DESIGNING FAST AND PROGRAMMABLE ROUTERS

by

ANIRUDH SIVARAMAN KAUSHALRAM

Master of Science, Massachusetts Institute of Technology (2012)
Bachelor of Technology, Indian Institute of Technology, Madras (2010)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2017

Author . . . . . .
Department of Electrical Engineering and Computer Science
August 31, 2017

Certified by . . . . . .
Hari Balakrishnan
Fujitsu Professor of Computer Science and Engineering
Thesis Supervisor

Certified by . . . . . .
Mohammad Alizadeh
TIBCO Career Development Assistant Professor
Thesis Supervisor

Accepted by . . . . . .
Leslie A. Kolodziejski
Professor of Electrical Engineering
Chair, Department Committee on Graduate Students

DESIGNING FAST AND PROGRAMMABLE ROUTERS
by
ANIRUDH SIVARAMAN KAUSHALRAM

Submitted to the Department of Electrical Engineering and Computer Science
on August 31, 2017, in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

Abstract

Historically, the evolution of network routers was driven primarily by performance. Recently, owing to the need for better control over network operations and the constant demand for new features, programmability of routers has become as important as performance. However, today's fastest routers, which have 10–100 ports each running at a line rate of 10–100 Gbit/s, use fixed-function hardware, which cannot be modified after deployment. This dissertation describes three router hardware primitives and their corresponding software programming models that allow network operators to program specific classes of router functionality on such fast routers.

First, we develop a system for programming stateful packet-processing algorithms such as algorithms for in-network congestion control, buffer management, and data-plane traffic engineering. The challenge here is the fact that these algorithms maintain and update state on the router. We develop a small but expressive instruction set for state manipulation on fast routers. We then expose this to the programmer through a high-level programming model and compiler.

Second, we develop a system to program packet scheduling: the task of picking which packet to transmit next on a link. Our main contribution here is the finding that many packet scheduling algorithms can be programmed using one simple idea: a priority queue of packets in hardware coupled with a software program to assign each packet's priority in this queue.

Third, we develop a system for programmable and scalable measurement of network statistics. Our goal is to allow programmers to flexibly define what they want to measure for each flow and scale to a large number of flows. We formalize a class of statistics that permit a scalable implementation and show that it includes many useful statistics (*e.g.,* moving averages and counters).

These systems show that it is possible to program several packet-processing functions at speeds approaching today's fastest routers. Based on these systems, we distill two lessons for designing fast and programmable routers in the future. First, specialized designs that program only specific classes of router functionality improve packet processing throughput by 10–100x relative to a general-purpose solution. Second, joint design of hardware and software provides us with more leverage relative to designing only one of them while keeping the other fixed.

Thesis Supervisor: Hari Balakrishnan
Title: Fujitsu Professor of Computer Science and Engineering

Thesis Supervisor: Mohammad Alizadeh
Title: TIBCO Career Development Assistant Professor

To my grandfather, the late Dr. V. Ramamurti

Contents

Previously Published Material

Chapter 4 revises a previous publication [188]: Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet Transactions: High-Level Programming for Line-Rate Switches. In *SIGCOMM*, Florianopolis, Brazil, August 2016.

Chapter 5 combines material from two previous publications [189, 190]:

1. Anirudh Sivaraman, Suvinay Subramanian, Anurag Agrawal, Sharad Chole, Shang-Tse Chuang, Tom Edsall, Mohammad Alizadeh, Sachin Katti, Nick McKeown, and Hari Balakrishnan. Towards Programmable Packet Scheduling. In *HotNets*, Philadelphia, U.S.A, November 2015.
2. Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. Programmable Packet Scheduling at Line Rate. In *SIGCOMM*, Florianopolis, Brazil, August 2016.

Chapter 6 revises a previous publication [167]: Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-Directed Hardware Design for Network Performance Monitoring . In *SIGCOMM*, Los Angeles, U.S.A, August 2017.

Acknowledgments

As a Ph.D. student, I was fortunate to work with two outstanding advisors, Hari Balakrishnan and Mohammad Alizadeh. Each contributed a unique perspective to my research. Hari was the master of the highest-order bit. He taught me to focus on the most important thing at every turn, whether it was an elusive sentence required to finish a paragraph, the key takeaway from a slide, or a single sentence summarizing a paper's novelty. Hari took me on as his student when my options were limited; this dissertation is my way of repaying that debt. Mohammad taught me to persist until a concept was clear to the point of being obvious. He made me appreciate the peace of mind that comes with mathematical rigor and showed me how to work with hardware engineers. Most importantly, he taught me how to listen and continuously incorporate feedback into my research.

Keith Winstein was a friend and mentor through grad school. He taught me to love clean code, corrected my writing, showed me how to give an engaging talk, and convinced me that it was always possible to program a computer to do your bidding. In April 2013, he wrote a blog post wondering why software-defined networking did not include the data plane. This dissertation is a belated answer to his question.

My thesis committee members, George Varghese and Nick McKeown, helped shape my research taste. George taught me to go beyond the surface in interdisciplinary research. He pushed me to discover precise connections to and differences from adjacent disciplines. Nick suggested that I spend some time interning at Barefoot Networks—an internship that significantly improved my dissertation. He showed me through his own example that it was possible to combine intellectual rigor and practical impact.

As part of my dissertation work, I spent a year interning at Barefoot Networks. It is not often that a fledgling startup gives an intern free rein to pursue open-ended research. My manager, Changhoon Kim, gave me the freedom to go after ideas that I thought were interesting, while ensuring that I was still productive. Mihai Budiu taught me how to engineer a compiler. Anurag Agrawal patiently explained a router's scheduler to me. In addition, I benefited from conversations with many other engineers and interns at Barefoot, including Antonin, KRam, Ravindra, CK, Pat, Mike, and Naga.

This dissertation is the result of joint work with many collaborators: Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, Steve Licking, Suvinay Subramanian, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Tom Edsall, Sachin Katti, Srinivas Narayana, Vikram Nathan, Prateesh Goyal, Venkat Arun, and Vimalkumar Jeyakumar. The broader P4 community provided an ideal setting for my work, both to get feedback and for translating some of these ideas into practice.

My MIT colleagues past and present, Ravi, Amy, Vikram, Tiffany, Shuo, Katrina, Lenin,

Jonathan, Peter, Hongzi, Pratiksha, Somak, Ameesh, Alvin, Eugene, Swarun, Jason, and Tushar, provided a great reason to come to work, whether to bounce off ideas, get feedback on a talk or paper draft, complain about grad school, get ice cream at Tosci's, or play an afternoon game of ping-pong. In particular, Ravi has been a great sounding board over the years, tempering my naive optimism with some practical reality checks.

Suvinay and I spent many years in grad school together. I am grateful for our friendship and for his ability to patiently explain hardware to me at all hours of the day and night. As undergrads, Srinivas and I spent many sleepless nights debugging problem sets together. Srinivas started his post doc at MIT just as I was looking around for ideas, allowing us to work together again on a slightly harder problem set: Chapter 6. My friends outside grad school, Raghav, Aakash, Akhil, Raghunandan, Siddharth, and Praneeth, provided me with much-needed breaks from work with the occasional catch ups.

Sheila Marian was a phenomenal admin assistant, assisting with paper work when I was away from MIT, booking travel for me, and ordering cakes for thesis defenses. Janet Fischer at the EECS graduate office patiently extended every Ph.D. deadline. My academic advisor, Piotr Indyk, looked out for me since my first year at MIT, especially when I was switching advisors. Sylvia Hiestand at the MIT ISO got all of my paper work in order during my year away from MIT.

My parents, Rama and Sivaraman, put up with my churlish ways during my time in grad school. I am grateful to them for their patience and for not asking me when I would graduate. My sister, Vibhaa, patiently proofread this dissertation and filed it on my behalf. My grandmother's utter lack of interest in my research is a much needed breath of fresh air. My in-laws were a steady source of support from afar during my job search. My wife, Tejaswi, went through both the highs and lows of my Ph.D. along with me. I am grateful for her ability to listen carefully, for believing in me for no good reason, and for her honest advice during difficult times.

My late grandfather, Dr. V. Ramamurti, was the reason I embarked on a Ph.D. Throughout high school, he spent an inordinate amount of time patiently teaching me mathematics and physics and indulging my pesky questions. As a faculty member who also consulted for industry he taught me not to stray too far from reality when conducting research. I dedicate this dissertation to him.

Chapter 1

Introduction

1.1 Background

Computer networks contain two types of elements: *end hosts* that generate packets and *routers*[1] that forward these packets between the end hosts. Historically, the Internet was architected so that most of the complexity resided in the end hosts, while the routers themselves were simple. According to Clark [95], this architecture was a result of the primary design goal of the Internet: the ability to easily interconnect a wide variety of existing networks (*e.g.,* long-haul networks, local-area networks, satellite networks, and radio networks) through a set of routers between these networks. Quoting Clark, "The Internet architecture achieves this flexibility by making a minimum set of assumptions about the function which the net will provide."

The minimum functionality assumed of and provided by the network's routers was best-effort and unreliable packet forwarding. Notably absent from a router's feature set were reliable packet delivery, packet prioritization, monitoring features to attribute a router's resource usage to specific end hosts, and security features to detect network breaches. As a result, the early routers were singularly dedicated to packet forwarding. A focus on packet forwarding alone made it simpler to design high-speed routers. It also helped broaden the Internet's reach by interconnecting existing networks with minimum friction. But, it sidelined other goals [95] such as network performance, security, and monitoring.

Today, four decades after ideas underlying the Internet were first published [90], it is clear that routers need to do much more than packet forwarding for at least two reasons. First, once the basic goal of interconnecting different networks is achieved, other goals like performance, security, and monitoring rise in prominence. Second, many large-scale private networks (*e.g.,* datacenters, private wide-area networks, and enterprise networks) do not need to concern themselves with interconnecting diverse networks as the Internet had to. Such private networks can expect more from the network. As a result, a typical router today implements many features beyond packet forwarding, pertaining to security (*e.g.,* access control), monitoring (*e.g.,* counting the number of packets belonging to each flow), and performance (*e.g.,* priority queues).

While a router's feature set has grown steadily with time, there's little consensus between

[1] We use the term router to refer to both switches and routers in this dissertation.

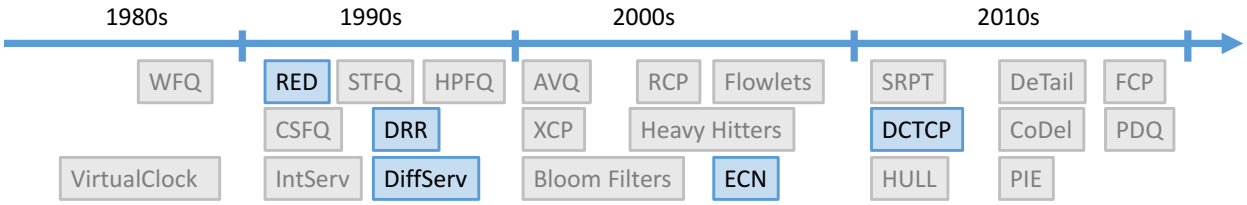


Figure 1-1: Timeline of prominent router algorithms since the 1980s. Only the ones shaded in blue are available on the fastest routers today.

network operators and router vendors on a router's feature set. Inevitably, there are network operators whose needs fall outside their router's feature set. But, because today's fastest routers are built out of specialized forwarding hardware, they are largely *fixed-function*[2] devices, *i.e.*, their functionality cannot be changed once the router has been built. In such cases, the operator has no alternative but to wait 2–3 years for the next generation of the router hardware. This is best illustrated by the lag time between the standardization and availability of new overlay protocols [34].

As a result, the rate of innovation in new router algorithms is outstripping our ability to get these algorithms into today's fastest routers, *i.e.*, routers with between 10 and 100 ports, each running at between 10 and 100 Gbit/s. Figure 1-1 shows a timeline of prominent router algorithms that have been developed since the 1980s. Of these, only a handful are available in the fastest routers today because there is no way to program a new router algorithm on these routers.

Against this backdrop, if an operator wants to introduce new network functionality, what are her choices? One is to give up on changing routers altogether and make all the required changes at the end hosts. Relying solely on the end hosts, however, results in cumbersome or suboptimal solutions. As a first example, imagine measuring the queueing latency at a particular hop in the network. One could do this by collecting end-to-end ping measurements between a variety of vantage points and then fusing these measurements together to estimate per-hop queueing latency—a process commonly called network tomography. Not only is this indirect, it is also inaccurate relative to directly instrumenting the router at that hop to measure its own queueing latency. As a second example, consider the problem of congestion control, which divides up a network's capacity fairly among competing users. There are many in-network solutions to congestion control [139, 195], which outperform the end-host-only approaches to congestion control used today [122, 198]. However, there is no way to deploy these in-network solutions using a fixed-function router today.

Another alternative is to use a *software router*: a catch-all term for a router built on top of some programmable substrate, such as a general-purpose CPU [144, 103], a network processor,[3] a graphics processing unit (GPU) [123], or a field-programmable gate array (FPGA) [216]. Figure 1-2 tracks the evolution of aggregate capacity of software routers and compares them to the fastest routers known at any point in time. The figure shows two trends. First, until the mid 1990s, software routers were in fact the fastest routers; the early routers [126] were minicomputers loaded with forwarding software. Second, since the mid 1990s, growing demands for higher link speeds, fueled

[2]The term fixed-function was first used to describe graphics processing units (GPUs) with limited or no programmability [16]. We use it in an analogous sense here.

[3]A CPU with an instruction set tailored to packet processing [29, 26].

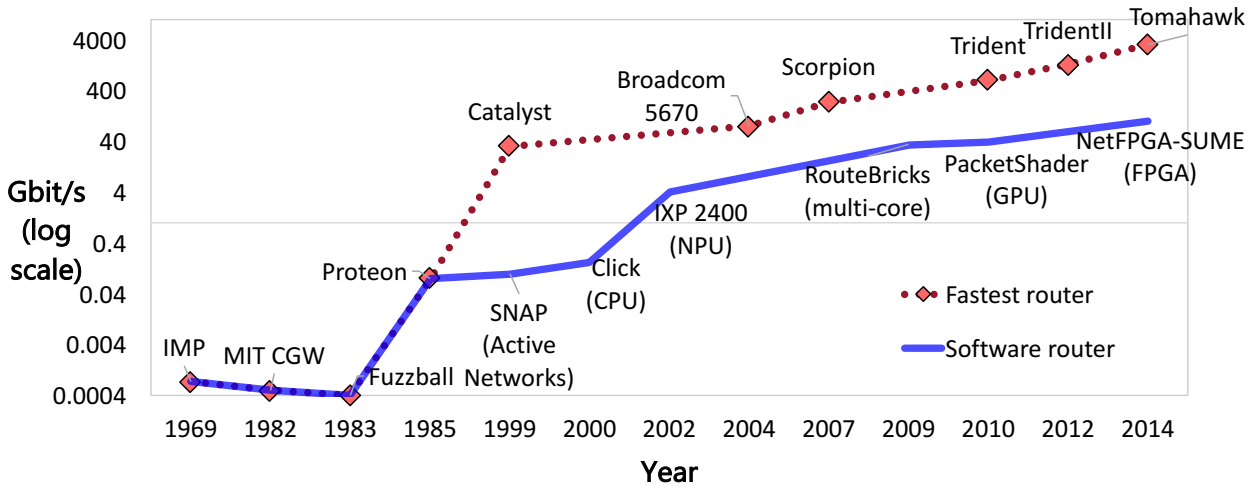


Figure 1-2: Aggregate capacity of software routers since the first router on the ARPANET in 1969 [126]. Until the mid 1990s, software routers were sufficient. Since then, however, the fastest routers have largely been fixed-function devices, built out of dedicated non-programmable hardware, which gives these routers a 10–100x performance improvement relative to the best software routers.

by the Internet's growth, have meant that the fastest routers are now built out of dedicated hardware.

Hardware specialization gives today's fastest routers a 10–100 $\times$ performance improvement relative to the fastest software routers. This performance improvement is the result of fully exploiting the abundant parallelism available in packet processing. First, data parallelism, the ability to simultaneously process either different parts of the same packet or packets belonging to different ports. Second, pipeline parallelism, the ability to simultaneously perform different operations on different packets. But, hardware specialization carries a cost: because routers are built out of specialized hardware, they are fixed-function devices that can not be programmed in the field.

Recent work in software-defined networking [108] (SDN) and programmable router chips [86, 60, 25] has endowed fast routers with limited flexibility. SDN allows operators to program the network's control plane, which is the part of the network that computes a network's routing tables. SDN does this by moving route computations out of the routers and on to a logically centralized programmable server running on a CPU.

Programmable router chips allow operators to program parts of the data plane: the part of the network that forwards packets based on the routing tables. For instance, these chips allow an operator to program the router's parser to recognize new packet headers, such as a new overlay format [34]. They also allow the operator to program packet header transformations (*e.g.,* decrementing the IP TTL field) so long as these transformations do not modify router state.

However, both SDN and programmable router chips are still insufficient to express the grayed-out algorithms shown in Figure 1-1 because (1) these algorithms programmatically manipulate router state and (2) they require flexibility in packet scheduling. State modification on routers and the router's scheduler are largely ignored by both SDN and programmable router chips.

1.2 Primary contributions

This dissertation considers the problem of designing routers that approach the speeds of today's fastest fixed-function routers, while also being programmable. My thesis is that *it is possible to design router hardware that is both fast and programmable, if we restrict ourselves to programming specific classes of router functionality*. It is this specificity that allows us to resolve the programmability-performance tension; indeed, our designs provide a much more restricted form of programmability than a Turing-complete processor.

The challenge here is to pick classes of router functionality that are simultaneously (1) practically useful to network operators, (2) broad enough to cover a range of current and future use cases within that class, and yet (3) narrowly focused enough to permit a high-speed hardware implementation. We will describe high-speed programmable hardware primitives and their corresponding programming models in software for three classes of router functionality: stateful data-plane algorithms, packet scheduling, and scalable network measurement. Table 1.1 summarizes our contributions. §1.3, §1.4, and §1.5 expand on each of the three contributions.

1.2.1 Evaluation metrics

The goal of this dissertation is to design fast and programmable routers. Accordingly, we measure each of our three systems on two attributes, performance and programmability, as described below.

The traditional approach to evaluating performance of a software system is to measure the system's throughput on some workloads. This approach does not apply to evaluating hardware designs, which are built for a specific clock rate or throughput. Hardware designs provide this clock rate or throughput even under worst-case conditions, regardless of the workload presented to them.

To provide worst-case guarantees, hardware designs exploit spatial parallelism—by chaining together computations in a pipeline (pipeline parallelism) and simultaneously performing multiple operations within a pipeline stage (data parallelism). Put differently, hardware designs spend circuit area in return for deterministic high performance. The question then is whether these designs consume a large amount of area to provide such performance guarantees and whether they can provide acceptable performance, as measured by the design's clock rate.

Hence, to evaluate performance, we estimate if the hardware components of our systems can run at a high clock rate while not taking up too much gate and wire area. To do so, we code any new hardware components as programs in the Verilog hardware description language. We then pass this Verilog program to a logic synthesis tool [12, 6] that produces a gate-level implementation from Verilog programs. The synthesis tool also reports whether the resulting implementation meets timing at a given clock frequency and the area taken up by the gates in the implementation.[4] When we say a hardware design is feasible, we mean that it meets timing at a high clock frequency (1 GHz in this dissertation) and its gate area is modest relative to the area of a router chip (200 mm^2 in this dissertation based on the minimum area estimates provided by Gibb et al. [115]).

[4] A full hardware implementation also requires a place-and-route [41] step after logic synthesis to physically place these gates and route wires between them. This increases the area of designs, especially if the design is dominated by wires, as is the case with crossbars. Because our designs are dominated by gates, not wires, we do not perform this place-and-route step when estimating area.

Stateful data-plane algorithms (Chapter 4)

Examples: In-network congestion control (*e.g.,* XCP [139] and RCP [195]), active queue management (*e.g.,* RED [111], BLUE [109], and CoDel [169])
Technical challenge: How do we allow programmable router state modification at the router's line rate, when a new packet can be received as often as every nanosecond?
Programming model: Packet transactions (§4.2)
Hardware primitive: Atoms (§4.1)
Key finding: A small set of atoms (Table 4.4) is simultaneously (1) expressive enough to serve as the instruction set for many stateful algorithms (Table 4.5) and (2) feasible in high-speed hardware (§4.4). Further, we find that these atoms can support several new use cases that were unanticipated at the time the atoms were designed (Table 4.6).

Scheduling algorithms (Chapter 5)

Examples: Weighted Fair Queueing [99] and priority scheduling [183]
Technical challenge: Can we find an abstraction that unifies many disparate scheduling algorithms?
Programming model: Scheduling trees (§5.2)
Hardware primitive: A priority queue data structure called a Push In First Out Queue (PIFO) (§5.4)
Key finding: A priority queue of packets with a program to set each packet's priority can express many scheduling algorithms (§5.3) and is feasible in high-speed hardware (§5.5).

Scalable per-flow statistics (Chapter 6)

Examples: Per-flow measurements of moving averages, counters, and loss rates
Technical challenge: Can we allow programmers to flexibly define the per-flow statistics they want to measure and also scale these measurements to a large number of flows?
Programming model: Performance queries (§6.1)
Hardware primitive: Programmable hardware key-value store. Keys correspond to flows and values to statistics. (§6.2)
Key finding: A class of statistics measurements, which we call the linear-in-state class (§6.2.2), can be scaled to a large number of flows without losing accuracy. This class covers many practically useful statistics such as counters, moving average filters, and conditional counters (§6.4).

Table 1.1: Contributions of this dissertation

To evaluate if our systems are programmable, we evaluate the expressiveness of the programming models in each of our systems. If we are able to express a diverse set of algorithms using a programming model, we say that the programming model is expressive. Our approach to expressiveness is empirical: beyond specific example programs, we do not have theoretical characterizations of the set of programs that can or cannot be programmed using our programming models (§7.1.2).

In addition, to assess the correctness of our designs, we built a C++ simulator of a programmable router that models the behavior of the essential computational elements of a programmable router at the level of individual clock cycles (§4.1).

1.3 Stateful data-plane algorithms

In Chapter 4, we consider the problem of programming *stateful data-plane algorithms* at high packet processing rates. These are algorithms that operate on a sequence of packets in a streaming manner, doing a bounded amount of work per packet and manipulating a bounded amount of router state in the process. They include algorithms for managing the router's buffer (*e.g.,* RED [111] and BLUE [109]), load balancing (*e.g.,* CONGA [65] and flowlet switching [187]), and in-network congestion control (*e.g.,* XCP [139] and RCP [195]).

High-speed data-plane programming on routers poses two challenges: (1) what hardware instructions are required to support programmable state modification at high speeds, and (2) what is the right programming model? To address these challenges, we develop a system for data-plane programming, Domino, which contains three main components: an instruction set (atoms), a programming model (packet transactions), and a compiler from packet transactions to atoms.

1.3.1 Atoms: Hardware for high-speed state manipulation

Atoms capture a router's instruction set. They specify atomic units of packet processing provided by the router hardware, *e.g.,* an atomic counter or an atomic test-and-set. Figure 1-3 shows an example atom. Atoms are atomic in the sense that if some state is updated by an atom as part of processing a packet, the next packet arriving at that atom will see the updated value of that state. The processing within atoms is constrained to meet the atomicity requirement. We enforce this constraint when designing atoms in hardware by ensuring that the input-to-output latency of the atom's digital circuit is under a clock cycle. This guarantees that any state updated by the atom has been updated to its correct value in time for the next packet arriving at the atom a clock cycle later.[5]

1.3.2 Packet transactions: A programming model for stateful algorithms

Packet transactions provide a programming model for data-plane algorithms. A packet transaction is an atomic and isolated block of code capturing an algorithm's logic written in a domain-specific language (DSL) called Domino. Figure 1-4 shows an example packet transaction. Packet transactions provide programmers with the illusion that the transaction's body executes serially from

[5]Note that this is a sufficient condition for atomicity and may not always be necessary.

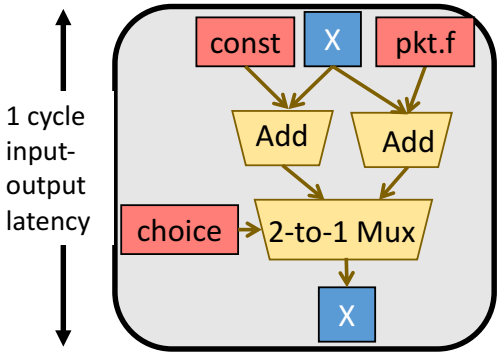


Figure 1-3: An atom that either adds either a constant or a packet field to a piece of state x and writes it back to x.

```
if (count == 9):
  pkt.sample = pkt.src
  count = 0
else:
  pkt.sample = 0
  count++
```

Figure 1-4: A packet transaction that samples the source IP address of every 10th packet. State variables (count) are in red.

start to finish on each packet, in the order in which packets arrive at the router. Conceptually, when programming with packet transactions, there is no overlap in packet processing across packets—akin to an infinitely fast single-threaded processor carrying out packet processing on each packet. Packet transactions are expressive and capture many important data-plane algorithms. Further, their serial semantics shield programmers from the router's data and pipeline parallelism.

1.3.3 Compiling from packet transactions to atoms

The Domino compiler compiles packet transactions written in the Domino DSL to a pipeline of atoms provided by the router. It rejects the code if the router's atoms cannot support the packet transaction. The compiler bridges the gap between the serial single-threaded execution model seen by the programmer and the data-parallel and pipeline-parallel execution model of the router.

The compiler has three passes (Figure 1-5). First (§4.3.1), it preprocesses the code to make it easier to infer dependencies between packet-processing operations. Second (§4.3.2), the compiler transforms the preprocessed, but still serial, packet transaction into a parallel pipeline of *codelets*. When pipelining, the compiler maintains the following invariant: if each codelet executes atomically and passes off its results (*e.g.,* pkt.tmp in Figure 1-5) to the next codelet, the behavior will be indistinguishable from the packet transaction itself executing atomically on each packet. Third (§4.3.3), the compiler maps each codelet one-to-one to an atom provided by the underlying router, rejecting the code if the codelet cannot be supported by an atom.

The compiler provides an *all-or-nothing* guarantee: if the compiler compiles a program, the program will run at the line rate of the router.[6] All other programs that cannot run at the router's

[6]We use the term line rate in this dissertation to mean both the maximum bit/packet rate of a particular router port/line (*e.g.,* 10 Gbit/s) and the maximum aggregate bit/packet rate of a router: the product of the per-port line rate and the number of ports (*e.g.,* 64×10 Gbit/s = 640 Gbit/s). It should be clear from the context which interpretation of line rate we are referring to.

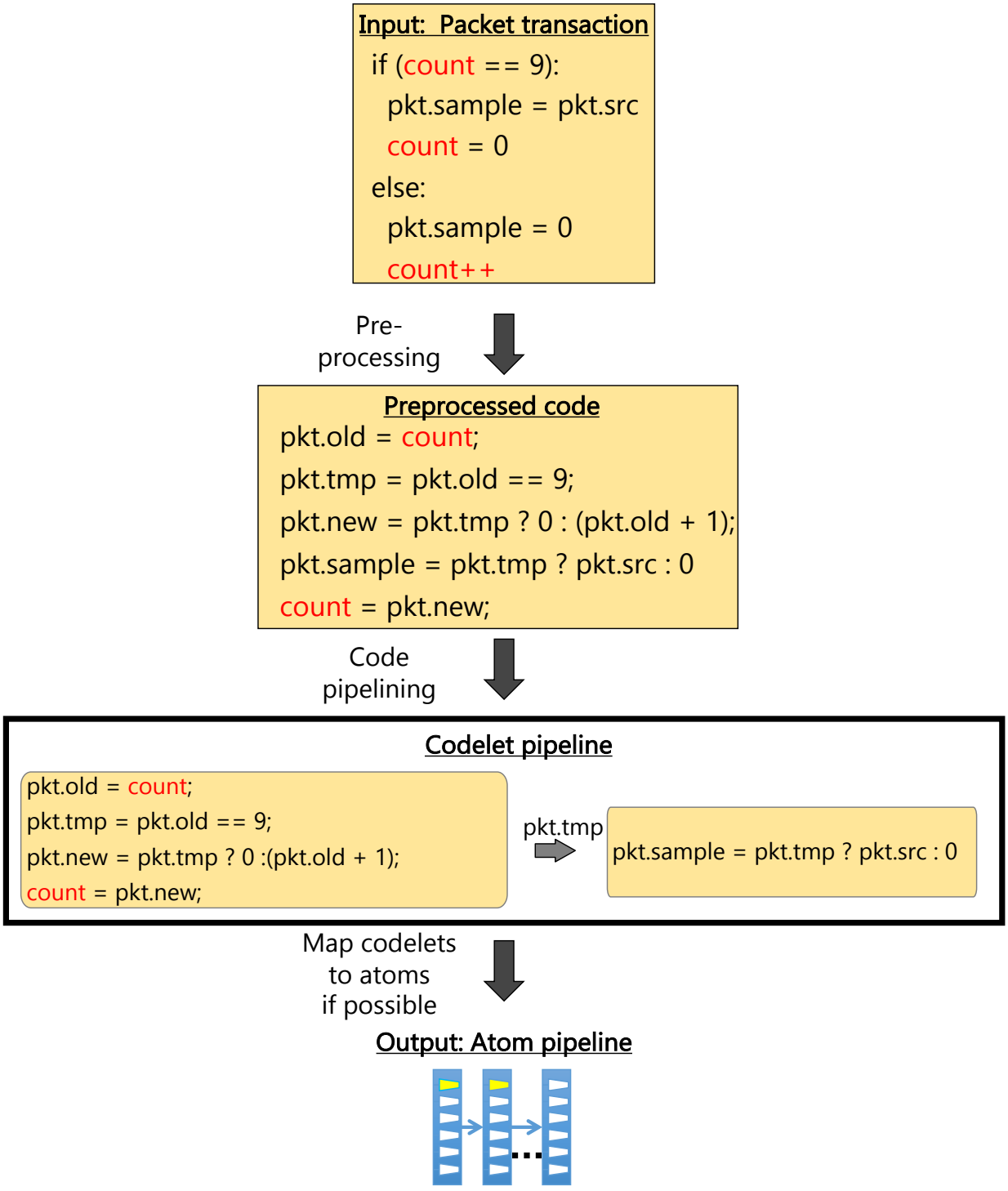


Figure 1-5: The three passes of Domino's compiler shown on the packet transaction from Figure 1-4.

line rate will be rejected. A program may be rejected either because the router's atoms are not expressive enough for the program's operations or there aren't enough atoms for all operations

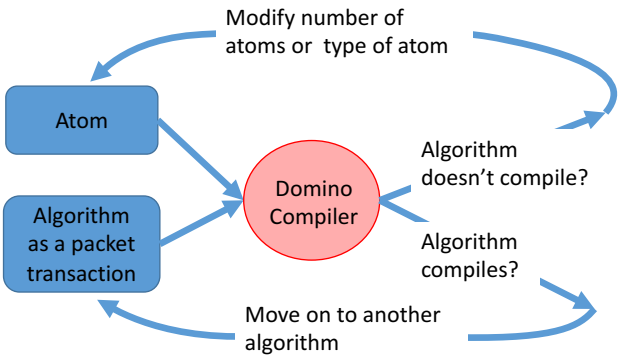


Figure 1-6: Iterative atom design process

in the program. A conventional compiler for a general-purpose CPU compiles all programs, but a program's run-time performance depends on its complexity. In Domino, only programs that are simple enough to run at the router's line rate will be compiled, obviating the need for any performance profiling.

