pipeline and a new packet enters in the first stage. While the data plane emits one packet per clock cycle, it is in fact processing a number of packets in parallel. The length of the pipeline finally defines the number of packets actually processed at the same time, in a given clock cycle. Clearly, the longer the pipeline, the longer the time spent by the packet into the switch.

Extending the TCAM-based OpenFlow pipeline. In an OpenFlow switch, one flow table represents one stage of the pipeline. Flow tables in OpenFlow implement ternary matching capabilities, i.e. the possibility to match only on a subset of the header bits, to support for example IP prefix matching and ACLs. For this reason, flow tables are usually implemented using a special type of content-addressable memories (CAM), called Ternary CAM (TCAM). Standard CAMs use data search words, i.e. a packet's header fields, consisting entirely of 1s and 0s. TCAMs, instead allow a third matching state of "don't care" for one or more bits in the stored search word, thus adding flexibility[7] to the search. More importantly, given a packet header, TCAMs are able to find a matching entry in only 1 clock cycle.

In order to support OpenState, the OpenFlow TCAM must be extended with two other logical stages: key extractors, and state table (Figure 2.8). In our FPGA prototype, such a pipeline spans 5 clock cycles, meaning that up to 5 packets can be found traveling at the same time. The state table and the flow table spans each one 2 clock cycles, meaning that those stages can be seen recursively as a pipeline of 2 stages each. For example, for the flow table 1 cycle is due to the TCAM match and the other for the lookup of the corresponding actions to apply, stored in a RAM block.

A major concern of such a pipeline architecture is due to the feedback-loop required to write back the state in the state table after a packet has finished processing in the flow table. Such a loop can create inconsistencies when processing packets of the same flow back-to-back, i.e. traveling one immediately after the other in the pipeline, as the second one might read the state entry before it can be updated by the first one. We discuss the risk of such inconsistencies and we propose methods to prevent them in Section 2.3.

Key extractors. This stage extracts both the lookup key and the update key. This is a fairly simple and standard operation performed by many switching architectures. A trivial implementation is that of applying consecutive shift-and-mask operations to extract and concatenate portions of the header. Given the stateless nature of this operation, the complexity of this block does not affect the scalability of the system. Indeed, the case of complex keys requiring numerous shift-and-mask iterations over the packet headers, can be handled with a dedicated pipeline spanning many clock cycles.

[7] As a side note, the added flexibility of TCAMs comes at a cost. TCAM's chip area is typically six to seven times that of an equivalent bitcount SRAM [20]. Hence, TCAMs are an expensive resource that should be dimensioned carefully in a switch.

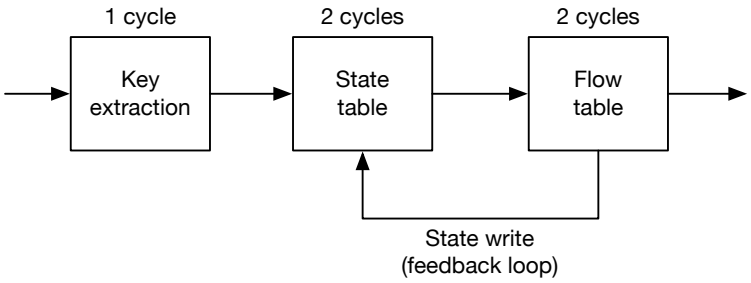


Figure 2.8 Logical pipeline of a stateful block in OpenState

State table. The state table must be able to uniquely map a key to a state entry. If an entry is found, then the corresponding timeouts must be evaluated, and the resulting state label returned (either the original or the expired). If no entry is found, then the state table should return a `DEFAULT` state. Moreover, the state table should permit the creation or update of existing entries as a consequence of a set-state action.

Uniquely mapping keys to values in constant time is not a trivial task. Assuming a 112 bits key (the length of a TCP/UDP 5-tuple), we can think of a memory with 2^{112} unique addresses, however such an addressing space is hardly feasible and inefficient with current hardware technology. Instead, we can use a hash table to map keys to a restricted space. Hash tables have the nice property of providing constant time $O(1)$ for read operations, however one should look out for the possibility of two keys colliding onto the same hash. In the case of a state table this would mean an inconsistent state for some flows. As such, we need to provide a mechanism to avoid collisions. For this reason, in the FPGA prototype, the state table is implemented by using a d-left hash table[25], with $d = 4$ sized for 4K entries. The choice of a d-left allows for constant, deterministic access times for both read and write operations with almost-zero risk of collisions and with around 70% memory efficiency. Other perfect hashing schemes that offers better memory efficiency exists. For example the well known cuckoo hashing [26] scheme attains around 99% efficiency, however, insertion time of these structure is usually not deterministic has they require multiple iterations to rearrange elements in case of collision.

Finally, in order to provide a non-blocking behavior of the hash table, the RAM blocks comprising the d-left table are configured to work as dual port memories, with one write port and one read port. This choice permits updating a state entry without blocking the lookup of another one. If the same entry of the RAM is concurrently read and written, the block RAM is configured to provide the new value to the read port.

State timeouts. Instantiating a timer thread for each state entry can be extremely resource expensive for a switch, instead timeouts should be evaluated as needed, i.e. each time a packet hits a given state entry. As such, a simple and efficient way for a switch to expire state entries is that of taking note of the time t_0 when each state entry was updated (or added). When a packet hits a given entry, the switch can compare the packet arrival timestamp with t_0, if this interval is greater than the idle or hard timeout, then the state table should return the expired state corresponding to the timeout that expired before. The complexity of this structure, shared by the whole state table, is that of few comparators, i.e. negligible complexity that does not represent a risk for the scalability of the system.

On supporting multiple update keys. There are use cases of OpenState in which a set-state can be directed to a key or another, depending on the state or the match event [27]. It could be possible to further generalize the API by allowing for more than one update key extractors, and having the set-state action pick one of them using a runtime parameter. After this time working on the design of OpenState, it is not clear what could be a reasonable number of key extractors to instantiate, a preliminary analysis of some use cases suggests it could be limited to only two update key extractors. However we do not exclude that there might be cases that require more. For simplicity and without loss of generality, we assume only one update key extractor.

2.1.4 Remarks

OpenState departs minimally from the well-known and successful OpenFlow abstraction. Indeed by adding a state table, two key extractors, and a set-state action[8], it is possible to deliver a level of flexibility in the switch, namely the configuration of stateful forwarding behaviors, which goes well beyond what is currently supported in today's OpenFlow switches. Moreover, OpenState can be readily implemented with today's commodity switch hardware components, i.e. simply prepending the ordinary OpenFlow TCAM with a hash table and two key extractors. The tricky part is the feedback-loop, which, if not controlled, can cause erroneous forwarding behaviors. We postpone this discussion to Section 2.3.

Limitations of OpenState

The main limitation of OpenState is that the set of stateful functions that can be programmed by means of a Mealy Machine, is relatively small. As a rule of thumb, OpenState fits well for those tasks that can be described with a reasonably small number of states and associated

[8]The state label can be considered a metadata header field, already supported by OpenFlow.

transitions, those that can be contained in a TCAM. There will always be functions that cannot be described, or which implementation would be inefficient if using a Mealy Machine.

A straightforward example is the following: consider a function that samples packets witch constant rate 1:1000, where sampled packets are sent to the controller for further inspection. Such a function can be described with a Mealy Machine with $|S| = 1000$ (number of state labels), where each packet triggers a state transition to the next state, i.e. increments the state, and when in state `1000` packets are sent to the controller and the state reset to `0`. This approach would require at least 1000 entries in the flow table. Clearly, such implementation is extremely inefficient due to the inherent cost of a TCAM supporting 1000 entries. It would be more efficient if the abstraction would allow for simple arithmetic computations on the state, e.g. integer sum to increment the value of a variable *count*. We could then have only two entries in the flow table: the first matching on $count < 1000$, the second matching on $count \geq 1000$.

The next data plane abstraction, Open Packet Processor, aims at fixing this limitation.

2.2 Open Packet Processor

Open Packet Processor (OPP) is an abstraction for stateful data plane processing that builds upon OpenState. OPP heavily generalize the OpenState concept allowing the definition of a configurable "flow context" in which several per-flow variables can be stored (opposed to a single state label), it extends the decision regarding the FSM transitions using logical/arithmetic conditions, and is able provide means for flexible state updates using Arithmetic and Logic Units (ALU) to execute elementary operations.

The design of OPP is based on the same design principles of OpenState, and the following properties:

- **Ability to process stateful information directly on the fast path**, i.e., similarly to OpenState, while the packet is traveling in the pipeline (nanoseconds time scale). The requirement of performing packet processing tasks in a deterministic, small, bounded number of hardware clock cycles hardly copes with the possibility to employ a standard CPU, thus calling for the definition of a domain-specific computing architecture from scratch. The abstraction should provide means to (i) evaluate metrics relevant in traffic control applications and (ii) describe and evaluate conditions on these metrics, e.g. the average flow rate is above a threshold, or the time elapsed since the last seen packet is greater than the average inter-arrival time. All this should happen entirely in the fast-path, without impacting the requirement of a high-performance data

plane.

- **Target independence**. The abstraction should serve as a general framework to perform advanced state management, i.e. provide a structured way to read, modify, and write data plane state. However, the design of the abstraction should not be tied to a finite set of packet actions. As it will be clear in the next section, applications will be ultimately enabled by the computational primitives exposed by the hardware target, e.g. the ability of the ALU to perform simple integer sums, rather than floating point operations. Each target can provide its own finite set of primitives, OPP provides a framework in which these capabilities can be used in a programmatic way to describe stateful data plane algorithms. Similarly, the abstraction should not make any assumption on the target capability to parse and modify packet headers, rather OPP should assume a target capable of performing flexible header parsing and manipulation as in modern programmable switching chips [19, 18].

2.2.1 Abstraction

Extended Finite State Machines

The EFSM model permits to formalize complex behavioral models, involving per-flow states, per-flow registers, conditions, state transitions, arithmetic and logic operations. Still, this model does not require us to know *what* primitives are available in the hardware platform and *how* such primitives are concretely implemented, but just permits us to combine them together so as to formalize a desired behavior. Hence, it can be *ported* across platforms which support a same set of primitives.

The model is formally specified by means of a 7-tuple $< I, O, S, D, F, U, T >$:

- I: the set of input symbols, i.e. all possible matches on packet header fields;
- O: the set of output symbols, i.e. actions that can be applied to a packet;
- S: the set of *symbolic* state labels, i.e. application specific states, defined by a programmer. Technically state labels are handled as a bit string of arbitrary length. Differently from Mealy Machines, the state label here is said to be symbolic as the actual state of the system depends also on a set of registers D;
- D: n-dimensional linear space $D_1 \times \cdots \times D_n$, i.e. all possible settings of n per-flow registers, i.e. variables. The state of each flow is then completely described by the value of the state label and its registers. Each register is a bit string of arbitrary length.

- F: set of enabling functions $f_i : D \rightarrow \{0, 1\}$, i.e. conditions (boolean predicates) evaluated on registers;
- U: set of update functions $u_i : D \rightarrow D$, i.e. arithmetic instructions to update registers' content;
- T: transition relation $T : S \times F \times I \rightarrow S \times U \times O$, i.e. transitions of the EFSM mapping $<$ state label, conditions, match event $>$ to $<$ next-state, register update functions, actions $>$.

With respect to Mealy Machines, EFSMs bring additional programming flexibility. Indeed, each flow is associated to a state label and n registers, while events are characterized by match on packet headers, state label and conditions evaluated on the registers. EFSM transitions describe the next state label, a new value for the registers (expressed as a function over the old value) and a set of actions to be applied to the packet.

The set of possible match I, packet actions O, update instructions U are provided by the hardware target, while state labels S, registers D, conditions F and transitions T are left to programmers' choice.

2.2.2 Pipeline model

Analogue to OpenState, OPP defines a pipeline of processing stages, either stateless, i.e. ordinary OpenFlow's flow table, or stateful, where the ordinary flow table is extended to provide means to program EFSMs. OPP assumes packets headers are already parsed when passed to a stage, therefore, OPP can leverage related work on programmable packet parsing and reconfigurable match tables [28, 20]. Figure 2.9 depicts the architecture of an OPP processing stage. The pipeline is an extension of the OpenState pipeline: two new blocks, namely *condition block* and *update logic block* are added to the OpenState pipeline comprising the *key extractors*, the state table, now called *flow context table*, and the flow table, now called *EFSM table*.

Flow context. First, when a packet enters the stage, a lookup key extractor builds a key that uniquely identifies a flow context for such packet. Analogue to OpenState, the extractor is programmed at runtime by specifying a list of relevant header fields and packet's metadata. The key is then used to extract a flow context from a *flow context table*[9]. The context includes a state label s and an array of flow registers $\vec{R} = \{r_0, r_1, ..., r_{(k-1)}\}$. If no context is found for a given key, a *default* context is used (i.e., with `DEFAULT` state and all registers set to 0).

[9] In somewhat analogy with context switching in ordinary operating systems.

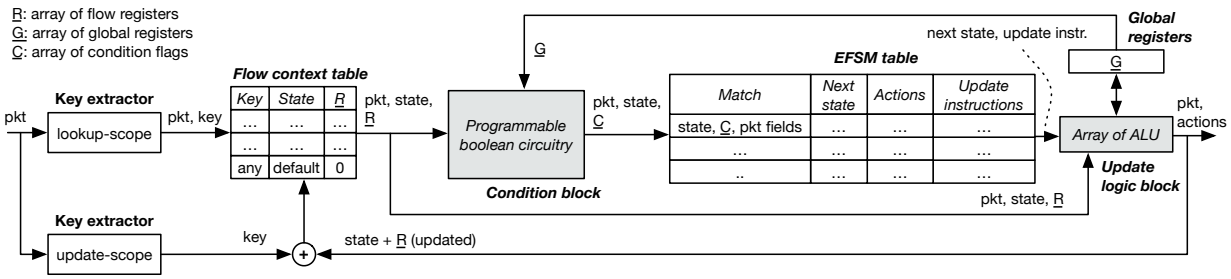


Figure 2.9 Abstract pipeline model of an OPP processing block.

Conditions. The packet's headers and metadata, which now include the flow context, are passed to a *condition block*. The condition block can be programmed at runtime for the evaluation of up to m expressions with mathematical operators like $>$, $<$, $=$. For example, conditions can be used to compare if a port number is greater than the value stored in a given flow context's register. Conditions can also be evaluated over an array of global registers $\vec{G}$, described later, which is shared by all flows. The output of the condition block is a Boolean vector $\vec{C} = \{c_0, c_1, ..., c_{(m-1)}\}$, where $c_i = 1$ if the i-th condition is true, otherwise $c_i = 0$.

EFSM table. The packet's headers and metadata, plus $\vec{C}$ and the state label s, are passed to the *EFSM table*. As the name recalls, this table describes the transitions of the state machine. Such table is a typical MAT that supports ternary matching on the just mentioned values. For each entry in the table, a programmer can specify (i) a list of OpenFlow-like actions to be executed on the packet, (ii) the next state label s in which the flow context shall be set, and (iii) a list of instructions to update the flow context registers $\vec{R}$.

Update logic block. The packet's headers and metadata, the update instructions and the new value of the state label (next state) are passed to the *update logic block*. Here, an array of ALUs performs the required update instructions to update the values stored in the flow context registers $\vec{R}$, using arithmetic functions. Such functions can range from simple integer sums, for instance to update the value of a register representing a packet or byte counter, to more complex ones, e.g., floating point processing, depending on the specific implementation and required performance. The result of this block is then used to update the flow context identified by the key generated by the *update key extractor*.

Global registers. The stage also maintains a vector $\vec{G} = \{G_0, G_1, ..., G_{(h-1)}\}$ of global registers. Differently from the flow context registers $\vec{R}$, $\vec{G}$ can be accessed by all flows *within*

the same OPP stage. These variables can be used in the condition block and their value can be updated by the instructions executed by the update logic block. As we will discuss in Section 2.3, since the variables $\vec{G}$ are global, to avoid inconsistencies, their read and update operations must happen atomically, i.e. without pipeline feedback-loop.

2.2.3 Programming examples

Long-lived flows detection A first trivial example is that of a simple mechanism which distinguishes short-lived flows from long-lived flows by considering "long" any flow that has transmitted at least N packets, and applies different DSCP tags. In this case, an OPP programmer would need to (i) define two states, `DEFAULT`, and `LONG`, (ii) one per-flow register r_0 to count packets, (iii) one global register g_0, storing the constant threshold N, (iv) a condition $c_0 = r_0 > g_0$ applicable when in state `DEFAULT`, and (v) an update function $r_0 \leftarrow r_0 + 1$ to increment the value of r_0.

Figure 2.10 shows the configuration of an OPP stage implementing such algorithm. In this example, we are assuming the target provides capabilities to (i) parse and increase the DSCP field of a packet, and (ii) perform integer sum in the ALU. For simplicity, we omit the actions necessary to forward a packet to an output port. We can imagine those being defined in a subsequent stateless OpenFlow-like stage.

Flowlet-based path load balancing. Figure 2.11 depicts an OPP configuration equivalent to the example of the flowlet-based path load balancer introduced in Section 2.1.2.

Support of this algorithm in OPP requires, for each packet being transmitted, to update a per-flow register r_0 with the quantity $pkt.t + D$, being $pkt.t$ the actual packet timestamp of the packet and D a constant value, stored in g_0, representing the average idle time between bursts. When a packet arrives, we check the condition $pkt.t > r_0$. If true, meaning that the packet is the first of a new burst, we compute a new pseudo-random choice as $pkt.t \ \% \ 2$ (modulo) and store it to r_1, the same port choice is used to forward the packet. If $pkt.t > r_0$ is false, meaning that the packet belongs to an existing burst, we route the packet using the previously assigned r_1. In this case we do not make use of any state label to take forwarding decisions, after the first packet, the state is set to a dummy value `ACTIVE`.

The assumptions of this example are that the hardware target can expose the packet timestamp as a metadata, and that it provides ALU capabilities to perform and integer sum, and a modulo operation.

Remarkably, OPP is able to generalize not only OpenState, but also OpenFlow primitives.

Long-lived flows detection

```
lookup_scope = [ipsrc,ipdst,ipproto,sport,dport]
update_scope = [ipsrc,ipdst,ipproto,sport,dport]
```

Conditions

c0	r0 < g0

Match	Next state	Update instr.	Actions
state=DEFAULT, c0=0	DEFAULT	r0 ← r0 + 1	set_dscp(EF), forward
state=DEFAULT, c0=1	LONG		set_dscp(AF)
state=LONG	LONG		set_dscp(AF)

Figure 2.10 OPP configuration to implement detection of long-lived flows. AF and EF represent DSCP standard DiffServ per-hob behaviors, respectively, "assured forwarding" and "expedited forwarding".

Flowlet-based path load balancing

```
lookup_scope = [ipsrc,ipdst,ipproto,sport,dport]
update_scope = [ipsrc,ipdst,ipproto,sport,dport]
```

Conditions

c0	pkt.t > r0

Match	Next state	Update instr.	Actions
c0=1	ACTIVE	r0 ← pkt.t + g0 r1 ← pkt.t % 2	output(r1)
c0=0	ACTIVE	r0 ← pkt.t + g0	output(r1)

Figure 2.11 OPP configuration to implement a flowlet-based load balancer.

This is made clear from the fact that the OPP configuration does not require structures such as the OpenFlow's group table for port selection, or state timeouts. An equivalent behavior is entirely described by means of an EFSM. Random port selection is described by means of a modulo operation over the packet timestamp, while state timeout expirations are described natively using a register and a condition.

Dynamic priority queuing based on rate estimation. An intuitive way of providing minimum rate guarantees to different users sharing the same link is that of prioritizing their traffic based on the rate history. If a user is sending at a rate higher than the guaranteed threshold, then he should be put in a lower priority queue, and vice versa. In Chapter 4, we discuss more in details the approach and provide experimental results with real TCP traffic.

Priority queuing based on rate estimation

```
lookup_scope = [ipsrc]
update_scope = [ipsrc]
```

Conditions

c0	pkt.t > r4
c1	r3 > g1

Match	Next state	Update instr.	Actions
c0=0, c1=0	ACTIVE	r0 ← r0 + pkt.bytes	set_prio(high)
c0=1, c1=0	ACTIVE	r1 ← pkt.t - r2 r3 ← r0 ÷ r1 r0 ← pkt.bytes r2 ← pkt.t r4 ← pkt.t + g0	set_prio(high)
c0=0, c1=1	ACTIVE	r0 ← r0 + pkt.bytes	set_prio(low)
c0=1, c1=1	ACTIVE	r1 ← pkt.t - r2 r3 ← r0 ÷ r1 r0 ← pkt.bytes r2 ← pkt.t r4 ← pkt.t + g0	set_prio(low)

Figure 2.12 OPP configuration to implement priority queuing based on rate estimation.

We present here an OPP configuration to implement such algorithm (Figure 2.12). At high level, we need to find a way to estimate the average arrival rate of an user at regular intervals, e.g. 1 second. To this purpose, we define two conditions, $c_0 = pkt.t > r_4$, where $pkt.t$ is the packet timestamp and r_4 is the time after which the rate should be estimated; $c_1 = r_3 > g_1$ where r_3 is the estimated rate and g_1 is the guaranteed rate threshold.

When c_0 is false, for each packet, we add the size of the packet ($pkt.bytes$) to r_0; when c_0 is true, i.e. it is time to evaluate the rate, the rate (r_3) is evaluated over the interval that goes from the packet timestamp ($pkt.t$) to the last time the rate was measured ($r2$), after which r_0 is reset, r_2 is set to the packet timestamp, i.e. now, and r_4 is set to now plus the estimation interval, i.e. the next time the rate should be estimated. These two entries (when c_0 is false or true) are repeated for the values of c_1, for a total of 4 entries in the EFSM table. If c_1 is true, the flow is put in a low priority queue, otherwise in high priority.

In this example we are assuming that the hardware target allows for priority queuing, i.e. a priority scheduler with at least two queues, and ALU capabilities to perform an integer sum (assume time in microseconds or milliseconds), an integer subtraction and an integer division operation. Moreover, we assume up to 2 operations can be pipelined, as the result of $r_3 =$

$r_0 \div r_1$ depends on the result of $r_1 = pkt.t - r_2$, which rewritten becomes $r_3 = r_0 \div (pkt.t - r_2)$

2.2.4 Hardware feasibility

A prototype hardware implementation of OPP has been described in [29], however, such a prototype is not a contribution of this thesis. Instead, in this section we discuss issues related to the feasibility and limitations of a switching chip supporting OPP, based on the experience of developing and using such prototype.

Prototype overview. The prototype is based on the NetFPGA-SUME [30] board. It can forward packets of 4 10GbE ports at line rate, providing a throughput of 156.25 Mpps (Million packets per second) when forwarding minimum size packets (64 bytes). The FPGA is clocked at 156.25 MHz, with a 64 bits data path from the Ethernet ports, corresponding to a 10 Gbit/s throughput per port. Analogue to OpenState, the hash table used to store the flow context has been implemented using a d-left table with $d = 4$, and based on RAM blocks. The flow context table can host up to 4K entries. As in OpenState, the RAM blocks are realized as dual port RAM, so as to provide a read and a write operation for each clock cycle. The EFSM table is instead implemented using a very small TCAM of 32 entries of 160 bits. Indeed, TCAM implementation over FPGAs is very inefficient and is currently an open research issue [31, 32, 33]. Still, the reader should recall that such TCAM has not to be used as an ordinary OpenFlow's flow table, but rather to describe the transitions of the EFSM. That is, its size should not depend on the number of flows, but on the number of EFSM transitions of a given algorithm. Thus it is reasonable to keep the size of this TCAM small.

The prototype is configured with the parameters shown in Table 2.1. The configuration of the ALU requires a special mention and will be described more in details later.

The whole system has been synthesized using the standard Xilinx design flow. Table 2.2 reports the logic (LUT) and memory resources (RAM) (in terms of absolute numbers and fraction of available FPGA resources) used by the OPP FPGA implementation, and compare these results with those required for the NetFPGA-SUME single-stage reference L2 learning switch [34] and a OpenState stage. The reader should recall that OPP is an evolution of the original OpenState stage. The synthesis results confirm the trend already shown by RMT [20]: the hardware area is dominated by memory, while adding intelligence/features in the logic requires a small silicon overhead. The OPP additional logic w.r.t. OpenState, i.e. condition and update blocks, uses a small fraction of the total area, of only 2% increment. Notice that the reported resources include the overhead of several blocks not described here,

Parameter	Value	Description
k	4 × 32 bit	Number of flow context's registers. Each register is 32bit long.
m	8	Each condition is in the form $var1$ op $var2$, with operand being one of $>$, $\geq$, $<$, $\leq$, or $==$, and variables being packet header's fields, registers (either flow or global) or constants.
n	32 bit	Size of packet's metadata moved between stages
h	8 × 32 bit	Number or global registers. Each register is 32 bit long.
ALUs	5	Number ALUs. Each ALU performs an operation in the form $res = var1\ op\ var2$. res and vars can be registers or packet fields (including metadata). op can be one of $+$, $-$, $shift$, etc. ALUs support also more advanced traffic control functions such a $ewma$, var, avg.

Table 2.1 Parameters of the OPP hardware prototype.

Resource type	Reference switch	OpenState	OPP switch
# Slice LUTs	49436 (11%)	62637 (14%)	71712 (16%)
# Block RAMs	194 (13%)	245 (16%)	393 (26%)

Table 2.2 Hardware cost comparison of OPP, NetFPGA SUME ref. switch and OpenState.

such as the micro-controller for the OPP configuration at runtime, the input/output FIFO queues for the 10GbE interfaces, which are required to operate the FPGA and do not need to be replicated for each stage. In fact, given the required resources, a NetFPGA-SUME can currently host a pipeline of up to 6 stateful OPP stages.