1.3.4 Evaluation

Developing atoms for a router is a chicken-and-egg problem. A router's atoms determine what algorithms the router can support, while the algorithms determine what atoms are required in the first place. Designing the right atoms is especially important for a programmable router because, in contrast to a general-purpose CPU, there is no way to "emulate" functionality in software when hardware support is not available: recall that if an atom does not exist to support a program's operations, the program will be rejected.

To develop atoms, we use the Domino compiler to experiment with different atoms and iteratively modify the atoms until they support enough algorithms (Figure 1-6). Using this process, we developed seven atoms of increasing complexity (Table 4.4) that allow us to progressively program more and more algorithms from Figure 1-1. For instance, measurement using Bloom filters only requires simple read and write operations on state. On the other hand, heavy-hitter detection [210] uses a count-min sketch [96] that relies on an atomic counter. Finally, algorithms like flowlet switching [187] require conditional writes to a state variable.

After freezing the design of these atoms, we found that our atoms could express several new use cases that were unanticipated when the atoms were designed (Table 4.6). This provides us with some evidence that these atoms are indeed programmable and can generalize to new algorithms beyond the initial algorithms that influenced their design in the first place.

1.4 Programmable packet scheduling

Packet scheduling is an important determinant of network performance. The choice of scheduling algorithm is tied to a network's performance goals. For instance, an algorithm that divides link

capacity fairly is ideal in a multi-tenant setting [99], while the shortest remaining processing time algorithm is ideal for a single tenant who desires low flow completion time [68]. Today's routers provide a fixed set of scheduling algorithms (*e.g.,* priority queues, Deficit Round Robin [186], and rate limiting [58]). While configuration settings on these scheduling algorithms can be tweaked, there is no way for an operator to program a new scheduling algorithm that is tailored to their needs.

Routers lack programmable scheduling because there is no single abstraction to express many scheduling algorithms [99, 183, 180, 68, 150] that appear so different at first brush. This lack of a single unifying abstraction is not merely an academic concern. It has practical consequences: in the absence of a single abstraction that can then be hardened in hardware, we are left with using a general-purpose substrate such as a CPU to program scheduling. As we mentioned earlier, this can hurt performance substantially.

Push In First Out Queues (PIFOs) (Chapter 5) provide such an abstraction.[7] They exploit our observation that in many practical schedulers the relative order of packets that are already buffered does not change in response to new packet arrivals (§5.1). Hence, when a packet arrives, it can be pushed into the right location based on a packet priority (push in), but packets are always dequeued from the head (first out). A PIFO is simply a priority queue of packets with a small program to assign each packet its priority.[8] Yet, by flexibly programming a packet's priority assignment, a network operator can use PIFOs to program a variety of previously proposed scheduling algorithms.

1.4.1 Programming model for packet scheduling

Our programming model for scheduling couples a PIFO with a program to determine a packet's *rank* in the PIFO. This rank can denote the packet's scheduling order (for work-conserving algorithms such as Weighted Fair Queueing [99]) or absolute wall-clock departure time (for non-work-conserving algorithms such as traffic shaping [58]). This program is written as a packet transaction, introduced earlier. Depending on whether the program determines the scheduling order or time, the program is called either a scheduling transaction or a shaping transaction respectively.

A single PIFO coupled with a scheduling or shaping transaction can express many classical scheduling algorithms, *e.g.,* Weighted Fair Queueing (Figure 1-7a) and Token Bucket Shaping (Figure 1-7b). But, a single PIFO is still restricted to scheduling algorithms with the property that the relative order of packets that are already buffered does not change in response to new packet arrivals. A canonical class of scheduling algorithms that violate this relative order property is the class of hierarchical scheduling algorithms. A well-known example of this class is Hierarchical Packet Fair Queueing (HPFQ) [76]. HPFQ first divides up the link's capacity fairly among classes, and then when each class is scheduled, divides up the class's transmission opportunities fairly among its constituent flows. It can be thought of a recursive version of fair queueing.

To support hierarchical scheduling, we extend our programming model from a single PIFO to a tree of PIFOs. We also allow an entry in a PIFO to be either a packet or a reference to another PIFO.

[7] PIFOs were used as a theoretical construct to establish the equivalence of combined input-output queued routers and output-queued routers [94]. We show here that they can be used practically for programmable packet scheduling.

[8] We use the term PIFO instead of priority queue because, within the context of networking, priority queues typically refer to an implementation of work-conserving priority scheduling. PIFOs can be used for both work-conserving and non-work-conserving algorithms (§5.3).

```
f = flow(p) # compute flow from packet p
if f in last_finish:
 p.start = max(virtual_time, last_finish[f])
else: # p is first packet in f
 p.start = virtual_time
last_finish[f] = p.start + p.length/f.weight
p.rank = p.start
```

(a) Scheduling transaction for the Start-Time Fair Queueing implementation [118] of Weighted Fair Queueing.

```
tokens = tokens + r * (now - last_time)
if (tokens > B):
  tokens = B
if (p.length <= tokens):
  p.send_time = now
else:
  p.send_time = now + (p.length - tokens) / r
tokens = tokens - p.length
last_time = now
p.rank = p.send_time
```

(b) Shaping transaction for Token Bucket Shaping.

Figure 1-7: Examples of scheduling and shaping transactions. `p.x` refers to a packet field `x` in packet `p`. `y` refers to a state variable that is persisted on the router across packets, *e.g.,* `last_finish` and `virtual_time` in this snippet. `p.rank` denotes the packet's computed rank.

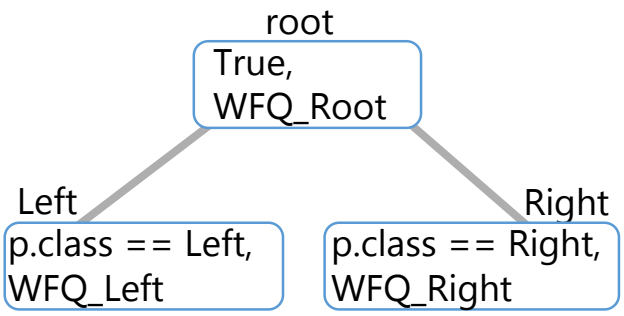


Figure 1-8: Scheduling tree for HPFQ. The scheduling transactions WFQ_Root, WFQ_Left, and WFQ_Right are similar to Figure 1-7a and differ only in how the flow is computed from the packet. This tree has no shaping transaction.

More formally, a node in a *scheduling tree* (Figure 1-8) has three attributes: (1) a predicate that determines which packets are handled by that node, (2) a scheduling transaction that determines a packet's rank in a scheduling PIFO attached to that node, and (3) an optional shaping transaction that determines a packet's rank in an optional shaping PIFO attached to that node. The shaping transaction is optional because it is only relevant to non-work-conserving schedulers. It can be disregarded entirely for work-conserving schedulers.

We now explain the semantics of a scheduling tree during dequeues and enqueues. During a dequeue, we walk the scheduling tree starting from the root. We dequeue the scheduling PIFO at the root, resulting in either a reference to another scheduling PIFO or a packet. If the result is a packet, we are done, and we transmit the packet. If not, we continue recursively, dequeueing from the scheduling PIFO that is pointed to until we find a packet.

When a packet is enqueued, we walk the tree from the leaf whose predicate captures the packet to the root. Along this leaf-to-root path, we execute scheduling and (optionally) shaping transactions attached to the path's nodes. Figure 1-9 illustrates the timing of various operations related to a node's scheduling and a shaping PIFO. A node's scheduling transaction inserts either a packet or a

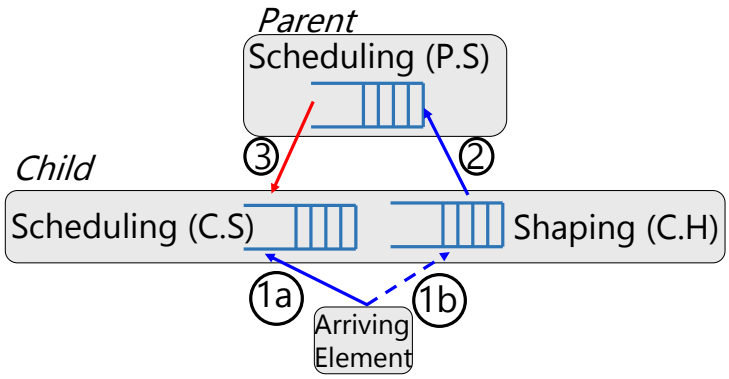


Figure 1-9: *Child's* shaping transaction (1b) *defers* enqueue into *Parent's* scheduling PIFO (2) until time T. Blue arrows show enqueue paths. Red arrows show dequeue paths.

PIFO reference into the node's scheduling PIFO. A node's shaping transaction inserts a reference R to the node's scheduling PIFO in the node's shaping PIFO. Once the shaping transaction has executed, execution of transactions for this packet is temporarily suspended. When the reference R is dequeued from the shaping PIFO, it is enqueued into the node's parent's scheduling PIFO using the parent's scheduling transaction and the rest of the leaf-to-root path is resumed. The shaping PIFO thus provides an optional mechanism for a node to *defer* enqueues into its parent's scheduling PIFO, which is useful when combining hierarchical scheduling with traffic shaping (Figure 5-4 in Chapter 5 shows an example.).

1.4.2 Expressiveness of our programming model

The PIFO-based programming model unifies several scheduling algorithms and allows us to program a wide variety of scheduling algorithms using the single PIFO primitive, *e.g.,* Weighted Fair Queueing [99], Token Bucket Shaping [58], Hierarchical Packet Fair Queueing [76], Least-Slack Time-First [150], the Rate-Controlled Service Disciplines [214], and fine-grained priority scheduling (*e.g.,* Shortest Job First).

1.4.3 Hardware for programmable scheduling

We designed high-speed hardware to support a PIFO-based programming model. When designing hardware for PIFOs, we set out to meet performance targets that are typical for a single-chip shared-memory router today. These are routers that are built out of a single chip,[9] and whose packet buffer and scheduling logic is shared across all ports. Sharing the packet buffer and scheduling logic reduces the memory and digital logic cost associated with packet scheduling.

[9] Also called an application-specific integrated circuit or ASIC.

For concreteness, we picked a target clock frequency of 1 GHz to reflect the requirement of performing one enqueue and one dequeue every nanosecond, typical of high-speed routers today [86]. We targeted a packet buffer of size 12 MByte based on the buffer size of the Broadcom Trident II [40], a commercial single-chip shared-memory router. With a cell size of 200 bytes,[10] a 12 MByte buffer can support up to 60000 packets in the worst case.

Hence we need a PIFO that can support up to 60K packets. A naive way to implement a 60K-entry PIFO is to use a flat sorted array of 60K elements and insert an incoming element into this array in a manner reminiscent of insertion sort. Concretely, an incoming element's rank would be compared in parallel to the ranks of all 60K elements. This would produce a bitmask denoting which elements were greater than or lesser than the incoming element. Because the array is always sorted, the bitmask would have a single 0-to-1 transition. The position of this 0-to-1 transition could be detected using a priority encoder. Finally, the new element could be inserted into this position. However, the difficulty with this approach of using a flat sorted array is that it is hard to lay out 60K parallel comparators, one for each element in the PIFO.

Instead, we exploit the observation that in most practical scheduling algorithms, scheduling is performed across flows and not individual packets. This is because, in most schedulers, packet ranks increase monotonically across consecutive packets within a flow. This ensures that packets within a flow are transmitted in the order in which they arrive, thereby preventing packet reordering within a flow.

Because ranks increase monotonically within a flow, we only need to look at the first packet of each flow to determine which packet to dequeue next. This reduces the number of elements that need to be sorted from 60K packets in the naive implementation to around 1K flows in the smarter implementation. We picked this 1K number with reference to the Broadcom Trident II that supports ~10 queues on each of its ~100 ports.

We find that transistor technology has evolved to the point where it is relatively cheap to build a sorted array of 1K flows, where a flow can be enqueued into or dequeued from the array every nanosecond. In a recent industry-standard 16 nm technology node, a hardware design for a programmable 5-level hierarchical scheduler costs less than 4% additional chip area relative to a 200 mm^2 baseline router chip [116].

1.5 Programmable and scalable network measurement

Chapter 6 considers the problem of programmable and scalable network measurement. Concretely, can we allow network operators to flexibly specify the per-flow statistics they want to measure (*e.g.,* a moving average over queueing latencies or a count of packets or bytes) at the flow granularity they desire (*e.g.,* at the 5-tuple level or at the level of each distinct destination IP address)? Current router solutions for measurement [8, 11] fix either the statistics that can be measured or the granularity at which the statistics can be measured. The challenge here is to provide a set of measurement primitives that can cover a range of statistics while also scaling to a large number of flows, because the granularity of a flow can be as fine-grained as the 5-tuple.

[10] A cell is the minimum unit of memory allocation in the packet buffer.

```
result = filter(pktstream, qid == Q && tout - tin > 1ms)
// Filter out packets whose queueing delay at queue Q exceeds one ms.

result = map(pktstream, [tin/epoch_size], [epoch])
// Round off packet time stamps to the nearest epoch of size epoch_size.

def count([c], []):
  c = c + 1
result = groupby(pktstream, [5tuple], count)
// Partition packets by 5-tuple and count the number of packets in each 5-tuple.
```

Figure 1-10: Three example Marple queries. `pktstream` refers to the original packet performance stream with one tuple for each packet seen in the network.

1.5.1 Programming model

To allow network operators to express a wide range of performance measurement questions, we provide them with the abstraction of a single performance packet stream for the entire network. Conceptually, the performance packet stream is a stream of tuples, one for each packet, containing information identifying the packet (source and destination address and ports) and its performance information (the timestamp at which the packet was enqueued/dequeued at each network queue).

Network operators can write queries that operate on this performance packet stream using a query language called Marple. Marple has functional constructs like `map`, `filter`, and `groupby` similar to functional APIs in programming languages like Python, Java, Scala, and Haskell. Figure 1-10 shows some example queries in Marple. These constructs take a stream of tuples as input and return an output stream of tuples. Because all constructs produce and consume streams, Marple queries can be easily composed.

The output stream either transforms each tuple in the input stream (`map`), drops certain tuples from the input stream based on a predicate (`filter`), or partitions the input stream into substreams and then aggregates the tuples within each substream based on a user-defined aggregation function (`groupby`). Marple's use of user-defined aggregation functions differentiates its groupby construct from the groupby construct supported by classical query languages like SQL. SQL only supports specific order-independent aggregation functions like counts and averages. Unlike Marple, SQL does not allow general, user-defined, and potentially order-dependent aggregation functions.

1.5.2 Hardware for programmable network measurement

A naive implementation of performance queries would stream relevant performance information for every packet to a centralized collection server, which would execute the performance query on the incoming packet stream. However, this would require enormous computational capacity on the server and a separate measurement network just to stream per-packet information to the server. For instance, modern stream processing systems support a throughput of ~100K–1M operations per second per core [4, 21, 125, 2, 48], but processing every single packet from a single 1 Tbit/s

router requires around ~100M operations per second even for relatively large 10000 bit packets. Instead, we use programmable routers as first-class citizens to perform early filtering, aggregation, and packet transformations. The net effect is a substantial reduction in the amount of data sent out to and processed by the collection server.

Many of Marple's constructs (*e.g.,* `map` and `filter`) are *stateless* in the sense that they do not need to maintain any router state. For instance, a `map` query could take an input packet stream, round off the enqueue timestamp of each packet in the stream to the nearest millisecond, and return the result as a new stream. This query is stateless because it can operate independently on each packet without maintaining any state on the router that persists between packets. Supporting stateless queries is relatively straightforward. Emerging programmable router chips [86, 60, 25, 3] already provide an instruction set that performs stateless transformations on packet headers. We leverage this instruction set as is to execute stateless queries because a packet-processing pipeline naturally fits the streaming model of our queries.

On the other hand, the `groupby` query is *stateful*. To see why, consider a `groupy` query that (1) partitions the packet stream into substreams based on the 5-tuple and (2) counts the number of packets within each substream. This query is stateful because it needs to maintain a dictionary mapping each 5-tuple to its count as part of the persistent router state. To implement the `groubpy` query, this dictionary is updated by incrementing some 5-tuple's counter on each packet. To support Marple's `groupby`, we design a programmable key-value store in hardware. The keys represent the flows and the values represent the state being aggregated according to the aggregation function as packets are processed. In the example above, the key would be a 5-tuple and the value would be a count of packets belonging to that 5-tuple.

This key-value store has two requirements that are at odds with each other: it needs to be fast to process packets at the router's line rate (a packet every nanosecond), and it needs to be large to store a large number of flows. Static random-access memory (SRAM) can be accessed once every nanosecond, but has low memory density and can only fit a small number of flows. On the other hand, dynamic random-access memory (DRAM) has much better density allowing it to support many more flows, but can be accessed only once every ten nanoseconds or so.

We use caching to address both requirements: a fast SRAM cache stores the currently active flows, while a backing store in DRAM handles evictions from the cache. However, cache misses in traditional processor designs lead to unpredictable memory access latencies while the data is being fetched from the main memory. A processor handles this by occasionally stalling its pipeline to handle a cache miss. But stalling a router pipeline prevents us from providing the deterministic latency guarantees that routers are known and benchmarked for.

Instead, as Figure 1-11 shows, we treat a cache miss as the arrival of a packet from a new flow and initialize the cache entry as though we have just started aggregating packets from a new flow. When this cache entry is eventually evicted, we *merge* it against the old value for that entry in the backing store. The merge operation requires reading and writing from DRAM, which does incur unpredictable access latencies. But, it happens off the critical path during evictions alone. Importantly, it does not affect the critical path of packet processing in the router's chip.

What exactly does this merge operation look like? As a simple example, if the value in the key-value store was a counter; the merge operation would then simply add up the old and new

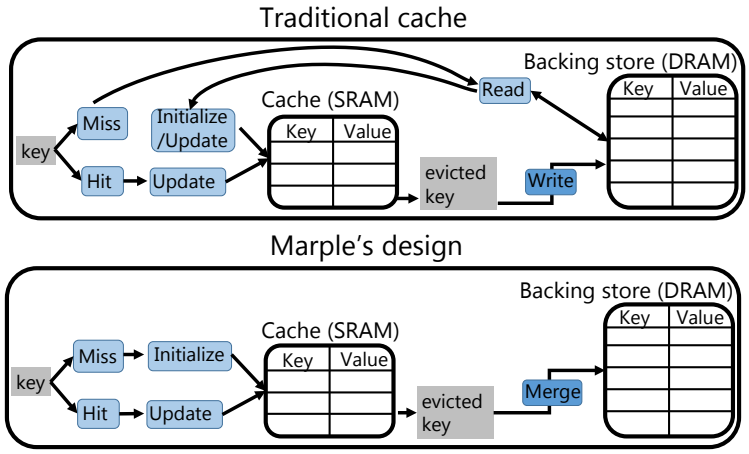


Figure 1-11: Marple's key-value store vs. a traditional cache

counts. But is it always possible to merge an aggregation function accurately *i.e.*, the value after the merge should be the same as the value that would be obtained in an ideal system with an infinitely large cache that needs no merging?

We can prove (§6.2.5.3) that there are aggregation functions for which it is not possible to merge efficiently without losing accuracy. However, we formally characterize a class of aggregation functions for which such merging *is* possible without losing accuracy. We call this class the *linear-in-state* class because aggregation functions in this class take the following form:

$$S = A(p) * S + B(p) \tag{1.1}$$

Here S is the state/value that is being updated by the aggregation function, while A and B are functions of a bounded number of packets into the past including the current one. The linear-in-state class captures a variety of aggregation functions including counters, predicated counters, moving average filters, and arbitrary functions on sliding windows.

1.5.3 Evaluation

We evaluate Marple's expressive power by using it to express several diverse performance queries, *e.g.,* tracking the loss rate on a per-flow basis, measuring the extent of reordering in a TCP flow, measuring a moving average of queueing latencies on a per-flow basis, and detecting the presence of TCP incast [202].

Our Marple compiler compiles queries to two different targets: an atom pipeline to determine the hardware feasibility of Marple queries and a P4-based software router [37] for end-to-end case studies of Marple in Mininet [149]. We find that Marple queries occupy a small fraction of the router's computational and memory resources. Computationally, queries require a small number of atoms. Memory-wise, the programmable key-value store requires an SRAM cache that occupies a modest amount of additional chip area. Our Mininet experiments show that Marple can be used to iteratively troubleshoot problems such as occasional latency spikes that occur because of bursty

background traffic.

We determine the eviction rate from the SRAM cache using a trace-driven simulation on publicly available packet traces from CAIDA [54, 53] and a university datacenter [78]. For a 64 Mbit on-chip cache, which occupies about 10% of the area of a 64×10-Gbit/s router chip, we estimate that the cache eviction rate from a single top-of-rack router can be handled by a single 8-core server running Redis [43]. Relative to a strawman that sends every packet to the collection server (*i.e.*, a 100% eviction rate), the on-chip cache results in a reduction in eviction rate of 34–81x (Figure 6-8).

1.6 Lessons learned

We conclude this chapter by distilling some general lessons for designing fast and programmable routers in the future that go beyond the specific contributions of each of the three systems.

1.6.1 The power of specialization

The most important lesson here is the power of specialization: the idea of targeting hardware and software to specific classes of router functionality. The three systems in the dissertation demonstrate how narrowing our focus to specific router functionality allows us to resolve the performance-programmability tradeoff that has affected software routers so far. Put differently, it allows us to get the best of both worlds, but for restricted classes of router functionality.

More formally, the three systems are specialized in the sense that they are not Turing-complete: they cannot simulate a Turing machine, an idealization of the general-purpose CPU. We refer the reader to §4.1.4 and Table 4.1, §5.3.5, and §6.3 for specific examples of what each of the three systems cannot express.

Yet, each covers a large number of use cases within specific functionality classes (stateful data-plane algorithms, scheduling, and measurement queries) without giving up performance relative to a fixed-function router. Viewed differently, by specializing, each system provides an order of magnitude improvement in performance relative to a general-purpose software router. Intellectually, these results suggest that there is a rich space of system designs that is as yet unexplored—if we choose to look beyond Turing-complete systems.

1.6.2 Jointly designing hardware and software

We are entering an era where Moore's law has slowed down or effectively stopped [127]. In this new era, transistors are no longer guaranteed to get smaller or faster every year. Besides, it may not be cost effective to move to smaller or faster transistors because of rising non-recurrent engineering costs associated with newer transistor technologies [142]. In the heyday of Moore's law, processor hardware automatically got faster year on year. Software automatically enjoyed the free lunch of improved performance because of improvements in the underlying hardware. With the imminent end of Moore's law, it is no longer apparent how these performance improvements will be sustained.

Jointly designing hardware and software in service of a higher level goal suggests a way to continue improving performance. Year on year, the greatest performance improvements could

now come from human creativity in specializing hardware to specific goals and then appropriately exposing this specialized hardware to software. This is already visible in domains such as machine learning. For instance, the tensor processing unit [135] is a chip tailored to deep learning, and TensorFlow [62] is its corresponding programming model.

The systems presented in this dissertation are examples of joint hardware and software design in the context of networking, where we develop both the underlying hardware (atoms, PIFOs, and hardware key-value stores) and the corresponding software (packet transactions, scheduling trees, and performance queries) for specific goals (stateful data-plane algorithms, scheduling, and statistics measurement).

1.7 Source code availability

Source code for the systems presented in this dissertation is available online at http://web.mit.edu/domino, http://web.mit.edu/pifo, and http://web.mit.edu/marple. Each URL contains links to the relevant source code for each project:

1. For Domino, the source code consists of a C++ simulator for high-speed router hardware (https://github.com/packet-transactions/banzai), C++ code for the Domino compiler (https://github.com/packet-transactions/domino-compiler), Domino code (https://github.com/packet-transactions/domino-examples) for the algorithms described in Tables 4.3 and 4.6, and Verilog code for the atoms (https://github.com/packet-transactions/atomsyn).
2. For PIFO, the source code consists of a C++ reference implementation of the PIFO hardware (https://github.com/programmable-scheduling/pifo-machine), and a Verilog implementation of the PIFO hardware (https://github.com/programmable-scheduling/pifo-hardware).
3. For Marple, the source code consists of a Java compiler for Marple queries (https://github.com/performance-queries/marple), instructions to setup a testbed for Mininet experiments using Marple (https://github.com/performance-queries/testbed), and the example Marple queries used in our evaluation (https://github.com/performance-queries/marple-domino-experiments).

Chapter 2

Background and Related Work

As background for the rest of this dissertation, we first review several prior approaches to network programmability (§2.1). We then review work that is closely related to and concurrent with the work presented in this dissertation (§2.2). This chapter focuses on literature that is generally related to the broad themes of this dissertation. Chapters 4, 5, and 6 discuss literature that is more specifically related to each of our three systems.

2.1 A history of programmable networks

2.1.1 Minicomputer-based routers (1969 to the mid 1990s)

The first router on a packet-switched network was likely the Interface Message Processor (IMP) on the ARPANET in 1969 [126]. The IMP described in the original IMP paper [126] was implemented on the Honeywell DDP-516 minicomputer. In today's terminology, such a router would be called a software router because it was implemented as software on top of a general-purpose computer.

This approach of implementing routers on top of minicomputers was sufficient for the modest forwarding rates required at the time. For instance, the IMP paper reports that the IMP's maximum throughput was around 700 kbit/s, sufficient to service several 50 kbit/s lines in both directions. Such minicomputer-based routers were also eminently programmable: changing the functionality of the router simply required upgrading the forwarding software on the minicomputer.

This approach of building production routers using minicomputers continued into the mid 1990s. A notable example of a software router during the 1970s was David Mills' Fuzzball router [160]. The most well known examples from the 1980s were Noel Chiappa's C Gateway [22], which was the basis for the MIT startup Proteon [14], and William Yeager's "Ships in the Night" multiple-protocol router [56], which was the basis for the Stanford startup Cisco Systems.

By the mid 1990s, software could no longer keep up with the demand for higher link speeds caused by the rapid adoption of the Internet and the World Wide Web. Juniper Networks' M40 router [55] was an early example of a hardware router in 1998. The M40 contained a dedicated chip to implement the router's data plane along with a control processor to implement the router's control plane. As we described in Chapter 1, since the mid 1990s, the fastest routers have predominantly

been built out of dedicated hardware because hardware specialization is the only way to sustain the yearly increases in link speeds (Figure 1-2).

2.1.2 Active Networks (mid 1990s)

The mid 1990s saw the development of active networks [206, 64], an approach that advocated that the network be programmable or "active" to allow the deployment of new services in the network infrastructure. There were at least two approaches to active networks. First, the programmable router approach [64], which allowed a network operator to program a router in a restricted manner. Second, the capsule approach [206, 205, 204], where end hosts would embed programs into packets as capsules, which would then be executed by the router.

Active networks came to be associated mostly with the capsule approach [108]. But the capsule approach raised some security concerns. Because programs were embedded into packets by end users, it was possible that a malicious or erroneous end user program could corrupt the entire router. One way to resolve security concerns was to execute the capsule program within an isolated application-level virtual machine like the Java virtual machine [206, 205, 204]. However, isolation using a virtual machine came at the cost of degraded forwarding performance.

Even with techniques to provide efficient isolation, such as SNAP [163], there was a significant performance hit when carrying out packet forwarding on a general-purpose processor. For instance, SNAP reported a forwarding rate of 100 Mbit/s in 2001, about two orders of magnitude slower than the Juniper M40 40 Gbit/s hardware router developed in 1998 [55].

The capsule approach—probably the most ambitious of all active networking visions—has not panned out in its most general form due to security concerns. However, recent systems [23] have exposed a far more restricted subset of a router's features to end hosts (*e.g.,* the ability for an end host to read router state, but not write to it), reminiscent of the capsule approach. On the other hand, the programmable router approach *has* seen adoption in various forms: software-defined networking and programmable router chips both provide network operators with different kinds of restricted router programmability.

2.1.3 Software routers (1999 to present)

One approach to programmability since the late 1990s has been to use a general-purpose substrate for writing packet processing programs, in contrast to fixed-function router hardware that can not be programmed. The general-purpose substrate has varied over the years. For instance, Click [144] used a single-core CPU in 2000. In the early 2000s, Intel introduced a line of processors tailored towards networking called network processors, such as the IXP1200 [27] in 2000 and the IXP2800 [26] in 2002. The RouteBricks project used a multi-core processor [103] in 2009, the PacketShader project used a GPU [123] in 2010, and the NetFPGA-SUME project used an FPGA in 2014 [216].

Software routers have found adoption as a means of programming routers at the expense of performance. They have been especially beneficial in scenarios where the link speeds are lower, but the computational requirements are higher. For instance, this approach has been used to implement MAC layer algorithms in WiFi [83, 82, 141] and signal processing algorithms in the wireless physical layer [197, 81].

Many domain-specific languages (DSLs) for packet processing were developed in parallel with the development of software routers. For instance, Click [144] uses C++ for packet processing on software routers. packetC [104] and Microengine C [24] target network processors. When designing the Domino DSL (Chapter 4), our syntax and semantics were inspired by these DSLs. However, because Domino targets high-speed routers, it is more constrained. For instance, because compiled programs run at the router's line rate, Domino forbids loops.

2.1.4 Software-defined networking (2004 to now)

Starting in the early 2000s, researchers argued for separating the router's control plane (the part of the router that runs a distributed routing protocol to computes its routing tables) from the router's data plane (the part of the router that actually forwards packets by looking them up in the routing table) [87, 107, 208, 119, 88]. As an example, an implementation of a link-state routing protocol would be part of the control plane, while an implementation of a longest-prefix-based table lookup would be part of the data plane.

The idea behind this approach, which later came to be called software-defined networking (SDN) [59], was that much of the flexibility desired by network operators when managing large-scale networks (*e.g.,* traffic engineering, access control, creating virtual networks) had to do with the control plane, not the data plane. Furthermore, the control plane executed relatively infrequently compared with the data plane: once every few milliseconds, as opposed to once every few nanoseconds. Hence, while the data plane had to be implemented in hardware for performance, the relatively infrequent nature of the control plane's operations allowed it to be moved out of the router and on to a commodity general-purpose processor, where it could be much more easily programmed.

SDN also introduced the idea of centralized control: the idea that by moving router control planes off the router and on to a general-purpose processor, the entire control plane for the network could be centralized at a few servers. This, in turn, allowed those few servers to compute routes for the entire network with the benefit of global network visibility. Effectively, SDN replaced an error-prone distributed route computation protocol with a much simpler centralized graph computation (*e.g.,* shortest paths using Dijkstra's algorithm).

SDN also required a mechanism for the control plane to populate the contents of the routing tables once the control plane had computed routes for each router. These tables would then be consulted by the data plane when forwarding packets. The most well-known of these mechanisms was the OpenFlow API [158], which exposed a minimal interface to the routing tables in router hardware. The goal of OpenFlow was to serve as a minimum common denominator across interfaces to routing tables in different router chips. This was so that existing chips could immediately support the OpenFlow API.

While the OpenFlow API made it possible to program the network's control plane, it did not necessarily make it easy. Hence, the development of SDN also led to the development of high-level programming languages to program a router's control plane [112, 162]. While SDN saw a considerable amount of research work in programming and verifying rich control plane policies, it saw much less work in enabling programmability in the data plane.

2.1.5 Network functions virtualization (2012 to now)

Network functions virtualization (NFV) [33] sought to move richer packet processing functionality, beyond vanilla packet forwarding, onto commodity general-purpose processors and the cloud infrastructure [185]. This packet processing functionality included deep-packet inspection, load balancing, intrusion detection, and WAN acceleration, colloquially unified under the umbrella term "middlebox." Several systems emerged to program both the data [174] and control [113] plane of such middleboxes.

One common use case for such middleboxes is at the edge of a network (*e.g.,* at a cellular base station), where a variety of packet-processing functions run on a cluster of processors [171] every time a client accesses the Internet. Because NFV carries out its packet processing on a software platform, it can also be viewed as a practical use case for software routers at the edge where the link rate requirements are relatively modest.

2.1.6 Edge/end-host based software-defined networking (2013 to now)

It soon became clear that the OpenFlow API was not expressive enough to capture all the needs of network operators because it was designed as a common minimum denominator API for easy adoption. The lack of expressiveness in OpenFlow led to the edge-based approach to software-defined networking [89, 145, 177].

In this approach, the network's routers were divided into two classes. The *edge routers* were located at the entrance or edge of a network and performed programmable packet manipulation. At the center of the network were the *core routers* that simply forwarded packets with little to no programmability. Because the edge routers were spatially distributed to serve clients in different locations, each edge router had to only handle a small slice of the overall aggregate traffic into and out of the network.

As a result, the edge routers had far less demanding performance requirements relative to the core, which allowed them to be implemented on general-purpose CPUs. Using a general-purpose CPU for programming edge routers allowed them to be far more programmable than the restricted OpenFlow API. Open Virtual Switch [177] is one well-known example of an edge router; it runs within the hypervisor on an end host and connects together multiple virtual machines on a single end host to the network. More recently, increasing performance requirements on the edge have led to implementing such virtual switches on an FPGA [110].

Taken to its logical extreme, the edge router could be the end host itself. Hence, when discussing edge-based approaches, we also include several recent proposals that use the end hosts to achieve network flexibility. For instance, Eden [75] provides a programmable data plane using commodity routers by programming end hosts alone. Tiny Packet Programs (TPP) [130] allow end hosts to embed small programs in packet headers, which are then executed by the router in a style similar to capsule-based active networks. TPPs use a restricted instruction set to alleviate the performance and security concerns of active networks. On the measurement and monitoring side, many systems monitor network performance from end hosts alone [73, 209, 114, 166, 31].

Edge-based or end-host-based solutions are necessary for application context that is not available within the network. For instance, knowledge of which application used the network, which

may be useful in monitoring, is only available at the end hosts. Similarly, many network security applications, such as filtering spam are best run on the end hosts because the information required to determine what is spam and what is not is best left on the end host for privacy reasons. Furthermore, quite a bit of programmable networking functionality can be implemented with edge programmability alone (*e.g.,* network virtualization, access control, security policies etc.).

However, the edge-based approach is inadequate for all network problems. For instance, there is a substantial improvement in performance from using network support for congestion control (*e.g.,* DCTCP [66] that uses explicit congestion notification support from the routers and XCP [139] that uses explicit information on the extent of congestion from the routers). Many other recent proposals for improving network performance rely on support from the routers within the core of the network [68, 74, 207, 195, 213, 65, 201]. There is also a considerable improvement in network visibility from direct in-network monitoring, in contrast to "triangulating" the root cause of a network problem using network measurements from different end host vantage points. In summary, the absence of a programmable network core leaves significant performance on the table and complicates network debugging.

One could argue for a hybrid network architecture that combines edge-based programmability with smarter, but fixed, core routers. Such an architecture still puts all the programmability at the edge but augments core routers with a small set of fixed features to support programmability from the edge. Examples of this approach include Universal Packet Scheduling [161] and In-band Network Telemetry [23, 143], which augment routers with a fine-grained priority queue and the ability to export queue size information into packets, but otherwise leave all the programmability to the edge.

This hybrid approach would be future proof and general *if* there existed a small set of features that could then be deemed sufficient for a fixed-function router. If such a small set of fixed features did indeed exist, it would be preferable and simpler to build a fixed-function router rather than a programmable router. But the last few decades of increasing feature creep in a router suggest that a "silver bullet" feature set is very unlikely. In such a setting, where features are constantly under churn, programmability within the network provides future proofness and peace of mind for the network operator: the ability to quickly add features to a router when required without lengthy hardware iteration cycles.

2.2 Concurrent research on network programmability

2.2.1 Programmable router chips

Besides the edge-based approach to network programmability, another response to the difficulty of repeatedly extending OpenFlow was to develop a programmable router chip. This router chip would have a minimal instruction set that would permit a network operator to express OpenFlow-like forwarding rules for *any* new protocol format. This is in contrast to OpenFlow, which only supports a fixed set of actions as part of the forwarding rules (drop, decrement TTL, forward, etc.) on a fixed set of packet headers (UDP, TCP, IP, etc.).

Recent academic work [86] and commercial router chips [3, 25, 60] embody this approach

of building routers that are both fast and programmable. The P4 programming language [85] has emerged as an industry effort towards a standard programming language for these chips. P4 compilers [134] then compile these P4 programs to programmable router architectures such as the RMT [86] and FlexPipe [25] architectures.

To the extent that we know based on publicly available documents, these chips provide flexibility on two counts: recognizing user-specific header formats and programmatically manipulating packet headers for functions such as forwarding, tunneling, and access control, *i.e.*, functions that do not modify router state.

In particular, they do not provide the programmability required to implement many of the grayed-out algorithms in Figure 1-1. These algorithms require the ability to programmatically manipulate router state on every packet, the ability to program which packet a router link must schedule next, and the ability to program what per-flow statistics a router must measure for a large number of flows.

2.2.2 Programmable traffic management

In an early position paper [191], we argued for the ability to program a router's traffic management algorithms: active queue management and packet scheduling algorithms. We presented a quantitative case for such programmability by considering a set of three router configurations and showing that none of the three configurations was unilaterally the best. This was because there were workloads and application objectives for which one configuration was better than the other, while for other workloads and application objectives, the situation was reversed. In other words, there did not seem to be a single "silver bullet" algorithm, making a case for multiple algorithms to be programmed based on a network operator's objectives.

As part of the same paper, we proposed that router designers add a small FPGA to each router, which could be programmed to support new traffic management algorithms. We estimated that this FPGA could run the CoDel [169] and RED [111] algorithms at the full line rate of a single 10 Gbit/s port—even with minimum-size packets.

The results in this dissertation realize the goals outlined in our earlier position paper at speeds approaching the fastest routers today. We do so by developing dedicated hardware that avoids the area, performance, and power penalties associated with an FPGA [147]. For instance, it is hard to clock an FPGA beyond 100 MHz, while custom hardware can run at 1 GHz or more. Using multiple FPGAs (*e.g.,* one for each port, like our position paper did [191]) adds considerable area, power, and cost to a high-speed router chip.

2.2.3 Centralized data planes

Fastpass [176] manages a datacenter network at fine time scales using a logically centralized arbiter server that regulates packet transmissions from all end hosts within the network. A logically centralized arbiter provides the global visibility needed to perform resource management for the entire network. Put differently, Fastpass adopts the idea of centralized control from SDN, but applies it to the data plane instead of the control plane.

In Fastpass, when an application on an end host calls send() on a socket, the end host sends a message to the arbiter specifying its network demands. Upon receiving this message, the arbiter specifies the times at which the end host must transmit and the network path that the end host's packets must follow during those times.

In subsequent work, Flexplane [170] uses the arbiter as an emulator to emulate existing resource management algorithms (*e.g.,* active queue management algorithms like RED to manage the router's buffer and scheduling algorithms like WFQ to schedule the router's links). Flowtune [175] applies Fastpass-style ideas to the centralized management of groups of packets called flowlets, instead of packets. This allows the centralized arbiter in Flowtune to handle larger networks than Fastpass.

Fastpass, Flexplane, and Flowtune are all applicable to networks of a modest size, until the centralized arbiter becomes a bottleneck. Microbenchmarks that stress the Flowtune arbiter alone show that an arbiter running on a 64-core server can manage a network with an aggregate capacity of 184 Tbit/s [175]. However, the largest network that a Fastpass-style approach has been benchmarked on (in the Flexplane paper [170]) is a network of 40 10-Gbit/s servers connected to a single top-of-rack router.

In summary, by leveraging a centralized data plane implemented on a commodity CPU, Fastpass provides more flexibility than a restricted interface exposed by a programmable router chip—but at a significant scalability cost.

2.2.4 Stateful packet processing

SNAP [71] programs stateful data-plane algorithms using a network transaction: an atomic block of code that treats the entire network as one router [137]. It then uses a compiler to translate network transactions into rules on each router. SNAP needs a lower level compiler to compile these router-local rules to a router's pipeline and can use Domino for this purpose. In fact, we compile several examples from SNAP to study the generality of our atoms in Table 4.6. FAST [164] provides router support and software abstractions for state machines. As §4.4 shows, atoms support more general stateful processing beyond state machines. In turn, this enables a much wider class of data-plane algorithms.

2.2.5 Universal Packet Scheduling (UPS)

UPS [161] shares our goal of flexible packet scheduling. UPS approaches this goal by seeking a single scheduling algorithm that is *universal* and can emulate any scheduling algorithm. Theoretically, UPS finds that the well-known LSTF scheduling discipline [150] is universal if packet departure times for the scheduling algorithm to be emulated are known up front. Practically, UPS shows that by appropriately initializing slacks, many different scheduling objectives can be emulated using LSTF. LSTF is programmable using PIFOs, but the set of schemes practically expressible using LSTF is limited. For example, LSTF cannot express:

1. Hierarchical scheduling algorithms such as HPFQ, because it uses only one priority queue.
2. Non-work-conserving algorithms. For such algorithms, LSTF must know the departure time of each packet up-front, which is not practical.

3. Short-term bandwidth fairness in fair queueing, because LSTF maintains no router state except one priority queue. As shown earlier in Figure 1-7a, programming a fair queueing algorithm requires us to maintain a virtual time state variable. Without this, a new flow could have arbitrary virtual start times, and be deprived of its fair share indefinitely. UPS provides a fix to this that requires estimating fair shares periodically, which is hard to do in practice in an environment where flows continuously start and stop.

The restrictions in UPS/LSTF are a result of a limited programming model. UPS assumes that routers are fixed and cannot be programmed to modify packet fields or manipulate router state. Further, it only has a single priority queue. By using atom pipelines to execute scheduling and shaping transactions, and by composing multiple PIFOs together, PIFOs express a wider class of scheduling algorithms.

2.2.6 Measurements using sketches

Several systems for network measurement use summary data structures that give up some accuracy to fit measurement information within limited router memory [153, 152, 154, 165, 212, 210]. These systems provide traffic volume statistics (*i.e.*, statistics derived from flow counts) using summary data structures like sketches that tradeoff accuracy for low memory consumption. Since counting is linear-in-state and can be scalably implemented in Marple, Marple sidesteps this tradeoff. Instead, Marple trades off memory size with cache eviction rate (§6.4). Marple also allows users to perform a broader set of aggregations (the linear-in-state class) beyond counters without losing accuracy.

2.2.7 Languages for network measurement

Prior network query languages [97, 112, 168, 121] allow users to ask questions primarily about traffic volumes, since their input data is collected using NetFlow and match-action rule counters [158]. In contrast, Marple enables expressive *performance* questions on data collected with purpose-built router hardware. Marple shares some language constructs with Gigascope [97] and Sonata [121], but supports aggregations directly in the router chip.

2.2.8 Recent router support for measurement

In-band Network Telemetry (INT) [23, 143] exposes queue lengths to end hosts by stamping it on the packet itself. Marple builds on INT and provides flexible filters and aggregations directly in routers. Marple's data aggregation in routers saves the bandwidth needed to collect INT data distributed over many end hosts. Besides, with INT alone, performance information may be lost, since packets carrying the INT data may be dropped on the way to end hosts. The Tetration chip [11] provides flow-level telemetry, exposing a fixed set of metrics including latency, window and packet size variation, and a "burst measurement" at a fixed granularity (*e.g.,* 5-tuple). In contrast, Marple allows both the metrics and granularity to be flexible using programmable aggregation functions and programmer-supplied `groupby` fields respectively.

2.3 Summary

This chapter summarized both the historic background that sets up the context for this dissertation and the related work that is concurrent with this dissertation. The main difference in approach between this dissertation and prior approaches to router programmability is the joint design of both hardware and software.

Jointly designing hardware and software provides us with more leverage than treating either as unchangeable. Past approaches either propose hardware or software improvements, while treating the other as fixed. We briefly describe 5 examples of how this additional leverage helps the systems in this dissertations relative to systems in prior work.

Universal Packet Scheduling. UPS adds a fine-grained priority queue to a *fixed* router, but otherwise treats the router hardware as unmodifiable, opting to carry out priority assignments in software at the edge. Relative to UPS, PIFOs are able to express a broader set of scheduling algorithms because we develop new programmable router hardware (atoms) to maintain and update state on the router.

Flexplane. Flexplane moves the data plane into slower commodity hardware to emulate algorithms like RED and WFQ. By modifying router hardware to support atoms, Domino allows such AQM schemes to be programmed directly without emulation on high-speed router hardware.

Hardware support for counters. Many router hardware designs optimize for the fixed software requirement of per-flow counters [184]. Marple develops an expressive programming language for measurements in software along with hardware support for this language. This allows Marple to support a wider range of queries beyond counters.

CONGA. CONGA [65] is a load-balancing algorithm that uses congestion-aware flowlet-based load balancing to handle asymmetry in the network topology. CONGA is implemented on a custom chip by developing new router hardware specifically for load balancing. In contrast, we developed a programming model in software as part of Domino in addition to the atom hardware. This allows us to reuse the same programmable atoms for many different algorithms including CONGA.

SONATA. SONATA [121] is a measurement system that sources its measurements from statistics exported by OpenFlow and NetFlow. By designing new router hardware for programmable measurement, Marple can express a wider range of measurements than what is supported by either OpenFlow or NetFlow.

Chapter 3

The Hardware Architecture of a High-Speed Router

This chapter describes the hardware architecture of a high-speed router to provide the reader with a mental model of a router chip for the rest of this dissertation. We first describe the main functions of a router. We then describe the performance requirements for a high-speed router today. Motivated by these performance requirements, we consider two strawman hardware architectures and explain their shortcomings. We then describe the predominant pipeline-based architecture of high-speed routers today. We conclude by describing how the programmable hardware primitives proposed by this dissertation fit within the hardware architecture of a high-speed router.

3.1 Overview of a router's functionality

A router's functionality can be divided into two major planes: the *control plane* and the *data plane*. The control plane is responsible for network management functionality such as running distributed routing protocols (*e.g.,* BGP, IS-IS, and OSPF) to enable end-to-end connectivity, creating access control rules, and setting up tunnels for network virtualization. The control plane writes rules into a router's *match-action tables*.[1] Each rule in one of these tables specifies that the router must carry out a specific *action* on a packet if the incoming packet's header *matches* a particular pattern. As an example, a rule could instruct the router to transmit a packet on a particular output port (action) if the packet has a certain destination IP address (match).

The data plane is responsible for reading the match-action table and carrying out the appropriate action operation on every packet. The data plane matches the relevant packet headers against the match half of a match-action table rule. If the packet matches a particular rule, the data plane carries out the action half of that rule on the packet's headers. If the packet does not match any rule, some default action is carried out.

The control plane typically runs when the network's topology or the network's policy changes. The data plane, on the other hand, needs to run on every packet. The rate of policy or topology changes is typically much less than the rate at which packets are processed at a router. Hence, the

[1] Also known as lookup tables, routing tables, or just tables.

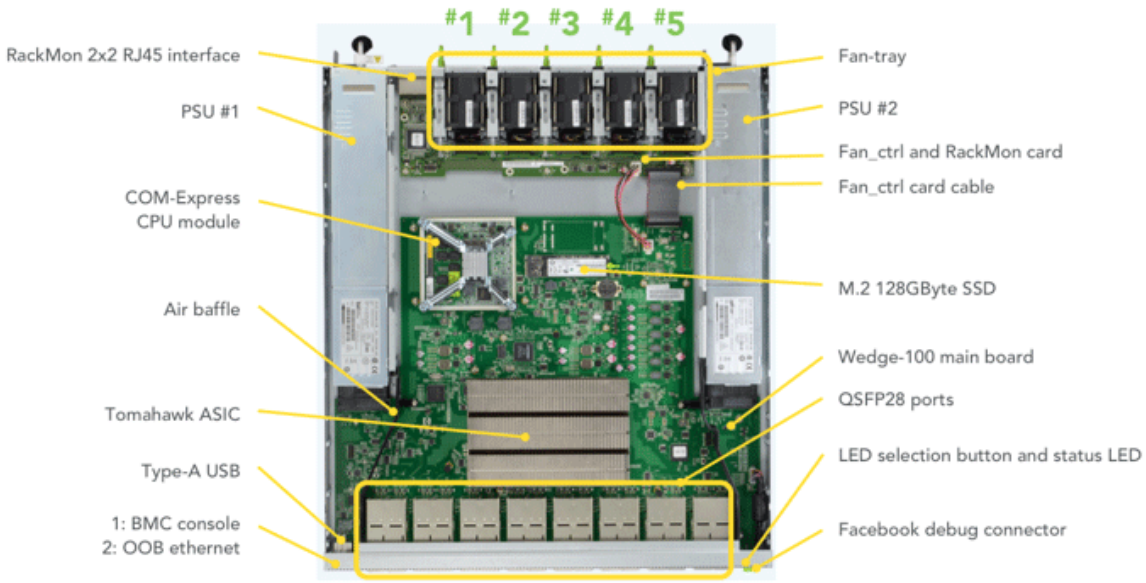


Figure 3-1: Facebook's Wedge 100 router from the Open Compute Project [20]. The COM-Express CPU module serves as the control plane, while the Tomahawk ASIC is a router chip that serves as the data plane. Image courtesy of Facebook [35].

control plane runs on a general-purpose CPU, while the data plane is implemented in dedicated hardware as part of a router chip. Figure 3-1 shows an example router.

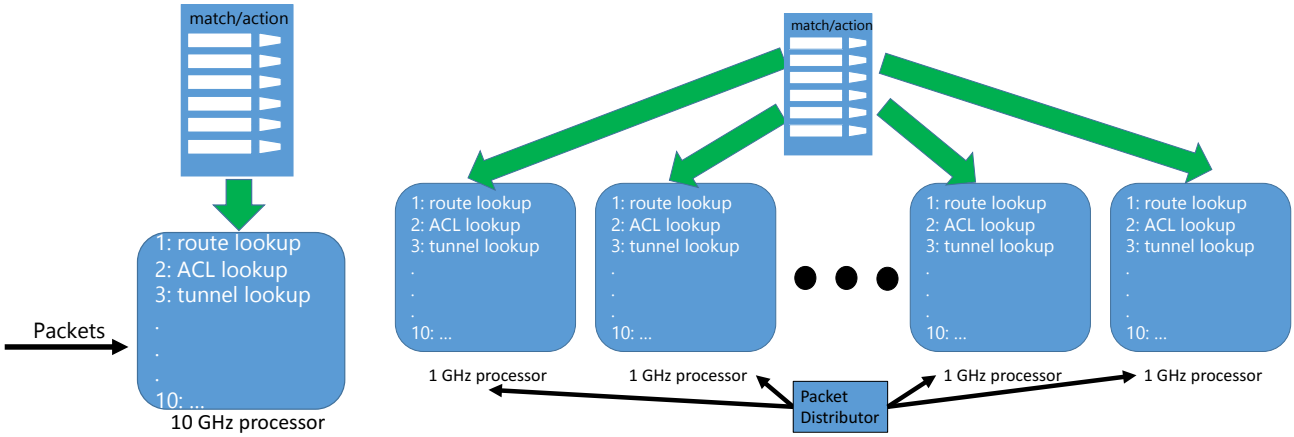
3.2 Performance requirements for a high-speed router

This dissertation focuses on programming router features that were previously implemented in fixed-function hardware, *i.e.*, the data plane of the router. Hence, we focus on the hardware architecture of the data plane here. To understand the hardware architecture of the router chip that implements the router's data plane, it is useful to have a sense of the performance requirements of a router.

For illustration, let's consider a 1 Tbit/s router, representative of many high-speed routers today [18, 19, 5]. Let's assume that the router needs to forward 1000 bit packets. Finally, let's assume that on each packet, the router needs to carry out ~10 operations, such as determining the output port based on the destination address, access control, tunneling, measurement, and decrementing the IP TTL field. These requirements imply that the router needs to support about 10 operations on a billion packets per second, or 10 billion operations per second.

3.3 Strawman 1: a single 10 GHz processor

One approach to architecting a router chip is to build a single in-order scalar processor that can run at 10 GHz and support 10 billion operations per second (Figure 3-2a). But a clock rate of 10 GHz is out of reach today. Even with painstaking manual design, the fastest general purpose processor



(a) 10 GHz processor

(b) An array of 10 1-GHz processors.

Figure 3-2: Two strawman designs for a high-speed router

chips today do not exceed 3.5 Ghz, and most other chips (*e.g.,* graphics processors and digital signal processors) have lower clock speeds in the 1 GHz range.

3.4 Strawman 2: an array of processors with shared memory

An obvious solution to this problem is to have many scalar in-order processors operate in parallel on different packets. In this architecture, when a packet comes in, a distributor sends the packet to a free in-order processor. This processor then runs that packet to completion, *i.e.*, it performs all operations for that packet before accepting a new packet. This architecture is similar to some network processors [27, 26, 9]. With such an approach, to handle 10 billion operations per second, we would need 10 1-GHz processors. Each processor would perform 1 billion operations per second, which is much more feasible (Figure 3-2b).

The problem with the processor array approach is that it needs shared memory: memory that is accessible from all processors. This memory would centrally store all match-action tables that need to be consulted during packet forwarding. This memory needs to be accessible from all processors so that any processor can match packets against any match-action table on every clock cycle.

The memory containing the match-action tables needs to support 10 billion matches (reads) per second. Memory designs typically run at the same clock frequency as the processor (1 GHz) and support a single read or write operation per clock cycle (single-ported memory). Supporting 10 billion matches per second at a 1 GHz clock requires a *multi-ported* memory that can handle multiple reads every clock cycle. Multi-ported memory consumes considerably more area than single-ported memory. Further, there is a substantial increase in wiring delay and wiring area when a single block of memory needs to be connected to many different processors.

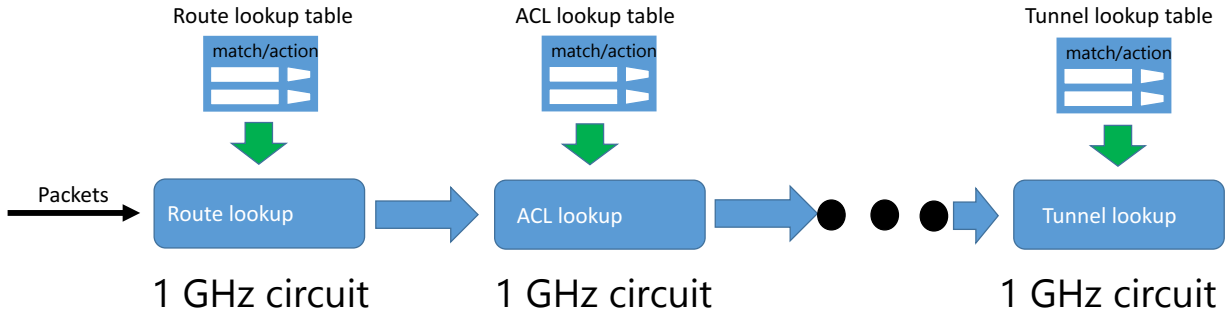


Figure 3-3: Pipeline architecture adopted by most high-speed routers.

3.5 A pipeline architecture for high-speed routers

To avoid having to use multi-ported memories, high-speed routers are typically architected as a shared-nothing *pipeline* (Figure 3-3). Each pipeline stage is dedicated to a fixed functionality, such as destination address lookup, tunneling, measurement, or access control. Each pipeline stage has its own dedicated local memory to store its own match-action tables. This architecture provides parallelism: at any point, each pipeline stage is working on a different packet. It also does so without sharing memory between processors: each match-action table is local and can be accessed only by the pipeline stage to which it belongs.

This pipeline architecture is the architecture followed by most high-speed routers today. Packets arriving at a router (Figure 3-4) are parsed by a parser that turns packets into header fields. These header fields are first processed by an ingress pipeline consisting of match-action tables arranged in stages. Processing a packet at a stage may modify its header fields, through match-action rules, as well as some persistent state at that stage, *e.g.,* packet counters. After the ingress pipeline, the packet is queued. A queue might build up any time two input ports have a packet for the same output port during the same clock cycle. Once the scheduler dequeues the packet, it is processed by a similar egress pipeline before it is transmitted.[2]

3.5.1 The internals of a single pipeline stage

Each pipeline stage implements the match-action model at high speed. Packets are received at a pipeline stage at the clock rate of the pipeline (~1 GHz). Within a pipeline stage, the match unit extracts the relevant packet headers from the packets so that the extracted headers can be matched against rules in a match-action table. For instance, a match unit might extract the TCP headers out of all the packet headers to match the TCP's source port against 22 to detect SSH traffic and, as part of the action, prioritize SSH traffic by setting the IP ToS field.

Once the relevant headers are extracted, they are matched against match-action rules stored in an on-chip match-action table. This match can either be exact (*e.g.,* determining the next hop based on the destination's MAC address) or can use wildcards to indicate don't-care bits (*e.g.,* matching

[2]In practice, the ingress and egress pipeline can be time multiplexed on top of a single shared physical pipeline to improve hardware utilization [86].

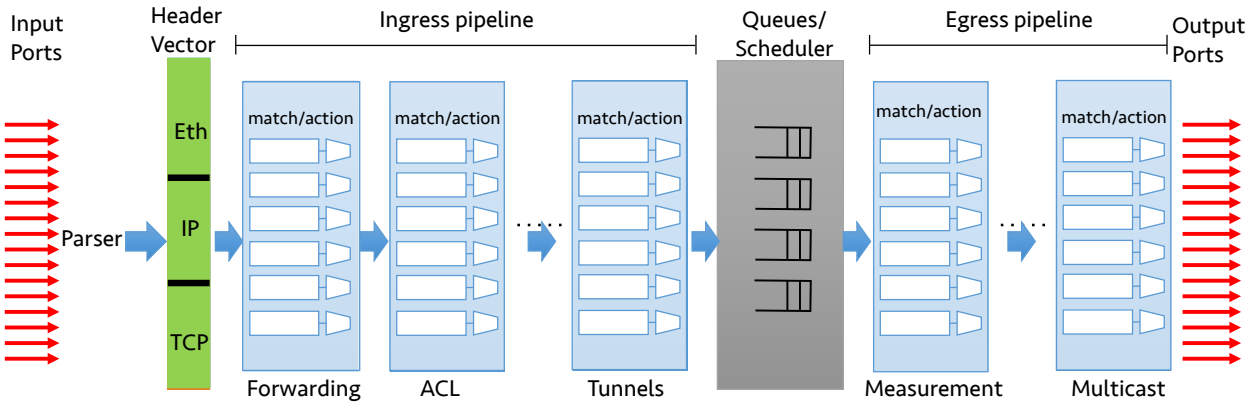


Figure 3-4: Architecture of a high-speed router showing the parser, pipelines, and scheduler.

a destination IP address against different possible IP subnets). The memory for the match-action table is structured as a hash table in SRAM for exact matches and as a ternary content-addressable memory (TCAM) for wildcard matches.