Pipeline feedback-loop Similarly to OpenState, the pipelined nature of OPP calls for a feedback-loop. In the FPGA prototype, such loop spans exactly 6 clock cycles, from when the flow context is read until it is written back, after being processed by the update logic block. Such loop requires 2 clock cycles more than OpenState, mostly due to processing performed by the update logic block. As already outlined, such a loop can create inconsistencies when processing packets of the same flow back-to-back, i.e. traveling one immediately after the other in the pipeline, as the second one might read the flow context entry before it can be updated by the first packet. Intuitively, the risk of such inconsistencies increase with longer loops. An evaluation of the risk and performance of such feedback loop is presented in Section 2.3.

Update logic block configuration. The update logic block is the OPP component in charge of providing computational primitives involving arithmetic processing. The capabilities offered by this block are not tied to the OPP abstraction, nevertheless, they are of primary importance. Indeed, as it should be clear from the examples, this block's capabilities define what algorithms can be ultimately supported. We present here an instance of the OPP update logic block as implemented in the FPGA prototype, however it should be made clear to the reader that optimizing this block is not a goal of this thesis, and further research should be conducted.

In our prototype, this block comprises 5 parallel ALUs able to perform a set of elementary instructions as listed in Table 2.3 and 2.4. Instructions in Table 2.3 are those of a typical RISC architecture, while instructions in Table 2.4 are specific for traffic control tasks. The specific instructions to be performed are provided by the output of the EFSM table. Each instruction is 32 bit long and comprises an *opcode*, followed by a variable number of operands that depend on the specific instruction. Both input operands (INi) and output operands ($OUTi$) can be flow registers, global registers, or packet fields. In some instructions, one or more of the operands (IOi) are both used as input and output. IMM represents the case of a constant value, specified directly in the instruction declared in the EFSM table.

The advanced instructions of Table 2.4 are domain-specific operations deemed useful in traffic control applications. Such operations include on line computation of running averages (avg), variances (var), and exponentially decaying moving averages ($ewma$[10]) which can serve the purpose of a moving average, but which can be incrementally computed and do not require to maintain a window of samples.

These instructions have been implemented as dedicated hardware primitives running at the system clock frequency and taking up to two clock cycles to complete execution. Dedicated circuitry is needed as they would would normally require several more clock cycles if implemented using more elementary operations. These instructions represent the "critical path" of the update logic block, as they take the most time to complete, affecting the total length of the feedback-loop. Interestingly, most of the complexity is due to a division operation that has been limited to support dividend and divisor of 16 bits, in order to complete execution in two cycles. Supporting larger operands, e.g. 32 bit, would require more clock cycles, and hence longer critical path delays.

[10]Being t_k the last sample time, and $x_{k'}$ a new sample occurring at time $t_{k'}$, for simplicity of the hardware implementation, $ewma$ is approximated as $m(t_{k'}) = m(t_k)\alpha^{t_{k'}-t_k} + x_{k'}$, where $\alpha = 1/2$ to compute powers as shift operations. The intermediate *decay* quantity in the second line is used just for clarity of presentation.

Type	**Instructions**	**Definition**
Logic	NOP	do nothing
	NOT	$OUT1 \leftarrow NOT(IN1)$
	XOR, AND, OR	$OUT1 \leftarrow IN1\ op\ IN2$
Arithmetic	ADD,SUB,MUL,DIV	$OUT1 \leftarrow IN1\ op\ IN2$
	ADDI,SUBI,MULI,DIVI	$OUT1 \leftarrow IN1\ op\ IMM$
Shift/ Rotate	LSL (Logical Shift Left)	$OUT1 \leftarrow IN1 << IMM$
	LSR (Logical Shift Right)	$OUT1 \leftarrow IN1 >> IMM$
	ROR (Rotate Right)	$OUT1 \leftarrow IN1\ ror\ IMM$

Table 2.3 ALU basic instruction set.

Instruction	**Definition**
avg()	$IO1 \leftarrow IO1 + 1$ $IO2 \leftarrow IO2 + (IN1 - IO2)/(IO1 + 1)$
var()	$IO1 \leftarrow IO1 + 1$ $IO2 \leftarrow IO2 + (IN1 - IO2)/(IO1 + 1)$ $IO3 \leftarrow IO3 + ((IN1 - IO2)^2 - IO3)/(IO1 + 1)$
ewma()	$IO1 \leftarrow IN1$ $decay = 1 << (IN1 - IO1)$ $IO2 \leftarrow IO2/decay + IN2$

Table 2.4 ALU advanced instruction set.

Performance achievable with an ASIC implementation

FPGA is a convenient platform to assess the feasibility of an hardware design, however, due to constraints of the technology itself, FPGAs cannot provide full scale and performance. To this purpose, it is worth discussing the performance and resources available on a dedicated OPP ASIC design.

In doing that, we make the same technology assumptions of state-of-the-art programmable forwarding ASIC RMT [20]. We can then assume the OPP micro-controller, ingress/egress queues, and memory for the packet buffers are the same of RMT, hence an OPP ASIC design would be able to manage 64 ports working at 10 Gbit/s if clocked at 1 GHz. This frequency can be easily achieved by the blocks composing the condition logic block and the update logic block. Indeed, these blocks are similar to the hardware primitives described in [35], where the hardware is also clocked at 1 GHz.

As already mentioned, memory accounts for the largest part in the chip area. Memory in OPP is needed for the TCAM realizing the EFSM Table and the SRAM memory realizing the flow context table. Concerning the TCAM, as already mentioned, the size of the EFSM

table depends on the number of state transitions, which we can estimate (while still being conservative) being an order of magnitude smaller than the number of flows in an ordinary OpenFlow's TCAM. Thus, a reasonable allocation could be 1K TCAM rows of 256 bits each, i.e. 256 Kbits of TCAM, which is only 20% of the 1.2 Mbits of TCAM available in an RMT stage.

Regarding the flow context table, assuming a total of 32 Mbytes on-chip SRAM as in in RMT, the number of flow context entries depends on the number and size of the flow registers. For example, a reasonable configuration with flow context entries of 256 bits could be the following: 128 bits key, 32 bits state label, and 128 bits to be arbitrary partitioned between all registers, e.g. 4×32 bits or 8×16 bits. With 32 Mbytes on-chip SRAM the ASIC would be able store up to 1 million flow contexts, to be distributed among OPP stages.

2.2.5 Remarks

As a final remark, it is important to stress that the specific instruction set provided by the update logic block is independent of the proposed OPP abstraction, i.e., its extension or improvement does not affect the overall OPP design. Indeed, looking closely at the FPGA implementation of OPP, one could notice that not all the programming examples presented before can be executed on such target. The configuration depicted in Figure 2.12, presents two issues: (i) estimating the rate requires pipelining of two instructions, and (ii) the prototype supports division on operand truncated to the first 16 bits. The latter represents a problem if we consider that one of the operand is a byte counter that can easily overflow the 16 bits, e.g. assuming to count bytes over 1-second interval with a 10 Gbit/s link. (i) can be solved simply pipelining two update logic blocks, providing the output of the first as the input of the second; while (ii) can be solved by implementing support for 32 bit division. In both cases, the added complexity of the logic block affects the length of the feedback-loop, which could affect the performance of the system.

In general, there will always be applications that requires more complex computations. The update logic block should be seen as an interchangeable piece of the pipeline. Essentially, it represents the *flow processor* of our system. Finding a configuration for such processor that is expressive enough to allow the execution of a broad range of packet processing algorithms is outside the scope of this thesis.

However, we argue that OPP provides a convenient framework where to research and experiment with such a flow processor. OPP provides a structured way for managing different kinds of state: per-flow, global and per-packet (i.e. headers and metadata) all fed to such flow processor. That of using a feedback-loop in the pipeline model is an important design

choice. As it will be hopefully clear after the next section, it is both a blessing and a curse. A blessing, because such loop does not put any constraint on the flexibility of the processor. As long as it can accept one packet each clock cycle, i.e. it can recursively pipeline packets[11], that processor can be used in an OPP pipeline. A curse, because if not controlled, this feedback-loop can generate state inconsistencies resulting in erroneous forwarding behaviors. In the next section we give guidelines on the maximum allowable time budget, i.e. how much time the processor is allowed to spend on a single packet, that maximizes both throughput and flexibility.

2.3 Understanding the performance of feedback-loop architectures

As it should be clear from the previous sections, the implementation of a high performance hardware data plane requires processing packets in parallel. Typically this is achieved using a pipelined architecture. OpenState and OPP are abstractions designed to provide the ability to program stateful algorithms that read and modify data plane's state, while keeping line rate performance. However, the pipeline architecture outlined for both abstractions includes a shared memory, *state table* or *flow context table*, which is accessed at different stages of the pipeline. This is a problematic scenario for the consistency of the state. In fact, the execution of both OpenState and OPP is split to be executed over multiple pipeline's stages. Since state is typically read in the first stage and written back, after modification, in the last stage, there is a risk the first stage may read an inconsistent state when a new packet enters the pipeline. That is, the read state is going to be invalidated by a result written back in a later stage.

When state read and write operations are quick enough to be executed in a single pipeline's stage, i.e., in a clock cycle, as in the case of the global registers in OPP, the state consistency problem is inherently solved, while a *data hazard* arises when more complex computations are required.

We argue that such risk is negligible when considering realistic traffic workloads.This hypothesis is based on two observations. First, data plane pipelines perform processing only acting on packets' header. For a given line rate, larger packet sizes actually mean a lower rate of packet headers to process. Hence, more time per packet header can be used to execute the pipeline operations. Second, the risk of a data hazard is limited to the risk of having in the pipeline two or more packets whose processing requires access to the same portion of the state, e.g. the same flow context in OPP.

[11] A flow processor spanning 10 clock cycles should be able to process 10 packets in parallel.

To check the actual data hazard probability when taking into account these observations, we designed a trace-based simulator and run it using real traffic traces, from both carrier and datacenter networks. Results confirm that, in most cases, there is just a little probability of incurring in a data hazard even if state read and write operations happen in different clock cycles. Of course, such probability depends on the packet size and network flows distributions, as well as on the aggregation level of the flow definition.

Given these findings, we provide as second contribution a sketch of a pipeline design that avoids such data hazards by stalling the pipeline, when needed. Simulation results show that such a design is able to provide line rate throughput, despite the stalls, even extending the time between state read and write to several clock cycles. Furthermore, this design introduces little overhead in terms of memory and circuitry complexity when compared to state-of-the-art solutions. On the flip side of the coin, when using locking it is impossible to provide line rate throughput in all the cases, therefore the data plane performance is dependent on the actual traffic load properties. Moreover, stalling the pipeline requires the introduction of small queues at the entrance of the pipeline stages. Dimensioning such queues introduces a new variable in the design space: small queues may provide lesser throughput, while big ones may introduce significant latency to the packet forwarding.

2.3.1 Background: packet processing pipelines

For simplicity and without loss of generality, from now until the end of this chapter, we will refer only to the OPP model. All considerations made from this point are equivalently applicable to OpenState.

We present here the data plane architecture used as reference model for the simulations. Programmable data plane solutions such as RMT [20], Intel's FlexPipe [18], and Cavium's XPliant Packet Architecture [17] implement a high level architecture similar to the one sketched in Figure 2.13. In such architectures, packets received from the input ports are stored (enqueued) in a per-port queue and served, with a round robin policy, by a mixer that feeds the packets to an ingress pipeline. After the ingress pipeline, packets are stored in a common data memory. A scheduler selects which packets should be transmitted to the egress queues. A packet selected for forwarding is first processed by an egress pipeline, which is in principle similar to the ingress one, before being finally transmitted to the egress queues[12].

Both ingress and egress pipelines are composed by a programmable packet parser [20] and by a variable number of MAT elements. For each new packet, the parser extracts the *headers*

[12]While we focus on store-and-forward mode of operation, such considerations are applicable also to data planes that work in a cut-through configuration.

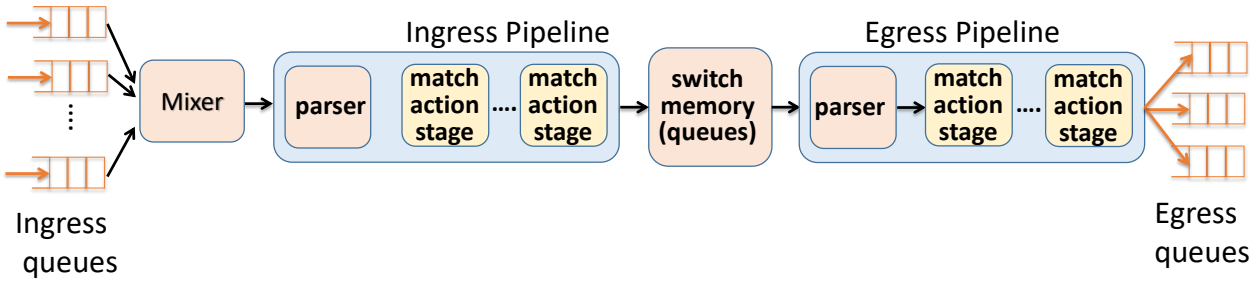


Figure 2.13 Programmable data plane architecture.

that are then processed by the MAT elements. A MAT element implements itself a pipeline, whose architecture may sensibly change from one implementation to the other. We can focus on the MAT element's pipeline, since it is in here that stateful operations are executed.

For the sake of the discussion, we can ignore most of the details of the actual MAT element and model its internal pipeline using just a sequence of stages plus memory. Each stage performs a limited amount of operations, such as read and write from/to memory, or some sort of computation. Since a complex operation can be split in a number of simpler operations executed in multiple stages, the number of stages finally defines the complexity of the operations that can be implemented by the MAT element. Also, since each stage adds a clock cycle to the latency, the number of stages directly impacts the forwarding latency of a packet traversing the MAT element. Finally, and most importantly for the state consistency problem, the number of stages between a memory read and a memory write operations has important implications on the probability of incurring in a race condition. Intuitively, the longer is the time to process a value read from memory, before writing it back, the more probable is the reception of a new packet whose processing requires access to that same memory area. Without loss of generality and to simplify the exposition, we can assume that the first MAT element's pipeline's stage reads from memory, while the last writes back to it.

Rethinking design assumptions

If we would like to design a stateful switching architecture capable of *guaranteeing* line rate performances and state consistency with *any possible* traffic workload, we should eliminate the risk of data hazards by allowing memory read and write operations only within the same stage of the pipeline. Such a design derives from the worst case assumption that all packets have minimum size, that they arrive back-to-back, i.e., with no inter-packet gaps, and that they all need to access the same memory area. More specifically, let us consider a data plane

with a throughput of 640 Gbit/s, with a chip clocked at 1 GHz, as it is the case of RMT [20]. As showed in Figure 2.14, we can assume that packets are read in chunks of at most 80 bytes (i.e., 80 × 8 bit × 1 GHz = 640 Gbit/s) when entering the data plane's pipeline. Consequently, it will take 1 clock cycle to read packets with minimum size ≤ 80 bytes, while it will take more cycles to read longer packets, e.g., 19 for 1500 bytes.

As mentioned earlier, the data plane's pipeline is dimensioned to accept the headers of a new packet at each clock cycle. However, even when all packets arrive back-to-back, the variability of the packet size will cause the pipeline to experience one or more idle cycles. For example, in the case of maximum size packets of 1500 bytes, the pipeline will receive a new packet's headers at intervals of 19 clock cycles[13].

It is known that packets produced by today's application have very variable size distributions. E.g., spanning from 64 bytes to 1500 bytes (in a typical case), which could leave some space for relaxing the constraint on the atomicity of memory read and write operations, when dealing with non corner-case traffic loads.

Per-flow concurrency

The second observation, which follows the OPP model, is that data plane state can be categorized in two types: *global state* and *flow state*. The first type is state that is shared among all packets, with no distinction, while flow state is shared only by packets of the same flow, where the definition of the flow can be arbitrary (recall the key extractor in OPP)

Usually packet processing functions use a combination of the two, or only one. For example, a stateful firewall needs to maintain state for each TCP connection. A source NAT (SNAT) that dynamically translates the source IP address and port of outgoing connections, needs to maintain both flow state and global state: flow state for each L4 connection in order to distinguish between packets of new or existing connections, and in the case of new ones, it needs to pick a source address and port from a pool of available ones. Maintaining a pool of addresses and ports, e.g. using a stack, is an example of global state. Advanced load balancing schemes such as CONGA [36] maintain state at different aggregation levels: i) the 5-tuple in order to distinguish between flowlets, i.e. burst of packets, of the same L4 connection, ii) tunnel ID to maintain real-time utilization levels of several paths and iii) global state to maintain the best path among all available ones, to assign new flowlets to.

[13]While in principle it is possible to widen the pipeline data-path to reduce the maximum number of clock cycles to process a packet, and to increase the throughput, the routing of such a big number of parallel wires prevents several technological challenges that actually limit the maximum data-path width. For example in RMT[20] it is explicitly mentioned that the data-path is limited due to these technology constraints. Furthermore, the maximum achievable throughput is finally limited by the network interfaces speed.

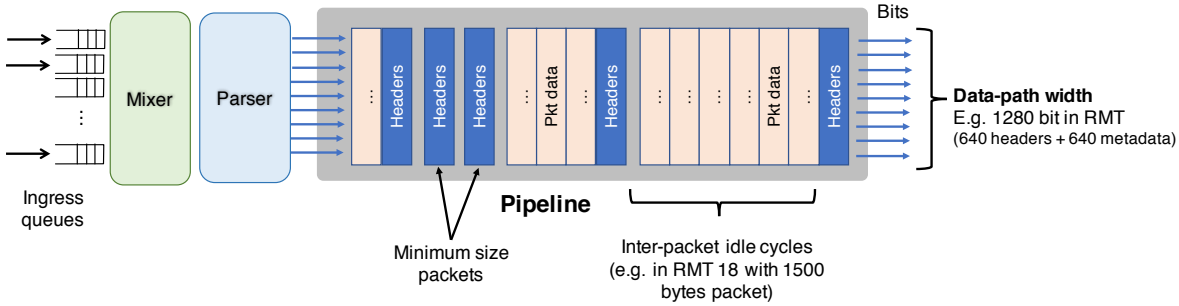


Figure 2.14 Processing of a RMT-like pipeline. Packets are read from input ports in chunk of 80 bytes. Longer packets cause idle cycles in the pipeline.

In OPP, before the packet is processed by a MAT, a flow key is derived by looking solely at a subset of header fields decided by the programmer at configuration-time. Moreover, the operation on state that a programmer can define are limited to the portion of the memory associated to the packet's flow key. As such, by using the OPP's flow context table abstraction, multiple packets accessing and modifying different entries of the table can be processed in parallel, with no harm for consistency. In other words, packets with *different flow keys* can be processed in *parallel*.

2.3.2 Measuring the risk of data hazards

We start by evaluating the actual probability of generating data hazards when processing packets in OPP, i.e. with stateful functions spanning many clock cycles. To do so we implemented a simulator that we feed with real traffic traces. The code of the simulator is available at [43].

Table 2.5 Packet traces used in simulations

Trace	**Source**	**Num. of packets**	**Distinct flows per 1 million pkts**		
			5-tuple	*ipdst*	*ipdst/16*
chi-15	CAIDA [37]	3.5 billions	100.6K	57.7K	4.6K
sj-12	CAIDA [38]	3.6 billions	429K	17K	2K
mawi-15	MAWI [39][40]	135 millions	40.8K	17.3K	1.7K
fb-web	Facebook [41][42]	447 millions	n/a	n/a	n/a

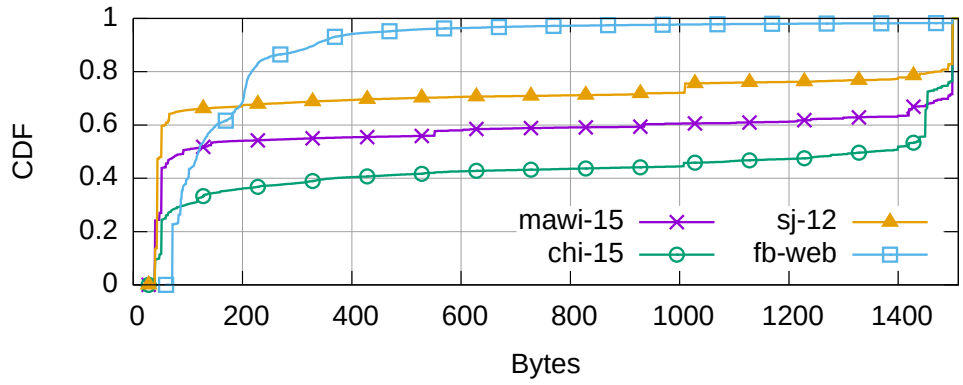


Figure 2.15 Packet size cumulative distribution.

Traffic traces

We used 4 publicly available traffic traces (Table 2.5). Three traces (*chi-15*, *sj-12*, *mawi-15*) are taken from backbone carrier networks and one from a datacenter (*fb-web*). Each trace presents different characteristics, in terms of packet size (Figure 2.15) and number of flows when using different aggregation keys (Table 2.5). CAIDA publishes 1-hour long traces 4 times per year. We selected *chi-15* as one representative of usual conditions as the packet size and number of flows is close to the majority of the other traces published by CAIDA in the recent years [44]. The packet size presents a bimodal distribution, 30% of packets have minimum size below 80 bytes, wile 50% have larger size close to 1500 bytes. On the other hand, *sj-12* and *mawi-15* represent an abnormal situation. In both traces there is a prevalence of smaller packets and, in the case of *sj-12*, also an unusual large number of 5-tuples w.r.t. the number of distinct IP destination address.

Finally, *fb-web* includes packets collected from a Facebook's datacenter's cluster that serves web requests. As such it presents a predominance of small packets (80% have size $<$ 200 bytes). Unfortunately, the traces provided by Facebook are the result of uniform sampling with rate 1:30K. As a consequence, we were not able to count the number of distinct flows. The reason is that the probability that two consecutive packets belong to different flows is higher than the other traces, not because of the traffic characteristics but because packets are indeed distant in time (a distance potentially greater than the average flow life). Hence, we use *fb-web* only to measure the effects of the variable packet size, not the flow distribution.

Results: fraction of data hazards

We simulate the case of a stateful processing block comprising N sequential (pipelined) stages, where each stage is executed in 1 clock cycle, and where the first stage reads from the memory while the last writes back. A data hazard is the event in which the first stage of the action pipeline processes a header, while another one is currently traveling in the same pipeline. Clearly, when $N = 1$, there is no risk of data hazards.

We simulate a "near-worst case" scenario, by feeding all packets back-to-back, i.e. 100% line rate utilization with no inter-packet gaps. Packets are read in chunks of 80 bytes (as in RMT), hence taking 19 clock cycles to read 1 packet of 1500 bytes. For $N \leq 18$, there is no risk of data hazard. Conversely, with small packets the pipeline will experience shorter idle gaps. In the worst case, the headers of minimum size packets arriving back-to-back, will be also processed back-to-back in the pipeline, hence causing a data hazard for any $N > 1$. When considering per-flow concurrency, if two headers belonging to *distinct* flows are processed back-to-back, this does *not* generate a data hazard.

The simulator process traffic in batches of 100K packets. For each batch, it computes the fraction of data hazards (FDH) over the total number of clock cycles needed to process 100K packets (which depends on the packet size). To reduce simulation time, for each trace it selects batches at a rate of 1:100, in other words it evaluates one batch of 100K consecutive packets every 10M packets. The observation here is that traffic characteristics vary slowly in a period of 10M packets, hence multiple batches close in time will produce similar results. For each trace, the simulator extracts the 99th percentile from all FDH samples. As an example, if for a given trace the 99th percentile of the FDH is 0.3, it means that in the 99% of batches evaluated, the FDH was below 30%.

Figure 2.16 shows the results for all traces when all packets are considered belonging to the same flow, i.e. accessing global state. Instead, in Figure 2.17 FDH values are plotted for each trace when aggregating packets with different flow keys. As expected, the FDH greatly depends on the packet size distribution and flow keys, with smaller probability of hazards for traces with higher prevalence of larger packets, and for longer, i.e. finer, flow aggregation keys. such a *chi-15*. In the second case, per-flow concurrency affects the results. For example, with *sj-12* when considering state associated to distinct 5-tuples, the risk on incurring in a data hazard is way below the case when state is associated to distinct destination IP addresses. This result follows the flow distribution showed in Table 2.5. For all traces, using 5-tuples performs better than other flow keys. For *chi-15*, in all cases the FDH is around 1%.

This result is important because it shows the probability of creating inconsistent state, and

hence, if memory locking is a viable approach, the probability of incurring in such locking, thus affecting both throughput and latency.

2.3.3 Preventing data hazards via memory locking

We propose here a general approach to perform memory locking among packets competing to access the same memory portion. This approach was designed around OPP, but it can be generalized to any architecture implementing a feedback-loop of multiple clock cycles.

Locking is implemented by stalling the pipeline. That is, if two packets of the same flow arrive back-to-back, processing is paused for the second packet until the first one has left the pipeline. This already affects throughput by a factor of $1/N$ per flow, hence aggregate throughput is maximized when at least N flows are active. Clearly, stalling calls for buffering which also introduces additional latency. We are interested in measuring the impact on throughput and latency when such locking approach is implemented.

Figure 2.18 depicts a simple but effective pipeline design that implements stalling in order to prevent data hazards. In this design, a stateful processing function spanning N clock cycles, e.g. an OPP stage of 6 clock cycles, is preceded by few Q queues and a scheduler. For each packet's headers, a first block extracts a flow key (FK), then a dispatcher stores the headers in the q-th queue, where $q = hash(FK)\ mod\ Q$, thus preserving the processing order between packets of the same flow. Each queue can store at most Q_{len} headers.

The scheduler decides which packet to admit in the processing pipeline by looking at the tip of each queue and comparing the head-of-line FK with the at most N FKs currently traveling in the pipeline. The scheduler admits a header if its FK is *not* currently in the pipeline. The scheduler is work-conserving, meaning that all non-empty queues are compared at the same time, if at least one header can be served it will do so. To avoid starvation of a queue, the scheduler serves queues in a round-robin fashion, i.e. with cyclic priority.

We know from OPP that the FK can have arbitrary length of FK_{len} bits, depending on the number of state memory cells available, for example $FK_{len} = 32$ bits for 2^{32} memory cells. As $Q < 2^{FK_{len}}$, multiple flows will end up sharing the same queue. Such an event may generate head-of-line blocking, in which all packets in a queue are held by the first one. Clearly, such a problem can be reduced by adding more queues, which has a cost in terms of silicon needed to implement both the queues and the scheduler.