If a match is successful, the rule that matched against the headers of the incoming packet is returned. This rule contains information required for the action, such as the concrete numeric value of the output port in the case of a destination-address lookup. If the packet does not match successfully against any rule in the match-action table, some default action is carried out. Typically, the default action sends the packet to the control plane CPU of the router.

This whole processing sequence within a pipeline stage: extracting relevant headers for a match, looking them up in a match-action table, and then carrying out the appropriate action, can take 10s of clock cycles for any given packet. At the same time, the pipeline stage must be ready to accept a new packet every clock cycle. To allow this, the pipeline stage is itself internally pipelined: the header extraction for one packet proceeds in parallel with the table lookup for a second packet and the action execution for a third packet.

3.5.2 Flexible match-action processing

Initially the set of fields on which matches could be performed were fixed and limited to a small set of standard header fields (*e.g.,* TCP, UDP, and IP). Similarly, the actions allowed on the packet headers were restricted to a fixed set of actions, such as dropping a packet, forwarding a packet through an output port, or setting a priority field on a packet. The common subset of match fields and actions that were supported by most fixed-function routers was standardized as part of the OpenFlow API [158], which provided control planes with a standardized and portable way to access the data planes of a variety of high-speed routers.

Over time, however, the fixed nature of matches and actions led to a progressive increase in the number of fields supported by OpenFlow [85]. This eventually lead to the development of flexible match-action processing as embodied in recent high-speed routers [60, 25, 3]. In these routers, a programmable parser [115] allows the user to specify a new protocol format. Using the programmable parser, the router parses bytes into this new protocol format, and the ingress and

egress pipelines can match on header fields that are part of the new protocol format.

This flexible match-action processing is enabled by an input multiplexer in a match-action stage, which can extract a user-specified field from anywhere in the packet's headers for matching, without being constrained to a fixed set of fields available at fixed locations in the packet's headers. With this hardware capability, a network operator can now program a router to recognize a new packet format. She can then match on fields within this new format and compose new user-defined actions out of a small set of primitive actions that support modifications on individual fields.

However, flexible match-action processing largely ignores the question of programmable manipulation of *state* in the data plane as part of the action. Instead, the primitive actions are restricted to *stateless* manipulation of packet headers, *e.g.,* adding two packet headers and writing the result into a third. As we will see in §4.1, one of our contributions is developing a machine model for programmable routers that supports programmable state modification in the data plane of the router.

3.6 Using multiple pipelines to scale to higher speeds

Both the ingress and egress pipelines are shared across a number of router ports. They handle aggregate traffic belonging to all these ports, regardless of packet size. When the aggregate traffic requirement is small enough, a single ingress and egress pipeline suffice. For instance, a 64-port router with a line rate of 10 Gbit/s per port and a minimum packet size of 64 bytes must process up to 1 billion packets per second, after accounting for minimum inter-packet gaps [86]. This can be supported by a single pipeline that runs at 1 GHz and is shared across all 64 ports.

For higher aggregate capacities, multiple 1-GHz pipelines are required because, as discussed earlier, it is technically challenging to achieve a clock rate higher than 1 GHz in most chips. While having multiple pipelines allows the router to scale to higher aggregate speeds, it comes at the cost of fragmenting memory for lookup tables across different pipelines. For instance, packets in one pipeline cannot access lookup tables present in another pipeline. Hence, if a lookup table needs to be accessed by two different pipelines, it must be replicated, wasting some memory in the process.

Even in a multi-pipeline router, the scheduler memory that stores packet payloads is typically shared to reduce the overall memory requirement. This memory needs to handle packet enqueues from multiple ingress pipelines and packet dequeues from multiple egress pipelines. To support the ability to handle enqueues and dequeues from multiple pipelines, the scheduler's memory is multi-ported. This is possible without considerable area overhead by exploiting the fact that the scheduler's memory is structured as a set of First In First Out Queues with a limited set of operations (*i.e.*, enqueues and dequeues). The limited set of operations on the scheduler's memory allows the development of multi-ported memory for the scheduler without considerable area expense.[3]

Throughout this dissertation, we assume a single pipeline router. For concreteness, we assume the pipeline runs at 1 GHz (*i.e.*, every pipeline stage handles a new packet every 1 ns), although our concepts generalize to other clock frequencies. We discuss how our ideas can be extended to multi-pipeline routers later (§7.1.4).

[3]By contrast, our strawman design for routers based on an array of processors with shared memory (§3.4) requires a more general-purpose, and hence expensive, form of multi-ported memory.

3.7 Clarifying router terminology

Routers are classified based on several attributes in the literature. In this subsection, we briefly discuss some of the major classifications to clarify the terminology used in this dissertation.

At a conceptual level, routers are classified into output-queued and input-queued routers depending on whether the queues build up at the input ports or the output ports of a router [138]. An output-queued router is preferable because it offers more predictable performance: if an output port has a large backlog of packets it only affects that output port and no other ports.

One implementation of an output-queued router is a *shared memory* router, where the scheduler's memory for buffering packets is shared across all output ports of the router. This memory needs to support enqueues from any of the input ports and dequeues from any of the output ports every clock cycle [128, 66]. In addition to emulating an output-queued router, a shared memory implementation has the benefit that the memory pool can be allocated to output ports on demand, without committing memory to output ports at hardware design time.

With a single ingress and a single egress pipeline shared across all ports, sharing memory is relatively straightforward: each pipeline enqueues or dequeues once every clock cycle, which can be supported by single-ported memory. With multi-pipeline routers, this requires specialized multi-ported memories in the scheduler as discussed earlier.

Routers with single and multiple pipelines are instances of a *single chip* router, where a single chip provides aggregate forwarding capacity for the full line rate at all ports, while emulating an output-queued router. For even higher aggregate speeds, such as those seen in core routers, multiple such chips are put together on a motherboard in a Fat Tree topology [129], but such multi-chip routers are not necessarily output-queued anymore.

Unless otherwise mentioned, a router in this dissertation refers to a single-chip, single-pipeline, shared-memory, output-queued router.

3.8 Summary

This chapter provided an overview of the hardware architecture of a high-speed router to serve as a mental model of a router chip for the rest of this dissertation. The programmable hardware designs that we propose are intended to replace fixed-function router hardware within the hardware architecture presented here.

For instance, atoms, introduced in §1.3.1 and detailed in §4.1, replace fixed action units within a match-action stage. Our hardware design for PIFOs, introduced in §1.4.3 and detailed in §5.4, replaces the First In First Out Queues usually found in a scheduler. Finally, the on-chip cache for our programmable key-value store, introduced in §1.5.2 and detailed in §6.2, takes the place of a match-action table within a match-action stage. The off-chip backing store in our key-value store runs on the control plane: either on the general-purpose CPU controlling the router or on a centralized measurement server.

Chapter 4

Domino: Programming Stateful Data-Plane Algorithms

This chapter focuses on the hardware and software required to program *stateful data-plane algorithms* on high-speed routers. These algorithms process and transform packets, reading and writing state in the router. Examples include active queue management [111, 146, 169], congestion control with router feedback [139, 195], network measurement [210, 106], and data-plane traffic engineering [65, 187]. Hence, the central question of this chapter is: *how do we enable programmable stateful packet processing on a high-speed router?* Concretely, what is the right instruction set for programmable state modification, what is the the right programming model for stateful algorithms, and how do we compile from the programming model to the instruction set?

In answering these questions, this chapter makes three new contributions. Our first contribution is *Banzai*, a machine model for programmable high-speed routers (§4.1). Banzai is inspired by existing programmable router chips, which provide programmable, but stateless packet transformations. Banzai extends such chips with functionality required for high-speed and programmable state modifications. Specifically, Banzai models two important constraints (§4.1.3) for stateful operations at the router's line rate: the inability to share state between different packet-processing units and the requirement that any router state modifications be visible to the next packet entering the router. Based on these constraints, we introduce *atoms* to represent a programmable router's packet-processing units.

Our second contribution is a new abstraction to program data-plane algorithms called *packet transactions* (§4.2). A packet transaction is a sequential code block that is atomic and isolated from other such code blocks. Packet transactions provide programmers with the illusion that the transaction's body executes serially from start to finish on each packet in the order in which packets arrive at the router's ingress or egress pipeline. Conceptually, when programming with packet transactions, there is no overlap in packet processing across packets—akin to an infinitely fast single-threaded processor carrying out packet processing on each packet. The code within a packet transaction is written in *Domino*, a new imperative domain-specific language (DSL).

Packet transactions let the programmer focus on the operations needed for each packet without worrying about other concurrent packets. Packet transactions have an *all-or-nothing* guarantee: all packet transactions accepted by the packet transactions compiler will run at the router's line rate, or

be rejected. There is no "slippery slope" of running network algorithms at lower speeds as with network processors or software routers: when compiled, a packet transaction runs at the router's line rate, or not at all. Performance is not just predictable, but guaranteed.

Our third contribution is a compiler from Domino packet transactions to a Banzai target (§4.3). The Domino compiler extracts *codelets* from transactions: code fragments, which if executed atomically, guarantee a packet transaction's semantics. It then uses program synthesis [193] to map codelets to atoms, rejecting the transaction if the atom cannot execute the codelet.

We evaluate Domino's expressiveness by programming a variety of data-plane algorithms (Table 4.3) in Domino. We find that Domino provides a concise and natural programming model for stateful data-plane algorithms.

Next, we design a small set of atoms that can express these algorithms (§4.4.2). We show that compiler targets with these atoms are feasible at a 1 GHz clock rate in a 32-nm standard-cell library with $< 2\%$ cost in area relative to a 200 mm^2 baseline router chip [115].

We compile data-plane algorithms written in Domino to these targets (§4.4.3) to show how a target's atoms determine the algorithms it can support (Table 4.5). To study the generality of our atoms, we froze the design of our atoms and compiled a new set of Domino algorithms to our atoms. As Table 4.6 shows, our atoms were able to generalize to new and unanticipated use cases, providing empirical evidence of their programmability. Based on our experiences with Domino, we distill several lessons for programmable router design (§4.4.5).

Code for the Domino compiler, the Banzai machine model, and the code examples listed in Tables 4.3 and 4.6 is available at http://web.mit.edu/domino.

4.1 A machine model for programmable state manipulation on high-speed routers

Banzai is a machine model for programmable high-speed routers that serves as the compiler target for Domino. Banzai is inspired by recent programmable router architectures such as Barefoot Networks' Tofino [3], Intel's FlexPipe [25], and Cavium's XPliant Packet Architecture [60]. Banzai abstracts these architectures and extends them with stateful processing units to implement data-plane algorithms. These processing units, called *atoms*, model atomic operations that are supported by a programmable high-speed router. In other words, atoms serve as the instruction set of the router.

Recall from Chapter 3 that high-speed routers are architected in hardware as a pipeline where each pipeline stage processes a new packet every clock cycle of 1 ns. Having to process a packet every clock cycle in each stage constrains the operations that can be performed on each packet. In particular, any packet operation that modifies state visible to the next packet is constrained to finish execution in a single clock cycle (§4.1.2 shows why). Any programmable router chip will provide a small set of processing units or primitives that respect these constraints. These primitives could be used for manipulating packets and/or state in a stage and determine which algorithms can or cannot run on the router. We now describe this concept of constrained computation in greater detail.

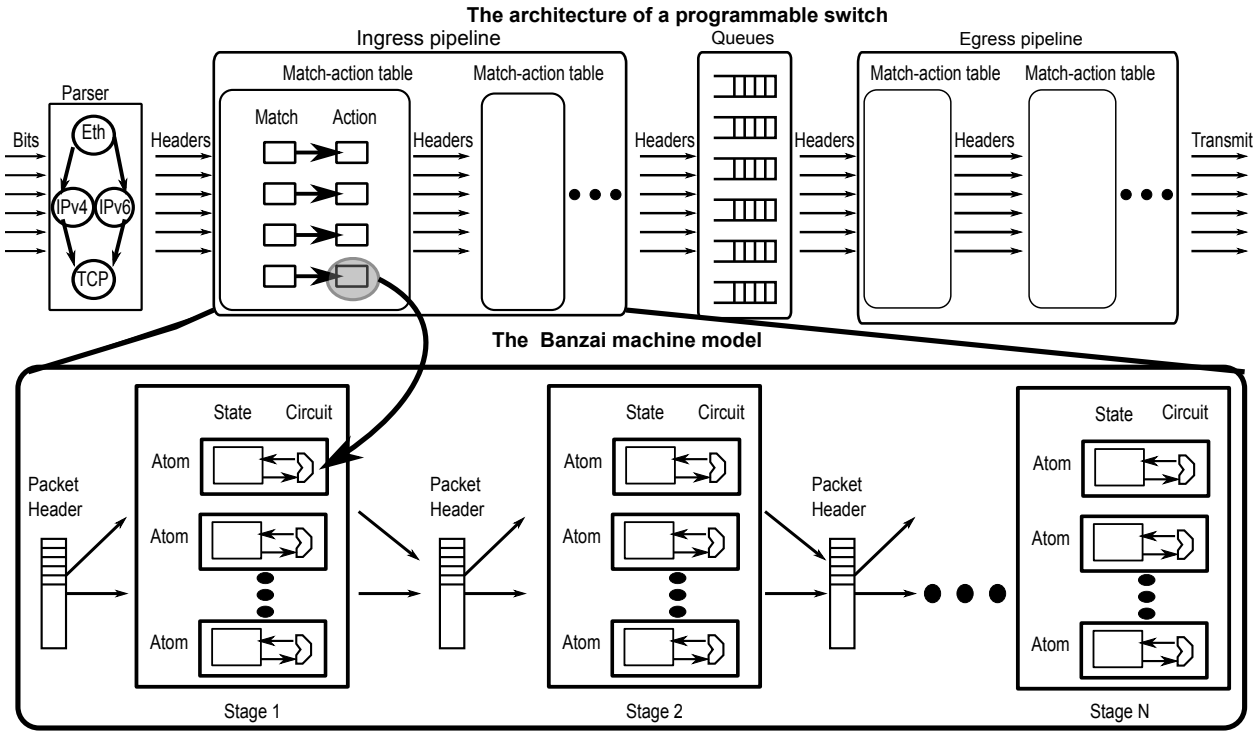
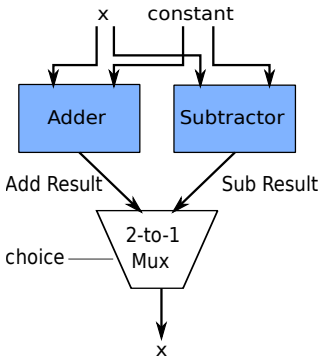


Figure 4-1: Banzai models the ingress or egress pipeline of a programmable router. An atom corresponds to an action in a match-action table. Internally, an atom contains local state and a digital circuit modifying this state. Figure 4-2 details an atom.



(a) Circuit for the atom

```
bit choice = ??;
int constant = ??;
if (choice) {
  x = x + constant;
} else {
  x = x - constant;
}
```

(b) Atom template.

Figure 4-2: An atom and its template. The atom above can add or subtract a constant from a state variable `x` based on two configurable parameters, `constant` and `choice`. Each "??" in the template represents a configuration parameter that must be instantiated so that the atom implements some computation. For instance, setting choice to 1 and constant to 1 allows us to implement x=x+1 using this atom.

4.1.1 The Banzai machine model

A programmable router that provides programmability in parsing and match-action processing [86, 60, 25, 3] has the architecture shown in the top half of Figure 4-1: a programmable parser, followed by an ingress pipeline, followed by a scheduler, followed by an egress pipeline. Banzai (the bottom half of Figure 4-1) models the ingress or egress router pipeline. It models the computation within a match-action table in a stage (*i.e.*, the action half of the match-action table), but not how packets are matched (*e.g.,* exact match or wildcard match). Banzai does not model packet parsing and assumes that packets arriving to Banzai are already parsed.

Concretely, Banzai is a feed-forward pipeline because it is hard to physically route backward-flowing wires that would be required for feedback. Banzai consists of a number of stages executing synchronously on every clock cycle. Each stage processes one packet every clock cycle and hands off one packet to the next stage every clock cycle. Note that while the throughput of a stage is one packet per clock cycle, the latency of a packet through a stage can be multiple clock cycles (to extract headers for a match, perform the match, and then perform the action). This is handled by internally pipelining each Banzai stage (§3.5.1).

Unlike a CPU pipeline, which occasionally experiences pipeline stalls, Banzai's pipeline is deterministic, never stalls, and always sustains the router's line rate even at the smallest packet size—and hence the highest packet processing rate. However, relative to a CPU pipeline, Banzai is restricted in the operations it supports (§4.1.3).

4.1.2 Atoms: Banzai's processing units

An *atom* is an atomic unit of packet processing supported natively by a Banzai machine, and the atoms within a Banzai machine form its instruction set. Each pipeline stage in Banzai contains a vector of atoms. Atoms in this vector modify mutually exclusive sections of the same packet header in parallel in every clock cycle, and process a new packet header every clock cycle.

In addition to packet headers, atoms may modify persistent state on the router to implement stateful data-plane algorithms. To support such algorithms at a router's line rate, the atoms for a Banzai machine need to be substantially richer (Table 4.4) than the simple RISC-like stateless instruction sets for programmable routers [86]. We explain why below.

Suppose we need to atomically increment a router counter to count packets. One approach is hardware support for three simple RISC-like single-cycle operations: (1) *read* the counter from memory in the first clock cycle, (2) *add* 1 in the next, and (3) *write* it to memory in the third. This approach, however, does not provide atomicity. To see why, suppose packet A increments the counter from 0 to 1 by executing its read, add, and write at clock cycles 1, 2, and 3 respectively. If packet B issues its read at time 2, it will increment the counter again from 0 to 1, when it should be incremented to 2.

Locks over the shared counter are a potential solution. However, locking causes packet B to wait during packet A's increment, and the router no longer provides a deterministic throughput of one packet every clock cycle. CPUs employ micro-architectural techniques such as operand forwarding for this problem. But these techniques still suffer pipeline stalls, which prevents line-rate performance from being achieved under worst-case conditions. Instead, Banzai's approach

to resolving this problem is to provide a small set of atomic operations that complete within a clock cycle. Programs that require multi-cycle atomic operations are rejected because line-rate performance cannot be guaranteed for them under worst-case conditions.

For instance, in the case of the counter, Banzai provides an atomic increment operation with an *atom* to read a counter, increment it, and write it back in a single stage within one clock cycle. It uses the same approach of reading, modifying, and writing back to implement a limited set of other stateful atomic operations that complete within a clock cycle. Figure 4-2a shows an example atom. Table 4.4 provides a representative set of atoms.

Unlike stateful atomic operations, stateless atomic operations are easier to support with simple RISC-like instructions for binary operations on a pair of packet fields. Consider, for instance, the operation `pkt.f1 = pkt.f2 + pkt.f3 - pkt.f4`. This operation does not modify any persistent router state and only accesses packet fields. It can be implemented atomically by using 2 atoms in a 2-stage pipeline. In the first stage, an atom adds fields f2 and f3 and writes the result to a temporary. In the second stage, another atom subtracts f4 from the temporary. Generalizing from this example, an instruction set designer can provide *simple* stateless instructions operating on a pair of packet fields. These instructions can then be composed into larger stateless operations, without designing atoms specifically for each stateless operation.

Representing atoms. An atom is represented by an *atom template* (Figure 4-2b): a body of sequential code that captures the atom's behavior. When executing, the atom template can make use of (1) a small number of packet fields, (2) internal state local to the atom, and (3) configuration parameters that provide the atom with its programmability. Atom templates allow us to create and experiment with Banzai machines with different atoms.

When implemented as a digital circuit, an atom completes execution of the entire atom template within a clock cycle, modifying a packet and any internal state before processing the next packet. The designer of a programmable router would develop these atoms, and expose them to a router compiler as the programmable router's instruction set, *e.g.,* Table 4.4.

Using this representation, a router counter that wraps around at a configurable value C can be written as the atom:[1]

```
int C = ??
if (counter < C)
  counter = counter + p.x;
else
  counter = 0;
```

Similarly, a stateless operation like setting a packet field to a configurable constant value C can be written as the atom:

```
int C = ??
pkt.field = C;
```

[1]We use `p.x` to represent field `x` within a packet `p` and a bare `x` to represent a state variable `x` that persists across packets. The ?? notation is used to denote a configuration parameter.

4.1.3 Constraints for line-rate operation

Memory limitations. State in Banzai is local to each atom. It can neither be shared by atoms within a stage, nor by atoms across stages. This is because building multi-ported memories (to store the state) accessible to multiple atoms is technically challenging and consumes additional chip area. However, state can be read into a packet header in one stage, for subsequent use by a downstream stage. Figure 4-3b shows an example: `last_time` is read into `pkt.last_time` in stage 2, for subsequent use by stage 3. But, the Banzai pipeline is feed-forward, so state can only be carried forward, not backward.

Computational limitations. Atoms need to execute atomically from one packet to the next, so any state internal to the atom must be updated before the next packet arrives at that atom. Because packets may be separated by as little as one clock cycle, we mandate that atom templates finish execution within one clock cycle, and constrain them at hardware design time to do so.

Termination within a clock cycle is enforced at hardware design time by ensuring that the atom's digital circuit meets timing at a given clock rate (§4.4.2). As programmable routers and transistor technology evolve, we expect that atoms will evolve as well, but constrained by the clock-cycle requirement.

Resource limits. We also limit the number of atoms in each stage (*pipeline width*) and the number of stages in the pipeline (*pipeline depth*). This is similar to limits on the number of stages, tables per stage, and memory per stage in programmable router architectures [134, 86].

4.1.4 Limitations of Banzai

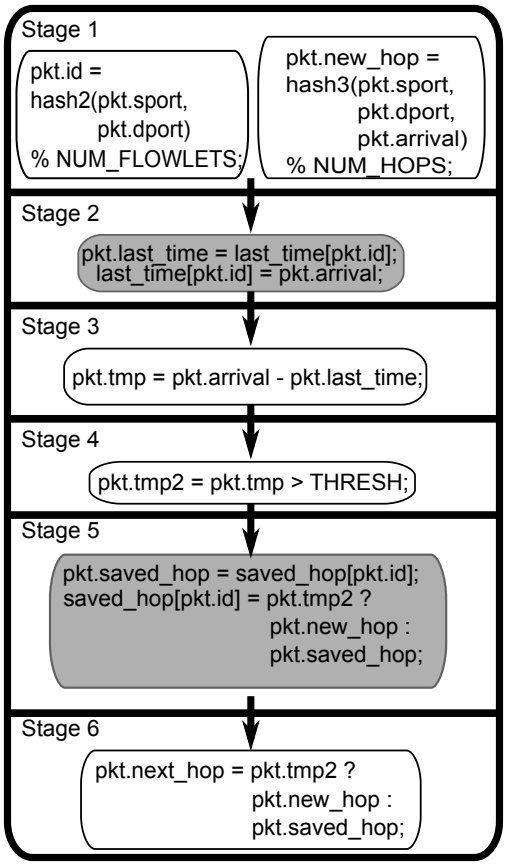
Banzai is a good fit for data-plane algorithms that modify a small set of packet headers and carry out small amounts of computation per packet. Data-plane algorithms like deep packet inspection and WAN optimization require a router to parse and process the packet payload as well, effectively parsing a large "header" consisting of each byte in the payload. This is challenging to implement on a router that processes packets at 1 GHz, and such algorithms are best left to general-purpose, but slower, CPUs [171]. Some algorithms require complex computations, but not on every packet, *e.g.,* a measurement algorithm that periodically scans a large table to perform garbage collection. Banzai's atoms model small operations that occur on every packet, and are unsuitable for such complex operations that span many clock cycles.

4.2 Packet transactions

A programmer programs a data-plane algorithm by writing it as a packet transaction in Domino (Figure 4-3a). The Domino compiler then compiles this transaction to an atom pipeline for a Banzai machine (Figure 4-3b). We first describe packet transactions in greater detail by walking through the example shown in Figure 4-3a (§4.2.1). Next, we discuss language constraints in Domino (§4.2.2) informed by the domain of high-speed routers. We then discuss how the execution of packet transactions is triggered (§4.2.3) and how we handle multiple transactions (§4.2.4).

```
#define NUM_FLOWLETS    8000
#define THRESH          5
#define NUM_HOPS        10

struct Packet {
  int sport;
  int dport;
  int new_hop;
  int arrival;
  int next_hop;
  int id; // array index
};

int last_time [NUM_FLOWLETS] = {0};
int saved_hop [NUM_FLOWLETS] = {0};

void flowlet(struct Packet pkt) {
  pkt.new_hop = hash3(pkt.sport,
                      pkt.dport,
                      pkt.arrival)
                % NUM_HOPS;

  pkt.id  = hash2(pkt.sport,
                  pkt.dport)
            % NUM_FLOWLETS;

  if (pkt.arrival - last_time[pkt.id]
      > THRESH)
  { saved_hop[pkt.id] = pkt.new_hop; }

  last_time[pkt.id] = pkt.arrival;
  pkt.next_hop = saved_hop[pkt.id];
}
```

(a) Flowlet switching written in Domino



(b) 6-stage Banzai pipeline for flowlet switching. Control flows from top to bottom. Stateful atoms are in grey.

Figure 4-3: Programming flowlet switching in Domino

4.2.1 Domino by example

We use flowlet switching [187] as our running example. Flowlet switching is a load-balancing algorithm that sends bursts of packets, called flowlets, from a TCP flow on a randomly chosen next hop. These bursts are chosen such that two consecutive bursts are separated by a large time interval of quiescence between them. This ensures that packets do not arrive out of order at a TCP receiver—even if packets in different bursts are reordered because they experience different delays along different load-balanced paths. For ease of exposition, we use only the source and destination

ports in the hash function that randomly computes the next hop for flowlet switching.

Figure 4-3a shows flowlet switching in Domino and demonstrates its core language constructs. All packet processing happens in the context of a packet transaction (the function `flowlet` starting at line 17). The function's argument type `Packet` declares the fields in a packet (lines 5–12)[2] that can be referenced by the function body (lines 18–32). The function body can also modify persistent router state using global variables (*e.g.,* `last_time` and `saved_hop` on lines 14 and 15, respectively). The function body may use *intrinsics* such as `hash2` on line 23 to directly access hardware accelerators on the router such as hash generators. The Domino compiler uses an intrinsic's signature to analyze read/write dependencies (§4.3.2), but otherwise considers an intrinsic a blackbox. The compiler directly passes intrinsics to the underlying router chip for execution, similar to intrinsics in GCC [52].

Packet transaction semantics. Semantically, the programmer views the router as invoking the packet transaction serially from start to finish on each packet in the order in which packets arrive, with no concurrent packet processing. Put differently, the packet transaction modifies the passed-in packet argument and runs to completion, before starting on the next packet. These semantics allow the programmer to program under the illusion that a single, extremely fast, processor is serially executing the packet processing code for all packets. The programmer does not have to worry about parallelizing the code within and across pipeline stages to run on a high-speed router pipeline.

4.2.2 The Domino language

Domino's syntax (Figure 4-4) is similar to an imperative language, but with several constraints (Table 4.1). These constraints are required so that the compiler can provide deterministic performance guarantees for Domino programs. Memory allocation, unbounded iteration counts, and unstructured control flow cause variable performance, which may prevent an algorithm from achieving the router's line rate. Hence, Domino forbids all three.

Additionally, within a Domino transaction, each array can only be accessed using a single packet field. Repeated accesses to the same array are allowed only if that packet field is unmodified between accesses. For example, all read and write accesses to `last_time` use the index `pkt.id`. `pkt.id` is not modified during the course of a single transaction execution (single packet); it only changes between executions (packets). This restriction on arrays mirrors restrictions on the stateful memories attached to atoms (§4.1.3), which are single ported and hence only allow a single memory address to be read/written every clock cycle.

4.2.3 Triggering the execution of packet transactions

Packet transactions specify *how* to process packet headers and state. To specify *when* to execute packet transactions, programmers use *guards*: predicates on packet fields that trigger a transaction if a packet matches the guard. For example, the guard `(pkt.tcp_dst_port == 80)` could trigger heavy-hitter detection [210] on packets with TCP destination port 80.

[2]A field is either a packet header, *e.g.,* source port (`sport`) and destination port (`dport`), or packet metadata (`id`).

No iteration (while, for, do-while).
No unstructured control flow (goto, break, continue).
No heap, dynamic memory allocation, or pointers.
At most one location in each array is accessed by a single execution of a transaction.
No access to unparsed portions of the packet (payload).

Table 4.1: Restrictions in Domino

$$
\begin{array}{rrl}
l \in \text{literals} \quad v \in \text{variables} & bop & \in \text{binary ops} \quad uop \in \text{unary ops} \\
e \in \text{expressions} & ::= & e.f \mid l \mid v \mid e \; bop \; e \mid uop \; e \mid e[d.f] \mid \\
 & & f(e_1, e_2, \ldots) \\
s \in \text{statements} & ::= & e = e \mid \texttt{if}\ (e)\ \{s\}\ \texttt{else}\ \{s\} \mid s\ ;\ s \\
t \in \text{packet txns} & ::= & name(v)\{s\} \\
d \in \text{packet decls} & ::= & \{v_1, v_2, \ldots\} \\
sv \in \text{state var inits} & ::= & v = e \mid sv\ ;\ sv \\
p \in \text{Domino programs} & ::= & \{d; sv; t\}
\end{array}
$$

Figure 4-4: Domino grammar. Types (void, struct, int, and Packet) are elided for simplicity.