For the scheduler to be work-conserving, it needs to compare all queues at the same time, hence increasing the number of wires with i) the number of queues and ii) the number of bits to compare for each queue. For this reason, to simplify the implementation of the scheduler's

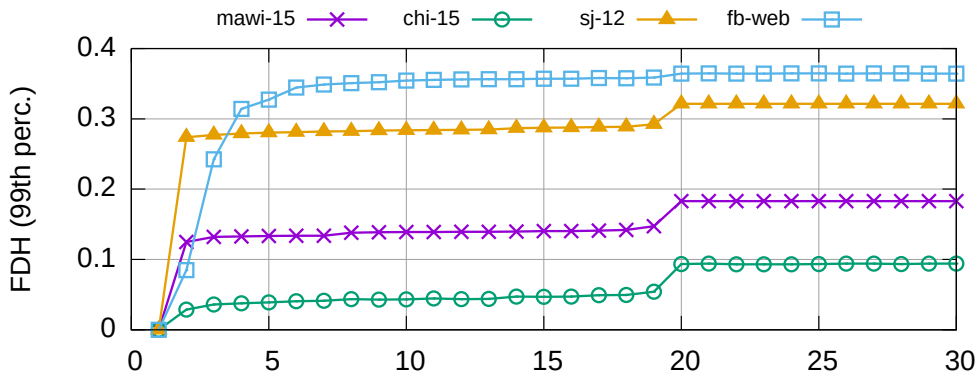


Figure 2.16 Fraction of data hazards (FDH) w.r.t. increasing pipeline depth when all packets are considered belonging to the same flow.

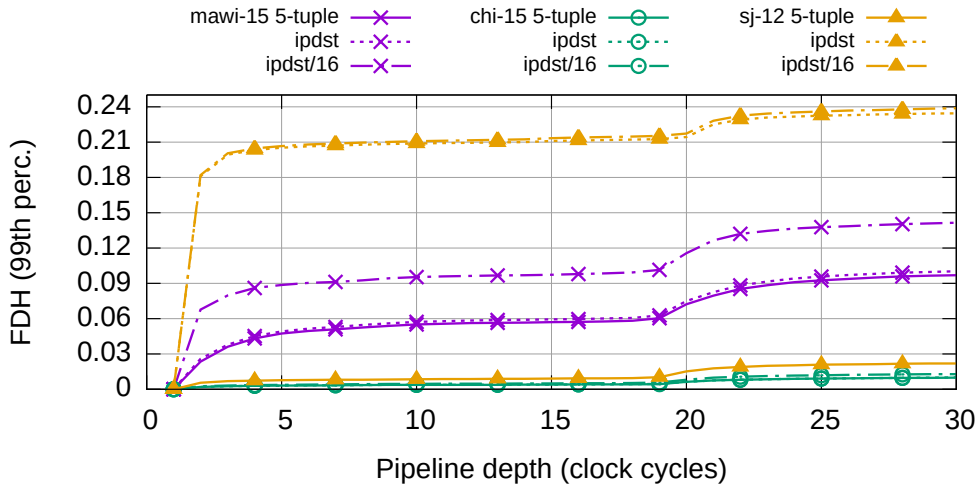


Figure 2.17 Fraction of data hazards (FDH) w.r.t. increasing pipeline depth when aggregating packets per different flow keys.

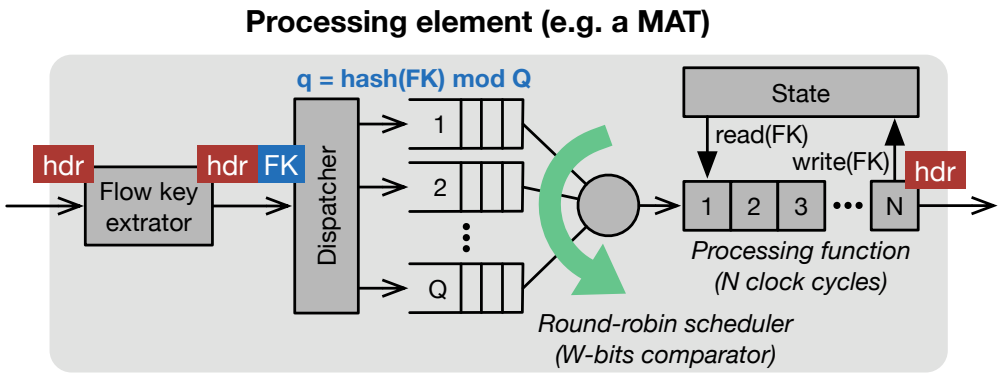


Figure 2.18 Architecture of a stateful processing block with memory locking. Headers are queued based on their flow key (FK). A scheduler compares the FK of the head of each queue with what is traveling in the function pipeline, admitting only one header per flow.

comparator, the FK is reduced to a smaller space of W bits. This operation can be performed by the flow key extractor, which along the FK (needed later to access the state) extends the headers with a field w. A trivial approach is that of hashing the FK in the reduced space, i.e. $w = hash(FK)\ mod\ W$. For example with $W = 4$, the scheduler is able to distinguish among $2^4 = 16$ flows. If $W < FK_{len}$ there will be different flows colliding onto the same value w, impacting performance. Flow collision also depends on the hash function, during simulations we set $hash() = crc16()$, which is a common function in packet processing architectures. However, we did not investigate the impact of other hash functions on the distribution of FKs among the different queues and values of w.

Silicon overhead

We provide in this section an evaluation of the feasibility of the proposed locking design. The combinatorial logic complexity of the locking scheme is that of Q comparators, each one comparing the W bits of the head of the queue with each of the N stages of the pipeline. We have synthesized the comparators using the Synopsys Design Compiler and a 45 nm standard-cell library. The VHDL code is available at [43]. Table 2.6 shows the area and minimum critical-path delay of one comparator when $W = 4$ and $N = 8, 16, 32$. In all the cases timing constraints are easily met at 1 GHz, which is the clock frequency of state-of-the-art architectures such as RMT and Banzai [35]. In terms of area overhead, when comparing our results to a 200 mm^2 chip as in [35, 28], we find that comparators area is negligible. For example, the area of 16 comparators ($Q = 16$) when $N = 16$ and $W = 4$, corresponds to the 0.01% of the total chip area.

Finally, notice that 45 nm is a fairly old technology for circuit synthesis, hence we expect both the area overhead and the minimum delay to be even smaller when using more recent libraries, e.g. 32 nm, 22 nm or 16 nm.

The memory requirements to implement queues is $H_{len} \times Q \times Q_{len}$ bits, where H_{len} is the length in bits of the data path. With $H_{len} = 88$ bytes (80 for the header and 8 for the metadata), $Q = 4$ and $Q_{len} = 100$, the locking scheme requires 35.2 KB of memory for the queues. Approximately a 3.5% memory overhead compared with the memory of a MAT stage in RMT.

Trace-based results

We evaluated the proposed architecture using the same traffic traces presented in Section 2.3.2. When collecting performance metrics, the simulator uses the same approach described in Section 2.3.2: traffic is processed in batches of 100K packets, with a distance of 10 millions packets between each batch; and 100% line rate utilization. For each batch of packets the simulator computes the *throughput* as the fraction of packets served by the scheduler, over the total number of packets received; and the *latency* as the number of clock cycles from when the packet is completely received to when it is served by the scheduler, i.e. it enters the function pipeline. For simplicity, when $N = 1$, i.e. no locking required, latency is 0. Latency is computed for each packet, for each batch the simulator takes the 99th percentile among all latency values, finally taking the maximum among all batches for a given trace. For example, a latency value of 5 means that in the worst case, 99% of the packets experienced a latency of no more than 5 clock cycles, e.g. 5 ns at 1 GHz.

We evaluated these metrics when varying the different parameters described in Section 2.3.3 for the different traces. For simplicity we limit the maximum size of the pipeline to 30 clock cycles. Table 2.7 summarize results in terms of clock cycle budget, which is the *maximum allowable* number of clock cycles for a stateful function to sustaining a given throughput. For example, to sustain 100% throughput, using queues of size 10 (headers) does not provide any benefit, as the clock cycle budget is 1 for each trace and flow key. However, by adding more

N	Area at 1 GHz	Minimum delay
8	196 µm^2	240 ps
16	929 µm^2	360 ps
32	1560 µm^2	400 ps

Table 2.6 Area and minimum critical-path delay of one comparator in a 45 nm standard-cell library when $W = 4$.

capacity to queues up to 100, budget improves even when using only 1 queue, allowing for functions spanning 20 clock cycles for all flow keys with *chi-15*, and 4 clock cycles with *sj-12*, but only when aggregating packets per 5-tuple. Clearly, long queues impact latency. Both clock cycle budget and latency improve when admitting a lower throughput of 99.9%, i.e. allowing for 0.1% drop probability. Clearly, reducing utilization (100% in our experiments) reduces further the risk of drops while maintaining the same cycle budget.

Figures 2.19, 2.20, 2.21 and 2.22 show the detailed results in terms of throughput and latency for all the traffic traces considered.

2.3.4 Discussion

Issues with blocking architectures While the proposed solution for memory locking enables the execution of more complex operations directly in the data plane, it implements a blocking architecture. That is, if applied to OPP, for particular workloads, the data plane is unable to offer line rate forwarding throughput. As a consequence, the data plane processing should be adapted to the expected network load characteristics, for the line rate to be achievable. The results presented help in defining the boundaries of the achievable performance, for a given workload and set of operations. An additional problem of the dependency on the workload is the possibility to exploit such dependency, e.g., to perform a denial of service attack on the data plane. However, the ability to program stateful algorithms in the data plane could help in detecting and mitigating such exploitations at little cost.

What can we do more with more clock cycles? As discussed in Section 2.2.4, the length of the pipeline, and thus the number of clock cycles, depend on the complexity of the update logic block. More clock cycles allow for more complex functions. To the far end, one can envision the possibility of using a general purpose packet processor, carefully programmed to complete execution in a longer, but bounded, clock cycle budget, with predictable performance when the traffic characteristics are known. Finally, another option is that of supporting larger, potentially off-chip memories like DRAMs which have slower access times to read and write values. The same multi-queue scheduling approach could be used to coordinate access to multiple parallel memory banks, where each bank is associated to a queue.

When is consistency really an issue? Lastly, but perhaps most important, one can argue that preventing data hazards in packet processing might not be such a strict requirement. State inconsistencies can cause one or more packets to be processed erroneously, for example,

Table 2.7 Clock cycle budget (and latency) when using memory locking

Maximum number of clock cycles (up to 30) per processing function, to sustain a given throughput. In all cases $W = 4\ bits$. "Global" represents the case when packets need to access global state. Latency values are given for 1 GHz clock frequency, i.e. 1 clock cycle = 1 ns.

			chi-15				sj-12				mawi-15				fb-web
Thrpt	Q_{len}	Q	5-tuple	ipdst	ipdst/16	global	5-tuple	ipst	ipdst/16	global	5-tuple	ipdst	ipdst/16	global	global
100%	10	1	1	1	1	1	1	1	1	1	1	1	1	1	1
		4	1	1	1		1	1	1		1	1	1		
		8	1	1	1		1	1	1		1	1	1		
		16	1	1	1		1	1	1		1	1	1		
	100	1	20 (174ns)	20 (190ns)	21 (230ns)	8 (282ns)	4 (49ns)	1	1	1	2 (18ns)	2 (20ns)	2 (35ns)	1	2 (10ns)
		4	30 (175ns)	30 (192ns)	30 (320ns)		8 (152ns)	1	1		2 (12ns)	2 (14ns)	2 (25ns)		
		8	30 (137ns)	30 (144ns)	30 (259ns)		8 (133ns)	1	1		2 (12ns)	2 (14ns)	2 (25ns)		
		16	30 (122ns)	30 (126ns)	30 (221ns)		8 (123ns)	1	1		2 (11ns)	2 (14ns)	2 (24ns)		
99.9%	10	1	8 (16ns)	8 (16ns)	8 (18ns)	4 (18ns)	2 (5ns)	1	1	1	1	1	1	1	2 (10ns)
		4	14 (33ns)	14 (31ns)	14 (38ns)		2 (4ns)	1	1		1	1	1		
		8	16 (39ns)	15 (30ns)	16 (44ns)		2 (4ns)	1	1		1	1	1		
		16	17 (37ns)	18 (43ns)	16 (42ns)		2 (5ns)	1	1		1	1	1		
	100	1	27 (568ns)	27 (618ns)	26 (605ns)	8 (282ns)	6 (143ns)	2 (86ns)	2 (84ns)	1	3 (42ns)	3 (48ns)	3 (100ns)	2 (60ns)	2 (10ns)
		4	30 (175ns)	30 (192ns)	30 (320ns)		15 (526ns)	2 (79ns)	2 (77ns)		4 (41ns)	4 (52ns)	4 (135ns)		
		8	30 (137ns)	30 (144ns)	30 (259ns)		22 (731ns)	2 (79ns)	2 (78ns)		4 (38ns)	4 (50ns)	4 (131ns)		
		16	30 (122ns)	30 (126ns)	30 (221ns)		25 (741ns)	2 (79ns)	2 (72ns)		4 (37ns)	4 (49ns)	4 (129ns)		
99%	10	1	21 (80ns)	21 (80ns)	21 (89ns)	7 (60ns)	3 (14ns)	1	1	1	2 (13ns)	2 (14ns)	1	1	2 (10ns)
		4	30 (142ns)	30 (148ns)	30 (184ns)		10 (45ns)	1	1		5 (34ns)	4 (31ns)	2 (18ns)		
		8	30 (129ns)	30 (138ns)	30 (184ns)		11 (47ns)	1	1		6 (42ns)	4 (31ns)	2 (18ns)		
		16	30 (116ns)	30 (122ns)	30 (180ns)		12 (47ns)	1	1		7 (52ns)	4 (31ns)	2 (18ns)		
	100	1	30 (950ns)	30 (922ns)	30 (1.1us)	9 (842ns)	8 (268ns)	2 (86ns)	2 (84ns)	2 (171ns)	9 (422ns)	8 (380ns)	5 (316ns)	4 (379ns)	2 (10ns)
		4	30 (175ns)	30 (192ns)	30 (320ns)		22 (1.1us)	3 (285ns)	3 (283ns)		24 (1.7us)	17 (1.3us)	7 (572ns)		
		8	30 (137ns)	30 (144ns)	30 (259ns)		30 (1.9us)	3 (290ns)	3 (292ns)		30 (2.1us)	23 (2.2us)	8 (753ns)		
		16	30 (122ns)	30 (126ns)	30 (221ns)		30 (940ns)	3 (296ns)	3 (293ns)		30 (1.3us)	25 (2.5us)	8 (759ns)		

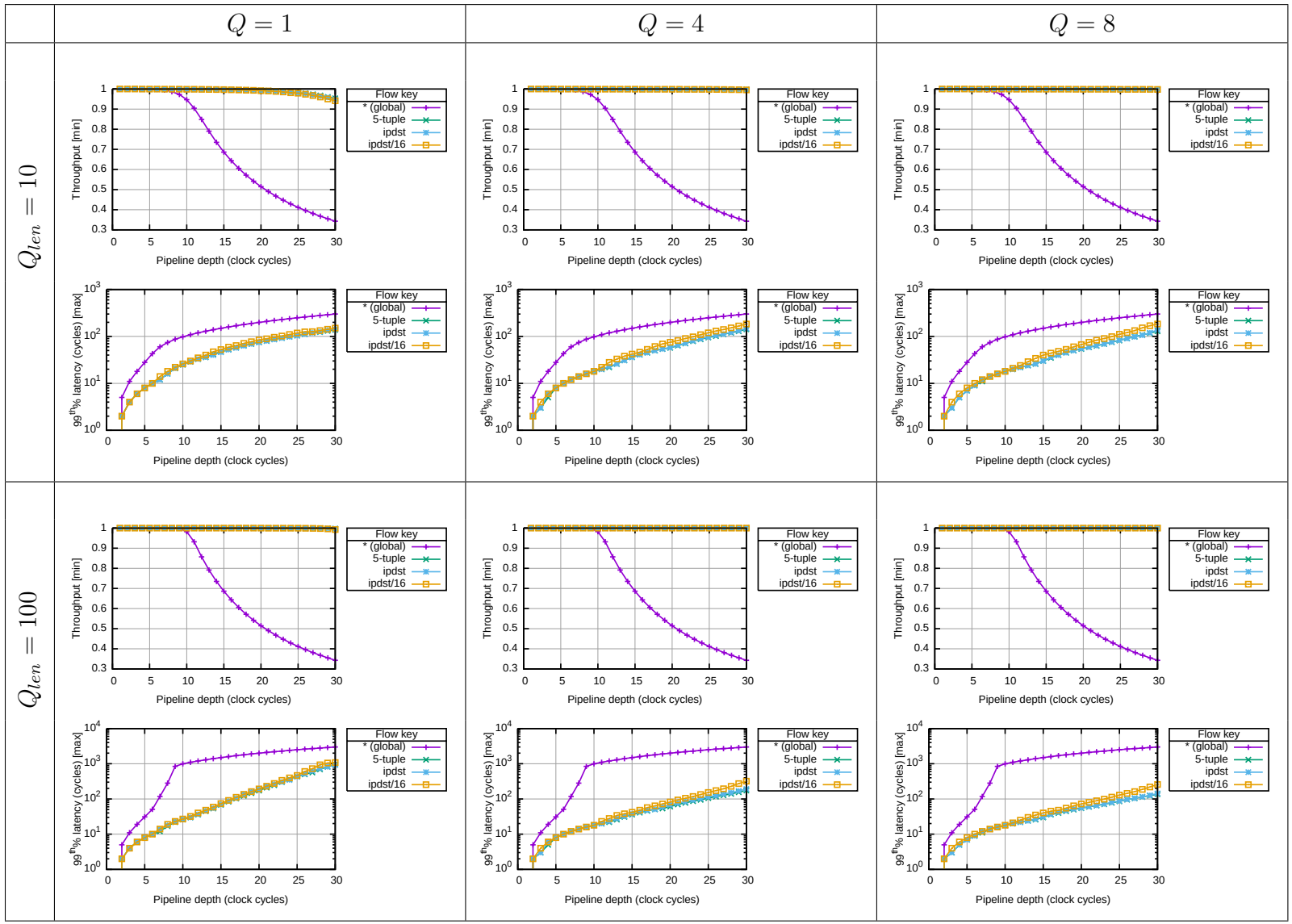


Figure 2.19 Throughput and latency for the *chi-15* traffic trace.

sj-12

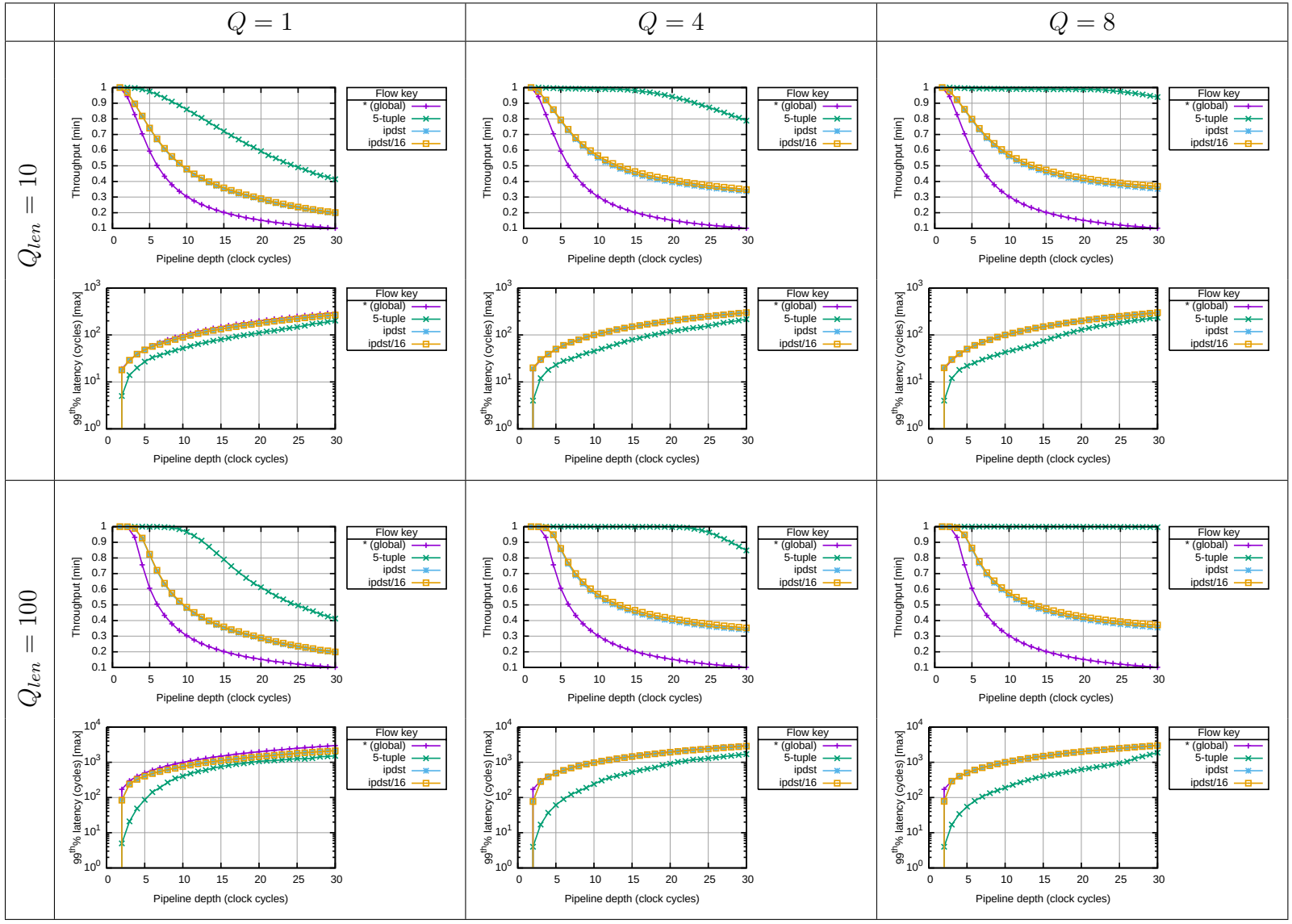


Figure 2.20 Throughput and latency for the *sj-12* traffic trace.

mawi-15

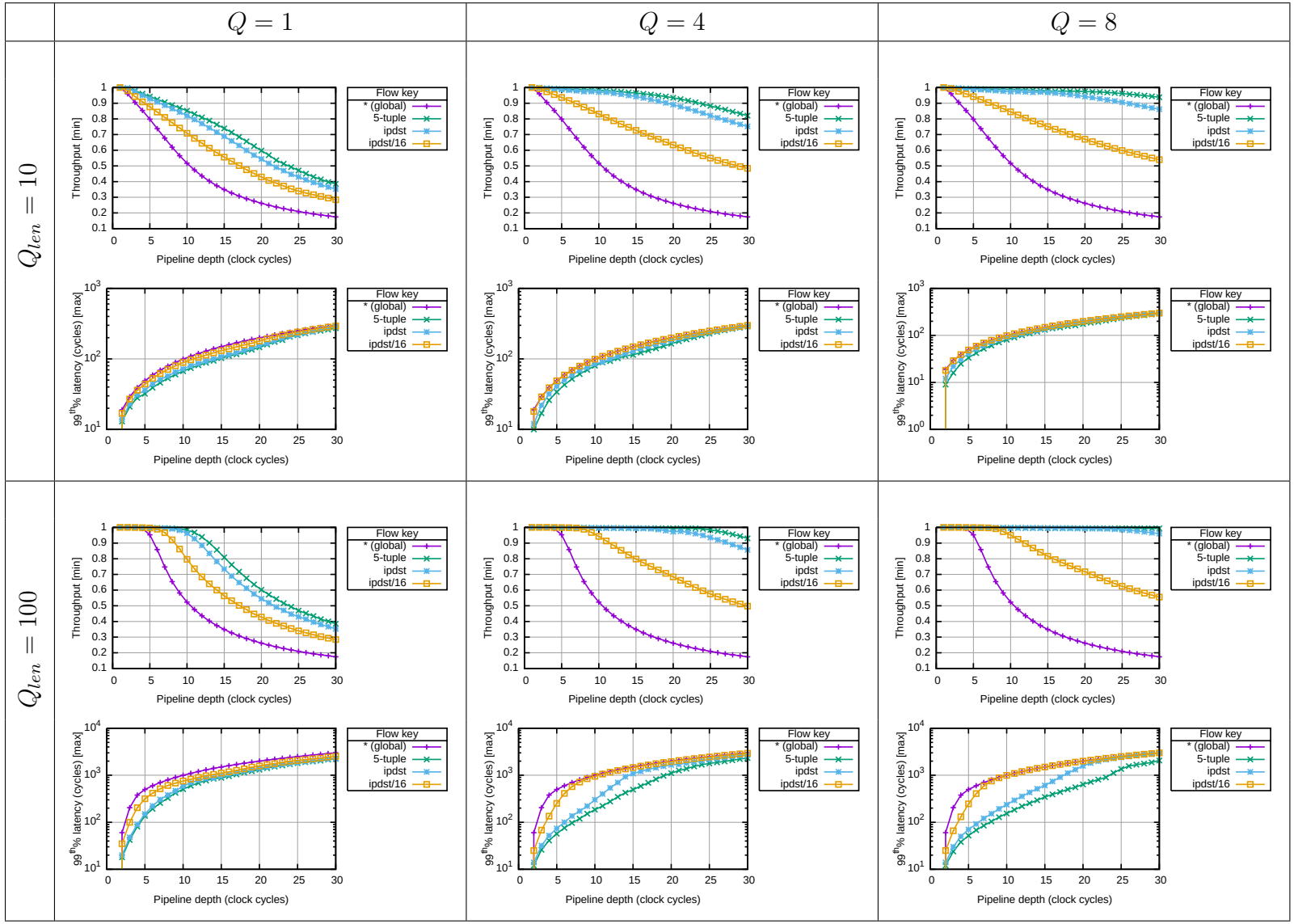


Figure 2.21 Throughput and latency for the *mawi-15* traffic trace.

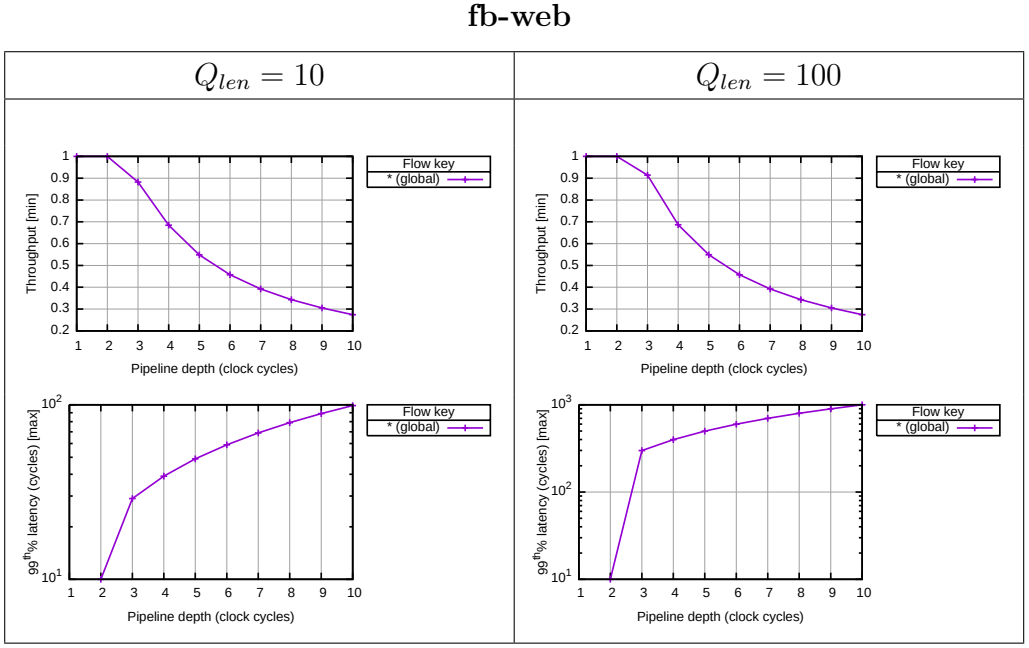


Figure 2.22 Throughput and latency for the *fb-web* traffic trace. Results are shown only for the case of global state, hence $Q = 1$, for the same reason introduced in Section 2.3.2.

dropped or sent to the wrong path, or their headers confused. Luckily, the unreliability of the network has been taken into account since the dawn of the Internet. Today's end-to-end applications use reliability mechanisms that are also end-to-end, such as TCP.

As an example, from Figure 2.17 we know that with an OPP-like pipeline, i.e. six clock cycles feedback-loop, applications that rely on state unique to the UDP/TCP 5-tuple are subject to less than 1% risk of data hazards in most cases, 9% with *mawi-15*. End-hosts relying on the data plane performing load balancing (ECMP-like or flowlet-based), stateful firewalling, or NAT would be minimally affected by such a small probability of inconsistencies. In the case of load balancing, few packets after the first one of a new flowlet might end up being forwarded to the old path, arriving out-of-order and causing the TCP to *temporarily* reduce its sending rate. Similarly, for a stateful firewall (or a NAT), if an inbound packet is dropped because the port is still closed (due to outdated NAT rules), the end-host application will likely detect the event and ask for retransmission. For most practical cases of these applications, 1% of such risk seems to be a reasonable tax to pay to avoid locking and get guaranteed throughput.

Another interesting case is that of cross-flow state updates. Considering the example of a L2 learning switch in Section 2.1.2, where `lookup-scope = [ethdst]` and `update-scope = [ethsrc]`, here an inconsistency would cause one or few packets to be flooded or sent to an old route. Now, regardless of the end-hosts asking for retransmission, in this case, it does not make sense to ask for strict consistency, as a *total order* between packets of different flows is not possible.

Assuming two flows A and B, where processing of packets of A updates the state of B and vice versa, it makes sense to talk about consistency only if a total order between packets of A and B is possible, i.e. if the switch is able to tell which packet of the two flows was originated first. Indeed, as for the L2 learning switch, if A and B are flows originated at physically distinct locations, each one affected by different network delays (i.e. different paths, queuing, etc.), if the switch receives two packets A and B, at two separate ports at the same time, it is very impractical to know which packet was generated first. The switch would have to take an arbitrary decision on which packet to process first, possibly creating an inconsistency. Thus, as a general rule, when dealing with applications based on cross-flow state handling, if the two flows are generated by two different, separate processes, then the switch should not need to enforce consistency via memory locking. Eventual consistency is enough, provided that a consistent state is reached in a timescale smaller than one RTT.

On the other hand, strict consistency is a legitimate concern for network tasks such as accounting. For example, considering an operator wanting to bill its customers per volume of traffic sent. To this purpose, the operator can instantiate a byte counter in OPP. Assuming a

feedback-loop of 10 clock cycles, and assuming a malicious user capable of having its packets to be delivered at the switch in batches of 10, all processed back-to-back. Without memory locking, the counter will account only for the last packet[14], end the user billed only for a tenth of her real usage. This application requires strict consistency.

The bottom line is that programmers of an OPP switch should be free of deciding their own tradeoff between consistency and performance. Enabling memory locking brings consistency guarantees but only predictable throughput, on the contrary, applications robust to eventual inconsistencies permit line rate guarantees all the time.

2.3.5 Remarks

We discussed the case of a stateful packet processing pipeline implementing a feedback-loop, i.e. when state reads are performed on the first stage, and writes on the last one. Data hazards are caused when multiple packets travel in the pipeline, potentially accessing the same memory portion. Prevention of data hazards is performed by stalling the pipeline, where packets waiting to be served are stored in queues. By using simulations on real traffic traces from both carrier and datacenter networks, we show that such model can be applied with little or no throughput degradation, at the expense of added latency of 10-500 nanoseconds in most cases (for an RMT-like pipeline operating at 1 GHz with 640 bits data-path width). The exact clock cycle budget and latency depends on the (i) pipeline data-path width and clock frequency, (ii) the granularity of the flow key used to access state, and, most important, (iii) the packet size and flow distribution of the traffic workload considered. The case of global state seems to be hard to handle with feedback-loops longer than 1-2 clock cycles, hence access to global state should always be performed atomically, i.e. in 1 clock cycle, as in OPP.

2.4 Related work

Driven by the invention of SDN, many researchers have tried to extend network programmability to the data plane, in an attempt to both improve OpenFlow-based SDNs and to enable a new class of applications. Similar to OpenState and OPP, several works address the feasibility of a programmable data plane that supports the execution of stateful data plane algorithms. We review the most relevant of these works in this section.

[14]Both the first and last packet read the same flow context register, and add to that.

Evolving OpenFlow

OpenFlow specification. OpenFlow, now at version 1.5, has undergone several extensions, but none of this work has targeted the possibility to program stateful forwarding behaviors. Rather, extensions have been devised to fix punctual shortcomings and accommodate specific needs, by incorporating *very specific* stateful primitives, such as meters for rate control, group tables for fast failover or load balancing, synchronized tables for supporting learning-type functionalities, etc. OpenState and OPP show how it is possible to extended and generalize the stateless OpenFlow match+action abstraction to allow the programming of custom stateful forwarding behaviors. It is worth mentioning that at the time of writing, OpenState is being considered for standardization in the next version of the OpenFlow 1.6 specification.

DevoFlow. Curtis et al. [10] were among the first to show how the stateless nature of the OpenFlow's data plane and the inherent overhead on the control plane, cannot meet the needs of high-performance networks, especially in data centers. They propose DevoFlow, an extension to OpenFlow with dedicated stateful actions for multipath load balancing, rapid rerouting after a failure, and triggers evaluated over counters. OPP and OpenState are more generalized abstractions that allow to describe DevoFlow's stateful functions *and* others.

FAST. Moshref et al. [45] propose FAST, a forwarding abstraction that shares the same idea of OpenState and OPP of using state machines to modify the forwarding behavior of switches. The FAST abstraction is also based on a feedback-loop between a state table and a flow table. However, the authors limited the evaluation to a prototype software implementation (based on the *learn* action of Open vSwitch [46]), instead, we provide an evaluation on the hardware feasibility of both OpenState and OPP, and the effects of the feedback-loop on state consistency and performance. Finally, while the work on FAST probably happened in parallel with OpenState, the latter was published before.

Abstractions for programmable high-speed switches.

The following works devise abstractions and hardware architectures for high-speed programmable data planes that offer capabilities well beyond OpenFlow.

RMT. Bosshart et al. [20] designed RMT, a first example of a high-performance programmable chip. RMT provides abstraction for a reconfigurable switching ASIC that can

parse and modify arbitrary packet headers in a pipeline of match+action tables. Interestingly, RMT shows that such programmability can be supported with performance comparable to state-of-the-art fixed-function chips: it can process packets at a line rate of 640 Gbit/s. However, RMT is a *stateless* forwarding abstraction, programmers cannot define stateful data plane algorithms in RMT. The only *statefulness* is relegated to dedicated counters.

P4. Bosshart et al. [47] proposed P4, a domain-specific language to ease the programming of chips like RMT. P4 allows to define forwarding behaviors in an hardware independent-manner. Interestingly, the first version of the language v1.0 (also called $P4_{14}$) [48] allows the definition of stateful behaviors via *registers* that can be accessed within the pipeline to keep state (and manipulate it). In principle, one could use P4 to describe a behavior equivalent to that of an OpenState or OPP stage.

However, $P4_{14}$ lacks of a consistency model, i.e. registers can be accessed at different stages of the pipeline without providing means to arbitrate concurrent access. As a result, in P4 the only way in which state consistency, hence algorithm correctness, is guaranteed, is to assume that the pipeline processes one packet at a time. Such assumption is of course not acceptable, since pipelining of packets is required to provide high performance in hardware.

In the latest version of the specification v1.1 (also called $P4_{16}$) [49], a special `@atomic` annotation has been introduced to require a portion of the code to be executed atomically, i.e. avoid data races that can ultimately generate state inconsistencies. However, it is up to the target-specific compiler to implement mechanism to guarantee atomicity, while maximizing throughput. To the best of our knowledge, we are not aware of any P4 target hardware platform implementing such specification.

The OPP abstraction, while at a lower level than P4, allows one to define stateful forwarding behaviors with strict consistency guarantees.

NetASM. Shahbaz et al. [50] proposed an intermediate representation (IR) for programmable data planes. IRs serves as an intermediate layer between high-level languages and the low-level, specific instruction set of different kind of network devices: FPGAs, virtual switches, and line-rate switches. NetASM models the low-level operations of a pipelined packet processing architecture, providing means of expressing stateful behaviors.

NetASM captures the need of differentiating between per-packet and persistent states. Similar to OPP, they provide abstractions that can identify and isolate *state contexts*, providing modes of atomic execution on a per-context level. However, similar to $P4_16$, optimization of the pipelined processing for atomic operations on persistent states, still requires a compiler to

perform careful analysis of the program (e.g. data dependence analysis), a task that might not be straightforward. To the best of our knowledge, we are not aware of any NetASM compiler for a hardware platform implementing such optimizations.

OPP is an abstraction that instead forces the programmer to think and operate only on two kind of persistent state context: per-flow state and global state. As shown in Section 2.3, consistent pipeline access can be optimized by knowing in advance the flow key of each packet. Ideally, an OPP stage could be used as a NetASM target, provided that a compiler is able to derive the necessary "flow-key dependence" from a NetASM program. At this point, it seems more straightforward to directly program the OPP stage.

Domino/Banzai. In [35], Sivaraman et al. propose Domino, a C-like language to program stateful data plane algorithms, and Banzai, an abstract pipeline model that extends RMT with support for stateful actions called *atoms*. This is probably the most related work to the content of this chapter, as such it deserves a longer analysis.

A Domino compiler maps a high level program to a pipeline of Banzai atoms. A program, if accepted by the compiler, is guaranteed to run at line rate with *any* traffic workload. The authors show how Domino can describe a number of data plane algorithms that can be then mapped to Banzai. However not all algorithms can be executed with Banzai, some will be rejected by the compiler. Indeed, Banzai is based on a very conservative *all-or-nothing* model: programmability is limited only to those atoms that can read-modify-write state in only 1 clock cycle, i.e. without any feedback-loop. Clearly, the advantage of this approach is that there is no need to implement *blocking* state arbitration mechanisms like in OpenState or OPP. However, due to silicon technology constraints, there will always be instructions (atoms) that cannot be executed in 1 clock cycle. This is especially true at the hardware clock frequencies necessary for terabit switches, usually 1 GHz. It is hard to meet hardware timing for some instructions. The same authors show how algorithms like CoDel [51] cannot be supported as they require a square root operation, which is hard to implement in 1 clock cycle at 1 GHz.

The OPP abstraction is fundamentally different because it offers a tradeoff instead of limiting programmability: more complex processing, i.e. complex ALUs in the update logic block, requires longer feedback-loops, which in *some cases* might not sustain line rate. However, simulation results in Section 2.3.3 show how the maximum time budget allowable greatly varies, up to 30 clock cycles (and more[15]) in normal traffic conditions, at the cost of added latency of few hundreds of nanoseconds. Banzai instead, impose a very conservative time

[15]We limited the maximum pipeline depth to 30 clock cycles.

budget for atoms of only 1 clock cycle.

Another limitation of Banzai, is that the model does not allow for shared-state between atoms. State is unique to each atom. The consequence is that Domino does not permit the definition of algorithms that require cross-flow state handling. As outlined by the same authors, this limitation is due to the fact that memories attached to atoms do not support multiple ports for read and writes (which instead OPP does), as this would require complex wiring given the numerous distinct memories (one per atom), hard to support at 1 GHz. This translates in the inability of a Domino programmer to implement a simple L2 learning switch. That is, the Domino compiler would reject such program.

Finally, having state attached to atoms can limit the size of the available memory. To support atomic read-modify-write operations, memories are required to run at the same frequency as the pipeline. For example, at 1 GHz only memories in the order of tens of megabytes can be supported. While in OPP the state table could be moved to a slower but larger memory block. The tax to pay in this case is a feedback-loop with several clock cycles. The value of the OPP abstraction is that it leaves freedom of choice in the tradeoff between performance, i.e. throughput, and capabilities, i.e. more complex computations or memory.

TPP. Jeyakumar et al. [52] propose a more radical approach where end hosts can embed small programs in packet headers that define processing in the switch data plane. Besides the concern of security (malicious end hosts that inject harmful programs), a very interesting aspect is the proposal of targeted ASIC implementations where an extremely small set of instructions and memory space can be used to define packet processing. The ASIC sequentially executes packet's instructions on a tiny CPU (TCPU). The TCPU implements a 5-stage pipeline, with memory reads and writes at different stages. TCPU does not enforce strict consistency guarantees, except for a specific `CSTORE` instruction that allows atomic state updates based on condition evaluation. The OPP update logic block allows for richer processing, while providing strict consistency guarantees.

EFSM. The usage of EFSMs was initially inspired by [53] where EFSMs were used to convey a desired medium access control operation into a specialized wireless interface card. While the abstraction (EFSM) is similar, the context (wireless protocols versus flow processing), technical choices (state machine execution engine versus table-based structures), and handled events (signals and timers versus header matching), are not comparable.

Other flexible switches. Another way of offering more flexible packet processing is that of using software router implementations [54, 55] or network processors [56]. However, these platforms are at least 10–100 times slower than programmable dedicated ASIC switches [17, 18, 19]. On the other end, FPGAs allow for hardware programmability, but they are significantly more expensive, while having lower performance than ASICs (lower total memory capacity, inefficient TCAMs simulation, and higher power consumption). For example, recent solution like the Xilinx Virtex-7 [57], while forwarding traffic at nearly 1 Tbit/s, they have a price of one order of magnitude above that of a programmable ASIC solution [58]. Both OpenState and OPP target dedicated ASIC implementations.

Other related work

Arashloo et al. [59] propose SNAP, a centralized programing model that permits the definition of stateful data plane algorithms at the network level, i.e. programs are written using a *one big switch* abstraction of the network, instead of dealing with *many* switches. A compiler then translates SNAP programs into switch configurations. OpenState and OPP could be used as switch target for SNAP.

Sharma et. al [11] analyze the case of stateful packet processing architectures that have limited state, support limited types of operations, and limit the type of per-packet computation, in order to forward packets at line rate. They provide approximation techniques for the implementations of some tasks/building blocks common in many data plane algorithms, such as metering, counting packets/bytes, slow arithmetic (e.g. multiplication or division), etc. These techniques permit to mask the aforementioned hardware limitations while guaranteeing accurate, but not perfect, results. OPP tackles a similar problem but from a different perspective. OPP design relaxes the line-rate constraints by allowing for more powerful processing that does not require approximation techniques. Simulation results show that line-rate can be achieved in most of the cases, but not always.

2.5 Conclusions

In this chapter, we presented abstractions for a programmable data plane specifically targeted for stateful processing, i.e. processing that requires to access and manipulate state maintained by the fast-path. Handling state in the fast-path, as opposed to that managed at the slower control plane (either centralized or switch local), permits this processing to happen at line rate throughput, i.e. 10-100 Gbit/s on 10-100 ports.

Abstractions can be categorized into two levels. At a higher level, we have Mealy Machines and EFSMs, two well-known abstractions in classical theory of state machines which turned out to be useful to describe formally packet processing tasks. At a second, lower level, we have abstract pipeline models for packet processing, OpenState and OPP, which also define the data plane API used by the network OS or end users to program packet processing tasks.

Mealy Machines and OpenState are useful abstractions because they show how the well-established OpenFlow match+action model can be easily extended to provide stateful capabilities. The OpenFlow's flow table itself can be used to describe a Mealy Machine entirely. This translates in a hardware implementation that minimally departs from TCAM-based OpenFlow pipelines, realized with commodity switch components such as a hash table with collision handling (in our case a d-left with $O(1)$ lookup/insert time) and a key extractor (as simple as shift-and-mask operations of the packet headers).

However, Mealy Machines suffer from a well-known state explosion problem, limiting the set of stateful tasks that can be described. As a rule of thumb, Mealy Machines, and hence OpenState, fits well for those algorithms that require a reasonably small number of states and transitions, those that can be contained in a TCAM, usually in the order of thousands.

EFSM is a more flexible model that extends and generalize Mealy Machines by adding custom registers, i.e. variables, and computational capabilities. EFSMs permits a more compact representation of Mealy Machines. The whole state space is partitioned in few symbolic state labels, and the evolution of the program can be described inside a symbolic state by means of arithmetic functions over registers. Clearly, this is a way more efficient representation than Mealy Machines, where one need to enumerate all possible transitions and states (recall the simple example of a packet counter).

OPP is an abstract pipeline model for packet processing that permits the execution of an EFSM. OPP extends and generalizes OpenState by adding two more stages to the pipeline: a condition block and an update logic block. The latter is a crucial piece of the whole architecture, as it ultimately define which application can run in the fast-path. Essentially this block is the *flow processor* of our system. We presented an instance of such processor

designed for the OPP FPGA-based prototype, which allows for up to five parallel ALUs, i.e. five parallel computations, where each ALU supports the basic instructions of a RISC architecture plus three more advanced instructions (*avg*, *var*, *ewma*). We do not claim this instruction set is the right one. Indeed we show how that is not suitable for at least one of the OPP programming examples provided, i.e. rate estimation. Finding an architecture for this processor that is expressive enough to allow the execution of a broad range of algorithms is outside the scope of this thesis work.

However, we argued that OPP provides a convenient framework where to research and experiment with such flow processor. OPP provides a structured way of managing different kinds of state: per-flow, global and per-packet (i.e. headers and metadata) all fed to the flow processor. Using a feedback-loop in the OPP pipeline model does not put any constraint on the flexibility of the processor, but it can limit throughput. The length of the feedback-loop (expressed in clock cycles) defines the tradeoff between programmability and performance.

This tradeoff is well known in computing architectures. It is well exemplified by the proliferation of specialized processors in domains where general purpose CPUs cannot offer the desired performance requirements, e.g. GPUs, DSPs, TPUs, etc. When it comes to packet processing, most of the related work that we presented focuses on architectures that privileges line rate guarantees at all the time, instead of enhancing computational capabilities or available memory for state. In this chapter we discussed ways to favor programmability, with good, predictable, but *not guaranteed*, throughput.

OPP's feedback-loop allows for concurrent reads and writes of the state memory at different stages of the pipeline. If not controlled, e.g. by stalling the pipeline when needed, such feedback-loop can cause inconsistencies in the processing of packets. When inconsistencies are really an issue is an open discussion, that we addressed only marginally. While this discussion evolves, it seemed necessary to propose a scheme to prevent this issue and evaluate its impact on throughput.

We proposed to perform memory locking using few queues and a scheduler that admits only one packet per flow in the pipeline. This avoids two consecutive packets of the same flow to access the same entry in the state label concurrently. In the case of such an event, the second packet would have to wait in line until its traveling companion has exited the feedback-loop (and potentially updated the state). However, packets of different flows can be processed in parallel without risk of inconsistencies as they access different portions of the memory.

In the end, we have made OPP a blocking architecture, i.e. an architecture which throughput depends on the traffic workload, the worst case being traffic made of all packets of the same

flow, all minimum size and arriving back-to-back[16]. By running simulations on real traffic traces, we showed how both the actual risk of incurring in data hazards, and the throughput of the memory locking scheme are highly variable. They depend on the level of flow aggregation (i.e. the key scopes) and the traffic characteristics (packet size and flow distribution). Results show that in most of the cases the time budget available to the feedback-loop is of several cycles, between 9 and 30 (forced cap) for 5-tuples, causing throughput degradation smaller than 1%, with 100% switch capacity utilization, i.e. a very uncommon near-worst scenario. Similarly, results suggest that global state updates should be handled in no more than 1-2 clock cycles to sustain high throughput. This explains the decision in OPP of allowing only atomic operations on the global state.

2.5.1 Future work

Flow processor design. Simulation results show that operations on global registers should be performed atomically, i.e. in 1 clock cycle as in Banzai [35]. However, processing flow states has more relaxed constraints. For example a reasonable time budget when processing distinct 5-tuples is between 9 and 30 clock cycles. What can we do with so many cycles? Is it enough? Do we need more? We leave for future work the answer to this question. Ideally, we should come up with at least one design of a good enough flow processor, i.e. the update logic block in OPP, that makes the best use of this budget, providing the most flexible instruction set capable of executing a large number of processing tasks.

Programming language for OPP. Programming functions in OPP is hard. Or at least, it is not straightforward as it could be with a high-level language. We believe that approaches such as Domino [35] are very promising to address the issue. Introducing support for specifying flow-level consistency requirements in such languages is indeed an interesting area for future research. It is worth mentioning that the original paper on EFSMs [60] presents ways of generating an EFSM description starting from a from a high-level C-like program. Generating one EFSM is relatively trivial, however, some tasks might require the pipelining of multiple OPP stages, hence the sequential execution of multiple EFSMs. It is unclear how an high-level algorithm could be split into multiple EFSMs.

Explore other implementation options. While we leveraged an FPGA-based hardware implementation of OPP (with a sketch of its performance in ASIC) it would be interesting to verify how different architectures would implement the OPP abstraction. For instance,

[16] In which case throughput would be scaled by a faction of $1/N$ where N is the length of the feedback-loop

using NPUs, GPUs or even software running on general purpose CPUs. Especially for the last case, while OPP proved to work well in heavily pipelined architectures, it is unclear if such pipeline model could be efficiently implemented in software, e.g. by pipelining multiple CPUs, or by intelligently steering traffic (looking at the flow key) between parallel CPUs. The applicability of OPP with different architectures is an interesting area for future research.

CHAPTER 3 APPLICATION: FAULT RESILIENCE

3.1 Introduction

Fault resilience mechanisms are among the most crucial traffic engineering instruments in operator networks since they insure quick reaction to connectivity failures with traffic rerouting.

So far, traffic engineering applications for SDN, and failure recovery solutions in particular, have received relatively little attention from the research community and networking industry which has focused mainly on other important areas related to security, load balancing, network slicing and service chaining. Not surprisingly, while SDN is becoming widely used in data centers where these applications are crucial, its adoption in operator networks is still rather limited. The support in current SDN implementations of features for failure recovery is currently rather weak and traditional technologies, e.g. Multi-Protocol Label Switching (MPLS) Fast Reroute, are commonly considered for carrier networks more reliable.

Most recent versions of OpenFlow include a mechanism, namely Fast-failover, for allowing quick and local reaction to failures without the need to resort on a central controller. OpenFlow Fast-failover group works only when a local alternative path is available from the switch that detected the failure. Unfortunately, such an alternative path may not be available, in which case the intervention of a controller, which reachability is not guaranteed, is required in order to establish a rerouting at another point in the network. Failure detection and recovery can be better handled locally in the fast-path assuming different sets of forwarding rules that can be applied according to the observed network state.

We propose Spider[1], a packet processing pipeline design based on OpenState, that allows the implementation of failure recovery policies with fully programmable detection and rerouting mechanisms directly in the switches' fast-path. Spider is inspired by well-known legacy technologies such as Bidirectional Forwarding Detection (BFD) [61] and MPLS Fast Reroute (FRR) [62]. Differently from other OpenFlow-based solutions, detection and rerouting in Spider are implemented entirely in the data plane, with no need to rely on the slower control plane. As such, Spider allows very short recovery delays (< 1 ms), with a configurable tradeoff between overhead and failover responsiveness.

The chapter is organized as follows. In Section 3.2 we discuss related work on fault resilience in SDN. In Section 3.3 we introduce the Spider approach, outlining its pipeline design and prototype implementation in Section 3.4. We provide experimental results in Section 3.5. In

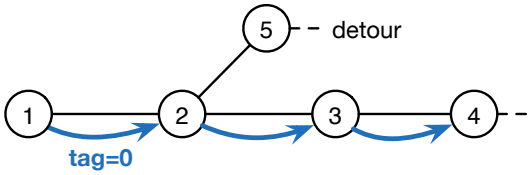
[1] **S**tateful **P**rogrammable fa**I**lure **DE**tection and **R**ecovery

Section 3.6 we discuss Spider w.r.t. legacy technologies and current SDN platforms. Section 3.7 concludes the chapter.