Guards can be realized using the match unit in a match-action table, with the actions being the atoms compiled from a packet transaction. Guards can take various forms, *e.g.,* exact, ternary, longest-prefix, and range-based matches, depending on the matches supported by the match units in the match-action pipeline. In this chapter, we focus only on compiling packet transactions, *i.e.*, generating the actions for the match-action tables. We assume that the programmer specifies a reasonable guard that can be supported using the match unit.

4.2.4 Handling multiple transactions

So far, we have discussed a single packet transaction corresponding to a single data-plane algorithm. In practice, a router would run multiple data-plane algorithms, each processing its own subset of packets. To address this, we envision a policy language that specifies pairs of guards and transactions. Realizing a policy is straightforward when all guards are disjoint: each guard-transaction pair can be compiled independently. When guards overlap, multiple transactions need to execute on the same subset of packets, requiring a mechanism to compose transactions.

One composition semantics is to run the two transactions one after another sequentially in a user-specified order. This can be achieved by concatenating the two transaction bodies to create a larger transaction. We leave a detailed exploration of alternative composition semantics to future work. We focus only on compiling a single packet transaction here.

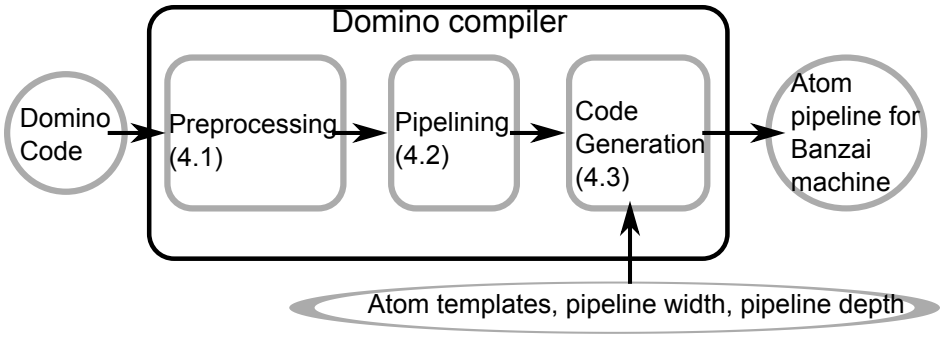


Figure 4-5: Passes in the Domino compiler

```
if (pkt.arrival - last_time[pkt.id] > THRESH) {
  saved_hop[pkt.id] = pkt.new_hop;
}
```

$\Longrightarrow$

```
pkt.tmp = pkt.arrival - last_time[pkt.id] > THRESH;
saved_hop[pkt.id] = pkt.tmp  // Rewritten
                  ? pkt.new_hop
                  : saved_hop[pkt.id];
```

Figure 4-6: Branch removal

4.3 The Domino compiler

The Domino compiler translates Domino programs to Banzai targets. The compiler provides an *all-or-nothing model*: if compilation succeeds, the program will run at the target router's line rate with packet transaction semantics. Otherwise, if the program cannot run at the router's line rate, it will not be compiled. This all-or-nothing model trades off diminished programmability for guaranteed line-rate performance. This in contrast to software routers that provide greater flexibility and compile any program, but result in lower and unpredictable performance at run-time [101].

The Domino compiler has three passes (Figure 4-5), which we illustrate using the flowlet switching example. *Preprocessing* (§4.3.1) simplifies packet transactions into a simpler form known as three-address code [57]. *Pipelining* (§4.3.2) transforms preprocessed code into code for a *Pipelined Virtual Router Machine (PVSM)*, an intermediate representation that models a router pipeline with no computational or resource limits. *Code generation* (§4.3.3) transforms this intermediate representation into configuration for a Banzai target, while respecting the target's computational and resource limits (§4.1.3). During the code generation step, the program is rejected if it either cannot be implemented by the target's atoms or there are an insufficient number of atoms. The Domino compiler uses many existing compilation techniques, but adapts them in important ways to suit the domain of high-speed routers (§4.3.4).

4.3.1 Preprocessing

Branch removal. A packet transaction's body can contain (potentially nested) branches (*e.g.,* lines 27 to 29 in Figure 4-3a). Branches alter control flow and complicate dependency analysis, *i.e.*, whether a statement should precede another. We transform branches into conditional assignments,

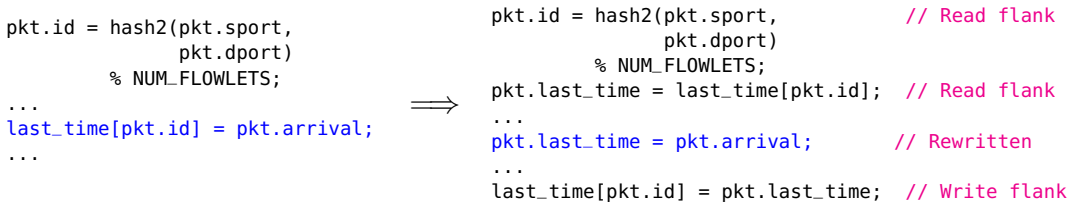


Figure 4-7: Rewriting state variable operations

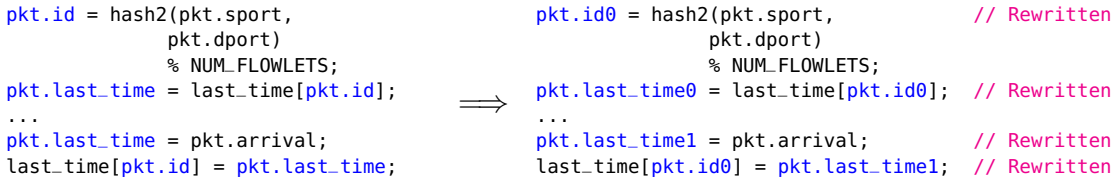


Figure 4-8: Converting to static single-assignment form

starting from the innermost `if` and proceeding outwards (Figure 4-6). This turns the transaction body into straight-line code with no branches, which simplifies dependency analysis during pipelining (§4.3.2).

Rewriting state variable operations. We now identify state variables in a packet transaction, *e.g.*, `last_time` and `saved_hop` in Figure 4-3a. For each state variable, we create a *read flank* to read the variable into a temporary packet field. For an array, we also move the index expression into the read flank using the fact that only one array index is accessed per packet (§4.2.2).

Within the packet transaction, we replace the state variable with the temporary packet field, and create a *write flank* to write this temporary packet field back to the state variable (Figure 4-7). After this step, the only operations on state variables are reads and writes that are present in the read and write flanks respectively; all arithmetic happens on packet fields. Restricting stateful operations to just reads and writes simplifies handling of state during pipelining (§4.3.2).

Converting to static single-assignment form. We next convert the code to static single-assignment form (SSA) [98], where every packet field is assigned exactly once. We do this by replacing every assignment to a packet field with a new packet field and propagating this until the next assignment to the same field (Figure 4-8) . Because fields are assigned once, SSA removes Write-After-Read and Write-After-Write dependencies. Only Read-After-Write dependencies remain during pipelining (§4.3.2).

Flattening to three-address code. Three-address code is a simplified representation of a program where all instructions are either (1) reads/writes into state variables present in the read/write flanks or (2) operations on packet fields of the form `pkt.f1 = pkt.f2 op pkt.f3`, where `op` can be an arithmetic, logical, relational, or conditional[3] operator. We also allow either one of `pkt.f2` or

[3]Conditional operations alone have four arguments.

`pkt.f3` to be an intrinsic function call. To convert to three-address code, we flatten expressions that are not in three-address code using temporary packet fields, *e.g.,* `pkt.tmp2` in Figure 4-9.

Flattening to three-address code breaks down statements in the packet transaction into a much simpler form that is closer to the atoms available in the Banzai target. For instance, there are no nested expressions. The simpler form of three-address code statements makes it easier to map them one-to-one to atoms during code generation (§4.3.3).

4.3.2 Pipelining

At this point, the preprocessed three-address code is still one sequential code block. Pipelining turns this sequential code block into a pipeline of *codelets*, where each codelet is a sequential block of three-address code statements. This codelet pipeline corresponds to an intermediate representation we call the *Pipelined Virtual Router Machine (PVSM)*. PVSM has no computational or resource limits, analogous to intermediate representations for CPUs [30] that have infinite virtual registers. Later, during code generation, we map these codelets to atoms available in a Banzai target while respecting its constraints.

We create PVSM's codelet pipeline using the steps below.

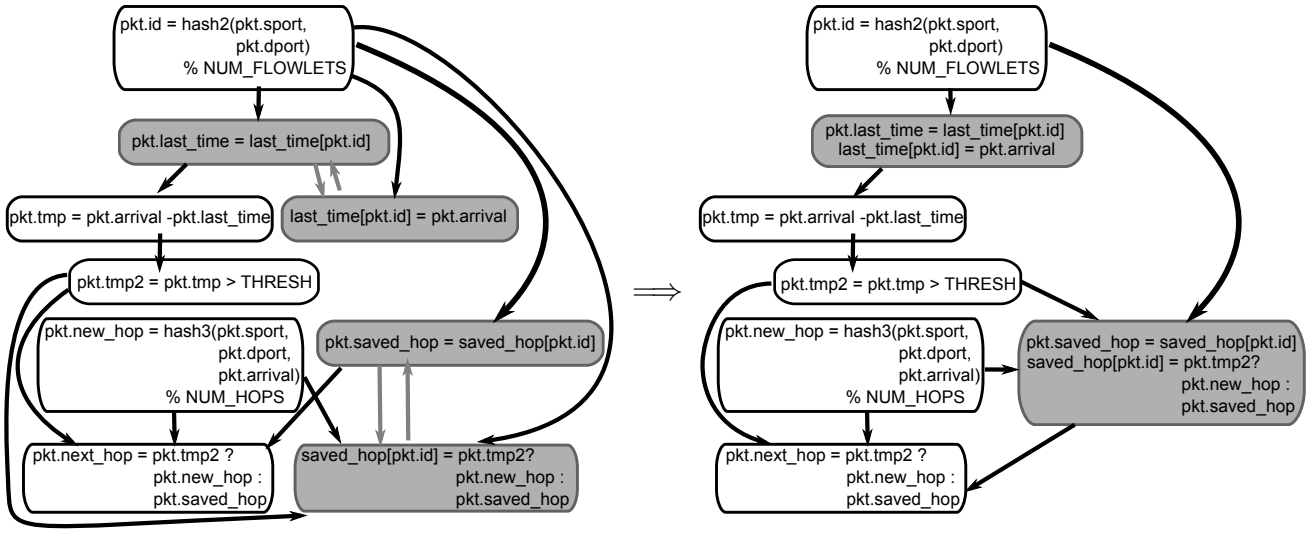
1. Create a graph with one node for each statement in the preprocessed code.
2. Now, add *stateful dependencies* by adding a pair of edges between the read and write flanks of the same state variable. For instance, in Figure 4-10a, we add a pair of edges between the pair of nodes `pkt.last_time = last_time[pkt.id]` and `last_time[pkt.id] = pkt.arrival`. Because of preprocessing, all stateful operations are paired up as read and write flanks. Hence, there is no risk of a "stranded" stateful operation.
3. Now, add *stateless dependencies* by adding an edge from any node that writes a packet variable to any node that reads the same packet variable, *e.g.,* from `pkt.tmp = pkt.arrival - pkt.last_time` to `pkt.tmp2 = pkt.tmp > THRESH` in Figure 4-10a. We only need to check for read-after-write dependencies because write-after-read and write-after-write dependencies do not exist after SSA, and we eliminate control dependencies [98] through branch removal.
4. Generate strongly connected components (SCCs) of this dependency graph and condense them into a directed acyclic graph (DAG). This captures the notion that all operations on a state variable must be confined to one codelet/atom because state cannot be shared between atoms. Figure 4-10b shows the DAG produced by condensing Figure 4-10a.
5. Schedule the resulting DAG by creating a new pipeline stage when one node depends on another. This results in the codelet pipeline shown in Figure 4-3b.[4]

The codelet pipeline implements the packet transaction on a router pipeline with no computational or resource constraints. We handle these next.

[4]We refer to this both as a codelet and an atom pipeline because codelets map one-to-one atoms (§4.3.3).

```
1 pkt.id             = hash2(pkt.sport, pkt.dport) % NUM_FLOWLETS;
2 pkt.saved_hop      = saved_hop[pkt.id];
3 pkt.last_time      = last_time[pkt.id];
4 pkt.new_hop        = hash3(pkt.sport, pkt.dport, pkt.arrival) % NUM_HOPS;
5 pkt.tmp            = pkt.arrival - pkt.last_time;
6 pkt.tmp2           = pkt.tmp > THRESH;
7 pkt.next_hop       = pkt.tmp2 ? pkt.new_hop : pkt.saved_hop;
8 saved_hop[pkt.id] = pkt.tmp2 ? pkt.new_hop : pkt.saved_hop;
9 last_time[pkt.id] = pkt.arrival;
```

Figure 4-9: Flowlet switching in three-address code. Lines 1 and 4 are flipped relative to Figure 4-3a because `pkt.id` is an array index expression and is moved into the read flank.



(a) Stateless dependencies in black, stateful in gray.

(b) DAG after condensing SCCs.

Figure 4-10: Dependency graphs before and after condensing strongly connected components

4.3.3 Code generation

To determine if a codelet pipeline can be compiled to a Banzai target, we consider two constraints specified by any Banzai target (§4.1.3). First, resource limits specify the number of atoms in a stage (pipeline width) and number of stages (pipeline depth). Second, computational limits specify the atom templates provided by a Banzai target and limit what computations can be carried out by an atom.

Resource limits. To handle resource limits, we scan each pipeline stage in the codelet pipeline starting from the first to check for pipeline width violations. If we violate the pipeline width, we insert as many new stages as required and spread codelets evenly across these stages. We continue until the number of codelets in all stages is under the pipeline width, rejecting the program if we exceed the pipeline depth.

Computational limits. Next, we determine if each codelet in the pipeline can be mapped to atoms provided by the Banzai target. In general, codelets have multiple three-address code statements that need to execute atomically. For instance, updating the state variable `saved_hop` in Figure 4-3b requires a read followed by a conditional write. It is not apparent whether such codelets can be mapped to an available atom. We develop a new technique to determine the implementability of a codelet, given an atom template.

Each atom template has a set of configuration parameters, where the parameters determine the atom's behavior. For instance, Figure 4-2a shows an atom that can perform stateful addition or subtraction, depending on the configuration parameters `choice` and `constant`. Each codelet can be viewed as a functional specification of the atom. With that in mind, the mapping problem is equivalent to searching for values of the atom's configuration parameters that result in the atom implementing the codelet.

We use the SKETCH program synthesizer [193] for this purpose, as the atom templates can be easily expressed using SKETCH. SKETCH also provides efficient search algorithms and has been used for similar purposes in other domains [91, 92]. As an illustration, assume we want to map the codelet `x=x+1` to the atom template shown in Figure 4-2b. SKETCH will search for possible parameter values so that the resulting atom is functionally identical to the codelet, for all possible input values of `x` up to a certain bound. In this case, SKETCH finds the solution with `choice=0` and `constant=1`. In contrast, if the specification is the codelet `x=x*x`, SKETCH will return an error as no parameters exist.

Using program synthesis for code generation frees the compiler developer from implementing custom code generators for different Banzai targets. Instead, the compiler developer only has to express the Banzai target's atom templates using SKETCH, and the SKETCH synthesizer automatically maps codelets to atoms.

4.3.4 Related compiler techniques

Table 4.2 shows the relationship between Domino's compilation techniques and prior work. The use of Strongly Connected Components (SCCs) is inspired by software pipelining for VLIW architectures [148]. The size of the largest SCC inversely affects the *maximum steady-state throughput* of the pipelined loop in software pipelining. For Domino, it affects the *circuit area* of the atom required to run a program at a *fixed* throughput: the target's line rate. There is a duality here: Domino trades off an increase in space (in terms of circuit area) for guaranteed performance in time (the target's line rate).

Program synthesis was used for code generation in Chlorophyll [178]. Code generation for Domino also shares similar goals to technology mapping [159] and instruction selection [63]. However, prior work maps a code sequence to *multiple* instructions/tiles, using heuristics to minimize instruction count. Domino's problem is simpler: we map each codelet to a single atom using SKETCH. The simpler problem allows a non-heuristic solution: if there is any way to map a given codelet to an atom, SKETCH will find it.

Branch removal resembles If-Conversion [70], a technique used in vectorizing compilers. This procedure is easier in Domino because there is no backward control transfer (`goto`, `break`, `continue`).

Technique	Prior Work	Differences
Conversion to straight-line code	If-Conversion [70]	No backward control flow (gotos, break, continue)
SSA	Cytron et al. [98]	SSA runs on straight-line code with no branches
Strongly Connected Components	Lam [148]	Scheduling in space vs. time
Code generation using program synthesis	Chlorophyll [178], technology mapping [159], instruction selection [63]	Optimal vs. best-effort mapping, One-to-one mapping vs. one-to-many mapping

Table 4.2: Domino's compiler in relation to prior work

Algorithm	Stateful operations	LOC	P4 LOC
Bloom filter (3 hash functions)	Test/Set membership bit on every packet.	29	104
Heavy hitters [210] (3 hash functions)	Increment count-min sketch [96] on every packet.	35	192
Flowlet switching [187]	Update saved next hop if flowlet threshold is exceeded.	37	107
RCP [195]	Accumulate RTT sum if RTT is under maximum allowable RTT.	23	75
Sampled NetFlow [45]	Sample a packet if packet count reaches N. Reset count to 0 when it reaches N.	18	70
HULL [67]	Update counter for virtual queue.	26	95
Adaptive Virtual Queue (AVQ) [146]	Update virtual queue size and virtual capacity.	36	147
Priority computation for weighted fair queueing (Chapter 5)	Compute packet's virtual start time using finish time of last packet in that flow.	29	87
DNS TTL change tracking [84]	Track number of changes in announced TTL for each domain.	27	119
CONGA [65]	Update best path's utilization/id if we see a better path. Update best path utilization alone if it changes.	32	89
CoDel [169]	Update whether we are marking or not, time for next mark, number of marks so far, and time at which minimum queueing delay will exceed target.	57	271

Table 4.3: Data-plane algorithms

4.4 Evaluation

We evaluate Domino's expressiveness by using it to program several data-plane algorithms (Table 4.3), and comparing it to writing them in P4 (§4.4.1). To validate that these algorithms can run

at the router's line rate, we design a concrete set of Banzai machines (Table 4.4) as compiler targets for Domino (§4.4.2). We estimate that these machines are feasible in hardware because their atoms incur modest chip area overhead. We use the Domino compiler to show that we can compile the algorithms in Table 4.3 to the targets in Table 4.4 (§4.4.3). We conclude with some lessons for programmable router design (§4.4.5).

4.4.1 Expressiveness

We program several data-plane algorithms (Table 4.3) using Domino. These algorithms encompass data-plane traffic engineering, in-network congestion control, active queue management, network security, and measurement. We also used Domino to express the priority computation for programming scheduling using PIFOs, as described in greater detail in Chapter 5.

In all these cases, the algorithms are already available as blocks of imperative code from online sources or published papers; translating them to Domino syntax was straightforward. In contrast, expressing any of them in P4[5] requires manually teasing out portions of the algorithm that can reside in independent match-action tables and then chaining these tables together.

Of the algorithms in Table 4.3, flowlet switching has a publicly available and manually written P4 implementation [17] that we can directly compare against. This implementation requires 231 lines of uncommented P4, compared to only 37 lines of Domino code in Figure 4-3a. Furthermore, using P4 also requires the programmer to manually specify tables, the actions within tables, and how tables are chained—all to implement a single data-plane algorithm. The Domino compiler automates this process; to demonstrate this, we developed a backend for Domino that generates the equivalent P4 code. We list the number of lines of code for these auto-generated P4 programs in Table 4.3.

4.4.2 Compiler targets

We design a set of compiler targets for Domino based on the Banzai machine model (§4.1). First, we describe how to assess the feasibility of atoms: whether they can run at a 1 GHz clock frequency, and what area overhead they incur in silicon. Next, we discuss the design of stateless and stateful atoms separately. Finally, we discuss how these stateless and stateful atoms are combined together in our compiler targets.

Atom feasibility. We synthesize a digital circuit corresponding to an atom template by writing the atom template in Verilog, and using the Synopsys Design Compiler [12] to compile the Verilog code. The Design Compiler checks if the resulting circuit meets timing at 1 GHz in a 32-nm standard-cell library and outputs its gate area. We use this gate area, along with the area of a 200 mm^2 baseline router chip [115], to estimate the area overhead for provisioning a Banzai machine with multiple instances of this atom.

Designing stateless atoms. Stateless atoms are relatively easy to design because complicated stateless operations can be broken up into multiple pipeline stages without violating atomicity (§4.1.2).

[5]We are referring to P4 at the time the Domino paper was published. As we describe in §8.1, many of Domino's ideas (including packet transactions) are part of the latest version of P4.

We design a stateless atom that can support simple arithmetic (add, subtract, left shift, right shift), logical (and, or, xor), relational (>=, <=, ==, !=), and conditional instructions (C's "?" operator) on a pair of packet fields. Any packet field can also be substituted with a constant operand. This stateless atom is similar to the stateless instruction set described by RMT [86]. When synthesized using the Synopsys Design Compiler, this atom meets timing at 1 GHz and occupies an area of 1384 μm^2 (Table 4.4).

Designing stateful atoms. As we have seen, a stateless computation can be easily pipelined over multiple stateless atoms in different pipeline stages. However, there is no easy way to pipeline a stateful computation over multiple stateful atoms and still retain atomicity. For instance, §4.1.2 illustrates how pipelining violates atomicity even for a counter.

However, designing the right set of stateful atoms for a router is a chicken-and-egg problem: the choice of stateful atoms determines which algorithms can execute on that target, while the choice of algorithms dictates what stateful atoms are required in the router. Indeed, other programmable substrates (*e.g.,* graphics processors and digital signal processors) go through an iterative process to design a good instruction set.

We use the Domino compiler to help us design a reasonable set of stateful atoms. Concretely, we pick a data-plane algorithm and execute the first two stages of the Domino compiler to generate a codelet pipeline. We then inspect the stateful codelets, and create an atom that expresses all the computations required by the stateful codelets. We verify that this atom can indeed express all these computations by executing all three stages of the compiler on the data-plane algorithm with that atom as the target. We then move on to the next algorithm, and check if the same atom suffices. If not, we repeatedly extend our atom through a trial-and-error process to capture more computations, all the while using the compiler to verify our intuitions on extending atoms.

In the process, we generate a containment hierarchy of atoms (Table 4.4), each of which works for a subset of algorithms. The complexity of the atoms increases as we proceed down this hierarchy: every atom in the hierarchy can express all the computations that its predecessors can. These atoms start out with the simplest stateful capability: the ability to read or write state alone (Read/Write). They then move on to the ability to read, add, and write back state atomically (RAW), a predicated version of the same (PRAW), a version that permits two independent read-add-writes depending on whether a predicate is true or false (IfElseRAW), and so on.

A more complex stateful atom can support more data-plane algorithms, but it occupies more area and may not meet timing at 1 GHz. When synthesized to a 32-nm standard-cell library, however, we find that all our stateful atoms comfortably meet timing at 1 GHz. However, the area and minimum input-to-output delay of the atom's circuit increases with the atom's complexity (Table 4.4).

The compiler targets. We design seven Banzai machines as compiler targets. A single Banzai machine has 600 atoms.

1. 300 are stateless atoms of the single stateless atom type from Table 4.4.
2. 300 are stateful atoms of one of the seven stateful atom types from Table 4.4 (Read/Write through Pairs).

These 300 stateless and stateful atoms are laid out as 10 stateless and stateful atoms per pipeline stage and 30 pipeline stages. While the number 300 and the pipeline layout are arbitrary, they are sufficient for all examples in Table 4.3, and incur modest area overhead as we show next. In the

Atom	Description	Area (μm^2) at 1 GHz	Minimum critical-path delay (ps)
Stateless	Arithmetic, logic, relational, and conditional operations on packet/constant operands	1384	387
Read/Write	Read/Write packet field/constant into single state variable.	250	176
ReadAddWrite (RAW)	Add packet field/constant to state variable (OR) Write packet field/constant into state variable.	431	316
Predicated ReadAddWrite (PRAW)	Execute RAW on state variable only if a predicate is true, else leave unchanged.	791	393
IfElse ReadAddWrite (IfElseRAW)	Two separate RAWs: one each for when a predicate is true or false.	985	392
Subtract (Sub)	Same as IfElseRAW, but also allow subtracting a packet field/constant.	1522	409
Nested Ifs (Nested)	Same as Sub, but with an additional level of nesting that provides 4-way predication.	3597	580
Paired updates (Pairs)	Same as Nested, but allow updates to a pair of state variables, where predicates can use the earlier values of both state variables.	5997	606

Table 4.4: Atom areas and minimum critical-path delays in a 32-nm standard-cell library. All atoms meet timing at 1 GHz. Each of the seven compiler targets contains 300 instances of one of the seven stateful atoms (Read/Write to Pairs) and 300 instances of the single stateless atom.

future, we anticipate these numbers being decided empirically based on what algorithms are most frequently run on such platforms.

We estimate the area overhead of these seven targets relative to a 200 mm^2 chip [115], which is at the lower end of chip sizes today. For this, we multiply the individual atom areas from Table 4.4 by 300 for both the stateless and stateful atoms. For 300 atoms, the area overhead is 0.2 % for the stateless atom and 0.9 % for the Pairs atom, the largest among our stateful atoms. The area overhead combining both stateless and stateful atoms for all our targets is at most 1.1%—a modest price for the programmability it provides.

4.4.3 Compiling Domino algorithms to Banzai machines

We now consider every target from Table 4.4 and every data-plane algorithm from Table 4.3 to determine if the algorithm can run at the line rate of a particular Banzai machine. Because every target is uniquely identified by its stateful atom type, we use the name of the stateful atom to refer to the target itself.

Algorithm	Most expressive atom required	Stages	Max. atoms/stage	Ingress or Egress Pipeline?	Amount of stateful memory proportional to?	Guard
Bloom filter	Read/Write	4	3	Either	(Size of bloom filter array) * (# of hash functions)	Match all packets
Heavy hitters	RAW	10	9	Either	(Size of sketch array) * (# of hash functions)	Match all packets
Flowlet switching	PRAW	6	2	Ingress	Size of flowlet table	Match all packets
RCP	PRAW	3	3	Egress	Constant: 3 scalar integers (total number of bytes, sum of RTTs, number of packets)	Match all packets
Sampled NetFlow	IfElseRAW	4	2	Either	Constant: 1 scalar integer for the counter	Match all packets
HULL	Sub	7	1	Egress	Number of output ports	Match on output port
AVQ	Nested	7	3	Ingress	Number of output ports	Match on output port
Priorities for weighted fair queueing	Nested	4	2	Ingress	Number of flows	Match on output port
DNS TTL change tracking [84]	Nested	6	3	Ingress	Number of DNS records tracked	Match all DNS packets
CONGA	Pairs	4	2	Ingress	Number of destination ToRs	Match all packets
CoDel	Doesn't map	15	3	Egress	Number of output ports or number of output queues	Match on output port or output queue

Table 4.5: Compiling Domino algorithms to Banzai machines with different atoms

We say an algorithm can run at line rate on a Banzai machine if every codelet within the data-plane algorithm can be mapped (§4.3.3) to either the stateful or stateless atoms provided by the Banzai machine. Because our stateful atoms are arranged in a containment hierarchy, we list the *most expressive* stateful atom/target required for each data-plane algorithm in Table 4.5.[6]

As Table 4.5 shows, the choice of stateful atom determines what algorithms can run on a router. For instance, with only the ability to read or write state, only the Bloom filter algorithm can run at line rate, because it only requires the ability to test and set membership bits. Adding the ability to increment state (the RAW atom) permits heavy hitter detection to run at line rate, because it

[6]Most expressive in the sense that we do not need a more expressive atom to run this algorithm.

employs a count-min sketch that is incremented on each packet. One example of an algorithm that cannot be supported by any of our atoms is CoDel, which requires a square root operation that is not provided by any of our atoms.

Table 4.5 also lists where each algorithm runs (ingress or egress), the algorithm's memory requirements for storing state, and guards (§4.2.3) for each algorithm. We explain each of these three columns below.

An algorithm may run either on the ingress pipeline (*e.g.,* load balancing algorithms like flowlet switching), egress pipeline (*e.g.,* active queue management algorithms like CoDel), or both (*e.g.,* packet sampling algorithms like sampled NetFlow). An algorithm's location depends on what information the algorithm needs to execute and what other router functionality depends on this algorithm. For instance, CoDel needs access to a packet's queueing delay, which is only available after the packet is dequeued from the scheduler and sent to the egress pipeline. On the other hand, flowlet switching is a load balancing algorithm that determines the next hop for a packet (and hence the output port). Because the output port determines which output queue the packet must be enqueued into, it needs to be determined in the ingress pipeline, before the packet enters the scheduler.

Different algorithms require different amounts of memory to store the state associated with the algorithm. For instance, the amount of stateful memory is proportional to the size of each array and the number of arrays (hash functions) in the case of heavy hitter detection and Bloom filters. On the other hand, the amount of stateful memory is proportional to the number of output ports for algorithms that run independently at each output port (*e.g.,* HULL, AVQ) or proportional to the number of output queues for algorithms that run independently at each output queue (*e.g.,* CoDel).

The guard column specifies how packets are selected for execution by an algorithm's packet transaction. For instance, the heavy hitter detection algorithm runs a single packet transaction on all packets, while the DNS TTL change tracking algorithm runs a single packet transaction on all *DNS* packets. To implement CoDel, the router needs to run a separate instance of CoDel's packet transaction for either each output port or each output queue, depending on whether the CoDel algorithm is running on a single queue for each output port or independently on multiple queues for each output port. In the case of CoDel, the guard selects packets by their output port or output queue and runs the appropriate instance of the same CoDel packet transaction on the selected packet.