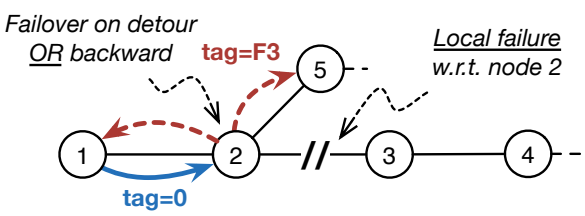
3.2 Related work on fault resilience in SDN

The concern of quickly recovering from failures in SDN has been already explored by the research community with the general goal of making SDN more reliable by reducing the need of switches to rely on the external controller to establish an alternative path. Sharma et al. in [63] shows how hard it is to obtain carrier grade recovery times (<50ms) when relying on a controller-based restoration approach in large OpenFlow networks. To solve this problem, the authors propose also a proactive protection scheme based on a BFD daemon running in the switch and integrated with the OpenFlow Fast-failover group type, obtaining recovery times within 50ms. Similarly, Van Adrichem et al. shows in [64] how by carefully configuring the BFD process already compiled in Open vSwitch, it is possible to obtain recovery times of few ms. The case of protection switching is also explored by Kempf et al. in [65], here the authors propose an end-to-end protection scheme based on an extended version of OpenFlow 1.1 to implement a specialized monitoring function to reduce processing load at the controller. Sgambelluri et al. proposed in [66] a segment-protection approach based on pre-installed backup paths. Also in this case, OpenFlow is extended in order to enable switches to locally react to failures by auto-rejecting flow entries of the failed interface. The concern of reducing load at the controller is also addressed by Lee at al. in [67]. A controller-based monitoring scheme and optimization model is proposed in order to reduce the number of monitoring iterations that the controller must perform to check all links. A completely different and more theoretical approach based on graph search algorithms is proposed by Borokhovich et al. in [68]. In this case the backup paths are not known in advance, but a solution based on the OpenFlow fast-failover scheme is proposed along an algorithm to randomly try new ports to reach traffic demands' destination. Similarly, McCauley et al. [69] propose AXE, a scheme entirely based on the data plane (via a P4-based implementation), to route packets in a L2 network using learning techniques, until they reach their destination. Interestingly, this approach can be used also in the face of failures. However, AXE does not say how failures are detected, instead it assumes the intervention of a separate, higher-level mechanism to declare a link as failed.

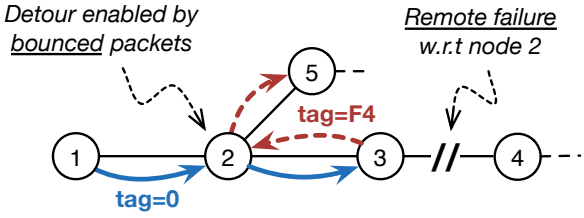
To the best of our knowledge, we are unaware of other prior work towards the use of programmable stateful data plane abstractions to implement both failure detection and recovery schemes directly in the fast-path.



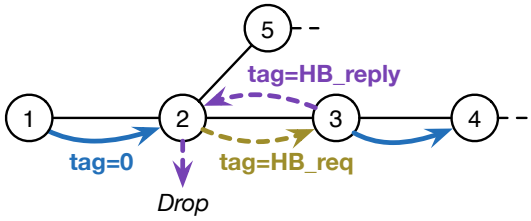
(a) Normal (no failures)



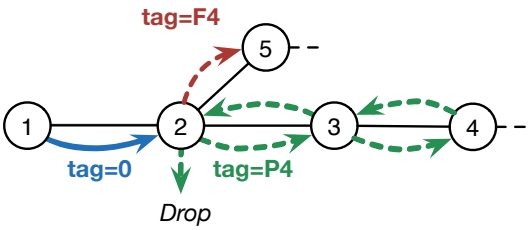
(b) Local failover



(c) Remote failover



(d) Heartbeat request/reply



(e) Path probing

Figure 3.1 Example of the different forwarding behaviors implemented by Spider.

3.3 Spider approach sketch

Spider provides mechanisms to perform failure detection and instant rerouting of traffic demands using a stateful proactive approach, without requiring the intervention of the controller. Interaction with the controller is needed only at boot time to provision switches' state tables and to fill flow tables with the different forwarding behaviors. No distributed protocols are required, instead the different forwarding behaviors are handled at the data plane level by labeling packets with special tags and by using the stateful primitives of OpenState. The features implemented by Spider are inspired by well-known legacy protocols such as BFD and MPLS FRR, in Section 3.6 we discuss more about the design of Spider w.r.t. these legacy technologies.

Backup path pre-planning. Spider does not distinguish between node or link failures, instead it defines with `Fi` a particular failure state of the network for which node i is unreachable. Given another node j, we can refer to the case of a "local" failure, when j is directly connected (1 hop) to i, otherwise a "remote" failure when node i is not directly connected to j. In Spider, the controller must be provided with the topology of the network and a set of primary paths and backup paths for each demand. Backup paths must be provided for each possible `Fi` affecting the primary path of a given demand. A backup path for state `Fi` can share some of the primary path, but it is required to offer a detour (w.r.t primary path) around node i. In other words, even in the case of a link failure making i unreachable from j, and even other links to j might exist, Spider requires that backup paths for `Fi` cannot use any of the links belonging to i. The reason of such a requirement is that, to guarantee very short ($< 1ms$) failover delays, a characterization of the failure, i.e. understanding if it is a node or a link failure, is not possible without the active involvement of the controller or other type of slow signaling. For this reason Spider assumes always the worst case where node i is down, hence it should be completely avoided. An example of problem formulation that can be used to compute an optimal set of such backup paths has been presented in [70]. Finally, if all backup paths are provided, Spider guarantees instantaneous protection from every single-failure `Fi` scenario, without requiring the controller to compute an alternative routing or to update flow tables. However, the unfortunate case of a second or multiple failures happening sequentially can be supported through the reactive intervention of the controller.

Failure detection. Spider uses tags carried in an arbitrary header field (e.g. MPLS label or VLAN ID) to distinguish between different forwarding behaviors and to perform failure

detection and switch-to-switch failure signaling. Figure 3.1 depicts the different forwarding scenarios supported by Spider. When in normal conditions (i.e. no failures), packets entering the network are labeled with `tag=0` and routed through their primary path (Figure 3.1a). To detect failures, Spider does not rely on any switch-dependent feature such OpenFlow's Fast-failover, instead it provides a simple detection scheme based on the exchange of bidirectional "heartbeat" packets. Spider assumes that as long as packets are received from a given port, that port can be also used to reliably transmit other packets. When no packets are received for a given interval, a node can request its neighbor to send an heartbeat. As shown in Figure 3.1d, heartbeat can be requested by labeling any data packet with `tag=HB_req`. A node receiving such a packet will perform 2 operations: i) set back `tag=0` and transmit the packet towards the next hop and ii) create a copy with `tag=HB_reply` and send it back on the same input. In this way, the node that requested the heartbeat will know that its neighbor is still reachable. Heartbeat are requested only when the received packet rate drops below a given threshold. If no packets (either data or heartbeat) are received for more than a given timeout, the port is declared `DOWN`. The state of the port will be set back to `UP` as soon as packets will be received again on that port.

Fast reroute. When a port is declared `DOWN`, meaning a local failure situation towards a neighbor node i, incoming packets are labeled with `tag=Fi` and sent to an alternative port (Figure 3.1b), this could be a secondary port belonging to a detour or the same input port where the packet was received. In the last case we can refer to a "bounced" packet. Bounced packets are used by Spider to signal a remote failure situation. Indeed, they are forwarded back along their primary path until they reach a node able to forward them along a detour. In Figure 3.1c, when node 2 receives a bounced packet with `tag=F4`, it updates the state of that demand to `F4` and forwards the packet along a detour. Given the stateful nature of Spider, state `F4` is maintained by node 2, meaning that all future packets of that demand with `tag=0`, will be labeled with `tag=F4` and transmitted directly on the detour. In the example, we can refer to node 2 as the "reroute" node of a given demand in state `F4`, while the portion of the path comprised between the node that detected the failure and the reroute node is called the "bounce path".

Path probing. Failures are temporary, for this reason Spider provides also a probe mechanism to establish the original forwarding as soon as the failure is resolved. When in state `Fi` the reroute nodes periodically generate probe packets to check the reachability of node i. As for heartbeat packets, probe packets are not forged by switches or the controller, instead, they are generated simply duplicating and labeling the same data packets processed by a

reroute node. In Figure 3.1e, node 2 duplicates a `tag=0` packet. One copy is sent on the detour with `tag=F4`, while the other is labeled with `tag=Pi` and sent on the original primary path. If node i becomes reachable again, it will bounce the probe packet towards the reroute node. The reception of a probe packet `Pi` from a node with a demand in state `Fi` will cause a state transition that will re-enable the normal forwarding on the primary path.

Flowlet-aware failover. Spider also addresses the issue of packet reordering that might occur during the remote failover. Indeed, in the example of Figure 3.1c, while new `tag=0` packets arrive at the reroute node, one or more (older) packets may be traveling backward on the bounce path. Such a situation might cause packets to be delivered out-of-order at the receiver, with the consequence of unnecessary throughput degradation for protocols such as TCP. For this reason Spider implements a flowlet-aware switching scheme [23]. While Spider is already aware of the failure, the same forwarding decision is maintained for packets belonging to the same burst; in other words, packets are still forwarded (and bounced) on the primary path until a given idle timeout (i.e. interval between bursts) is expired. Such a timeout can be evaluated by the controller at boot time and should be set as the maximum RTT measured over the bounce path of a given reroute node for state `Fi`. Effectively waiting for such an amount of time before enabling the detour, maximizes the probability that no more packets are traveling back on the bounce path, thus minimizing the risk of mis-ordered packets at the receiver.

3.4 Implementation

In this section we present the design of the pipeline and the configuration of the flow tables necessary to implement Spider. The pipeline (Figure 3.2) is based on 4 different flow tables. An incoming packet is first processed by table 0 and 1. These two blocks perform only stateless forwarding (i.e. legacy OpenFlow), which features will be described later. The packet is then processed by stateful tables 2 and 3. These tables implement respectively the Remote Failover FSM, and the Local Failover FSM described later. Packets are always processed by table 2 which is responsible for rerouting packets when the primary path of a given demand is affected by a remote failure. If no remote failure has been signaled to table 2, packets are submitted to table 3 which handles the failover in the case of local failures (i.e. directly seen on local ports). State updates in table 2 are triggered by bounced packets, while table 3 implements the heartbeat-based detection mechanisms introduced in Section 3.3. Although table 1 is stateless and for this reason doesn't need to maintain any state, it is responsible for triggering state updates on tables 2 and 3. State updates from one table

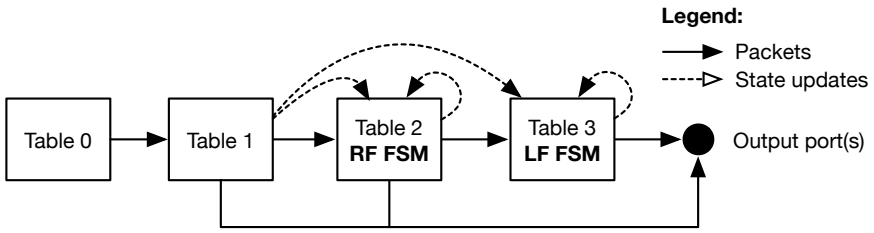


Figure 3.2 Spider pipeline architecture.

to another further in the pipeline are possible by using packet metadata fields. Indeed, one table can set a special value that if matched in the second table will trigger a state transition.

Table 0. It performs the following stateless processing before submitting packets to table 1:

- For packets received from an edge port (i.e. directly connected to a host): push an initial MPLS label to store the tag.
- For packets received from a transport port (i.e. connected to another switch): write the input port in the metadata field (used later to trigger state updates from table 1).

Table 1. It handles the processing of those packets which requires only stateless forwarding, i.e. which forwarding behavior doesn't depend on states:

- Data packets received at an edge port: set `tag=0` , then submit it to the next table.
- Data packets received at the last node of the primary path: pop the MPLS label, then directly transmitted on the corresponding output port (where the destination host is located).
- Packets with `tag=Fi`: directly transmitted on the detour port (unique for each demand and value of `Fi`); set `tag=0` on the last node of the detour before re-entering the primary path. An exception is made for the reroute node of demand in state `Fi`, in this case the routing decision for these packets is stored in table 2.
- Heartbeat requests (`tag=HB_req`): packets are duplicated, one copy is set with `tag=HB_reply` and transmitted through the input port, the other is set with `tag=0` and then submitted to the next table.
- Heartbeat replies (`tag=HB_reply`): dropped (used only to update the state on table 3).

- Probe packets (`tag=Pi`): directly transmitted on the corresponding output port belonging to the probe path (i.e. the primary path, unique for each demand and value of `Pi`) (e.g. Figure 3.1e).

Finally, table 1 performs the following state updates on table 2 and 3:

- For all packets: a state update is performed on table 3 so to declare the port on which the packet has been received as `UP`.
- Only for probe packets: a state update is performed on table 2 to transition a flow state from `Fi` to `Normal`.

Table 2 (Remote Failover FSM). Figure 3.3 shows a simplified version of the FSM. A state is maintained for each different traffic demand served by the switch. As outlined by the lookup and update scopes, in this case the origin-destination demands are identified by the tuple of Ethernet source and destination address, a programmer might specify different aggregation fields to describe the demands (e.g. IP source/destination tuple, or the 4-tuple transport layer protocol). Given the support for only single-failure scenarios, transitions between macro states `Fi` are not allowed (state must be set to `Normal` before transitioning to another state `Fi`). Figure 3.4 depicts a detailed version of the Remote Failover FSM with macro state `Fi` exploded. At boot time the state of each demand is set to the default value `Normal`. Upon reception of a bounced packet with `tag=Fi`, the latter is forwarded on the detour and state set to `Fault signaled`. The flowlet-aware routing scheme presented before, is here implemented by means of state timeouts. When in state `Fault signaled`, packets arriving with `tag=0` (i.e. from the source node) are still forwarded on the primary path. This behavior is maintained until the expiration of the idle timeout δ_1, i.e. after no packets of that demand have been received for a δ_1 interval, which should be set equal to the RTT

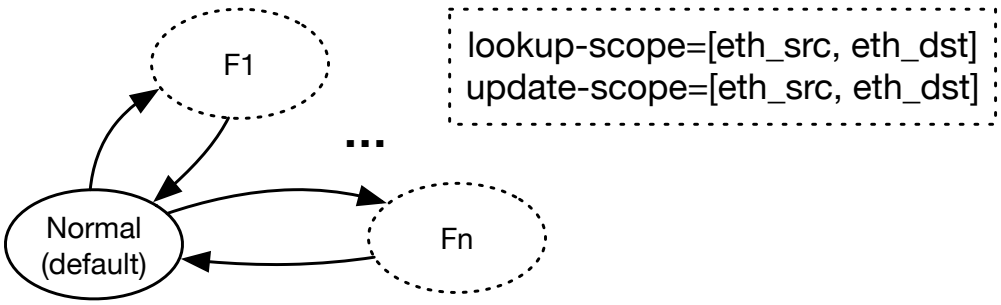


Figure 3.3 Macro states of the Remote Failover process.

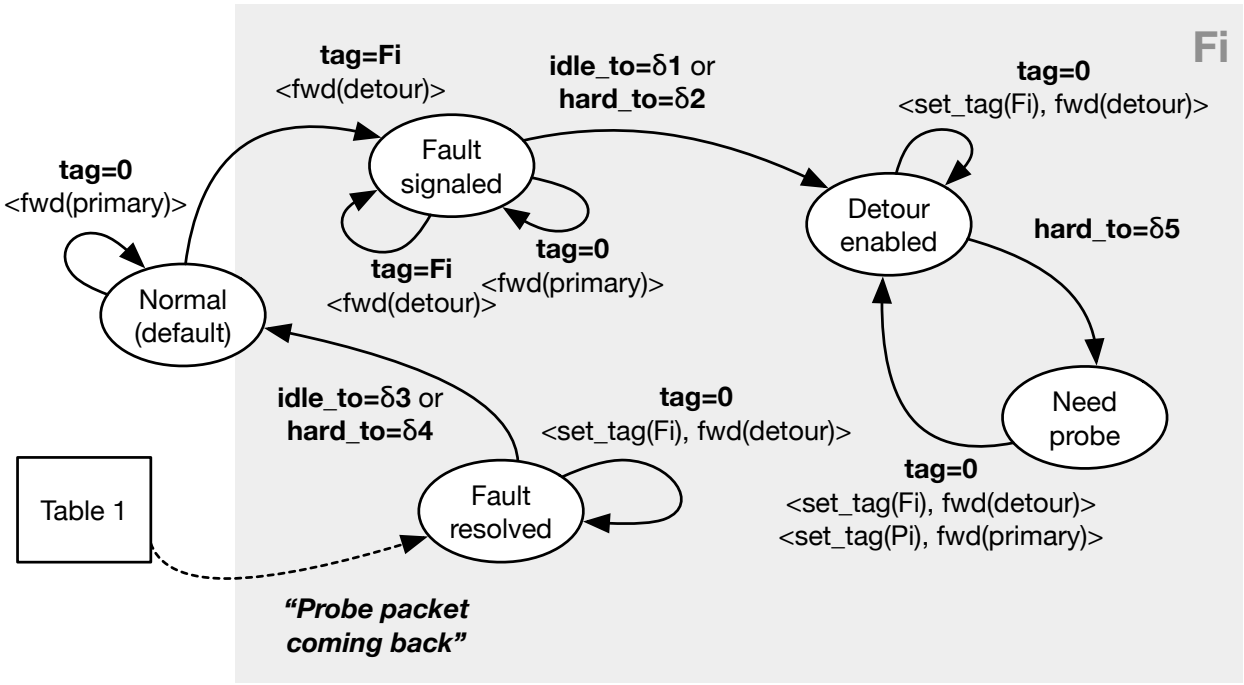


Figure 3.4 Detail of the macro state `Fi` for the Mealy Machine of the Remote Failover process.

measured on the bounce path[2].To avoid a situation where the demand remains locked in state `Fault signaled`, a hard timeout $\delta_2 > \delta_1$ is set so that the next state `Detour ready` is always reached after at most a δ_2 interval. When in state `Detour enabled`, packets are set with `tag=Fi` and transmitted directly on the detour. In this state a hard timeout δ_5 assures the periodic transmission of probe packets on the primary path. The first packet matched when in state `Need probe` is duplicated: one copy is sent on the detour towards its destination, another copy is set with `tag=Pi` and sent to node i through the original primary path of the demand. If node i becomes reachable again, it responds to the probe by bouncing the packet (`tag=Pi` is maintained) to the reroute node that originated it. The match of the probe packet at table 1 of the reroute node will trigger a reset of the Remote Failure FSM to state `Fault resolved`. When in state `Fault resolved`, the same flowlet-aware routing scheme of state `Fault signaled` is applied. In this case an idle and hard timeout are set in order to maintain the alternative routing until the end of the current burst of packets. In this case δ_3 must be set to the maximum delay difference between the primary and the backup path. After the expiration of δ_3 or δ_4, the state is set back to `Normal`, hence the transmission on the detour stops and packets are submitted to table 3 to be forwarded on their primary port.

[2]Assuming the OpenState target hardware supports state timeouts with microseconds or less resolution.

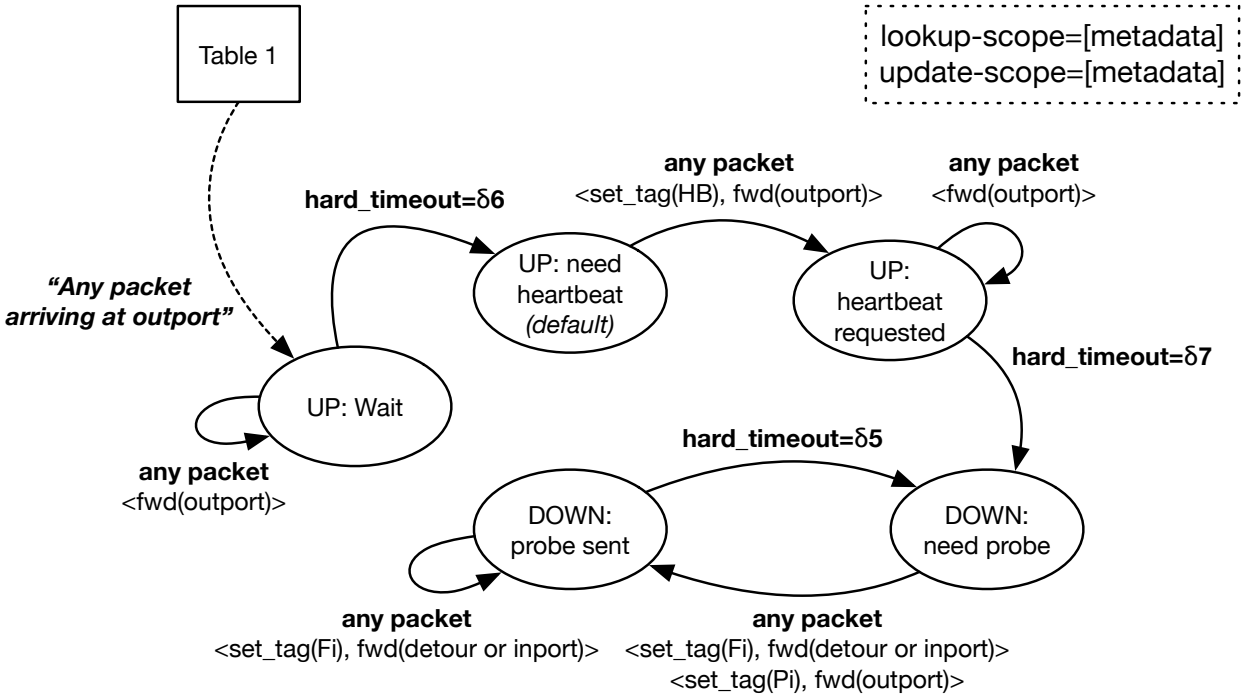


Figure 3.5 Mealy machine of the Local Failover process.

Table 3 (Local Failover FSM). Figure 3.5 depicts the FSM implemented by this table. Here flows are aggregated per output port (encoded in the metadata field)[3] , meaning that all packets destined to the same port will share the state. This FSM has two macro states, namely `UP` and `DOWN`. When in state `DOWN`, packets are forwarded to an alternative port (belonging to a detour or to the input port in case of bounced packets, according to the pre-planned backup strategy). At boot time all flows are in default state `UP: need heartbeat`, meaning that a heartbeat packet must be generated and a reply received, so that the port keeps being declared UP. Indeed, the first packet matched in this state will be sent with `tag=HB_req` and the state updated to `UP: heartbeat requested`. While in this state, packets will be transmitted on the primary output port, until a hard timeout δ_7 expires, in which case the port will be declared `DOWN`. The timeout δ_7 represents the maximum interval admitted between the generation of the heartbeat request and the reception of the corresponding reply. Every time a packet (either a data, probe or heartbeat) is received at table 1 the state of that port is reset to `UP: wait`. The Local Failover FSM will stay in this state for an interval δ_6 (hard timeout), after which the state will be set back to `UP: need heartbeat`. Hence, δ_6 represents the inverse of the minimum received packet rate required

[3]In the prototype OpenState implementation of Spider based on OpenFlow 1.3, matching on the outport is not supported, for this reason we can use the metadata field to carry this information across tables.

for a given port to avoid the generation of heartbeats. If the timeout δ_7 expires, the port is declared `DOWN`. Here, packets will be tagged with `Fi` (where i is the node directly connected through the port) and forwarded on an alternative port. Similarly to the Remote Failover FSM, a hard timeout δ_5 assures that probe packets will be generated even when the port is declared `DOWN`.

In conclusion, Table 3.1 summarizes the different timeouts used in Spider. We emphasize how, by tweaking these values, a programmer can explicitly control and impose i) a precise detection delay for a given port ($\delta_6 + \delta_7$), ii) the level of traffic overhead caused by probe packets of a given demand (δ_5 and δ_6), the risk of packets reordering in the case of a remote failover (δ_1, δ_2, δ_3, and δ_4). Experimental results based on these parameters are presented in the following section.

OpenState-based prototype

A prototype implementation of Spider is available at [71]. It uses a modified version of the OpenFlow Ryu controller [8] extended to support OpenState [72]. For the experimental performance evaluation we used Mininet [73] with a version of the CPqD OpenFlow 1.3 softswitch [74] as well extended with OpenState support.

3.5 Performance evaluation

Flow entries analysis

While detection and recovery times are guaranteed and topology-independent, a potential barrier for the applicability of the solution is represented by the number of flow entries, which can be limited by the switch memory and depends on the network topology. We can evaluate the resources required by a switch to implement Spider in terms of flow table entries and memory required for flow states. Let us start by defining as D the maximum number of demands served by a switch, F the maximum number of failures that can affect a demand (i.e. length of the longest primary path), and P the maximum number of ports of a switch. We can easily model the number of flow entries required by means of Big-O notation as $O(D \times F)$. Indeed, for table 0 the number of entries is equal to P; for table 1 in the worst case we have one entry per demand per fault ($D \times F$); for table 2 we always have exactly $7 \times D \times F$, and for table 3 exactly $P \times (3 + 2 \times D)$. In total, we have a number of entries order of $P + D \times F + D \times F + D \times P$ and then of $D \times F + D \times P$. Assuming $F >> P$ we can conclude that the number of entries is $O(D \times F)$.

Table 3.1 Summary of the configurable state timeouts of the Spider pipeline

Timeout	**Type**	**Description**	**Value**
δ_1	Idle	Flowlet idle timeout before switching packets from the primary path to the detour	Maximum RTT measured on the bounce path for a specific demand and `Fi`
δ_2	Hard	Maximum interval admitted for the previous case before enabling the detour	$> \delta_1$
δ_3	Idle	Flowlet idle timeout before switching packets from the detour to the primary path	Maximum end-to-end delay difference between the backup path and the primary path
δ_4	Hard	Maximum interval admitted for the previous case before re-enabling the primary path	$> \delta_3$
δ_5	Hard	Probe generation timeout	Arbitrary interval between each periodic check of the primary path in case of remote failure
δ_6	Hard	Heartbeat requests generation timeout	Inverse of the minimum rx rate for a given port before the generation of heartbeat requests and the corresponding replies
δ_7	Hard	Heartbeat reply timeout before the port is declared down	Maximum RTT for heartbeat requests/replies between two specific nodes (1 hop)

Table 3.2 Number of flow entries per node.

Net	**D**	**E**	**C**	**min**	**avg**	**max**	$E^2 \times N$
5x5	240	16	9	443	775	968	6400
6x6	380	20	16	532	1115	1603	14400
7x7	552	24	25	795	1670	2404	28224
8x8	756	28	36	1069	2232	3726	50176
9x9	992	32	49	1368	2884	4509	82944
10x10	1260	36	64	1188	3584	6153	129600
11x11	1560	40	81	1409	4249	7558	193600
12x12	1892	44	100	1185	5124	9697	278784
13x13	2256	48	121	2062	6218	11025	389376
14x14	2652	52	144	1467	7151	15436	529984
15x15	3080	56	169	3715	8461	16347	705600

If we want to evaluate the complexity according to network size, we can observe that in the worst case $F = N = E + C$, where N is the number of nodes, E edge nodes and C core nodes. Assuming a protection scheme that uses disjoint paths, which is the most demanding in terms of rules since all `Fi` states are managed by the ingress edge nodes, and a full traffic matrix, we have $D = E(E-1) \approx E^2$. In the worst case we have a single node managing all faults of all demands, where the primary path of each demand is the longest possible, thus $F = N$. In this case the number of entries will be $O(E^2 \times N)$.

In Table 3.2 we report the values for grid networks $n \times n$ where edge nodes are the outer ones of the grid and there is a traffic demand for each pair of edge nodes. In addition to the $O(E^2 \times N)$ values, we include in the table the values per node (min, max, average) calculated for the case of end to end protection where the primary path is the shortest (number of hops) and the backup path is the shortest node disjoint from the primary. The number of rules is generated according to the Spider implementation described in Section 3.4 and available at [71]. We can observe that even the max value is always much smaller than the values estimated by the complexity analysis, moreover, we can safely say that these are reasonable numbers well below the capabilities of programmable chips such as RMT. Obviously, for more efficient protection schemes based on a distributed handling of states Fi (e.g. segment protection), we can expect an even lower number of rules per node.

As far as the state table is concerned, table 2 for node n needs D_n entries, where D_n is the number of demands for which n is a reroute node. For the width of the table we need to consider the total number of possible states that is $1+4F_n$, where F_n is the number of remote failures managed by n. Similarly, for stage 3 there are only 5 possible states and a number of entries equal to P.

Detection mechanism

To evaluate the effectiveness of the Spider heartbeat-based detection mechanism, we have considered a simple experimental scenario of two nodes and a link with traffic of 1000 pkt/sec sent in one direction only. Figure 3.6 shows the number of packets lost after a link failure versus δ_6 (heartbeat interval) and δ_7 (heartbeat timeout). As expected, the number of losses decreases as the heartbeat interval and timeout decreases. In general, the number of dropped packets depends on the precise instant the failure occurs w.r.t. δ_6 and δ_7. The curves reported are obtained averaging the results of 10 different tries with failures reproduced at random instants.

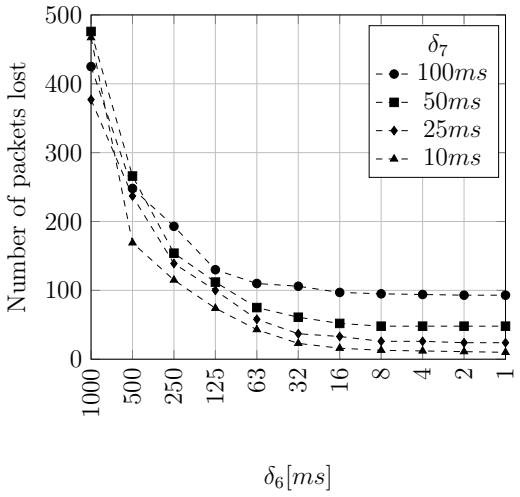


Figure 3.6 Packet loss (data rate 1000 pkt/sec)

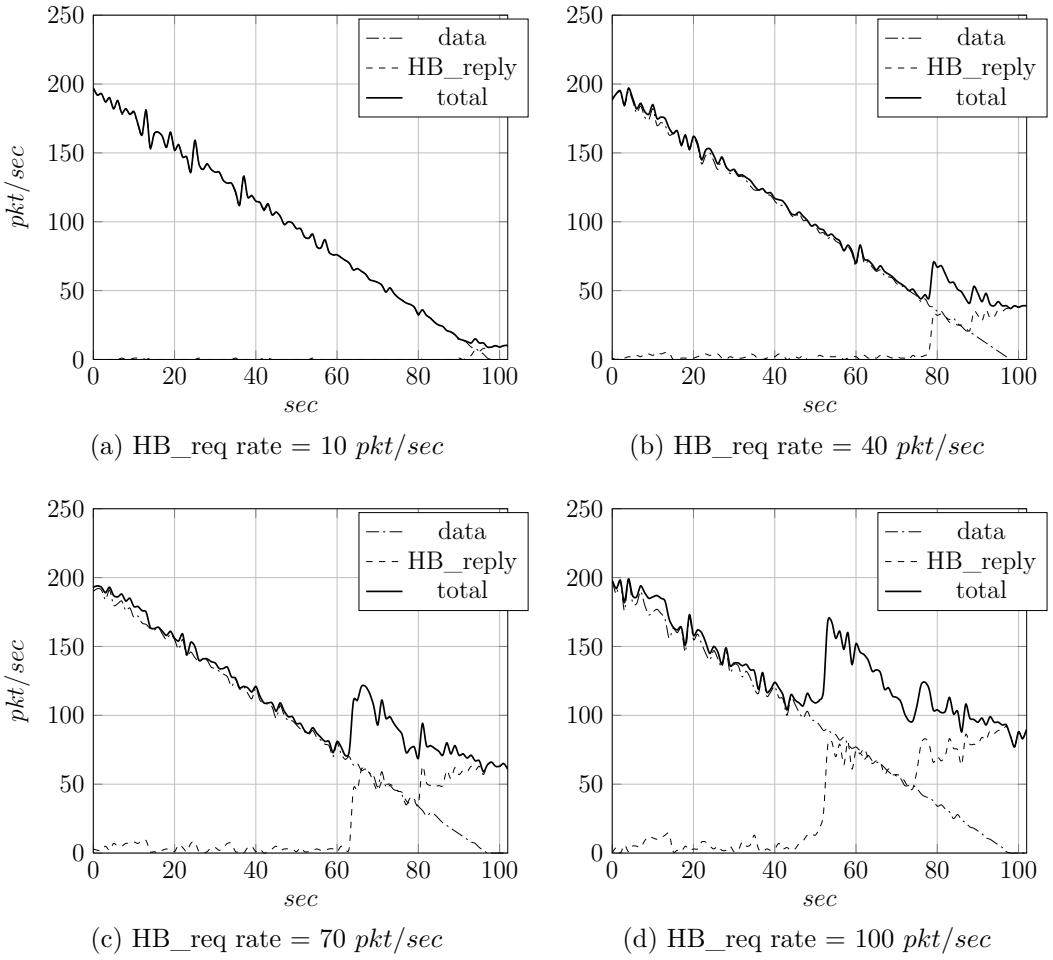
Overhead

Obviously, the price to pay for a small number of losses is the overhead due to heartbeat packets. However, Spider exploits the traffic in the reverse direction for failure detection, and this reduces the amount of heartbeat packets. For the same two nodes scenario in the previous section, we have evaluated the overhead caused when generating a decreasing traffic profile of 200 to 0 pkt/sec, with different values of δ_6. Results are reported in Figure 3.7.

We can see that, as long as the reverse traffic rate is higher than the heartbeat request rate ($1/\delta_6$), zero or low signaling overhead is observed. When the traffic rate decreases, the overhead due to heartbeats tends to compensate for the missing packets up to the threshold. However, this overhead does not really affect the network performance since it is generated only when reverse traffic is low.

Comparison with a reactive OpenFlow approach

We now compare a Spider-based solution with a strawman implementation corresponding to a reactive OpenFlow (OF) application able to modify the flow entries only when the failure is detected and notified to the controller. We have considered the network shown in Figure 3.8a. For the primary and backup paths, as well as the link failure indicated in the figure, we have considered an increasing number of demands with a fixed packet rate of 100 pkt/sec each one. For the OF case, we used the detection mechanism of the Fast-failover (FF) group type implemented by the CPqD softswitch, and different RTTs between the switch that detects the failure and the controller. For Spider we used a heartbeat interval (δ_6) of 2 ms and



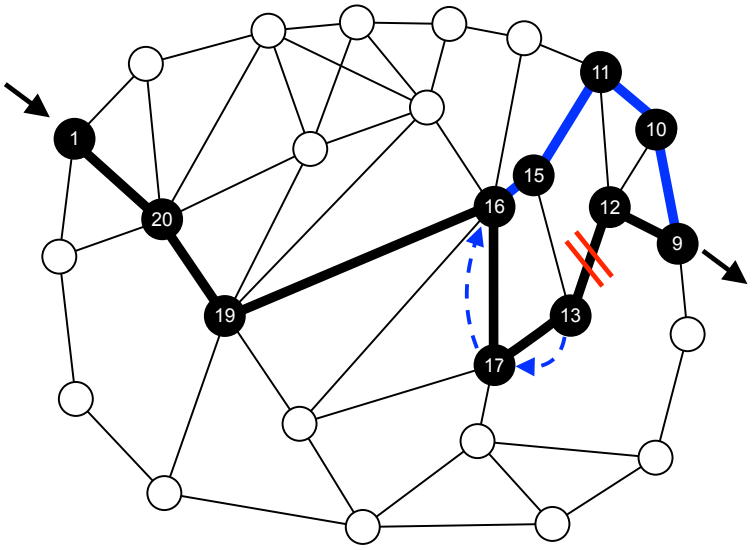
(a) HB_req rate = 10 pkt/sec

(b) HB_req rate = 40 pkt/sec

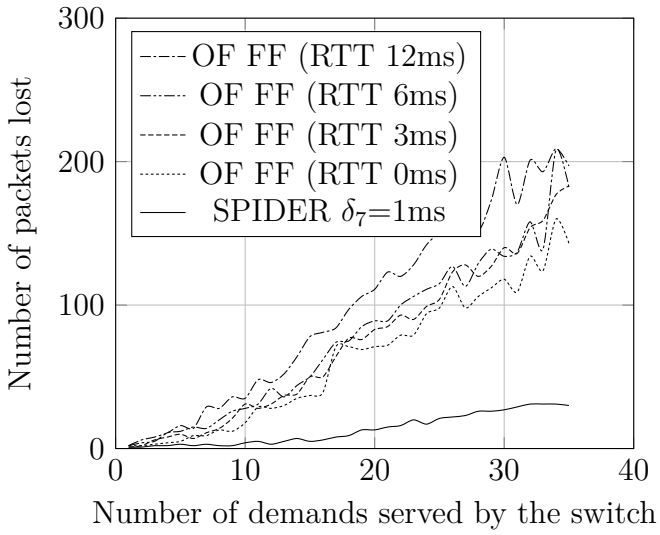
(c) HB_req rate = 70 pkt/sec

(d) HB_req rate = 100 pkt/sec

Figure 3.7 Heartbeat overhead with decreasing data traffic 200-0 pkt/sec and heartbeat request rates (inverse of δ_6) of 10, 40, 70, and 100 pkt/sec.



(a)



(b)

Figure 3.8 Comparison with OpenFlow: (a) test topology used in experiments and (b) number of packets lost

timeout (δ_7) of 1 ms. For all the considered flows, no local backup path is available: in the Spider case the network is able to autonomously recover from the failure by bouncing packets on the primary path, while in the OF case the controller intervention is needed to restore connectivity.

The results obtained are shown in Figure 3.8b. We can see that the losses with Spider are always lower than OF. Note that, even if the heartbeat interval used is small, this is not actually an issue for the network since in the presence of reverse traffic the overhead is proportionally reduced so that it never affects the link available capacity. The value of the timeout actually depends on the maximum delay for heartbeat replies to be delivered, which in high speed links mainly depends on propagation and can be set to low values by assigning maximum priority to heartbeat replies. In the case of OF, the number of losses increases as the switch-controller RTT increases. Obviously, losses also increase with the number of demands since the total number of packets received before the controller installs the new rules increases as well.

3.6 Discussion

3.6.1 Comparison with BFD

BFD [61] is a widely-used protocol to provide fast failure detection that is independent from the underlying medium and data protocol. When using BFD, two systems, i.e. forwarding entities, establish a session where control packets are exchanged to check the liveness of the session itself. In the common case the session to be monitored represents a bidirectional link, but it could also be a multi-hop path. The main mode of detecting failures in BFD is called *Asynchronous Mode*, where a session endpoint sends BFD packets at a fixed rate. A path is assumed to have failed if one system stops receiving those packets for a given detection timeout. Both packets send rate and detection timeout can be enforced by a network administrator to produce short (in the order of μs[4]) guaranteed detection delays. Optionally, an endpoint can explicitly request the other to activate/deactivate transmission of control packets using the so called *Demand Mode*. In both modes, the ability of a party to detect a failure depends on the ability of the device-local control plane to keep track of the elapsed time between the received control packets, and hence on the liveness of the control plane itself. For this reason, a third way of operation, namely the *Echo Function* is defined in order to test the forwarding plane of a device. When using this function, special Echo packets are emitted at arbitrary intervals by the control plane of one of the two parties,

[4]The BFD specification at [61] defines timestamps with μs granularity

with the expectation to have these packets looped-back by the forwarding plane of the other endpoint.

In the context of SDN, the devices' control plane is separated and logically centralized at a remote, geographical distant location. Current SDN platforms [6, 7] already provide means of detecting failures that are similar to BFD's Asynchronous Mode, where specially forged packets are requested to be emitted by the remote controller from a specific device port (via OpenFlow packet-out) and expected to be received (via OpenFlow packet-in) by the adjacent node in a given interval. However, due to the latency and overhead of the SDN control channel, it is hard to guarantee the same short detection delays as in BFD.

Spider improves SDN by providing ways to detect failure without relying on the slow control channel. Indeed, in Spider, whose mode of operation based on heartbeats resembles the BFD's Echo Function, detection delays can be enforced by appropriately setting timeouts δ_6 and δ_7, which are unique for a given switch and port. Moreover, we argue that Spider represents an improvement over BFD. Indeed, Spider operations are performed solely on the fast-path, i.e. at TCAM speed, differently from a BFD implementation based on the slower, device-local control plane. As such, the minimum detection delay of a target implementing Spider depends for the most part on the timestamp granularity provided by the target and the propagation delay between two devices. The other general advantage of Spider over BFD is that it does not require the definition of a separate control protocol, rather the same data packets are recycled to piggy-back heartbeats using an arbitrary header field (a MPLS label in our prototype implementation).

Some disadvantages of Spider over BFD are:

- **Security:** BFD defines means to authenticate a session to avoid the possibility of a system to interact with an attacker falsely reporting session states. In other words, all control and echo packets that cannot be validated as coming from a safe source are discarded. On the contrary, Spider does not use any mechanism to check for the the validity of the tag carried by data packets. For this reason, Spider tags should be used only inside the same authoritative domain, dropping any incoming packet at the edge carrying any unexpected header, and controlling physical access to the network to prevent the intrusion of an attacker.

- **False positives**: BFD allows to prevent false positives (i.e. erroneously declaring a session down) by setting a minimum number of consecutive dropped packets before declaring the session down. In fact, in presence of transmission errors, some control packets might be unrecognized and echo packets not looped-back. On the contrary, in

Spider failure state for a port is triggered after the first missed heartbeat request, that could be caused by a corrupted heartbeat request, thus causing unnecessary fluctuation between the backup and primary path (due to the probe mechanism). A similar mechanism could be implemented by using OPP flow registers to keep count of the number of consecutive dropped packets. However, heartbeat packets are smartly requested only when input traffic is low. In a real network, we can expect that most of the time the traffic will be flowing in both directions of the same link, hence the transmission of heartbeat packets itself is a quite rare event.

- **Administrative down:** BFD allows a network operator to administratively report a link as down, e.g. for maintenance, thus triggering a fast reaction of the device. On the contrary, the implementation of Spider presented here, allows down state only as a consequence of a failure detection event. However, the implementation could be easily extended to accept an additional state both in the Local Failover and Remote Failover FSMs to declare a flow or port as affected by failure without triggering the periodic link probing process. In this case the controller should be able to directly add or replace an entry in the state table.

- **Down state synchronization:** In some cases, only 1 of the 2 directions of a link might break, an event that is common in fiber optics. When using Spider, the party which incoming direction is down will detect first the failure after the configured detection timeout, thus stopping sending traffic on that port, after which the other party will trigger the down state after another detection timeout, resulting in twice the time for the failover to take place. BFD instead, applies a mechanism for session state synchronization, such that when a first endpoint detects the failure, it notifies the other of the down event, in which case (if one direction of the two is still up) the other endpoint will immediately trigger the failover procedure. In this case, the LS FSM could be extended to emit such an extra signaling message (via the tagging of the first packet matching the `DOWN: need probe` state) and to trigger a forced down state upon receiving such a packet.

3.6.2 Comparison with MPLS Fast Reroute

Fast Reroute (FRR) [62] is a technique used in MPLS networks to provide protection of Label-switched Paths (LSP)in the order of tens of milliseconds. Similarly to Spider, backup LSPs are established proactively for each desired failure scenario, such that, when a router detects a failure on one of its local ports, it swaps the label at the top of the MPLS stack with the one of the detour LSP, forwarding the packet to an alternative port. Packets are

forwarded on a detour until they reach a merge point with the primary path, where the label is swapped back to the primary LSP. RSVP-TE signaling is used to establish backup LSPs between routers in a distributed fashion.

Differently from FRR, Spider does not need a separate complex signaling protocol (which is described in around 30 pages in the original FRR RFC [62]) to establish backup paths. Instead, computation and provisioning of both primary and backup paths is performed by the remote controller with all the benefits of the SDN logically centralized paradigm, such as access to a global topology graph and a centralized API to provision forwarding rules on switches. It must be noted that in the proposed prototype implementation of Spider, MPLS labels are used for the sole purpose of carrying failure tags `Fi`, and must not be confused with their role in LSPs as the only parameter of the router forwarding function. In fact, in Spider the forwarding function is independent on the data protocol and can be based on arbitrary header fields. For example, as in our implementation, the output port of each packet is decided looking at the 3-tuple comprising Ethernet source address, Ethernet destination address, and failure tag.

When an alternative path is not available from the node that detected the failure, Spider allows to bounce back packets on the primary path until they reach a predefined reroute node, in which case a detour path is enabled. A similar approach is implemented by FRR when used in combination with another RSVP-TE extension for crankback signaling [75]. Differently from Spider, data packets are dropped before the failure point, while a separate failure notification is sent back on the primary path. Signalization in Spider is performed using the same data packets, with the added benefit of avoiding dropping extra traffic, a feature particularly useful when dealing with geographical distant nodes (e.g. 100 Mbytes otherwise lost at 10 Gbit/s with 80 ms signalization latency).

3.6.3 Data plane reconciliation

A stateful data plane seems to disagree with the architectural principles of OpenFlow and SDN, where all the state is handled at the logically centralized control plane, so that devices do not need to implement complex software to handle state distribution. In fact, when dealing with legacy distributed protocols (e.g. OSPF), an important concern is how to handle state reconciliation, for example after a device reset or failure, in which case the state of the device (e.g. topology graph and link metrics in OSPF) might not be in sync with the rest of the network devices, causing loops or black holes.

Handling data plane reconciliation with OpenFlow is relatively easy given the stateless nature of the flow tables. Indeed, modern SDN platforms [6, 7] follow an approach were applications

operate on a distributed flow rule data store that is then used to keep the data plane in sync, for example periodically polling the devices' flow tables so that missing flow rules are re-installed and extraneous ones removed. Spider forwarding decisions are based not only on flow rules but also on flow states, maintained by the switch and updated as a consequence of packets and timeout events. From here the question if this additional state needs to be synchronized (e.g. notify the controller of every state transition) and re-applied during reconciliation.

We argue that the reliability of Spider operations does not require support for flow state synchronization and reconciliation. In other words, when not used, the flow states are guaranteed to converge to the expected value in relatively short time with no risk of traffic loops or black holes. In fact, the per-flow state maintained by both the Remote and Local Failover FSMs can be learned by observing the incoming traffic (tag value) and does not depend on other means of state distribution. Moreover, the correctness of a forwarding decision of a switch does not depend on the flow state of any other switch.

As an example, we can analyze the case of switch j implementing Spider being reset, e.g. the content of the state and flow table wiped out during a situation of remote failure, i.e. while serving some flows on a detour, and hence in macro-state `Fi` for the Remote Failover FSM. If we assume the controller is able to re-install the flow rules in a time shorter than the detection delay configured on the upstream switch connected to j, so that it will not generate a failure state `Fj` itself, we end up with switch j forwarding traffic according to an empty state table, i.e. all flows in default state for both the Remote and Local Failover FSM. In this case, when a packet of a traffic flow affected by the failure state `Fi` arrives, it will be initially forwarded as in `Normal` state on the primary path, meaning that the switch directly connected to the unreachable node i will bounce back the packet appropriately tagged with Fi, triggering a state transition to `Fault signaled` on switch j, and hence initiating the failover procedure. Similarly, if node j is affected by a local failure state `Fj`, resetting the state table of the Local Failover FSM will have as a consequence that packets will be forwarded according to the default state (`UP: need heartbeat`), initiating the failure detection procedure, finally converging to the expected failure state. The tax to pay in this case is a few more packets dropped, depending on the detection delay configured for that node. If the time to re-provision the switch configuration (flow tables) after a reset takes more than the detection delay, this situation can be interpreted as a multiple concurrent failure for which a rerouting of flows is required to be performed by the controller.

3.6.4 P4-based implementation

In order to prove the feasibility of the Spider pipeline design, an implementation of it is also provided using the P4 language. This implementation can be found at [76] and is based on `openstate.p4`, a library that can be re-used by other P4 programs to easily express stateful packet processing using an table-based abstraction equivalent to OpenState.

The P4 based implementation of Spider has been tested with the reference P4 software switch BMv2 [77]. In the following we discuss some concerns related to the feasibility of Spider and `openstate.p4` on a P4-based programmable target:

- **State table:** it is needed in order to maintain per-flow states, indexed according to a flow key that is extracted from each packet according to a given lookup-scope or update-scope, depending on the type of access performed (read or write). The state table can be implemented using P4's register arrays. Hash functions can be used to efficiently map flow keys to the limited number of memory cells. Obviously, when using hash functions the main concern is related to collisions, where multiple flows can end up sharing the same memory cell. In the case of Spider, collisions should be properly handled to avoid the situation of a flow being forwarded according to a failure state set by another flow. Such an issue can be solved either by defining a collision handling mechanism in P4 or by delegating such a function to an "extern" object. The latter is a mechanism introduced in the more recent versions of the P4 language that allows a programmer to reference target-specific structure, for example, a key-value store which uniquely maps flow keys to state values, transparently handling collisions. Instead, `openstate.p4` provides native support for a trivial collision handling scheme by implementing an hash table with chaining that allows a fixed number of key-value couples to share the same index. We do not provide any insight about the performances of the approach, rather we use it to prove to feasibility of Spider for a P4 target.

- **State timeouts:** the ability of Spider to detect failures depends on the ability to evaluate timeout events (e.g. no packets received for on a given port for δ_7 time). State timeouts in `openstate.p4` are implemented comparing the timestamp of incoming packets with the idle or hard timeout value stored in the state table. However, packets timestamping is not a feature supported by the P4 specification itself. In our implementation, we rely on the ability of the BMv2 target to add a timestamp metadata to incoming packets. Moreover, the failure detection delay depends on the timestamp granularity, for example a target offering seconds granularity will not be able to detect failures in less than a second.

3.7 Conclusions

In this chapter we presented Spider, a new approach to failure recovery in SDN that provides a fully programmable abstraction to application developers for the definition of the re-routing policies and for the management of the failure detection mechanism. The use of a stateful data plane abstraction, minimizes the recovery delay while guaranteeing the failover even when the controller is not reachable. We believe that the proposed approach can close one of the gaps between the required and supported features that at the moment are slowing down the adoption of SDN in carrier grade networks for telco operators.

Spider has been implemented using OpenState and P4. The prototype implementation has been used to validate the proposed scheme and to experimentally assess its basic performance in a few example scenarios. The results have shown the potential advantages of Spider with respect to fully centralized applications where the controller is notified of failure events and is required to modify all affected forwarding rules.

CHAPTER 4 APPLICATION: FAIR BANDWIDTH SHARING

4.1 Introduction

It has been reported that TCP traffic represents 80-90% of the packets and bytes flowing today in the Internet[78]. It follows that most of the traffic sources adapt their sending rate according to the perceived available bandwidth. Indeed, TCP is the instantiation of an important design choice that contributed to the success of the Internet: to leave congestion control to the end-systems, thus permitting a relatively simpler implementation of the interconnection devices. TCP rate control algorithms, such as Additive-Increase-Multiplicative-Decrease (AIMD), help in maintaining a fair allocation of network resources on a *per-flow* basis. In the simplest case of multiple TCP streams, all experiencing the same RTT and sharing the same FIFO queue, each flow tends to occupy the same portion of the link bandwidth [79].

However, relying only on end-systems to guarantee fairness is not enough due to ill-behaving users and issues intrinsic to TCP-like algorithms. Example of such situations of unfairness are: (i) applications that open a large number of parallel TCP connections, e.g. peer-to-peer, or that tweak TCP to get better performances; (ii) non-TCP-like protocols, i.e. protocols that do not respond to congestion signals such as drops, and (iii) the dependence of standard TCP to the round-trip times (RTT) [79].