4.4.4 Generality or future proofness of atoms

One concern with designing atoms using the trial-and-error method described in §4.4.2 is the potential for "overfitting:" the atoms only work for the algorithms that inspired the design of these atoms and do not generalize to new and unseen algorithms. The main argument for a programmable router is the ability to change its functionality without hardware redesign. However, if the atoms need to change with every new algorithm, this argument no longer holds.[7]

[7]Router instruction sets, and hence atoms, will evolve with time. For instance, §6.2.7 describes a new atom we added for scalable and programmable measurement. However, to justify a programmable router, the rate at which atoms need to change should be slower than the rate at which new algorithms are developed.

[8]The HULA algorithm requires some coordination between the ingress and egress pipelines to ensure the link utilization is available in the ingress. This coordination can be achieved by occasionally creating a fake packet to carry

Algorithm	Guard	Most expressive atom required
HULA probe processing logic [140][8]	Match all HULA probe packets	Pairs
HULA forwarding logic [140]	Match all data packets	PRAW
BLUE increase logic [109]	Match all packets whose enqueue queue depth exceeds a threshold	PRAW
BLUE decrease logic [109]	Match all packets where the link utilization is low	Sub
FTP monitoring [71]	Match all packets	PRAW
HashPipe (first stage) [192]	Match all packets	IfElseRAW
HashPipe (second and subsequent stages) [192]	Match only packets that make it past the first stage	Pairs
Checking for frequent domain name changes for a given IP address [71]	Match all DNS packets	PRAW
Detect first packet of a flow (Table 6.2)	Match packets belonging to a particular flow	PRAW
Counting packet reordering within a TCP connection (Table 6.2)	Match packets belonging to a particular flow	PRAW
Heavy hitter detection [71]	Match packets with the TCP SYN flag set	Pairs
Spam detection [71]	Match all packets	Pairs
Stateful firewall [71]	Match all packets	PRAW
Superspreader detection [71]	Match all packets	NestedIf

Table 4.6: The atoms in Table 4.4 generalize to new and unanticipated use cases.

To evaluate the generality of our atoms, we wrote a new set of algorithms in Domino after freezing our set of atoms to the set in Table 4.4. Using the Domino compiler, we again evaluated the most expressive atom required to run each of these algorithms at the router's line rate.

The results, shown in Table 4.6, suggest that the atoms we have designed are general enough to support several unanticipated use cases. However, for many of these algorithms, we had to expend a considerable amount of effort in manually rewriting these algorithms so that our compiler would compile them to our frozen atoms. This is because our current compiler is not intelligent enough to carry out these rewrites automatically. We illustrate this problem using an example drawn from the BLUE queue management algorithm [109].

Rewriting algorithms so that they compile. The BLUE algorithm increments a packet marking probability whenever the queue exceeds a certain size or on a packet drop. However, BLUE adds

information from one pipeline to another, but comes at the cost of the link utilization being slightly out of date at any given time. However, a slightly stale value of the link utilization doesn't severely affect the performance of the forwarding algorithm.

some hysteresis to ensure that the marking probability doesn't change too frequently. It does so by checking that a certain amount of time has elapsed since the last increment. The code for this probability increment is given below:

```
if (p.now - last_update > FREEZE_TIME) {
  p_mark = p_mark + DELTA1;
  last_update = p.now;
}
```

Unfortunately, in the above form, the BLUE algorithm cannot be compiled to the PRAW atom. The conditional check in the PRAW atom only allows a piece of state to be compared to another packet field or constant, *i.e.*, only checks of the form $A \{> / < / == /! =\} B$. The conditional check above is the of the form $A - B > C$, which is beyond the PRAW atom. However, we can manually rewrite the algorithm to precompute p.now - FREEZE_TIME in a previous stage:

```
p.tmp = p.now - FREEZE_TIME;
if (p.tmp > last_update) {
   p_mark = p_mark + DELTA1;
   last_update = p.now;
}
```

This algorithm *does* compile, because the conditional check is of the form supported by the PRAW atom. Detecting and performing such precomputation is beyond our compiler's current abilities, requiring us to assist the compiler by manually rewriting code in some cases.

Algorithms that are not supported by our atoms. When evaluating the generality of our atoms, we found two classes of algorithms that are not supported by any of our atoms. We describe these classes below.

One class of algorithms requires operations on real numbers. Our earlier example, CoDel, belongs to this class because it computes an inverse square root to decide when to next drop a packet. A second example is PIE [173], which computes an enqueue drop probability using a sequence of multiply and divide operations. A third example is Core Stateless Fair Queueing [194], which requires real-valued multiply and divide operations to estimate the fair share rate and aggregate arrival rate at a router. Other examples include Approximate Fair Dropping [172] and PERC [133]. We should note that these algorithms were not explicitly designed with hardware in mind, and it is possible that they can be modified to accommodate the constraints of high-speed hardware without losing performance. It may also be sufficient to use fixed-point or integer arithmetic instead of floating point arithmetic for some of these operations. We leave these investigations to future work.

The other class of unsupported algorithms falls outside the boundary of our atoms, as determined by the SKETCH-based code generator within the Domino compiler (§4.3.3). An example is the two-color three-rate meter [44] (trTCM), which does not compile to Pairs, our most expressive atom. We have found that explaining *why* an algorithm cannot compile to a target with a particular atom is challenging. Below, we describe our initial attempts at explaining why an algorithm does not compile.

Explaining why algorithms do not compile. So far, to explain why an algorithm does not compile, we have had to manually pare down an algorithm until we are left with a problematic "core" of the algorithm that does not compile to the underlying atom. We show an example of this procedure applied to trTCM.

trTCM marks packets with one of three colors depending on whether the packet's rate falls above a peak rate (red), between a peak and committed rate (orange), or below a committed rate (green). Its code is shown below, where tp and tc are two token buckets filling at the peak and committed rate respectively, and last_time is the last time either of the two buckets was updated.

```
if (tp < pkt.size) {
  // exceeds peak rate (PIR) (red)
  pkt.color = 1;
} else if (tc < pkt.size) {
  // exceeds commitited, but not peak (orange)
  pkt.color = 2;
  tp = tp - pkt.size;
} else {
  // within both peak and committed rates (green)
  pkt.color = 3;
  tp = tp - pkt.size;
  tc = tc - pkt.size;
}

// Refill logic
tp = tp + PIR * (pkt.time - last_time);
if (tp > PBS) tp = PBS;

tc = tc + CIR * (pkt.time - last_time);
if (tc > CBS) tc = CBS;

last_time = pkt.time;
```

Figure 4-11: Domino code for two-color three-rate meter

We repeatedly pared down the algorithm and removed last_time, tp, and pkt.color.[9] We were left with the code snippet below that still does not compile to the Pairs atom.

However, paring down just a little bit further leads us to a code snippet that *does* compile to the Pair atom:

Based on these observations, we conjecture that trTCM does not compile because it requires one more conditional check than is supported by the Pairs atom. We admit that it would be helpful to have a more precise explanation along with suggestions on how to fix trTCM so that it compiles.

[9]This paring down process is quite ad hoc at this point.

```
if (PBS < pkt.size) {
} else if (tc < pkt.size) {
} else {
  tc = tc - pkt.size;
}

tc = tc + CIR * (pkt.time);
if (tc > CBS) tc = CBS;
```

Figure 4-12: Uncompilable core of trTCM

```
if (tc < pkt.size) {
} else {
  tc = tc - pkt.size;
}

tc = tc + CIR * (pkt.time);
if (tc > CBS) tc = CBS;
```

Figure 4-13: Slightly simpler version of Figure 4-12 that *does* compile

Our method of paring down an algorithm to find a non-compilable subset of the algorithm is similar to how some SAT solvers provide an unsatisfiable core [155] when an input boolean formula is unsatisfiable. This unsatisfiable core is a subset of the clauses in the input boolean formula that is still unsatisfiable and helps a user understand why a particular input formula is unsatisfiable. Because our SKETCH-based code generator uses a SAT solver internally [193], we hope to draw on algorithms for finding unsatisfiable cores [155] to automate the paring down process in Domino.

4.4.5 Lessons for programmable routers

To conclude our evaluation, we distill some lessons for designing programmable routers.

Atoms that allow modifications to a single state variable support many algorithms. The algorithms from Bloom filters through DNS TTL Change Tracking in Table 4.5 can run at line rate using the Nested Ifs atom that modifies a single state variable.

But, some algorithms modify a pair of state variables atomically. An example is CONGA; we reproduce CONGA's relevant code snippet below:

```
  if (p.util < best_path_util[p.src]) {
    best_path_util[p.src] = p.util;
    best_path[p.src] = p.path_id;
  } else if (p.path_id == best_path[p.src]) {
    best_path_util[p.src] = p.util;
```

```
}
```

Here, `best_path` (the path id of the best path for a particular destination) is updated conditioned on `best_path_util` (the utilization of the best path to that destination)[10] and vice versa. These two state variables cannot be separated into different stages and still guarantee a packet transaction's semantics because they are mutually dependent on each other. The Pairs atom, where the update to a state variable is conditioned on a predicate of a pair of state variables, allows CONGA to run at a router's line rate.

Atom design is constrained by timing, not area. Atoms are affected by two factors: their area and their timing, *i.e.*, the delay on the critical path of the atom's combinational circuit. For the few hundred atoms that we require, atom area is insignificant (< 2%) relative to chip area. Further, even for future atoms that are larger, area may be controlled by provisioning fewer atom instances.

However, atom timing is critical. Table 4.4 shows a 3.4 × range in minimum critical-path delay (*i.e.*, the lowest achievable critical path on a particular standard-cell library) between the simplest and the most complex atoms. This increase can be explained by looking at the simplified circuit diagrams for the first three atoms (Table 4.7), which show an increase in circuit depth with atom complexity.

Because the clock frequency of a circuit is at least as small as the reciprocal of the critical-path delay, a more complex atom results in a lower clock frequency and a lower line rate. Although all our atoms have a minimum critical-path delay under 1 ns (1 GHz), it is conceivable that they can be extended with functionality that causes the atom to violate timing at 1 GHz.

In summary, for a router designer, the critical path of atoms is the most important metric to optimize. The most programmable routers will have the highest density of useful stateful functionality squeezed into a critical path budget of 1 clock cycle.

4.5 Summary

This chapter focused on the hardware and software techniques required for the problem of programming stateful algorithms on a high-speed router. Our solution to this problem resulted in three new contributions. Our first contribution was Domino, an imperative domain-specific language that allows programmers to write packet-processing code using packet transactions, which are sequential code blocks that are atomic and isolated from other such code blocks. Our second contribution was Banzai, which is a machine model based on programmable router architectures [25, 60, 3]. Banzai models the essential computational elements of a programmable router and extends them with the ability to perform programmable high-speed state manipulation. Our third contribution was the Domino compiler, which compiles packet transactions to hardware configurations for Banzai targets.

As part of our evaluation (§4.4), we showed that Domino offers a concise and natural model to program stateful algorithms (Table 4.3). We designed a set of 7 atoms (Table 4.4) and showed that they can be used to express a variety of stateful algorithms (Table 4.5). Furthermore, these atoms

[10] `p.src` is the address of the host originating this message, and hence the destination for the host receiving it and executing CONGA.

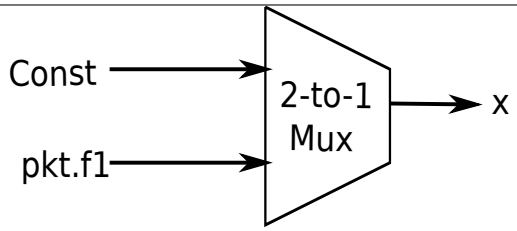
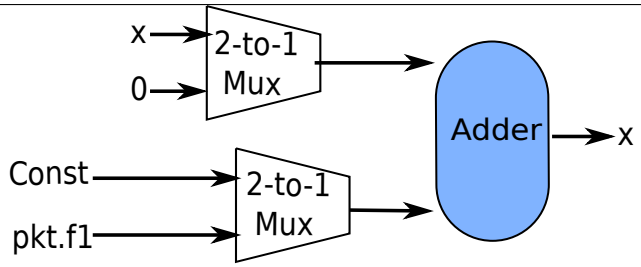
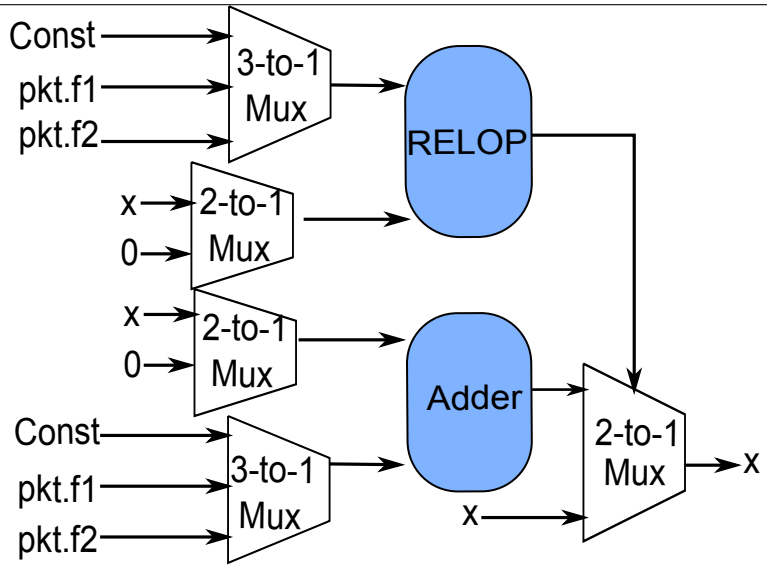
Atom	Circuit	Min. delay (ps)
Read/Write		176
ReadAddWrite (RAW)		316
Predicated ReadAddWrite (PRAW)		393

Table 4.7: An atom's minimum critical-path delay (*i.e.*, , the lowest possible critical-path delay on a particular standard cell library) increases with circuit depth. Mux is a multiplexer. RELOP is a relational operation (>, <, ==, !=). `x` is a state variable. `pkt.f1` and `pkt.f2` are packet fields. `Const` is a constant operand.

generalized to new and unanticipated use cases that were programmed after the design of these atoms was frozen (Table 4.6). These results suggest that it possible to program a wide variety of stateful algorithms at high speeds using the right combination of hardware (atoms) and programming models (packet transactions).

Chapter 5

PIFOs: Programmable Packet Scheduling

Packet scheduling is the task of picking the next packet to transmit on a router's outgoing links. The research literature is replete with scheduling algorithms [99, 180, 150, 186, 68] tailored to different network performance objectives (*e.g.,* fairness, low flow completion times, low tail latencies, etc.). However, today's fastest routers provide only a small menu of scheduling algorithms: typically, a combination of Deficit Round Robin (DRR) [186], strict priority scheduling, and traffic shaping. A network operator can change parameters (*e.g.,* weights in DRR, or rate limits in traffic shaping) in these algorithms. But, an operator cannot change the core logic in an existing algorithm, nor program a new one, without building new router hardware.

This chapter develops hardware primitives and software programming models for *programmable packet scheduling* in high-speed routers. With a *programmable* packet scheduler, network operators could customize scheduling algorithms to application requirements, *e.g.,* minimizing flow completion times [68] using Shortest Remaining Processing Time (SRPT) [183], allocating bandwidth flexibly across flows or tenants [131, 179] using Weighted Fair Queueing [99] or minimizing tail packet delays using Least Slack Time First [150]. With a programmable packet scheduler, router vendors could implement scheduling algorithms as programs running on a programmable router chip, making it easier to verify and modify these algorithms compared to baking in the same algorithms into a chip as rigid hardware.

We begin this chapter by deconstructing packet scheduling (§5.1). First, we observe that all scheduling algorithms make two basic decisions: in *what order* packets should be scheduled and *when* they should be scheduled, corresponding to work-conserving and non-work-conserving algorithms respectively. Second, we observe that for many scheduling algorithms, these two decisions can be made when a packet is enqueued. These observations suggest a natural hardware primitive for packet scheduling: a *Push In First Out Queue (PIFO)* [94]. A PIFO is a priority queue that allows elements to be pushed into an arbitrary position based on an element's *rank* (the scheduling order or time),[1] but always dequeues elements from the head.

We then develop a PIFO-based programming model for scheduling (§5.2) with two key ideas. First, we allow an operator programming the scheduler to set a packet's rank in a PIFO by supplying

[1] When the rank denotes the scheduling time, the PIFO implements a calendar queue; we distinguish between PIFOs and priority queues for this reason.

a small program for computing packet ranks (§5.2.1). Coupling this program with a single PIFO allows the operator to program any scheduling algorithm where the relative scheduling order of buffered packets does not change with future packet arrivals. Second, operators can compose PIFOs together in a tree to program several hierarchical scheduling algorithms that violate this relative ordering property (§5.2.2 and §5.2.3).

We find that a PIFO-based scheduler lets us program many scheduling algorithms (§5.3), *e.g.,* Weighted Fair Queueing [99], Token Bucket Filtering [58], Hierarchical Packet Fair Queueing [76], Least-Slack Time-First [150], the Rate-Controlled Service Disciplines [214], and fine-grained priority scheduling (*e.g.,* Shortest Job First). Until now, any high speed implementations of these algorithms—if they exist at all—have been baked into the router's hardware. We also describe the limits of the PIFO abstraction (§5.3.5) by presenting examples of scheduling algorithms that cannot be programmed using PIFOs.

To evaluate the hardware feasibility of PIFOs, we implemented a hardware design for PIFOs in Verilog (§5.4). We synthesized this design to an industry standard 16-nm standard-cell library (§5.5). The main operation in our design is sorting an array of PIFO elements at the router's line rate to determine the next packet to be dequeued. To implement this sort, traditionally considered hard [157, 186], we exploit two observations. One, most scheduling algorithms schedule across flows, with packet ranks increasingly monotonically within each flow. Hence, we only need to sort the head (earliest) packets of all flows to dequeue from a PIFO. Two, transistor scaling now makes it feasible to sort a surprisingly large number of packets (in this case, the head packets in a PIFO) at high speed.

We evaluate our hardware design (§5.5) and find that it is feasible to build a programmable scheduler, which

- supports 5-level hierarchical scheduling, where the scheduling algorithms at each level are programmable;
- runs at a clock frequency of 1 GHz—sufficient for a single-pipeline 64-port shared-memory router with a line rate of 10 Gbit/s on each port;
- uses only 4% additional chip area compared to a shared-memory router that supports only a small menu of scheduling algorithms; and
- has the same buffer size as a typical shared-memory router in a datacenter (~60K packets, ~1K flows) [18].

C++ code for a hardware reference model of our programmable scheduler and Verilog code for our hardware design are available at http://web.mit.edu/pifo/.

5.1 Deconstructing scheduling

We observe that any scheduling algorithm makes two basic decisions: in *what order* and *when* should packets leave the router, broadly corresponding to work-conserving and non-work-conserving schedulers respectively. Scheduling algorithms only differ in how the order and departure time are computed. Further, in many cases, the order or departure time can be determined when the packet is enqueued. To see why, we look at three popular packet-scheduling algorithms: pFabric [68], Weighted Fair Queueing (WFQ) [99], and traffic shaping [58]. We then extract similarities between

these three algorithms to provide intuition for a programming model for packet scheduling.

5.1.1 pFabric

pFabric [68] is a recent datacenter transport design that attempts to minimize average flow completion time by scheduling packets according to their remaining flow size at each router, *i.e.*, it implements the SRPT scheduling algorithm at each router [183].[2] For each packet, end hosts insert the remaining flow size as a packet field. At each router, packets are dequeued in increasing order of their remaining flow size.

5.1.2 Weighted Fair Queueing

Weighted Fair Queueing (WFQ) provides weighted max-min bandwidth allocation across flows sharing a link. Numerous implementations of WFQ exist, including Start-Time Fair Queueing (STFQ) [118] and Deficit Round Robin [186]. For concreteness, we consider STFQ.[3]

STFQ computes a *virtual start time* (`p.start`) for each packet using the algorithm below.

```
On enqueue of packet p of flow f:
-----------------------------------------
if f in T
  p.start = max(virtual_time, T[f].last_finish)
else
  p.start = virtual_time
T[f].last_finish = p.start + p.length / f.weight

On dequeue of packet p:
--------------------------------
virtual_time = p.start
```

Here, `last_finish` is a state variable maintained for each flow in table `T` that tracks the virtual finish time of its latest packet. `virtual_time` is a queue-wide state variable updated on each dequeue. Packets are scheduled in increasing order of their virtual start time (`p.start`).

5.1.3 Traffic shaping

Besides packet order, some non-work-conserving scheduling algorithms determine the time at which packets depart from a queue. Traffic shaping is a canonical example and is used to limit flows to a desired rate. A shaper has two parameters: a shaping rate, r, and a burst allowance, B. The standard implementation uses a *token bucket* [58], which is incremented at a rate r, subject to a cap of B tokens. A packet is transmitted immediately if the bucket has enough tokens when it is enqueued; otherwise, it has to wait until sufficient tokens accumulate. Transmitted packets decrement the token bucket by the packet size.

[2]There are two variants of pFabric [68], with and without starvation prevention. We consider the one without starvation prevention.

[3]The original WFQ implementation [99] is similar to STFQ, but uses a more complex virtual time calculation.

Alternatively, the transmission time of each packet can be calculated on enqueue as follows:

```
tokens = min(tokens + r * (now - last_arrival), B)
if p.length <= tokens
  p.send_time = now
else
  p.send_time = now + (p.length - tokens) / r
tokens = tokens - p.length
last_arrival = now
```

Here, `tokens` and `last_arrival` are two state variables, initialized to B and an initial time respectively. While a standard token bucket has only positive token counts, `tokens` can fall below zero in the algorithm above. It is easy to show that the transmission times calculated are still identical to those of the standard token bucket. With this alternative algorithm shown above, packets shaped by a token bucket are transmitted in increasing order of their transmission times (`p.send_time`).

5.1.4 Summary

The schedulers presented above (pFabric, WFQ, and traffic shaping) determine the order or time of departure of a packet using a single number computed at enqueue time. For pFabric, this number is the remaining flow size and is inserted by an end host tracking the number of remaining bytes for each flow. For WFQ, it is the virtual start time computed by the router. For token bucket shaping, it is the wall-clock departure time computed by the router. This ability to determine scheduling order or time at enqueue unifies a variety of diverse scheduling algorithms, and it is at the heart of our programming model for packet scheduling.

5.2 A programming model for packet scheduling

Our programming model builds on the observation we made in the previous section: the scheduling order or time for many schedulers can be determined at enqueue. It has two components:

1. The *Push In First Out Queue (PIFO)* [94], which maintains the scheduling order or scheduling time for enqueued elements. A PIFO is a priority queue that allows elements to be enqueued into an arbitrary position based on the element's *rank* (ranks could reflect either scheduling order or time), but dequeues elements from the head. Elements with a lower rank are dequeued first; if two elements have the same rank, the element enqueued earlier is dequeued first.
2. The computation of an element's rank before it is enqueued into a PIFO. We model this computation as a packet transaction (§4.2), an atomically executed block of code that is executed once for each element before enqueueing it in a PIFO.

We note that scheduling with the PIFO abstraction does not require packets to be stored in per-flow queues.

We now describe the three main abstractions in our programming model. First, we show how to use a *scheduling transaction* to program simple work-conserving scheduling algorithms using a single PIFO (§5.2.1). Second, we generalize to a *scheduling tree* to program hierarchical work-

```
f = flow(p) # compute flow from packet p
if f in last_finish:
 p.start = max(virtual_time, last_finish[f])
else: # p is first packet in f
 p.start = virtual_time
last_finish[f] = p.start + p.length/f.weight
p.rank = p.start
```

Figure 5-1: Scheduling transaction for STFQ. `p.x` refers to a packet field `x` in packet `p`. `y` refers to a state variable that is persisted on the router across packets, *e.g.,* `last_finish` and `virtual_time` in this snippet. `p.rank` denotes the packet's computed rank.

conserving scheduling algorithms (§5.2.2). Third, we augment nodes of this tree with a *shaping transaction* to program non-work-conserving scheduling algorithms (§5.2.3).

5.2.1 Scheduling transactions

A *scheduling transaction* is a block of code associated with a PIFO that is executed once for each packet before the packet is enqueued. The scheduling transaction computes the packet's rank, which determines its position in the PIFO. Scheduling transactions can be used to program work-conserving scheduling algorithms. In particular, a single scheduling transaction and PIFO are sufficient to specify any scheduling algorithm where the relative scheduling order of packets already present in the buffer does not change with the arrival of future packets.

WFQ, described earlier (§5.1.2), is one example. It achieves weighted max-min allocation of link capacity across flows sharing a link. We define a flow to be any set of packets sharing common values for specific packet fields. Approximations to WFQ[4] include Deficit Round Robin (DRR) [186], Stochastic Fairness Queueing (SFQ) [157], and Start-Time Fair Queueing (STFQ) [118]. We consider STFQ here, and show how to program it using the scheduling transaction in Figure 5-1.

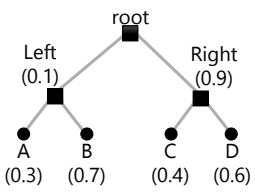
Before a packet is enqueued, STFQ computes a *virtual start time* for that packet (`p.start` in Figure 5-1) as the maximum of the *virtual finish time* of the previous packet in that packet's flow (`last_finish[f]` in Figure 5-1) and the current value of the *virtual time* (`virtual_time` in Figure 5-1), a state variable that tracks the virtual start time of the last dequeued packet across all flows.[5] Packets are scheduled in order of increasing virtual start times, which is the packet's rank in the PIFO.

5.2.2 Scheduling trees

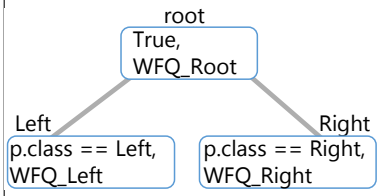
Scheduling algorithms that require changing the relative order of buffered packets when a new packet arrives cannot be programmed using a single scheduling transaction and PIFO. An important

[4] An approximation is required because the original WFQ algorithm [99] has a complex virtual time calculation.

[5] §5.5.5 discusses how this state variable can be accessed on the enqueue side.



(a) Algorithm



(b) Scheduling tree

```
1 f = flow(p) # see caption below
2 if f in last_finish:
3   p.start = max(virtual_time,
      last_finish[f])
4 else:
5   p.start = virtual_time
6 last_finish[f] = p.start + p.length
      / f.weight
7 p.rank = p.start
```

(c) Scheduling transaction for WFQ_Root, WFQ_Left, and WFQ_Right.

Figure 5-2: Programming HPFQ using PIFOs. "Left" and "Right" are classes. A, B, C, and D are flows. Within each tree node in the scheduling tree, the first line is the packet predicate and the second is the scheduling transaction. All three nodes execute the same code for the scheduling transaction except for their flow() function, which returns a packet's flow/class. For WFQ_Root, it returns the packet's class: Left/Right. For WFQ_Left and WFQ_Right, it returns the packet's flow: A/B or C/D.

class of such algorithms are *hierarchical* schedulers that compose multiple scheduling policies at different levels of the hierarchy. We introduce a *scheduling tree* for such algorithms.

To illustrate a scheduling tree, consider Hierarchical Packet Fair Queueing (HPFQ) [76], a recursive variant of fair queueing. HPFQ first divides link capacity fairly between classes, then recursively between sub classes in each class, all the way down to the leaf nodes. Figure 5-2a provides an example; the number on each child indicates its weight relative to its siblings in its parent's fair scheduler. HPFQ cannot be realized using a single scheduling transaction and PIFO because the relative scheduling order of packets that are already buffered can change with future packet arrivals. Section 2.2 of the HPFQ paper [76] provides an example illustrating how this can happen.

HPFQ *can*, however, be realized using a tree of PIFOs, with a scheduling transaction attached to each PIFO in the tree. To see how, observe that HPFQ executes WFQ at each level of the hierarchy, with each node using WFQ among its children. As discussed in §5.2.1, a single PIFO encodes the current scheduling order for WFQ, *i.e.*, the scheduling order if there are no further arrivals. Similarly, a tree of PIFOs (Figure 5-3), where each PIFO's elements are either packets or references to other PIFOs, can be used to encode the current scheduling order of HPFQ and other hierarchical scheduling algorithms. To determine this scheduling order, we inspect the root PIFO to determine the next child PIFO to schedule. Then, we recursively inspect the child PIFO to determine the next grandchild PIFO to schedule, until reaching a leaf PIFO that determines the next packet to schedule.