For these reasons, most Internet service providers (ISPs) tend to enforce direct control over their customers, by throttling their traffic at the network edge, limiting the maximum bandwidth of each user to a feasible, but *static* network allocation. This approach allows ISP to leave their core and interconnections with other ISPs uncongested all the time. The downside of such static bandwidth allocation is that the excess bandwidth remains unused, even in common situations of very low usage, such at night.

Researchers have proposed solutions that enforce a more *dynamic* bandwidth allocation in the network interconnection devices. In these approaches, instead of capping the maximum sending rate at all times, network devices are able to redistribute the unused capacity (if any) to those users asking for more. The trick here is to design a bandwidth enforcement scheme that (i) guarantees that all users can obtain at least the level of service they paid for, i.e. minimum rate guarantees, and (ii) when unused capacity is available, that is shared by all users, with no one prevailing on others. Today such bandwidth sharing scheme should be performed on a link of 10-100 Gbit/s.

Fair Queuing (FQ) scheduling [80, 81] is the textbook approach to enforce almost perfect

fairness among different traffic sources, independently of the behavior of the end-hosts. A switch implementing FQ works by assigning users to different queues, where a user is an arbitrary aggregate of packets, e.g. with the same IP source address or the same TCP/UDP 5-tuple. FQ provides high precision of bandwidth partitioning, but unfortunately, such precision comes at a considerable expense: (i) the time to process a packet depends on the number N of active users, precisely $O(\log(N))$; and (ii) N per-user queues are required.

The first limitation is important with today's throughput requirements which drastically reduce the maximum processing time allowed for a packet, e.g. a switching chip with aggregate throughput of 1 Tb/s has a time budget of 1 ns to process a minimum size packet. The second limitation affects switching hardware implementations. Here the number of queues impacts both the memory requirements and the combinatorial logic necessary to implements the circuitry of the scheduler[1]. As a consequence, it is hard to scale FQ implementations to hundreds, thousands or more users. For this reason the number of queues available in commercial hardware switches is usually bounded to less than 10 [82]. This consideration is also at the base of legacy quality of service (QoS) approaches such as DiffServ, where traffic is aggregated into few classes.

To solve this problem, we designed a bandwidth enforcing scheme in which both time and implementation complexity do not depend on the number of active users N. To this purpose we do not modify the scheduler, instead we use a widely-deployed strict priority (SP) scheduler with only few queues. Fairness can be enforced by dynamically assigning priorities to users according to the sending rate history of those. We call this design FDPA (Fair Dynamic Priority Assignment). In FDPA, packets belonging to a user whose arrival bitrate is equal or less than its fair share are given priority over those users generating traffic at higher rates. FDPA does not provide precise bit-level or packet-level fairness, but it approximates a fair repartitioning over longer timescales, in the order of few RTTs.

FDPA can be implemented using abstractions for stateful programmable data planes such as OPP. The scalability of FDPA does not depend on the number of queues, but instead on the number of rate estimators that can be instantiated in the switch. Precisely, while the circuitry to implement a rate estimator can be shared among many flows[2], the switch is required to maintain per-user state, i.e. the measured rate. Hence, the only limit of FDPA is the memory available in a switching chip. In the case of OPP, this limit is represented by the maximum size of the flow context table.

[1]For a scheduler to be work-conserving, i.e. to serve a packet if at least one can be served, all N queues must be examined at the same time. As such, the number of wires to implement such a structure depends on N.

[2]In a typical pipelined hardware architecture, that would be a stage of the pipeline.

In this work we address the feasibility of the FDPA approach by performing experiments on a 10 Gbit/s testbed using a software prototype implementation. Results show that FDPA produce fairness comparable to other schemes such as Deficit Round Robin (DRR). We found that such an approach produce a tradeoff between fairness and throughput, in which one or the other are penalized.

The rest of the chapter is organized as follows. We begin by reviewing the related work in Section 4.2, we then introduce the FDPA design in Section 4.3 and discuss its implementation options with programmable data planes. In Section 4.4, we present the experimental results from the 10 Gbit/s testbed, before concluding with a discussion on open questions and future work in Section 4.5.

4.2 Related work on enforcing fair bandwidth sharing

To reduce implementation and time complexity of FQ, a number of algorithms have been proposed in the literature. Deficit Round Robin (DRR) [83] is probably the most well known and widely-deployed algorithm. DRR was proposed to address the time complexity of FQ achieving $O(1)$ execution time per packet. However, DRR still requires per-user queues.

To overcome DRR's limitations, further approximations have been proposed. Stochastic Fair Queuing (SFQ) [84] is a probabilistic variant of FQ. Here traffic streams are hashed onto a smaller number of queues, and the hash function is periodically perturbed to minimize the time where two users collide onto the same queue. Here the quality of the approximations depends on the number of queues, and the perturbation interval. Finally, Approximate Fair Dropping (AFD) [85] employs a form of active queue management (AQM) by dropping packets before being stored on a simple FIFO queue. Dropping decisions are based on the recent history of packet arrivals, with higher probability of drop for users sending at higher rates. AFD has been used in several switch and router platforms at Cisco Systems [86].

FDPA shares the same design principles of AFD: (i) avoid using per-user queues in favor of per-user soft state, and (ii) achieve bandwidth partitioning by opportunistically dropping or delaying packets rather than by enforcing rate by using scheduling. However, while the AFD design allows for an efficient implementation in a fixed-function ASIC, its realization with programmable data plane primitives might not be straightforward. Specifically, AFD requires the implementation of a shadow buffer in which packets are removed at random. Instead as it will be discussed in Section 4.3.2, FDPA requires much simpler primitives already exposed by programmable data plane abstractions.

Finally, a more recent approach named PIFO has been proposed to address the need of a

programmable scheduler [87]. However, similarly to fixed-function schedulers, in PIFO the number of distinct flows that can be served with a fair queuing discipline is bounded by the number of queues. In their proposed design, such bound is 2048 in total or 32 per port in a 64 port switch. While one could imagine dedicating all 2048 queues to a port, the authors do not provide any evaluation of their scheduler with realistic traffic traces.

4.3 FDPA design

In this section we describe the design of a packet forwarding pipeline implementing FDPA. For the sake of clarity and without loss of generality, we assume a switch with rate controlled only on one egress port.

Figure 4.1 depicts the design of the pipeline. Packets are first classified per user and then processed by a rate estimator which measures the arrival bitrate of the specific user. Packets are then stored in one of the Q priority queues such that the higher is the arrival rate, the lower will be the priority. A strict priority scheduler (SP) serves queues in priority order: packets of priority q are dequeued only if all other queues with higher priority are empty, where $q = 1$ is the highest priority.

The measured arrival rate for a given user at a given point in time, determines an active "band" for that user. Packets arrived in band B_q will be assigned priority q (Figure 4.2). The first band B_1 represents the minimum guaranteed portion of the link capacity allocated to each user, for this reason a feasible allocation requires $N \times B_1 \leq LinkCapacity$. Moreover, to further penalize ill-behaving users, each queue has a different size L_q, with smaller values for low priority queues.

4.3.1 Rationale

To discuss the rationale behind this design, it is useful to first analyze the case of a scheduler with only 2 queues ($Q = 2$), high priority and low priority, and then move to the case of more queues.

Two priorities. When congestion occurs, those users sending packets below their fair share are prioritized over others sending at higher rates. Given the limited size of the queues, packets with low priority are delayed and in the case of a full buffer, dropped upon arrival. Such an event signals the TCP source to reduce the transmit rate. With FDPA, this reduction is expected to continue until the transmit rate hits the first band, in which case the user is prioritized again. Assuming that all sources are TCP-like and produce long-lived flows, under severe congestion we can expect traffic sources to shape their transmit rate around their fair

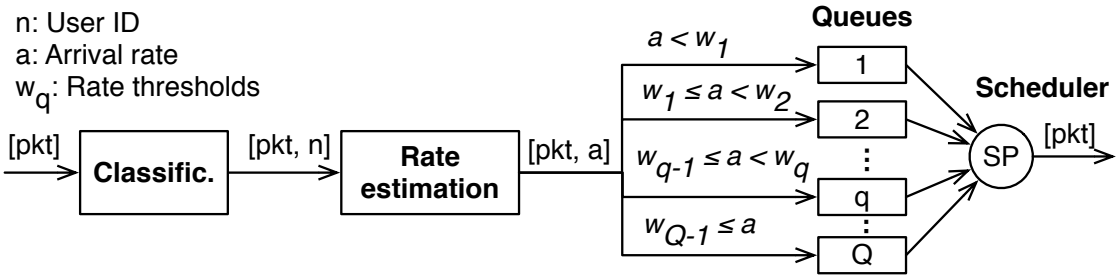


Figure 4.1 FDPA forwarding pipeline.

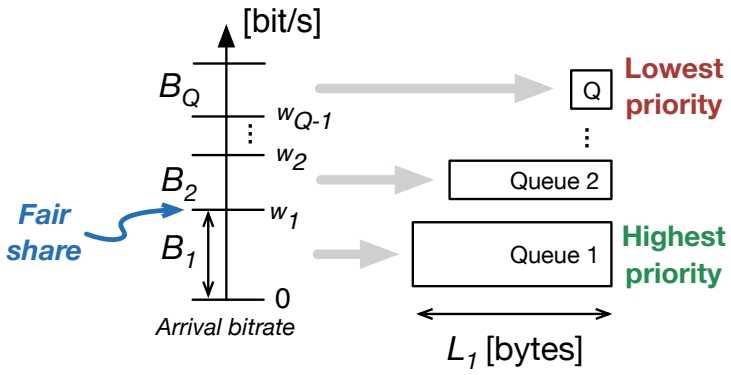


Figure 4.2 Rate bands and queue size in FDPA.

share, i.e. the upper threshold of the first band. Swapping queues frequently, can cause packet reordering at the receiver, confusing TCP congestion control and affecting throughput.

In the case of non-elastic sources, e.g. constant bitrate, B_1 represents the maximum rate that a source can send with guarantees of bounded latency and minimum drop probability. Indeed when a user hits the first band, packets are always served by the same, maximum priority queue, hence preventing disruption from other TCP sources when aiming to transmit at higher rates.

However, if some sources are using less than their fair share or because not all the link capacity has been reserved, i.e. $N \times B_1 < LinkCapacity$, using only 2 priorities does not enforce equal distribution of the excess bandwidth. Indeed, assuming that capacity has been allocated for many users, but only few of them are active and sending TCP traffic, we can expect that those users will be competing in the same low priority FIFO queue, without any guarantee of fairness.

More priorities. To enforce equal distribution of the excess bandwidth, we need to introduce more priorities, so that the more a source increases its rate, the less priority it will get

with respect to other users. When all sources are TCP-like, following the same rationale of the previous case, we can expect the transmit rate of each user to converge to a fair share that considers the excess bandwidth. Such fair share will lay in a rate band other than B_1

Figure 4.3 illustrates the expected behavior of 2 TCP-like sources competing for the excess bandwidth. In this example, one source (1) is ill-behaving as it uses a more aggressive rate control algorithm[3]; the other source (2) is well behaving, as for each congestion signal it halves its transmit rate. At steady state, both sources tend to share the same queue with priority 3, however the different rate-control behavior that they implement causes them to oscillate around different average values. Indeed, (1) always tends to increase its rate until it falls to the 4th band, which cause its packets to timeout as the scheduler will spend as much time as needed to serve packets of higher priority; (2) instead has higher drop probability when it falls in band B_3, as here the queue is monopolized by packets of (1). However, by always assuring a higher priority for lower rates, increases of (2) are always guaranteed at least until hitting the lower threshold of band B_3. Intuitively, we can expect that the difference between the average transmit rate of both sources ($\Delta rate$) will be smaller with narrower bands, hence producing a fairer allocation.

Unfortunately, as in the case with only 2 priorities, multiple narrower bands can increase the risk of packet reordering, affecting the overall throughput. This generates a tradeoff between fairness and throughput.

4.3.2 Implementation with programmable data planes

Classifying packets per user is easy and can be done using a match+action table common to many data plane abstractions. Using such tables one can match on specific header fields and write the corresponding user ID on the packet's metadata.

Estimating the bitrate of a flow might be tricky at high speed. Using the OPP configuration for rate estimation introduced in Section 2.2.3, the switch needs to maintain for each user a byte counter and a timestamp of the last time the rate estimation was updated. Updates of the rate values are triggered by packets arrival if the timestamp of the packet exceeds a predefined interval, i.e. the minimum interval over which the average bitrate is evaluated. The rate is then computed dividing the number of bytes by the interval between the packet's timestamp and the stored timestamp. While division is an operation that might be hard to perform in a line rate switch, in [11] it is shown how this operation can be approximated with good precision using primitives available in the update logic block of OPP and other

[3]The behavior of this source is similar to the case of a user opening multiple TCP streams.

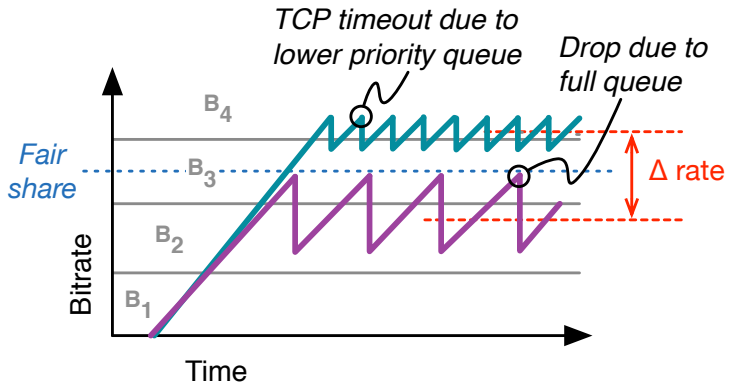


Figure 4.3 Example of 2 TCP sources competing for the excess bandwidth when using more than 2 priorities.

programmable data planes. A second match-action table can then be used to direct packets to the different queues according to the estimated rate band, written in the packet's metadata.

Along with programmable data planes, it should be noticed how FDPA can be implemented in switches supporting standard OpenFlow v1.3+. Indeed, OpenFlow define "meters" that can be configured with different bands as defined by FDPA, such that packets hitting a given rate can be marked using the the DSCP field, used later to assign packets to priority queues.

Finally, priority schedulers are a common structure available today in switching hardware.

4.4 Experimental results

To prove the applicability of the proposed approach, we implemented a software-based prototype of FDPA and used it with real traffic at 10 Gbit/s, evaluating the effects of different bands assignment on both fairness and throughput.

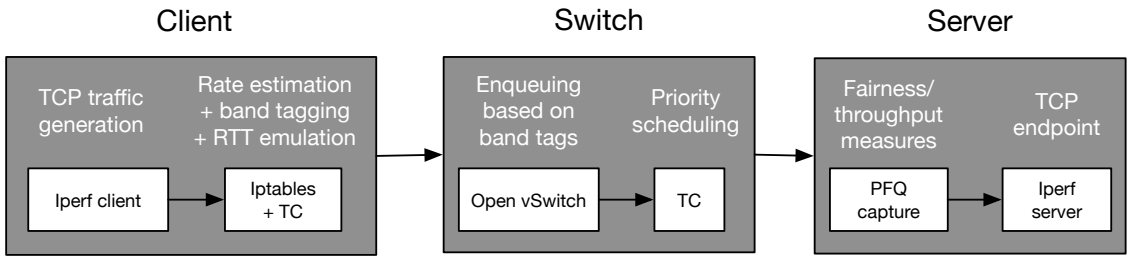


Figure 4.4 Software-based processing pipeline used in experiments.

Testbed

For the experiments we used 3 desktop machines with 8-core Intel Xeon E51660V3 CPUs (3.0GHz), equipped with multiple Intel 82599 10G NICs. One machine acts as a switch with 4 10G ports, another machine is used to generate traffic from 2 ports, while the last is used to both generate and receive traffic form different ports. Each machine runs a Debian 9.0 Stretch based on a Linux Kernel v4.9.16.

Figure 4.4 shows the processing pipeline used to emulate FDPA. We used `iperf` to generate TCP traffic, Linux's `iptables` to estimate the rate and tag packets accordingly. In FDPA, rate estimation should happen in the switch, however, to simplify the prototype implementation we decided to move it to the client machines. We used Linux's `tc` (Traffic Control) to emulate different RTTs at the clients and to perform priority scheduling at the switch. Open vSwitch was used to steer packets to the different queues based on the band tags. Finally, we used PFQ [88], a framework for accelerated packet I/O, to measure the bitrate of each user. Both clients and server use TCP Cubic, with the default parameters found in the Linux Kernel v4.9.16, changing only the memory available to TCP buffers to allow for a large number of connections. The MTU of all interfaces was set to 1500 bytes.

We configured traffic sources to experience an emulated RTT of around 5 ms with maximum 0.25 ms of variable jitter with 25% correlation. TCP increases its sending rate at RTT timescales, hence for FDPA to promptly respond to rate variations, the estimation interval should be in the order of few RTTs. For this reason we used estimation intervals of around 30 ms.

Metrics

We measured the quality of an experiment using two metrics: (i) the aggregate throughput (TPut) normalized over the link capacity, i.e. bounded between 0 and 1, and (ii) the Jain's Fairness Index (JFI) [89]. The JFI is a popular fairness measure defined as:

$$JFI = \frac{(\sum_n x_n)^2}{N \cdot \sum_n x_n^2}$$

where x_n is the normalized rate of a user n and N is the total number of users. The normalized rate x_n is defined as:

$$x_n = \frac{MeasuredRate_n}{FairRate_n}$$

In the experiments each user is assigned the same fair share. The JFI is bounded between 0

and 1, where 1 is a fair distribution and 0 is a discriminating one. FDPA should maximize both TPut and JFI.

Results

Figure 4.5 shows the results obtained from the experiments. Long-lived TCP traffic was generated from a varying number of users, 50, 100 and 200[4], and varying the number of TCP connections per user based on 4 scenarios: (i) all users have the same number 1 of TCP connections, i.e. they all well-behave, (ii) when 1/4th of the users mis-behave by opening 10 parallel TCP connections, (iii) when half of the users mis-behave, and (iv) when each user has a number of connections uniformly distributed between 1 and 10.

When testing FDPA, we vary the number and size of rate bands. The following notation describes a specific FDPA configuration: $F(FirstBand + NumBands * BandSize)$, where $FirstBand$ is the size of B_1, $NumBands$ is the number of bands following the first one, each one of size $BandSize$, except for the last one that has infinite size, i.e. up to the link capacity. $FirstBand$ and $BandSize$ are expressed as a proportion of the fair share, e.g. $F(1 + 4 * 0.5)$ describes a configuration where the first band is exactly the fair share, and the other 4 bands have size half of the latter. We performed experiments with $FirstBand \in \{0.75, 0.85, 1, 1.15, 1.25\}$, $NumBands \in \{3, 4\}$ and $BandSize \in \{0.25, 0.33, 0.50, 0.67, 0.75\}$. For the queue size we found that the following sizing provides the best performances: $L_q = \min(20, BDP/q^q)$, where BDP is the bandwidth delay product $\overline{RTT} \times LinkCapacity$. With $\overline{RTT} = 5ms$, the sizing for 5 queues is $L_1 = 4166$ (MTU-size packets), $L_2 = 1041$, $L_3 = 154$, $L_4 = 20$, and $L_5 = 20$.

Samples of the average bitrate measured at the server are collected over a 1-second interval, each second at the same time for all sources, for 50 seconds. Sampling starts 30 seconds after launching `iperf`, allowing all TCP sessions to converge to their average bitrate. For each second, a value of JFI and TPut is computed. The plots show the median of the JFI and TPut samples for each experiment, along with a 80% confidence interval. For each traffic scenario, only three configurations of FDPA are plotted, the one with the best TPut, the one with the best JFI, and the one that maximizes the product of both. Moreover, a scatter plot (black dots) of all JFI and TPut values obtained in all FDPA configurations explicitly shows a clear tradeoff between TPut and JFI.

Results are compared with the following cases:
FIFO. All users are served using 1 FIFO queue of size $L = BDP$, e.g. 4166 MTU-size

[4]We put a limit to 200 as the experimental setup suffers of performance degradation when emulating more users

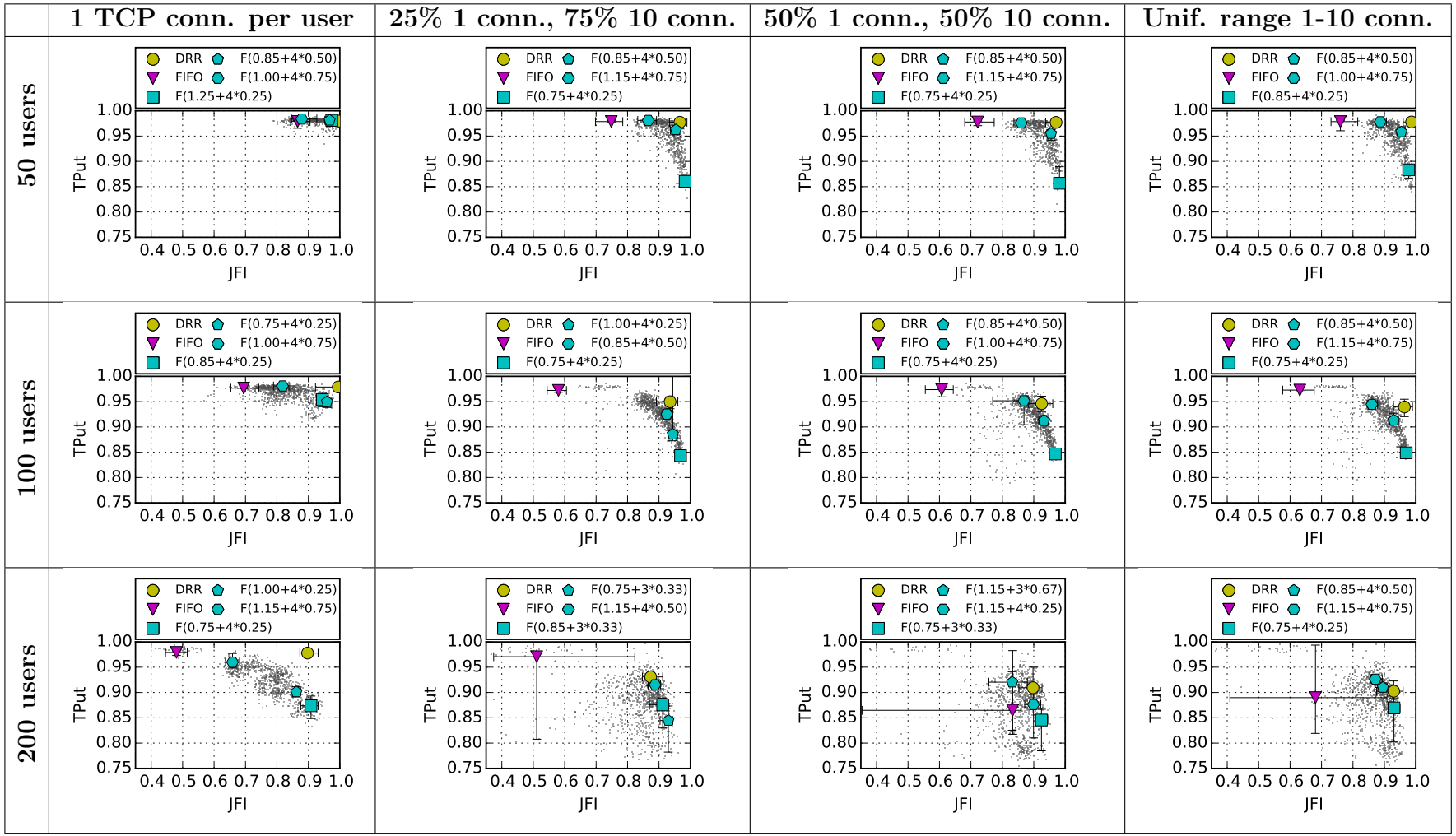


Figure 4.5 FDPA experimental results.

packets with $\overline{RTT} = 5ms$. This is our worst case, when fairness is not enforced.
DRR. The switch performs DRR scheduling with per-user queues, using the `tc-drr` implementation provided as part of the Linux's `tc` suite. DRR represent the best case scenario, however it is important to note that while it is still feasible to provide a large number of per-user queues in software, the same does not apply to hardware switches, where an a priori instantiation of hardware resources (memory and logic circuitry) is required. Remember that the majority of today switching chips provide 10 or less output queues per port [82].

As expected, FDPA holds the promise of enforcing fairness w.r.t. a single FIFO queue in all scenarios, producing results comparable to the ideal case of a DRR scheduler with per-user queues. However, fairness comes at the cost of throughput. Configurations of FDPA that use narrower bands provide more fairness, between 0.95 and 0.99 in most cases. Unfortunately, these configuration systematically incur in throughput degradation, down to 0.85 in some cases, when for the same scenario DRR achieves almost perfect fairness with throughput comparable to that of a FIFO queue, i.e. optimal around 0.98, or little less around 0.95. Vice versa, larger bands improve throughput, at the expense of fairness.

4.5 Discussion

How to improve throughput? Preliminary analysis show that throughput degradation is mostly caused by packet reordering due to frequent changes in the queue assignment, which confuses the TCP congestion control. The problematic part is when users are prioritized again. Here a burst of consecutive packets is swapped from a low priority queue to a higher priority one, with the effect of having subsequent packets being transmitted before those arrived earlier. A solution to this problem could be that of using a flowlet-based approach[23], in which queue assignments are valid for the whole burst of packets, where bursts are separated by an idle time usually comparable to the RTT. This would increase the probability of having all packets from the low priority queue sent before the arrival of the new burst. As shown in Section 2.2.3, flowlet detection is easily implemented in OPP. We leave exploring such a more advanced design for future work.

Rate estimation. An alternative to the average estimators used in the experiment are token buckets-based estimators. The advantage of token buckets lays in their ability to immediately respond to rate spikes and bursts of packets, while an average estimator leaves enough time to aggressive user to congest the first queue. The downside of frequent band variations correspond to higher risk of packet reordering, and preliminary results on our testbed using token buckets show that this is the case. However, we believe that using token

buckets along with per-flowlet queue assignment could help in improving both fairness and throughput. We leave this for future work.

How to compute the fair share? We can envision an external controller (or switch-internal control plane) that periodically adjusts band sizes by counting the number of active users. In the case of a service provider network, where the number of active users varies slowly, the frequency of the estimation process is not a limit for the scalability of the approach. Indeed, using many priorities helps in absorbing temporary rate spikes and minor variation of the fair share. How to efficiently implement user estimation is outside the scope of this work, however, it must be noted that a controller could use the same counter instantiated at the switch for the rate estimation process.

4.6 Conclusions

In this chapter we introduced FDPA, a design for a packet forwarding pipeline to enforce approximate fair bandwidth sharing among many users sharing the same link of fixed capacity. FDPA is based on abstractions for stateful programmable data planes such as OPP. We performed experimental evaluation on a 10 Gbit/s testbed. Results show that performance are close to that of an ideal DRR scheduler with dedicated queues per user (50, 100, 200). We identified a tradeoff between fairness and throughput, in which throughput is penalized when configuring FDPA for more fairness. Preliminary analysis show that packet reordering is the cause of such effect. We identified potential solutions to such problem that we leave for future work.

CHAPTER 5 CONCLUSION

Ideally, network operators would like to buy a switch or router and use it for many years, without paying for expensive infrastructure upgrades to support new protocols and features. It would be much more convenient for them to optimize or re-purpose the infrastructure by simply pushing software or firmware upgrades.

SDN enables programmability of the network's control plane. Operators can implement new routing protocols, or network-level services such as network virtualization, by simply writing software on top of a network OS. OpenFlow abstracts the data plane through an API that exposes forwarding tables. The network OS controls the devices' behavior by dynamically changing forwarding rules. OpenFlow is a powerful abstraction of the data plane that fostered the adoption of SDN.

However, the data plane is responsible for many other tasks on top of simple OpenFlow-like forwarding. Operators would like to program those tasks too. Data plane programmability would enable network owners and researchers to make a more efficient use of network resources, by programming for instance more advanced load balancing schemes than legacy ECMP; to reduce network congestion, by tweaking and optimizing existing rate control mechanisms; to take better decisions on where to route packets, by offloading fine-grained measurements directly in the data plane; to minimize the effects of failures on premium customers, by programming devices to detect failures more quickly. Most of these tasks require the data plane to keep and manipulate state.

Recent advances in the design and implementation of programmable switching chips provide enhanced flexibility of the forwarding tasks. Programmers can define new headers to match and operate on non-standard protocols. However, they marginally address the problem of programming stateful data plane algorithms, and if they do, they favor line rate guarantees at all times, instead of enhanced flexibility.

In Chapter 2, we presented two abstractions for programmable data planes that are specifically targeted for stateful processing, providing programming examples, discussing their feasibility for line rate hardware implementations, and evaluating their performance on real traffic traces.

OpenState extends OpenFlow by permitting to express forwarding behaviors as Mealy (finite state) Machines operating on flow-states. The main finding of OpenState is that the same OpenFlow's flow table can be used to describe a Mealy Machine entirely, provided support

for a state memory and means to quickly lookup/update such memory. The hardware implementation of OpenState departs minimally from that of a TCAM-based OpenFlow pipeline and can be realized with commodity switch components. The second abstraction, OPP, tries to address a key limitation of Mealy Machines, namely state explosion. In OPP, processing tasks can be defined using EFSMs, which extend and generalize Mealy Machines by adding support for custom flow registers, i.e. variables, and computational capabilities. Similarly, OPP extends and generalizes the OpenState packet processing pipeline by adding means to evaluate conditions on per-flow registers and, most important, to perform computations on different kinds of state: per-flow, global and per-packet (i.e. headers and metadata).

OPP (and so OpenState) is characterized by an important design choice: to allow for a feedback-loop in the pipeline. Such loop is both a blessing and a curse. It permits to use flow processors, i.e. OPP's update logic blocks, of arbitrary flexibility, i.e. taking several clock cycles, but it also represents a harm for state consistency. The feedback-loop is what makes OPP different from the related work. Indeed, OPP implicitly favors programmability over line rate guarantees at *all* the time with *any* traffic workload, rather performance in OPP are *just* predictable. To prevent state inconsistencies, we designed a memory locking scheme, thus making OPP a *blocking* architecture. To convince the reader that there is not much harm in such design choice, we implemented a simulator and run it using real traffic traces, from both carrier and datacenter networks. Results show that differentiating state contexts is useful to enable different levels of programmability, i.e. it is safe to permit feedback-loops of several clock cycles when dealing with per-flow states, while access to global states should be always performed in 1 cycle.

We left for future work the design of a flow processor that is expressive enough to execute a large number of processing tasks, the design of a programming language for OPP, and the implementation of OPP on other architectures, such as general purpose CPUs, GPUs, or NPUs.

The devised abstractions should be able to execute existing data plane algorithms and promote the definition of *new* ones. In the remaining chapters we presented two *novel* applications of OpenState and OPP: Spider and FDPA.

Spider, introduced in Chapter 3, is based on OpenState and permits to detect and react to link/node failures at fast-path timescales, i.e. micro/nanoseconds, by routing packets to predefined backup routes, also in the case of distant failures, i.e. enabling a detour at a node that is several hops distant from the failure. Detection of failures in Spider is heavily based on OpenState's state table timeouts. Interestingly, Spider provides functionalities equivalent to legacy control plane protocols such as BFD and MPLS Fast Reroute, but without the need of

a control plane (excluding the provisioning of the primary and backup paths). Detection and rerouting happen entirely in the data plane. Spider is an example of how a simple extension to OpenFlow such as OpenState can open to a whole new dimension of programmability in SDN.

FDPA, introduced in Chapter 4, is based on OPP and allows to enforce approximate fair bandwidth sharing among many TCP-like senders. This is a feature desirable in networks where unused capacity could be redistributed, without the risk of having some users obtaining more than others. FDPA is an example of how an abstraction like OPP can help trading one complexity for another. Scalability of classical fair queuing scheduling algorithms is limited by the available number of per-user queues. We propose an approach in which per-user queues are traded with per-user state, necessary to perform rate estimation in FDPA. Results on a 10 Gbit/s testbed show that performance of FDPA is comparable to that of an ideal DRR scheduler with per-user queues. During the experiments, we identified a tradeoff between fairness and throughput, in which throughput is penalized when configuring FDPA for more fairness. Preliminary analysis shows that packet reordering is the cause of such effect. We identified potential solutions to such problem that we left for future work.

BIBLIOGRAPHY

[1] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, "Openstate: Programming platform-independent stateful openflow applications inside the switch," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 2, pp. 44–51, Apr. 2014. [Online]. Available: http://doi.acm.org/10.1145/2602204.2602211

[2] C. Cascone, L. Pollini, D. Sanvito, A. Capone, and B. Sansó, "Spider: Fault resilient sdn pipeline with recovery delay guarantees," in *2016 IEEE NetSoft Conference and Workshops (NetSoft)*, June 2016, pp. 296–302. [Online]. Available: http://ieeexplore.ieee.org/document/7502425/

[3] C. Cascone, D. Sanvito, L. Pollini, A. Capone, and B. Sansò, "Fast failure detection and recovery in sdn with stateful data plane," *International Journal of Network Management*, vol. 27, no. 2, 2017. [Online]. Available: http://dx.doi.org/10.1002/nem.1957

[4] C. Cascone, N. Bonelli, L. Bianchi, A. Capone, and B. Sansò, "Towards approximate fair bandwidth sharing via dynamic priority queuing," in *2017 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*, June 2017, pp. 1–6. [Online]. Available: http://ieeexplore.ieee.org/document/7972168/

[5] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008. [Online]. Available: http://doi.acm.org/10.1145/1355734.1355746

[6] Open Network Operating System (ONOS). [Online]. Available: http://onosproject.org/

[7] OpenDaylight: Open Source SDN Platform (ODL). [Online]. Available: http://www.opendaylight.org/

[8] Ryu OpenFlow Controller. http://osrg.github.io/ryu/. [Online]. Available: http://www.osrg.net/ryu/

[9] N. Feamster, J. Rexford, and E. Zegura, "The road to sdn: An intellectual history of programmable networks," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 2, pp. 87–98, Apr. 2014. [Online]. Available: http://doi.acm.org/10.1145/2602204.2602219

[10] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "Devoflow: Scaling flow management for high-performance networks," in *Proceedings of the ACM SIGCOMM 2011 Conference*, ser. SIGCOMM '11. New York, NY, USA: ACM, 2011, pp. 254–265. [Online]. Available: http://doi.acm.org/10.1145/2018436.2018466

[11] N. K. Sharma, A. Kaufmann, T. Anderson, A. Krishnamurthy, J. Nelson, and S. Peter, "Evaluating the power of flexible packet processing for network resource allocation," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017, pp. 67–82. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/sharma

[12] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "Clickos and the art of network function virtualization," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, 2014, pp. 459–473. [Online]. Available: https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/martins

[13] L. Peterson, A. Al-Shabibi, T. Anshutz, S. Baker, A. Bavier, S. Das, J. Hart, G. Palukar, and W. Snow, "Central office re-architected as a data center," *IEEE Communications Magazine*, vol. 54, no. 10, pp. 96–101, October 2016.

[14] AT&T, BT, CenturyLink, China Mobile, Colt, Deutusche Telekom, KDDI, NTT, Orange, Telefom Italia, Telefonica, Telstra, and Verizon, "Network function virtualization: An introduction, benefits, enablers, challenges and call for action," October 2012. [Online]. Available: https://portal.etsi.org/nfv/nfv_white_paper.pdf

[15] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker, "Netbricks: Taking the v out of nfv," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. GA: USENIX Association, 2016, pp. 203–216. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/panda

[16] N. Zilberman, P. M. Watts, C. Rotsos, and A. W. Moore, "Reconfigurable network systems and software-defined networking," *Proceedings of the IEEE*, vol. 103, no. 7, pp. 1102–1124, July 2015. [Online]. Available: http://ieeexplore.ieee.org/document/7122247/

[17] XPliant ethernet switch product family. [Online]. Available: http://www.cavium.com/XPliant-Ethernet-Switch-ProductFamily.html

[18] Intel FlexPipe. [Online]. Available: http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/ethernet-switchfm6000-series-brief.pdf

[19] Barefoot Networks. The world's fastest and most programmable networks. [Online]. Available: https://www.barefootnetworks.com/media/white_papers/Barefoot-Worlds-Fastest-Most-Programmable-Networks.pdf

[20] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, ser. SIGCOMM '13. New York, NY, USA: ACM, 2013, pp. 99–110. [Online]. Available: http://doi.acm.org/10.1145/2486001.2486011

[21] G. H. Mealy, "A method for synthesizing sequential circuits," *Bell System Technical Journal*, vol. 34, no. 5, pp. 1045–1079, 1955.

[22] Z. Cao, Z. Wang, and E. Zegura, "Performance of hashing-based schemes for internet load balancing," in *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (Cat. No.00CH37064)*, vol. 1, 2000, pp. 332–341 vol.1. [Online]. Available: http://ieeexplore.ieee.org/document/832203/

[23] S. Kandula, D. Katabi, S. Sinha, and A. Berger, "Dynamic load balancing without packet reordering," *SIGCOMM Comput. Commun. Rev.*, vol. 37, no. 2, pp. 51–62, Mar. 2007. [Online]. Available: http://doi.acm.org/10.1145/1232919.1232925

[24] S. Pontarelli, M. Bonola, G. Bianchi, A. Capone, and C. Cascone, "Stateful openflow: Hardware proof of concept," in *2015 IEEE 16th International Conference on High Performance Switching and Routing (HPSR)*, 2015, pp. 1–8. [Online]. Available: http://ieeexplore.ieee.org/document/7483105/

[25] A. Broder and M. Mitzenmacher, "Using multiple hash functions to improve ip lookups," in *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No.01CH37213)*, vol. 3, 2001, pp. 1454–1463 vol.3. [Online]. Available: http://ieeexplore.ieee.org/document/916641/

[26] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen, "Scalable, high performance ethernet forwarding with cuckooswitch," in *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, ser.

CoNEXT '13. New York, NY, USA: ACM, 2013, pp. 97–108. [Online]. Available: http://doi.acm.org/10.1145/2535372.2535379

[27] BEBA Project, "Project deliverable D5.3: final applications' design and development," 2016. [Online]. Available: www.beba-project.eu/public_deliverables/BEBA_D5.3.pdf

[28] G. Gibb, G. Varghese, M. Horowitz, and N. McKeown, "Design principles for packet parsers," in *Architectures for Networking and Communications Systems*, Oct 2013, pp. 13–24. [Online]. Available: http://ieeexplore.ieee.org/document/6665172/

[29] G. Bianchi, M. Bonola, S. Pontarelli, D. Sanvito, A. Capone, and C. Cascone, "Open packet processor: a programmable architecture for wire speed platform-independent stateful in-network processing," *CoRR*, vol. abs/1605.01977, 2016. [Online]. Available: http://arxiv.org/abs/1605.01977

[30] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore, "Netfpga sume: Toward 100 gbps as research commodity," *IEEE Micro*, vol. 34, no. 5, pp. 32–41, Sept 2014. [Online]. Available: http://ieeexplore.ieee.org/document/6866035/

[31] J.-L. Brelet, "Using Virtex-II block RAM for high performance read/write CAMs," 2012. [Online]. Available: https://www.xilinx.com/support/documentation/application_notes/xapp260.pdf

[32] Z. Ullah, M. K. Jaiswal, Y. C. Chan, and R. C. C. Cheung, "Fpga implementation of sram-based ternary content addressable memory," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*, May 2012, pp. 383–389. [Online]. Available: http://ieeexplore.ieee.org/document/6270666/

[33] W. Jiang, "Scalable ternary content addressable memory implementation using fpgas," in *Architectures for Networking and Communications Systems*, Oct 2013, pp. 71–82. [Online]. Available: http://ieeexplore.ieee.org/document/6665177/

[34] Netfpga sume reference learning switch. [Online]. Available: https://github.com/NetFPGA/NetFPGA-SUME-public/wiki/NetFPGA-SUME-Reference-Learning-Switch

[35] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown, and S. Licking, "Packet transactions: High-level programming for line-rate switches," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16. New York, NY, USA: ACM, 2016, pp. 15–28. [Online]. Available: http://doi.acm.org/10.1145/2934872.2934900

[36] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese, "Conga: Distributed congestion-aware load balancing for datacenters," vol. 44, no. 4. New York, NY, USA: ACM, Aug. 2014, pp. 503–514. [Online]. Available: http://doi.acm.org/10.1145/2740070.2626316

[37] The CAIDA UCSD anonymized internet traces - Chicago 2015-02-19. [Online]. Available: http://www.caida.org/data/passive/passive_2015_dataset.xml

[38] The CAIDA UCSD anonymized internet traces - San Jose 2012-11-15. [Online]. Available: http://www.caida.org/data/passive/passive_2012_dataset.xml

[39] R. Fontugne, P. Borgnat, P. Abry, and K. Fukuda, "Mawilab: Combining diverse anomaly detectors for automated anomaly labeling and performance benchmarking," in *Proceedings of the 6th International COnference*, ser. Co-NEXT '10. New York, NY, USA: ACM, 2010, pp. 8:1–8:12. [Online]. Available: http://doi.acm.org/10.1145/1921168.1921179

[40] MAWILab traffic trace - 2015-07-20. [Online]. Available: http://www.fukuda-lab.org/mawilab/v1.1/2015/07/20/20150720.html

[41] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM '15. New York, NY, USA: ACM, 2015, pp. 123–137. [Online]. Available: http://doi.acm.org/10.1145/2785956.2787472

[42] FBFlow dataset - cluster b. [Online]. Available: https://www.facebook.com/network-analytics

[43] OPP-SIM source code and results repository. [Online]. Available: https://github.com/ccascone/opp-sim

[44] The CAIDA UCSD statistical information for the CAIDA anonymized internet traces. [Online]. Available: http://www.caida.org/data/passive/passive_trace_statistics.xml

[45] M. Moshref, A. Bhargava, A. Gupta, M. Yu, and R. Govindan, "Flow-level state transition as a new switch primitive for sdn," in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '14. New York, NY, USA: ACM, 2014, pp. 61–66. [Online]. Available: http://doi.acm.org/10.1145/2620728.2620729

[46] Open vSwitch. [Online]. Available: http://openvswitch.org

[47] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014. [Online]. Available: http://doi.acm.org/10.1145/2656877.2656890

[48] The P4 language Consortium, "The P4 Language Specification, version 1.0.2." [Online]. Available: http://p4.org/wp-content/uploads/2015/04/p4-latest.pdf

[49] ——, "The P4 Language Specification, version 1.1.0." [Online]. Available: http://p4.org/wp-content/uploads/2016/03/p4_v1.1.pdf

[50] M. Shahbaz and N. Feamster, "The case for an intermediate representation for programmable data planes," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, ser. SOSR '15. New York, NY, USA: ACM, 2015, pp. 3:1–3:6. [Online]. Available: http://doi.acm.org/10.1145/2774993.2775000

[51] K. Nichols and V. Jacobson, "Controlling queue delay," *Queue*, vol. 10, no. 5, pp. 20:20–20:34, May 2012. [Online]. Available: http://doi.acm.org/10.1145/2208917.2209336

[52] V. Jeyakumar, M. Alizadeh, Y. Geng, C. Kim, and D. Mazières, "Millions of little minions: Using packets for low latency network programming and visibility," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, ser. SIGCOMM '14. New York, NY, USA: ACM, 2014, pp. 3–14. [Online]. Available: http://doi.acm.org/10.1145/2619239.2626292

[53] I. Tinnirello, G. Bianchi, P. Gallo, D. Garlisi, F. Giuliano, and F. Gringoli, "Wireless mac processors: Programming mac protocols on commodity hardware," in *2012 Proceedings IEEE INFOCOM*, March 2012, pp. 1269–1277. [Online]. Available: http://ieeexplore.ieee.org/document/6195488/

[54] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy, "Routebricks: Exploiting parallelism to scale software routers," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 15–28. [Online]. Available: http://doi.acm.org/10.1145/1629575.1629578

[55] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," vol. 18, no. 3. New York, NY, USA: ACM, Aug. 2000, pp. 263–297. [Online]. Available: http://doi.acm.org/10.1145/354871.354874

[56] Intel IXP4XX product line of network processors. [Online]. Available: http://www.intel.com/content/www/us/en/intelligent-systems/previous-generation/intel-ixp4xx-intel-network-processor-product-line.html

[57] Xilinx Virtex-7 family overview. [Online]. Available: https://www.xilinx.com/products/silicon-devices/fpga/virtex-7.html

[58] G. Gibb, "Reconfigurable hardware for software-defined networks," Ph.D. dissertation, Stanford University, 2013.

[59] M. T. Arashloo, Y. Koral, M. Greenberg, J. Rexford, and D. Walker, "Snap: Stateful network-wide abstractions for packet processing," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16. New York, NY, USA: ACM, 2016, pp. 29–43. [Online]. Available: http://doi.acm.org/10.1145/2934872.2934892

[60] K. T. Cheng and A. S. Krishnakumar, "Automatic functional test generation using the extended finite state machine model," in *Proceedings of the 30th International Design Automation Conference*, ser. DAC '93. New York, NY, USA: ACM, 1993, pp. 86–91. [Online]. Available: http://doi.acm.org/10.1145/157485.164585

[61] D. Katz and D. Ward, "Bidirectional Forwarding Detection (BFD)," RFC 5880 (Proposed Standard), Jun. 2010. [Online]. Available: https://tools.ietf.org/html/rfc5880

[62] P. Pan, G. Swallow, and A. Atlas, "Fast Reroute Extensions to RSVP-TE for LSP Tunnels," RFC 4090 (Proposed Standard), May 2005. [Online]. Available: https://tools.ietf.org/html/rfc4090

[63] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester, "Openflow: Meeting carrier-grade recovery requirements," *Comput. Commun.*, vol. 36, no. 6, pp. 656–665, Mar. 2013. [Online]. Available: http://dx.doi.org/10.1016/j.comcom.2012.09.011

[64] N. L. M. v. Adrichem, B. J. v. Asten, and F. A. Kuipers, "Fast recovery in software-defined networks," in *2014 Third European Workshop on Software Defined Networks*, Sept 2014, pp. 61–66.

[65] J. Kempf, E. Bellagamba, A. Kern, D. Jocha, A. Takacs, and P. Sköldström, "Scalable fault management for openflow," in *2012 IEEE International Conference on Communications (ICC)*, June 2012, pp. 6606–6610. [Online]. Available: http://ieeexplore.ieee.org/document/6364688/

[66] A. Sgambelluri, A. Giorgetti, F. Cugini, F. Paolucci, and P. Castoldi, "Openflow-based segment protection in ethernet networks," *IEEE/OSA Journal of Optical Communications and Networking*, vol. 5, no. 9, pp. 1066–1075, 2013. [Online]. Available: http://ieeexplore.ieee.org/abstract/document/6608645/

[67] S. S. W. Lee, K. Y. Li, K. Y. Chan, G. H. Lai, and Y. C. Chung, "Path layout planning and software based fast failure detection in survivable openflow networks," in *2014 10th International Conference on the Design of Reliable Communication Networks (DRCN)*, April 2014, pp. 1–8. [Online]. Available: http://ieeexplore.ieee.org/document/6816141/

[68] M. Borokhovich, L. Schiff, and S. Schmid, "Provable data plane connectivity with local fast failover: Introducing openflow graph algorithms," in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '14. New York, NY, USA: ACM, 2014, pp. 121–126. [Online]. Available: http://doi.acm.org/10.1145/2620728.2620746

[69] J. McCauley, M. Zhao, E. J. Jackson, B. Raghavan, S. Ratnasamy, and S. Shenker, "The deforestation of l2," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16. New York, NY, USA: ACM, 2016, pp. 497–510. [Online]. Available: http://doi.acm.org/10.1145/2934872.2934877

[70] A. Capone, C. Cascone, A. Q. T. Nguyen, and B. Sansò, "Detour planning for fast and reliable failure recovery in sdn with openstate," in *2015 11th International Conference on the Design of Reliable Communication Networks (DRCN)*, 2015, pp. 25–32. [Online]. Available: http://ieeexplore.ieee.org/document/7148981/

[71] SPIDER source code repository. [Online]. Available: https://github.com/OpenState-SDN/spider

[72] OpenState SDN project home page. [Online]. Available: http://www.openstate-sdn.org

[73] Mininet. [Online]. Available: http://mininet.org/

[74] CPqD OpenFlow 1.3 software switch. [Online]. Available: http://cpqd.github.io/ofsoftswitch13/

[75] A. Farrel, A. Satyanarayana, A. Iwata, N. Fujita, and G. Ash, "Crankback Signaling Extensions for MPLS and GMPLS RSVP-TE," RFC 4920 (Proposed Standard), Jul. 2007. [Online]. Available: https://tools.ietf.org/html/rfc4920

[76] openstate.p4 source code repository. [Online]. Available: https://github.com/OpenState-SDN/openstate.p4

[77] BMv2 source code repository. [Online]. Available: https://github.com/p4lang/behavioral-model

[78] CAIDA, "Analyzing UDP usage in Internet traffic," 2009. [Online]. Available: https://www.caida.org/research/traffic-analysis/tcpudpratio/

[79] L. Qiu, Y. Zhang, and S. Keshav, "Understanding the performance of many {TCP} flows," *Computer Networks*, vol. 37, no. 3–4, pp. 277 – 306, 2001. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389128601002031

[80] J. Nagle, "On packet switches with infinite storage," RFC 970 (Proposed Standard), 1985. [Online]. Available: https://tools.ietf.org/html/rfc970

[81] A. Demers, S. Keshav, and S. Shenker, "Analysis and simulation of a fair queueing algorithm," pp. 1–12, 1989. [Online]. Available: http://doi.acm.org/10.1145/75246.75248

[82] Packet buffers. [Online]. Available: https://people.ucsc.edu/~warner/buffer.html

[83] M. Shreedhar and G. Varghese, "Efficient fair queueing using deficit round robin," pp. 231–242, 1995. [Online]. Available: http://doi.acm.org/10.1145/217382.217453

[84] P. E. McKenney, "Stochastic fairness queueing," in *INFOCOM '90, Ninth Annual Joint Conference of the IEEE Computer and Communication Societies. The Multiple Facets of Integration. Proceedings, IEEE*, Jun 1990, pp. 733–740 vol.2. [Online]. Available: http://ieeexplore.ieee.org/document/91316/

[85] R. Pan, L. Breslau, B. Prabhakar, and S. Shenker, "Approximate fairness through differential dropping," *SIGCOMM Comput. Commun. Rev.*, vol. 33, no. 2, pp. 23–39, Apr. 2003. [Online]. Available: http://doi.acm.org/10.1145/956981.956985

[86] R. Pan, B. Prabhakar, F. Bonomi, and B. Olsen, "Approximate fair bandwidth allocation: A method for simple and flexible traffic management," in *2008 46th Annual Allerton Conference on Communication, Control, and Computing*, Sept 2008, pp. 1081–1085. [Online]. Available: http://ieeexplore.ieee.org/document/4797679/

[87] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, S.-T. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, S. Katti, and N. McKeown, "Programmable packet

scheduling at line rate," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16. New York, NY, USA: ACM, 2016, pp. 44–57. [Online]. Available: http://doi.acm.org/10.1145/2934872.2934899

[88] N. Bonelli, S. Giordano, and G. Procissi, "Network traffic processing with pfq," *IEEE Journal on Selected Areas in Communications*, vol. 34, no. 6, pp. 1819–1833, June 2016. [Online]. Available: http://ieeexplore.ieee.org/document/7460204/

[89] R. Jain, D. Chiu, and W. Hawe, "A quantitative measure of fairness and discrimination for resource allocation in shared computer systems," *CoRR*, vol. cs.NI/9809099, 1998. [Online]. Available: http://arxiv.org/abs/cs.NI/9809099