The current scheduling order of the PIFO tree can be modified as packets are enqueued, by executing a scheduling transaction at each node in the PIFO tree. This is our second programming abstraction: a *scheduling tree*. Each node in this tree is a tuple with two attributes. First, a packet predicate that specifies which packets execute that node's scheduling transaction before enqueueing

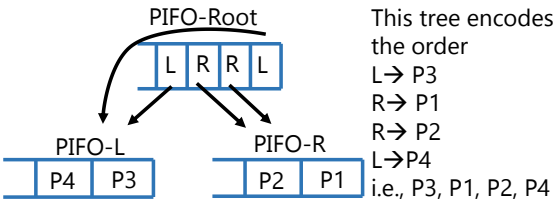
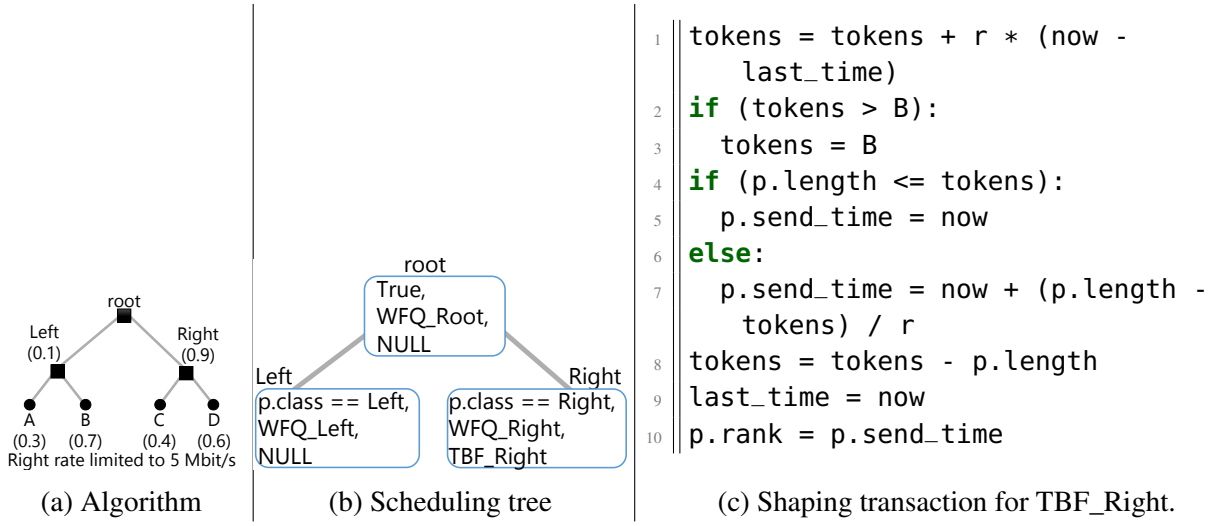


Figure 5-3: PIFO trees encode current scheduling order for hierarchical schedulers.



(a) Algorithm (b) Scheduling tree (c) Shaping transaction for TBF_Right.

Figure 5-4: Programming Hierarchies with Shaping using PIFOs. The third line within each tree node in the scheduling tree is the shaping transaction. The scheduling transactions for WFQ_Right, WFQ_Left, and WFQ_Root are identical to Figure 5-2.

an element into that node's PIFO; this element is either a packet or a reference to a child PIFO of the node. Second, a scheduling transaction that specifies how the rank is computed for elements (packet or PIFO references) that are enqueued into the node's PIFO. Figure 5-2b shows an example for HPFQ.

When a packet is enqueued into a scheduling tree, it executes one transaction at each node whose packet predicate matches the arriving packet. These nodes form a path from a leaf to the root of the tree and the transaction at each node on this path updates the scheduling order at that node. One element is enqueued into the PIFO at each node on the path from the leaf to the root. At the leaf node, that element is the packet itself; at the other nodes, it is a reference to the next PIFO on the path towards the leaf. Packets are dequeued in the order encoded by the tree of PIFOs (Figure 5-3).

5.2.3 Shaping transactions

So far, we have only considered work-conserving scheduling algorithms. *Shaping transactions* allow us to program non-work-conserving scheduling algorithms. Non-work-conserving algorithms differ from work-conserving algorithms in that they decide the *time* at which packets are scheduled

as opposed to the scheduling *order*. As an example, consider the algorithm shown in Figure 5-4a, which extends the previous HPFQ example with the requirement that the Right class be limited to 5 Mbit/s. We refer to this example as *Hierarchies with Shaping*.

To motivate our abstraction for non-work-conserving algorithms, recall that a PIFO tree encodes the current scheduling order, by walking down the tree from the root PIFO to a leaf PIFO to schedule packets. With this encoding, a PIFO reference can be scheduled only if it resides in a PIFO and there is a chain of PIFO references from the root PIFO to that PIFO reference. To program non-work-conserving scheduling algorithms, we provide the ability to *defer* when a PIFO reference is enqueued into the PIFO tree, and hence is available for scheduling.

To defer enqueues into the PIFO tree, we augment nodes of the scheduling tree with an optional third attribute: a *shaping transaction* that is executed on all packets matched by the node's packet predicate. The shaping transaction on a node determines when a reference to the node's PIFO is available for scheduling in the node's *parent's* PIFO. The shaping transaction is implemented using a separate *shaping PIFO* at the child—distinct from the scheduling PIFO at all nodes—that holds references to the child's scheduling PIFO until they are released to the parent's scheduling PIFO. The shaping transaction uses the wall-clock departure time as the rank for the shaping PIFO, unlike the scheduling transaction that uses the relative scheduling order as the rank.

Once a reference to the child's scheduling PIFO has been released to the parent's scheduling PIFO from the child's shaping PIFO, it is scheduled by executing the parent's scheduling transaction and enqueueing it in the parent's scheduling PIFO. If a node has no shaping transaction, references to that node's scheduling PIFO are immediately enqueued into its parent's scheduling PIFO with no deferral. The dequeue logic during shaping still follows Figure 5-3: dequeue recursively from the root until we schedule a packet.

Figure 5-4c shows an example of a shaping transaction that defers enqueues based on a Token Bucket Filter (TBF) with a rate-limit of r and a burst allowance B. Here, the packet's wall clock departure time (`p.send_time`), is used as its rank in the shaping PIFO. Figure 5-4b shows how to use this shaping transaction to program Hierarchies with Shaping: the TBF shaping transaction (TBF_Right) determines when PIFO references for Right's scheduling PIFO are released to Root's scheduling PIFO.

Timing of operations during shaping. We now describe the timing of operations during shaping in greater detail. When a packet is enqueued in a tree of PIFOs, it executes a scheduling transaction at the leaf node whose predicate matches this packet. It then continues upward towards the root executing scheduling transactions along the path, until it reaches the first node that also has a shaping transaction attached to it. Figure 5-5 shows the operations that occur at this node, *Child*, and its parent, *Parent*, to implement shaping.

Two transactions are executed at *Child*: the original scheduling transaction to push an element into *Child*'s scheduling PIFO (step 1a in Figure 5-5) and a shaping transaction to push an element R (step 1b), which is a reference to *Child*'s scheduling PIFO, into *Child*'s shaping PIFO. After R is pushed into *Child*'s shaping PIFO, further transactions for this packet are suspended until R's rank, the wall-clock time T, is reached.

At T, R will be dequeued from *Child*'s shaping PIFO and enqueued into *Parent*'s scheduling PIFO (step 2), making it available for scheduling at *Parent*. The rest of the packet's path to the root

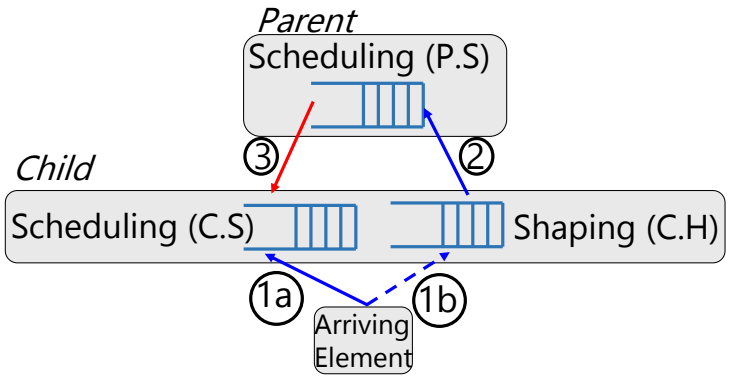


Figure 5-5: *Child's* shaping transaction (1b) *defers* enqueue into *Parent's* scheduling PIFO (2) until time T. Blue arrows show enqueue paths. Red arrows show dequeue paths.

is now resumed starting at *Parent*. This suspend-resume process can occur multiple times if there are multiple nodes with shaping transactions along a packet's path from its leaf to the root.

5.3 The expressiveness of PIFOs

In addition to the three examples from §5.2, we now provide several more examples of scheduling algorithms that can be programmed using our programming model (§5.3.1 through §5.3.4) and also describe the limitations of our programming model (§5.3.5).

5.3.1 Least Slack-Time First

Least Slack-Time First (LSTF) [150, 161] schedules packets at each router in increasing order of packet slacks, *i.e.*, the time remaining until each packet's deadline. Packet slacks are initialized at an end host or edge router and are decremented by the wait time at each router's queue. We can program LSTF using a simple scheduling transaction:

```
p.rank  = p.slack + p.arrival_time
```

The addition of the packet's arrival time to the slack already carried in the packet ensures that packets are dequeued in order of what their slack *would be* at the time of dequeue, not what their slack time *is* at the time of enqueue. Then, after packets are dequeued, we subtract the time at which the packet is dequeued from the packet's slack, which has the effect of decrementing the slack by the wait time at the router's queue. This subtraction can be achieved by programming the egress pipeline to decrement one header field by another using the techniques from Chapter 4.

5.3.2 Stop-and-Go Queueing

```
if (now >= frame_end_time):
  frame_begin_time = frame_end_time
  frame_end_time   = frame_begin_time + T
p.rank = frame_end_time

```

Figure 5-6: Shaping transaction for Stop-and-Go Queueing.

Stop-and-Go Queueing [117] is a non-work-conserving algorithm that provides bounded delays to packets using a framing strategy. Time is divided into non-overlapping frames of equal length `T`, where every packet arriving within a frame is transmitted at the end of the frame, smoothing out any burstiness in traffic patterns induced by previous hops.

The shaping transaction in Figure 5-6 can be used to program Stop-and-Go Queueing. `frame_begin_time` and `frame_end_time` are two state variables that track the beginning and end of the current frame in wall-clock time. When a packet is enqueued, its departure time is set to the end of the current frame. Multiple packets with the same departure time are sent out in first-in first-out order, as guaranteed by a PIFO's semantics for breaking ties with equal ranks (§5.2).

5.3.3 Minimum rate guarantees

A common scheduling policy on many routers today is providing a minimum rate guarantee to a flow, provided the sum of such guarantees does not exceed the link capacity. A minimum rate guarantee can be programmed using PIFOs with a two-level PIFO tree, where the root of the tree implements strict priority scheduling across flows. Flows below their minimum rate are scheduled preferentially to flows above their minimum rate. Then, at the next level of the tree, the PIFOs implement the First In First Out (FIFO) discipline for each flow, by using the packet's arrival time as its rank.

When a packet is enqueued, we execute a scheduling transaction corresponding to the FIFO discipline at its leaf node, setting its rank to the wall-clock time on arrival. At the root, a PIFO reference (the packet's flow identifier) is pushed into the root PIFO using a rank that reflects whether the flow is above or below its rate limit after the arrival of the current packet. To determine this, we run the scheduling transaction in Figure 5-7 that uses a token bucket (the state variable `tb`) that can be filled up until `BURST_SIZE` to decide if the arriving packet puts the flow above or below `min_rate`.

Note that a single PIFO node with the scheduling transaction in Figure 5-7 is not sufficient. It causes packet reordering within a flow: an arriving packet can cause a flow to move from a lower to a higher priority and, in the process, leave before low priority packets from the same flow that arrived earlier. The two-level tree solves this problem by attaching priorities to transmission opportunities for a specific flow, instead of attaching priorities to specific packets. Now, if an arriving packet causes a flow to move from low to high priority, the next packet scheduled from this flow is the

```
# Replenish tokens
tb = tb + min_rate * (now - last_time)
if (tb > BURST_SIZE):
  tb = BURST_SIZE

# Check if we have enough tokens
if (tb > p.size):
  p.over_min = 0 # under min. rate
  tb = tb - p.size
else:
  p.over_min = 1 # over min. rate

last_time = now
p.rank = p.over_min
```

Figure 5-7: Scheduling transaction for minimum rate guarantees.

earliest packet of that flow chosen in FIFO order, not the packet that just arrived.

5.3.4 Other examples

We now briefly describe several more scheduling algorithms that can be programmed using PIFOs.

1. **Fine-grained priority scheduling.** Many algorithms schedule the packet with the lowest value of a field initialized by the end host. These algorithms can be programmed by setting the packet's rank to the appropriate field. Examples of such algorithms and the fields they use are: strict priority scheduling (IP TOS field), Shortest Flow First (flow size), Shortest Remaining Processing Time (remaining flow size), Least Attained Service (bytes received for a flow), and Earliest Deadline First (time until a deadline).
2. **Service-Curve Earliest Deadline First (SC-EDF) [182]** schedules packets in increasing order of a deadline computed from a flow's service curve, which specifies the service a flow should receive over any given time interval. We can program SC-EDF using a scheduling transaction that sets a packet's rank to the deadline computed by the SC-EDF algorithm.
3. **Rate-Controlled Service Disciplines (RCSD) [214]** such as Jitter-EDD [203] and Hierarchical Round Robin [136] are a class of non-work-conserving schedulers. This class of schedulers can be implemented using a combination of a rate regulator to shape traffic and a packet scheduler to schedule the shaped traffic. An RCSD algorithm can be programmed using PIFOs by programming the rate regulator using a shaping transaction and the packet scheduler using a scheduling transaction.
4. **Incremental deployment of programmable scheduling.** Operators may wish to use programmable scheduling only for a subset of their traffic. This can be programmed as a hierarchical scheduling algorithm, with one FIFO class dedicated to legacy traffic and another to experimental traffic. Within the experimental class, an operator could program any

scheduling tree, *e.g.,* WFQ, LSTF, HPFQ.

5.3.5 Limitations of the PIFO programming model

Changing the scheduling order of all packets of a flow. A tree of PIFOs can enable some algorithms (*e.g.,* HPFQ in §5.2.2) where the relative scheduling order of buffered packets changes in response to new packet arrivals. However, it does not permit arbitrary changes to the scheduling order of buffered packets. In particular, it does not support changing the scheduling order for *all* buffered packets of a flow when a new packet from that flow arrives.

An example of an algorithm that needs this capability is pFabric [68]. pFabric introduces "starvation prevention" to schedule the packets of the flow with the shortest remaining size in FIFO order, to prevent packet reordering within a flow. To see why this is beyond the capabilities of PIFOs, consider the sequence of arrivals below, where pi(j) represents a packet from flow i with remaining size j. The remaining size is the number of unacknowledged bytes in a flow.

1. Enqueue p0(7).
2. Enqueue p1(9), p1(8).
3. The scheduling order is: p0(7), p1(9), p1(8).
4. Enqueue p1(6).
5. The new order is: p1(9), p1(8), p1(6), p0(7).

Specifying these semantics are beyond the capabilities of PIFOs.[6] For instance, adding a level of hierarchy with a PIFO tree does not help. Suppose we programmed a PIFO tree implementing FIFO at the leaves and picking among flows at the root based on the remaining flow size. This would result in the scheduling order p1(9), p0(7), p1(8), p1(6), after enqueueing p1(6). The problem is that there is no way to change the scheduling order for *multiple* references to flow 1 in the root PIFO by enqueueing only one reference to flow 1.

A single PIFO *can*, however, implement pFabric without starvation prevention, which is identical to the Shortest Remaining Processing Time (SRPT) discipline (§5.3.4). It can also implement the Shortest Flow First (SFF) discipline (§5.3.4), which performs almost as well as pFabric [68].

Traffic shaping across multiple nodes in a scheduling tree. Our programming model attaches a single shaping and scheduling transaction to a tree node. This lets us enforce rate limits on a single node, but not across multiple nodes.

As an example, PIFOs cannot express the following policy: WFQ on a set of flows A, B, and C, with the additional constraint that the aggregate throughput of A and B combined does not exceed 10 Mbit/s. One work around is to implement this as HPFQ across two classes C1 and C2, with C1 containing A and B, and C2 containing C alone. Then, we can enforce the rate limit of 10 Mbit/s on C1 as in Figure 5-4. However, this is not equivalent to our desired policy. More generally, our programming model for programmable scheduling establishes a one-to-one relationship between the scheduling and shaping transactions, which excludes some scheduling algorithms.

Output rate limiting. The PIFO abstraction enforces rate limits using a shaping transaction, which

[6] This is ironic because we started this project to program pFabric on a high-speed router, but have ended up being able to do almost everything but that!

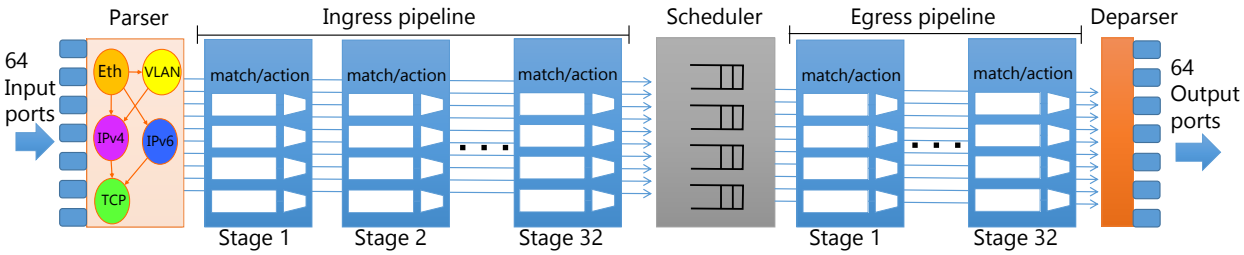


Figure 5-8: A router with a single pipeline. Combinational logic and memory are shared across all ports, both in the ingress and egress pipelines and in the scheduler.

determines a packet or PIFO reference's scheduling time *before* it is enqueued into a PIFO. The shaping transaction permits rate limiting on the *input* side, *i.e.*, before elements are enqueued. An alternate form of rate limiting is on the *output*, *i.e.*, by limiting the rate at which elements are scheduled.

To illustrate the difference between input and output rate limiting, consider a scheduling algorithm with two priority queues, `LO` and `HI`, where `LO` is to be rate limited to 10 Mbit/s. To program this using input rate limiting, we would use a shaping transaction to impose a 10 Mbit/s rate limit on `LO` and a scheduling transaction to implement strict priority scheduling between `LO` and `HI`. Now, assume packets from `HI` starve `LO` for a long period of time. During this time, packets from `LO`, after leaving the shaping PIFO, accumulate in the PIFO shared with `HI`. Now, if there are suddenly no more `HI` packets, all packets from `LO` are transmitted at the router's line rate, and are no longer rate limited to 10 Mbit/s until all instances of `LO` are drained out of the PIFO shared with `HI`. Input rate limiting still provides long-term rate guarantees, while output rate limiting provides short-term guarantees as well.

5.4 Design

We now present a hardware design for a programmable scheduler based on PIFOs. As discussed earlier (Chapter 3), the number of packet processing pipelines depends on the aggregate capacity of the router. For our PIFO design, we focus on a 64-port single-pipeline router with each port running at 10 Gbit/s. Here, a single pipeline handles the aggregate processing requirements of all ports at minimum packet size, *i.e.*, 64 10 Gbit/s ports each transmitting 64 byte packets. This translates into ~1 billion packets per second, after accounting for minimum inter-packet gaps, or a 1 GHz clock frequency for the pipeline. Figure 5-8 reviews the architecture of such a router.

We first describe how scheduling and shaping transactions can be implemented (§5.4.1). Then, we show how a tree of PIFOs can be realized using a full mesh of *PIFO blocks* by appropriately interconnecting these blocks (§5.4.2). We also describe how a compiler could automatically configure this mesh from a scheduling tree (§5.4.3).

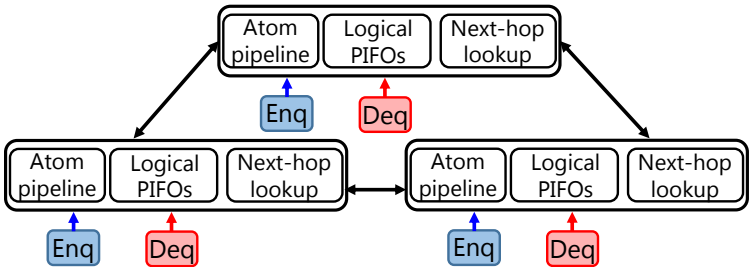


Figure 5-9: Three PIFO blocks in a PIFO mesh

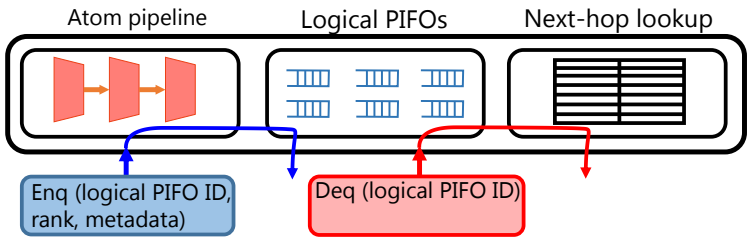


Figure 5-10: A single PIFO block. Enqueue operations execute transactions in the atom pipeline before enqueueing elements into a logical PIFO. Dequeue operations dequeue elements from logical PIFOs before looking up their next hop in the lookup table within each PIFO block.

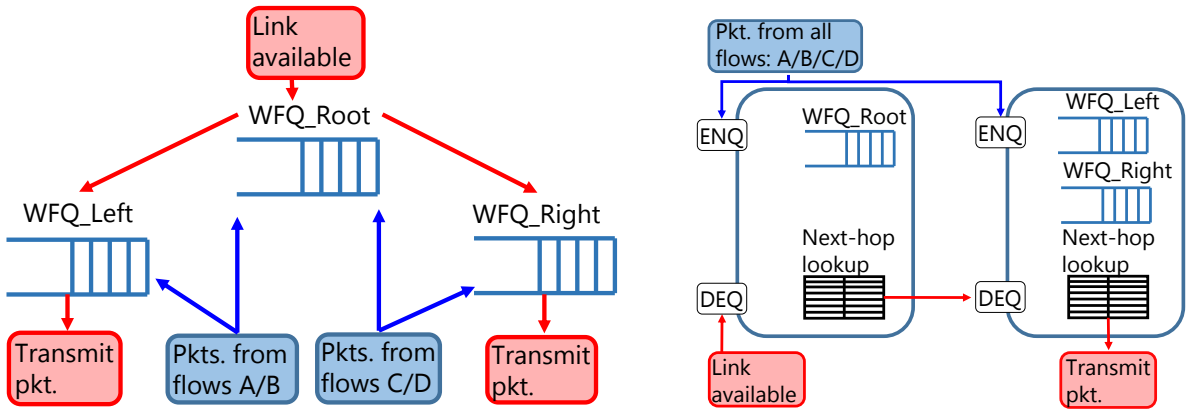
5.4.1 Scheduling and shaping transactions

To program and implement scheduling and shaping transactions, we use Domino (Chapter 4). Domino allows us to express these transactions using Domino's domain-specific language. We then use the Domino compiler to compile the transactions down to a pipeline of Banzai atoms. One of the examples used to evaluate Domino in §4.4 is the STFQ transaction from Figure 5-1. Specifically, Table 4.5 shows that the STFQ transaction can be run at 1 GHz on a router pipeline with the Pairs atom.

Similarly, we could use the Domino compiler to compile other scheduling and shaping transactions to an atom pipeline. For example, the transactions for Token Bucket Filtering (Figure 5-4c), minimum rate guarantees (§5.3.3), Stop-and-Go queueing (§5.3.2), and LSTF (§5.3.1), can all be expressed as Domino programs. They can then be compiled to a router pipeline with a sufficiently expressive atom. An important restriction in Domino is the absence of loops, which precludes rank computations containing a loop with an unbounded iteration count. We have not, however, encountered a scheduling or shaping transaction requiring this capability.

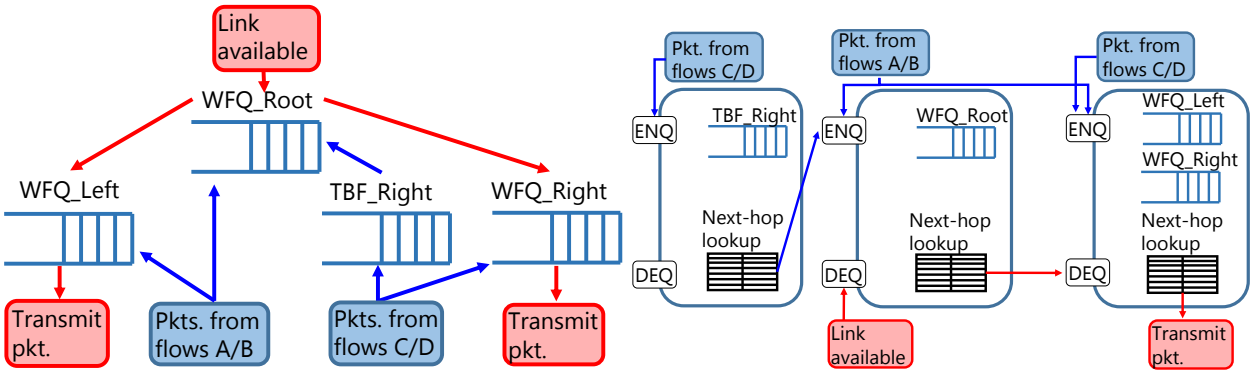
5.4.2 The PIFO mesh

We lay out PIFOs physically as a full mesh (Figure 5-9) of *PIFO blocks* (Figure 5-10). Each PIFO block supports multiple *logical PIFOs*. These logical PIFOs correspond to PIFOs for different output ports or different classes in a hierarchical scheduling algorithm (*e.g.,* the Left and Right class in Figure 5-2), which share physical resources such as memory and combinational logic required



(a) Logical PIFO tree for HPFQ (b) Physical PIFO mesh for HPFQ

Figure 5-11: Compiling HPFQ (Figure 5-2) to a PIFO mesh. On the left, the logical PIFO tree captures relationships between PIFOs: which PIFOs dequeue from or enqueue into which PIFOs. Red arrows indicate dequeues, blue indicates enqueues. On the right, we show the physical PIFO mesh for the logical PIFO tree on the left, following the same notation.



(a) Logical PIFO tree for Hierarchies with Shaping (b) Physical PIFO mesh for Hierarchies with Shaping

Figure 5-12: Compiling Hierarchies with Shaping (Figure 5-4) to a PIFO mesh. Same comments as Figure 5-11 apply.

for a PIFO. This is analogous to multiple virtual machines being multiplexed on top of a single physical machine.

We expect a small number of PIFO blocks in a typical router (*e.g.,* fewer than five) because each PIFO block corresponds to a different level of a hierarchical scheduling tree, and most practical hierarchical scheduling algorithms we know of do not require more than 2 or 3 levels of hierarchy. As a result, a full mesh between these blocks is feasible (§5.5.3 has more details).

PIFO blocks run at 1 GHz and contain an atom pipeline to execute scheduling and shaping transactions before enqueueing into a logical PIFO. In every clock cycle, each PIFO block supports one enqueue and dequeue operation on a logical PIFO residing within that block (shaping transactions require more than one operation per clock cycle and are discussed in §5.4.4). We address a logical

PIFO within a block with a logical PIFO ID.

The interface to a PIFO block is:

1. Enqueue an element (packet or reference to another PIFO) given a logical PIFO ID, the element's rank, and some metadata that will be carried with the element such as the packet length required for STFQ's rank computation. The enqueue returns nothing.
2. Dequeue from a specific logical PIFO ID within the block. The dequeue returns either a packet or a reference to another PIFO.

After a dequeue, besides transmitting a packet, a PIFO block may communicate with another PIFO block for two reasons:

1. To dequeue a logical PIFO in another block, *e.g.,* when dequeueing a sequence of PIFOs from the root to a leaf of a scheduling tree to transmit packets.
2. To enqueue into a logical PIFO in another block, *e.g.,* when enqueueing a packet that has just been dequeued from a shaping PIFO.

We configure these post-dequeue operations using a small lookup table, which looks up the "next hop" following a dequeue. This lookup table specifies an operation (enqueue, dequeue, transmit), the PIFO block for the next operation, and any arguments the operation needs.

5.4.3 Compiling from a scheduling tree to a PIFO mesh

A programmer should not have to manually configure a PIFO mesh. Instead, a compiler would translate from a scheduling tree to a PIFO mesh configuration implementing that tree. We have not yet prototyped this compiler, but we illustrate how it would work using HPFQ (Figure 5-2) and Hierarchies with Shaping (Figure 5-4).

The compiler first converts the scheduling tree to a *logical PIFO tree* that specifies the enqueue and dequeue operations on each logical PIFO. Figures 5-11a and 5-12a show the logical PIFO tree for the scheduling trees in Figures 5-2 and 5-4 respectively. It then overlays this tree over a PIFO mesh by assigning every level of the tree to a PIFO block and configuring the lookup tables to connect PIFO blocks as required by the tree. Figure 5-11b shows the PIFO mesh for Figure 5-2, while Figure 5-12b shows the PIFO mesh for Figure 5-4.

If a particular level of the tree has more than one enqueue or dequeue from another level, which arises in the presence of shaping transactions (§5.4.4), we allocate new PIFO blocks to respect the constraint that any PIFO block provides only one enqueue and dequeue operation per clock cycle. For instance, Figure 5-12b has an additional PIFO block containing TBF_Right alone. Finally, we use the Domino compiler to compile scheduling and shaping transactions.

5.4.4 Challenges with shaping transactions

Each PIFO block supports one enqueue and dequeue operation per clock cycle. This suffices for any algorithm that only uses scheduling transactions (work-conserving algorithms) because, for such algorithms, each packet needs at most one enqueue and one dequeue at each level of its scheduling tree, and we map the PIFOs at each tree level to a different PIFO block.

However, shaping transactions pose additional challenges. Consider Hierarchies with Shaping (Figure 5-12a). When the shaping transaction enqueues elements into TBF_Right, these elements

will be released into WFQ_Root at a future time T. The external enqueue into WFQ_Root may also happen at T, if a packet arrives exactly at T. This creates a conflict because there are two enqueue operations in the same clock cycle. Conflicts may also occur on dequeues. For instance, if TBF_Right shared its PIFO block with another logical PIFO, dequeue operations to the two logical PIFOs could occur at the same time because TBF_Right can be dequeued at any arbitrary wall-clock time.

In a conflict, only one of the two operations can proceed. We resolve this conflict in favor of scheduling PIFOs. Shaping PIFOs are used for rate limiting flows to a rate lower than the output port's line rate. Therefore, they can afford to be delayed by a few clocks until there are no conflicts. By contrast, delaying scheduling decisions of a scheduling PIFO would mean that the router would idle and not satisfy its line-rate guarantee on the port the scheduling PIFO belongs to.

As a result, shaping PIFOs only get best-effort service. There are workarounds to this. One is overclocking the pipeline at (say) 1.25 GHz instead of 1 GHz, providing spare clock cycles for such best-effort processing of shaping PIFOs. Another is to provide multiple ports to a PIFO block to support multiple operations every clock. These techniques are commonly used in routers for background tasks such as reclaiming buffer space, and can be applied to the PIFO mesh as well.

5.5 Hardware Implementation

This section describes the hardware implementation of a PIFO-based programmable scheduler. We discuss performance requirements for a single PIFO block (§5.5.1), the implementation of a single PIFO block (§5.5.2), and the full-mesh interconnect between PIFO blocks (§5.5.3). Finally, we estimate the additional chip area incurred by our design (§5.5.4).

5.5.1 Performance requirements for a PIFO block

Our goal is a programmable scheduler competitive with routers such as the Broadcom Trident II [18], used in many datacenters today. Based on the Trident II, we target 1000 flows that can be flexibly allocated across logical PIFOs. We target a 12 MByte packet buffer size [40] with a cell size[7] of 200 bytes. In the worst case, every packet is a single cell. Therefore, up to 60K packets/elements per PIFO block can be spread out over multiple logical PIFOs.

Based on these requirements, our baseline design targets a PIFO block that supports 64K packets and 1024 flows that can be shared across 256 logical PIFOs. These 256 logical PIFOs could support up to 256 output ports at the root level of a scheduling hierarchy. At the intermediate levels, the logical PIFOs could support up to 256 classes spread out across all output ports. Further, we target a 16-bit rank field and a 32-bit metadata field (*e.g.,* `p.length` in Figure 5-1) for our PIFO block. We put 5 such blocks together into a 5-block PIFO mesh that can support up to 5 levels of hierarchy in a scheduling algorithm, sufficient for all practical hierarchical schedulers we know of.

[7]Packets in routers are allocated in small units called cells.

5.5.2 A single PIFO block

A PIFO block supports 2 operations: an enqueue that inserts an element into a logical PIFO within the block and a dequeue to remove the head of a logical PIFO within the block. We first consider two strawman designs for a PIFO block: a flat sorted array and a heap. We show how neither meets our requirements. We then describe our implementation of a PIFO block. We start with a PIFO block supporting a single logical PIFO and then show how it easily extends to support multiple logical PIFOs in the same block.

One strawman hardware implementation of a single PIFO is a flat sorted array. In this implementation, an incoming element would be compared against all array elements in parallel to determine a location for the new element, and then inserted there by shifting the array. However, each comparison needs one comparator circuit; supporting 64K parallel comparators is infeasible in hardware.

Another hardware implementation for a PIFO, which is essentially a priority queue, is a heap in hardware. For instance, P-heap is a pipelined binary heap scaling to 4-billion entries [79, 80]. A capacity of 4 billion is far more than what we need (64K). However, each P-heap supports traffic belonging to a *single* 10 Gbit/s input port in an input-queued router; there is a separate P-heap instance for each port [79]. Having a separate P-heap per port incurs prohibitive area overhead on a single-pipeline shared-memory router, where the packet buffer and scheduling logic are shared across all ports to reduce area overhead. We also found it challenging to overlay multiple logical PIFOs over a single physical P-heap, which would have allowed the same physical P-heap to be shared across ports.

In contrast to the two strawman designs, our design for the PIFO exploits two domain-specific characteristics. First, the packet buffers on routers used in datacenters today are much smaller (tens of megabytes) than those in deep-buffered core routers (hundreds of megabytes) in the past. This permits a simpler, albeit less scalable, design relative to heaps because the design needs to support far fewer entries.

Second, there is considerable structure in the pattern of ranks within a PIFO. Nearly all practical scheduling algorithms group packets into flows or classes, *e.g.,* based on traffic type, ports, or addresses. They then schedule a flow's packets in FIFO order because packet ranks increase across a flow's consecutive packets to prevent packet reordering within a flow. Thus, a PIFO needs to only sort across the head packets of all flows, as opposed to all packets in the buffer.[8]

This motivates a design with two parts (Figure 5-13):

1. A *flow scheduler* that picks the next element to dequeue based on the ranks of the *head* (earliest) elements of all flows. The flow scheduler is effectively a PIFO consisting of the head elements of all flows.
2. A *rank store*, a FIFO bank that stores the ranks of elements beyond the head for each flow in FIFO order.

This decomposition reduces the number of elements requiring sorting from the number of packets (64K) to the number of flows (1024). Maintaining a flat sorted array of 1024 elements turns out to be much more feasible in today's transistor technology.

[8]Last-In First-Out (LIFO) is a counterexample. We can handle LIFO by creating a new flow for every packet, so long as there are fewer than 1024 packets (flows) in the buffer at any time.

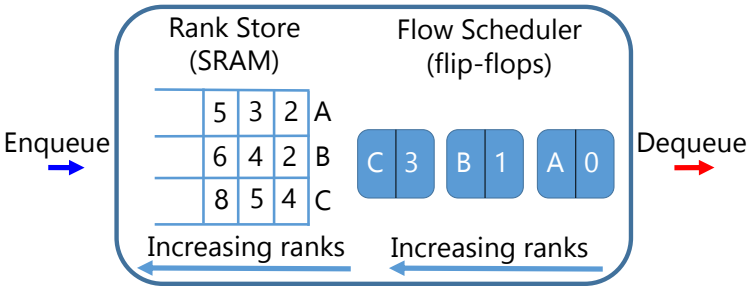


Figure 5-13: Block diagram of PIFO block with a flow scheduler and a rank store. Logical PIFOs and metadata are not shown for simplicity.

During an enqueue, an element (both rank and metadata) is appended to the end of the appropriate FIFO in the rank store. For a flow's first element, we bypass the rank store and directly push it into the flow scheduler. To permit enqueues into this PIFO block, we also supply a flow ID argument to the enqueue operation. During dequeues, we dequeue the element with the earliest rank among the head elements of all flows in the flow scheduler. We then insert the next element after the head for the flow that was just dequeued.

The FIFO bank needed for the rank store is a well understood hardware design. Such FIFO banks are used to buffer packet payloads in router queues and substantial engineering effort has gone into optimizing them. As a result, we focus our discussion here on the flow scheduler alone.

The flow scheduler. The flow scheduler sorts an array of flows using the ranks of the head elements of all flows. It supports one enqueue *and* one dequeue to its enclosing PIFO block every clock cycle, which translates into the following operations on the flow scheduler every clock cycle.

1. Enqueue operation: Inserting a flow into the flow scheduler when the flow goes from empty to non-empty.
2. Dequeue operation: Removing a flow from the flow scheduler that empties once it is scheduled, (or) removing and reinserting a flow into the flow scheduler with the rank of the next element if the flow is still backlogged.

The operations above require the flow scheduler to internally support two kinds of operations every clock cycle.

1. *Push* up to two elements into the flow scheduler: one each for an enqueue's insert and a dequeue's reinsert.
2. *Pop* one element: for the removal of a flow because of a dequeue.

These internal operations access all of the flow scheduler's elements in parallel. To facilitate this, we implement the flow scheduler in flip flops, unlike the rank store, which is in SRAM.

The flow scheduler is organized in hardware as a sorted array, where a push is implemented by executing the three steps below (Figure 5-14).

1. Compare the incoming rank against all ranks in the array in parallel using a comparator. This produces a bit mask of comparison results indicating if the incoming rank is greater/less than an array element's rank.
2. Find the first 0-to-1 transition in this bit mask, using a priority encoder. The location of this transition determines the index to push into.

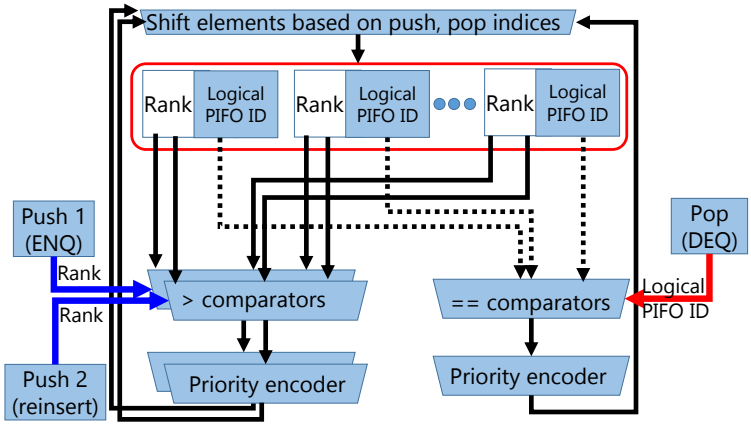


Figure 5-14: Hardware implementation of flow scheduler. Each element in the flow scheduler is connected to two > comparators (2 pushes) and one == comparator (1 pop).

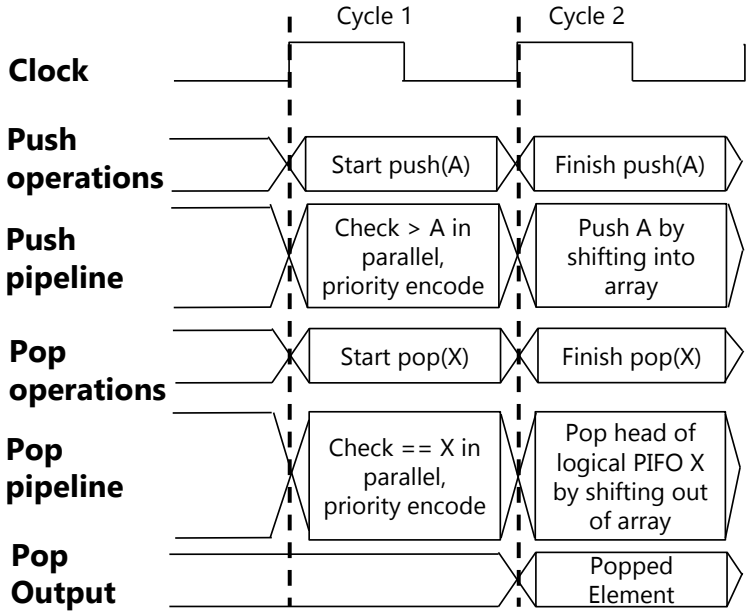


Figure 5-15: 2-stage pipeline for flow scheduler

3. Push the element into this index, by shifting the array.

A pop is implemented by shifting the head element out of the sorted array.

So far, we have focused on a flow scheduler implementation handling a single logical PIFO. We can extend this to handle multiple logical PIFOs backed by the same underlying physical hardware. To do so, we keep elements sorted by rank, regardless of the logical PIFO they belong to; hence, the push logic does not change. To pop from a specific logical PIFO, we compare the dequeue's logical PIFO ID against all elements to find elements with that logical PIFO ID alone. Among these elements, we find the first using a priority encoder, and remove this element by shifting the array. The rank store implementation does not change when introducing logical PIFOs; however, we do

require that a flow belong to exactly one logical PIFO.

Recall that to support one enqueue and one dequeue every clock cycle, the flow scheduler needs to support 2 pushes and 1 pop every clock cycle. To concurrently issue 2 pushes and 1 pop every clock cycle, we provision 3 parallel digital circuits (Figure 5-14). Both the push and pop require 2 clock cycles to complete and need to be pipelined to maintain the required throughput (Figure 5-15). For pushes, the first stage of the pipeline executes the parallel comparison and priority encoder steps to determine an index; the second stage pushes the element into the array using the index. Similarly, for pops, the first stage executes the equality check (for logical PIFO IDs) and priority encoder steps to compute an index; the second stage pops the head element out of the array using the index.

Our implementation meets timing at 1 GHz and supports up to one enqueue/dequeue operation on a logical PIFO within a PIFO block every clock cycle. Because a reinsert operation requires a pop, followed by an access to the rank store for the next element, followed by a push, our implementation supports a dequeue from the same logical PIFO only once every 4 clock cycles. This is because if a dequeue is initiated in clock cycle 1, the pop for the dequeue completes in 2, the rank store is accessed in 3, and the push is initiated in 4. This makes clock cycle 5 the earliest time to reissue a dequeue. This restriction is inconsequential in practice. A dequeue every 4 clock cycles from a logical PIFO is sufficient to service the highest link speed today, 100 Gbit/s, which requires a dequeue at most once every 5 clock cycles for a minimum packet size of 64 bytes. Dequeues to distinct logical PIFO IDs are still permitted every clock cycle.

5.5.3 Interconnecting PIFO blocks

An interconnect between PIFO blocks allows PIFO blocks to enqueue into and dequeue from other blocks. Because the number of PIFO blocks is small, we provide a full mesh between them. For a 5-block PIFO mesh as in our baseline design, this requires 5*4 = 20 sets of wires between PIFO blocks. Each set carries all the inputs required for specifying an enqueue and dequeue operation on a PIFO block.

We now calculate the size of the inputs required to specify enqueue and dequeue operations. For our baseline design (§5.5.1), for an enqueue, we require a logical PIFO ID (8 bits), the element's rank (16 bits), the element meta data (32 bits), and the flow ID (10 bits). For a dequeue, we need a logical PIFO ID (8 bits) and wires to store the dequeued element's metadata field (32 bits). Combining both enqueue and dequeue, this adds up to 106 bits per set of wires, or 2120 bits for the mesh. This is a small number of wires for an entire chip. For example, RMT's match-action pipeline uses 4000 1-bit wires between a *a pair of pipeline stages* to move its 4K packet header vector between stages [86].

5.5.4 Area overhead

Because we target a single-pipeline router, the scheduling logic is shared across all ports and a single PIFO mesh services an entire router. Therefore, to estimate the area overhead of a programmable scheduler, we estimate the area overhead of a single PIFO mesh. Our overhead does not have to be multiplied by the number of ports and is the same for two single-pipeline routers with equal aggregate packet rates, *e.g.,* a 6-port 100G router and a 60-port 10G router.

To determine the area of a PIFO mesh, we compute the area of a single PIFO block and multiply it by the number of blocks because the area of the interconnect itself is negligible (§5.5.3). For a single block's area, we separately estimate areas for the rank store, atom pipelines, and flow scheduler. We ignore the area of the small next-hop lookup tables. We estimate the rank store's area by using SRAM estimates [50] and the atom pipeline's area using the individual atom area numbers from Table 4.4. We estimate the flow scheduler's area by implementing it in Verilog [51] and synthesizing it to a gate-level netlist in a 16-nm standard cell library using the Cadence Encounter RTL Compiler [6]. The RTL Compiler also verifies that the flow scheduler meets timing at 1 GHz.

Overall, our baseline design consumes about 7.35 mm^2 of chip area (Table 5.1). This is about 3.7% of the chip area of a router chip, using the minimum chip area estimate of 200 mm^2 provided by Gibb et al. [116]. In return for this 3.7%, we get a significantly more flexible packet scheduler than current routers, which provide *fixed* two or three-level hierarchical scheduling. Our 3.7% area overhead is similar to the overhead for other programmable router functions, *e.g.,* 2% for programmable parsing [116] and 15% for programmable header processing [86].

Component	Area in mm^2
Router chip	200–400 [116]
Flow Scheduler	0.224 (from synthesis)
SRAM (1 Mbit)	0.145 [50]
Rank store	64 K * (16 + 32) bits * 0.145 mm^2 / Mbit = 0.445
Next pointers for linked lists in dynamically allocated rank store	64 K * 16 bit pointers * 0.145 = 0.148
Free list memory for dynamically allocated rank store	64 K * 16 bit pointers * 0.145 = 0.148
Head, tail, and count memory for each flow in the rank store	0.1476 (from synthesis)
One PIFO block	0.224 + 0.445 + 0.148 + 0.148 + 0.1476 = 1.11 mm^2
5-block PIFO mesh	5.55
300 atoms spread out over the 5-block PIFO mesh for rank computations	6000 $\mu\mathrm{m}^2$* 300 = 1.8 mm^2 (§5.4.1, Table 4.4)
Overhead for 5-block PIFO mesh	(5.55 + 1.8) / 200.0 = 3.7 %

Table 5.1: A 5-block PIFO mesh needs 3.7% additional chip area relative to a baseline router.

Varying the flow scheduler's parameters from the baseline. The flow scheduler has four parameters: rank width, metadata width, number of logical PIFOs, and number of flows. Among these, increasing the number of flows has the most impact on whether the flow scheduler meets timing at 1 GHz. This is because the flow scheduler uses a priority encoder, whose size is the number of flows and whose critical path delay increases with the number of flows. With other parameters set to their baseline values, we vary the number of flows to determine the eventual limits of a flow scheduler with today's transistor technology (Table 5.2), and find that we can scale to 2048 flows while still meeting timing at 1 GHz.

The remaining parameters affect the area of a flow scheduler, but have little effect on meeting timing at 1 GHz. For instance, starting from the baseline design of the flow scheduler that takes up

# of flows	Area (mm^2)	Meets timing at 1 GHz?
256	0.053	Yes
512	0.107	Yes
1024	0.224	Yes
2048	0.454	Yes
4096	0.914	No

Table 5.2: The flow scheduler's area increases with the number of flows. The flow scheduler meets timing until 2048 flows.

0.224 mm^2, increasing the rank width to 32 bits increases it to 0.317 mm^2, increasing the number of logical PIFOs to 1024 increases it to 0.233 mm^2, and increasing the metadata width to 64 bits increases it to 0.317 mm^2. In all cases, the flow scheduler continues to meet timing.

5.5.5 Additional implementation concerns

Coordination between enqueue and dequeue. When computing packet ranks on enqueue, some scheduling algorithms access state modified on packet dequeues. An example is STFQ (§5.2.1) that accesses the `virtual_time` variable when computing a packet's virtual start time. This enqueue-dequeue coordination can be implemented in two ways. One is shared state that can be accessed on both enqueue and dequeue, similar to queue occupancy counters. Another is to periodically synchronize the enqueue and dequeue views of the same state, potentially by creating and recirculating a packet carrying this information from the dequeue to the enqueue pipeline. For STFQ, the degree of short-term fairness is directly correlated with how up-to-date the `virtual_time` information on the enqueue side is.

Buffer management. Our design focuses on programmable scheduling and does not manage the allocation of a router's data buffer across flows. Buffer management can use static buffer limits for each flow. The limits can also be dynamic, *e.g.,* RED [111] and dynamic buffer sizing [93].

In a single-pipeline shared-memory router, buffer management is orthogonal to scheduling, and is implemented using counters that track flow occupancy in a shared buffer. Before a packet is enqueued into the scheduler, if any counter exceeds a static or dynamic threshold, the packet is dropped. A similar design for buffer management could be used with a PIFO-based scheduler as well.

Priority Flow Control. Priority Flow Control (PFC) [42] is a standard that allows a router to send a *pause* message to an upstream router requesting it to cease transmission of packets belonging to particular flows. PFC can be integrated into our hardware design by excluding certain flows in the flow scheduler during the dequeue operation if they have been paused because of a PFC pause message. These flows can be included again when a PFC *resume* message is received. This exclusion-inclusion process can be accomplished by using a bitmask to indicate which flows are currently excluded or included and ANDing with this bitmask when dequeueing from the flow scheduler.

5.6 Summary

Existing research into programmable and fast routers has looked at programming the router's parser [116] and its match-action pipeline [86]. But, so far, it has been considered off-limits to program the packet scheduler—in part because the desired algorithms are so varied, and because the scheduler sits at the heart of the shared packet buffer where timing requirements are tightest. It has been widely assumed too hard to find a useful abstraction that can also be implemented in fast hardware.

The results in this chapter suggest a promising abstraction for programmable packet scheduling. Our abstraction exploits the fact that in many practical schedulers, the relative order of packets that are already buffered does not change in response to new packet arrivals. Put differently, when a packet arrives, it can be pushed into the right location based on a packet priority (push in), but packets are always dequeued from the head (first out). This observation suggests a natural primitive for programmable scheduling called a Push In First Out Queue (PIFO). A PIFO is a priority queue where packets are pushed in based on a rank field, and the next packet to be dequeued is the packet with the earliest rank.

A single PIFO expresses many schedulers, *e.g.,* token bucket shaping, weighted fair queueing, and strict priority scheduling. Further, PIFOs can be combined to express hierarchical schedulers. Finally, PIFOs are feasible in hardware: a hardware design for a programmable 5-level hierarchical scheduler costs less than 4% additional chip area.

Chapter 6

Marple: Programmable and Scalable Network Measurement

Effective performance monitoring of large networks is crucial to quickly localize problems like high queueing latency [23], TCP incast [202], and load imbalance across network links [65]. A common approach to network monitoring is to collect information from the end host network stack [209, 196, 166] or to use end-to-end probes [120] to diagnose performance problems. While end hosts provide application context, they lack visibility to localize performance problems at links deep inside the network. For example, using end host approaches alone, it is challenging to localize queue buildup to a particular router or pinpoint traffic causing the queue buildup, forcing operators to infer the network-level root causes indirectly [120].

Router-based monitoring could allow operators to diagnose problems with more direct visibility into performance statistics. However, traditional router mechanisms like sampling [8, 47], mirroring [10, 124, 215], and counting [96, 153] are quite restrictive. Sampling and mirroring miss events of interest as it is infeasible to collect information on all packets, while counters only track traffic volume statistics. None of these mechanisms provides relevant performance data, like queueing delays.

Some upcoming technologies recognize the need for better performance monitoring using routers. In-band network telemetry [23] writes queueing delays experienced by a packet on the packet itself, allowing end hosts to localize delay spikes. The Tetration chip [11] provides a flow cache that measures flow-level performance metrics. These metrics are useful, but they are exposed at a fixed granularity (*e.g.,* per 5-tuple), and the metrics themselves are fixed. For example, the list of exposed metrics includes flow-level latency and packet size variation, but not latency variation, *i.e.*, jitter.

Operator requirements are constantly changing, and redesigning hardware for every new requirement is expensive. We believe that the trajectory of adding fixed-function router monitoring piecemeal is unsustainable. Instead, we advocate building performance monitoring primitives that can be flexibly reused for a variety of monitoring needs.

In this chapter, we apply *language-directed hardware design* to the problem of flexible performance monitoring, inspired by early efforts on designing hardware to support high-level languages [156, 100, 200]. Specifically, we first design a language that can express a broad variety of

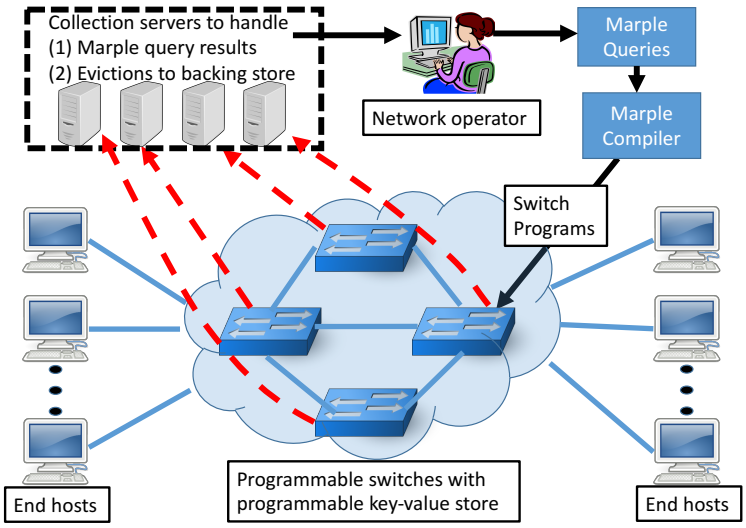


Figure 6-1: Operators issue Marple queries. These are compiled into router programs for programmable routers augmented with our new programmable key-value store. Routers stream results from this query to collection servers that also house the backing store for the key-value store.

performance monitoring use cases. We then design high-speed router hardware primitives in service of this language. By designing hardware to support an expressive language, we believe the resulting hardware design can support a wide variety of current and future performance monitoring needs.

Figure 6-1 provides an overview of our performance monitoring system. To use the system, an operator writes a query in a domain-specific language called *Marple,* either to implement a long-running monitor for a statistic (*e.g.,* detecting TCP timeouts), or to troubleshoot a specific problem (*e.g.,* incast [202]) at hand. The query is compiled into a router program that runs on the network's programmable routers, augmented with new router hardware primitives that we design in service of Marple. The routers stream results out to collection servers, where the operator can retrieve query results. We now briefly describe the three components of our system: the query language, the router hardware, and the query compiler.

Performance query language (§6.1). Marple uses familiar functional constructs like `map`, `filter`, `groupby`, and `zip` for performance monitoring. Marple provides the abstraction of a stream that contains performance information for *every packet at every queue* in the network (§6.1.1). Programmers can focus their attention on traffic experiencing interesting performance using `filter` (*e.g.,* packets with high queueing latencies), aggregate information across packets in flexible ways using `groupby` (*e.g.,* compute a moving average over queueing latency per flow), compute new stateless quantities using `map` (*e.g.,* binning a packet's timestamp into an epoch), and detect simultaneous performance conditions using `zip` (*e.g.,* when the queue depth is large and the number of connections in the queue is high).

Hardware design for performance queries (§6.2). A naive implementation of Marple might stream every packet's metadata from the network to a central location and run streaming queries against it. Modern scale-out data-processing systems support ~100K–1M operations per second per

core [4, 21, 125, 2, 48], but processing every single packet (assuming a relatively large packet size of 1 KB) from a single 1 Tbit/s router would need ~100M operations per second—2–3 orders of magnitude more than what existing systems support.

Instead, we leverage high-speed programmable routers [86, 60, 3, 25] as first-class citizens in network monitoring, because they can programmatically manipulate multi-Tbit/s packet streams. Early filtering and flexible aggregation on routers drastically reduce the number of records per second streamed out to a standard data-processing system running on the collection server.

While programmable routers support many of Marple's stateless language constructs that modify packet fields alone (*e.g.,* `map` and `filter`), they do not support aggregation of state across packets for a large number of flows (*i.e.*, `groupby`). To support flexible aggregations over packets, we design a programmable key-value store in hardware (§6.2), where the keys represent flow identifiers and the values represent the state computed by the aggregation function. This key-value store must update values at the rate of 1 packet per clock cycle (at 1 GHz [86, 7]) and support millions of keys/flows. Unfortunately, neither SRAM nor DRAM is simultaneously fast and dense enough to meet both requirements.

We split the key-value store into a small but fast on-chip cache in SRAM and a larger but slower off-chip backing store in DRAM. Traditional caches incur variable write latencies due to cache misses; however, line-rate packet forwarding requires deterministic latency guarantees. Our design accomplishes this by never reading back a value into the cache if it has already been evicted to the backing store. Instead, it treats a cache miss as the arrival of a packet from a new flow. When a flow is evicted, we *merge* the evicted flow's value in the cache with the flow's old value in the backing store. Because merges occur off the critical packet processing path and only during evictions, the backing store can be implemented in software on a separate collection server or on off-chip DRAM on the router.

While it is not always possible to merge an aggregation function without losing accuracy, we characterize a class of affine aggregation functions, which we call *linear-in-state*, for which accurate merging is possible. Many useful aggregation functions are linear-in-state, *e.g.,* counters, predicated counters (*e.g.,* count only TCP packets that saw timeouts), exponentially weighted moving averages, and functions computed over a finite window of packets. We design a router instruction to support linear-in-state functions, finding that it easily meets timing at 1 GHz, while occupying modest silicon area.

Query compiler (§6.3). We implement a compiler that takes Marple queries and compiles them into router configurations for two targets (§6.3): (1) the P4 behavioral model [37], an open source programmable software router that can be used for end-to-end evaluations of Marple on Mininet [149], and (2) Banzai [188], a simulator for high-speed programmable router hardware that can be used to experiment with different instruction sets. The Marple compiler detects linear-in-state aggregations in input queries and successfully targets the linear-in-state router instruction that we add to Banzai.

Evaluation (§6.4). We show that Marple can express a variety of useful performance monitoring examples, like detecting and localizing TCP incast and measuring the prevalence of out-of-order TCP packets. Marple queries require between 4 and 11 pipeline stages, which is modest for a 32-stage router pipeline [86]. We evaluate our key-value store's performance using trace-driven simulations. For a 64 Mbit on-chip cache, which occupies about 10% of the area of a 64×10-Gbit/s

Construct	Description
`pktstream`	Stream of packet performance metadata.
`filter(R, pred)`	Output tuples in `R` satisfying predicate `pred`.
`map(R, [exprs], [fields])`	Evaluate expressions, `[exprs]`, over fields of `R`, emitting tuples with new fields, `[fields]`.
`groupby(R, [fields], fun)`	Evaluate function `fun` over the input stream `R` partitioned by `fields`, producing tuples on `emit()`.
`zip(R, S)`	Merge fields in incoming `R` and `S` tuples.

Table 6.1: Summary of Marple language constructs.

router chip, we estimate that the cache eviction rate from a single top-of-rack router can be handled by a single 8-core server running Redis [43]. We evaluate Marple's usability through two Mininet case studies that use Marple to troubleshoot high tail latencies [61] and measure the distribution of flowlet sizes [65]. Marple is open source and available at http://web.mit.edu/marple.

6.1 The Marple Query language

This section describes the Marple query language. §6.2 then covers the router implementation of the language constructs, while §6.3 describes the compiler. Marple provides the abstraction of a network-wide stream of performance information. The tuples in the stream contain performance metadata, such as queue lengths and timestamps when a packet entered and departed queues, for each packet at each queue in the network. Network operators write queries on this stream as if the entire stream is processed by a single hypothetical server running the query. In reality, the compiler partitions the query across the network and executes each part on individual routers.

Marple programs process the performance stream using familiar functional constructs (`filter`, `map`, `groupby`, and `zip`), all of which take streams as inputs and produce a stream as output. This functional language model is expressive enough to support diverse performance monitoring use cases, but still simple enough to implement in high-speed hardware. Marple's language constructs are summarized in Table 6.1.

6.1.1 Packet performance stream

As part of the base input stream, which we call `pktstream`, Marple provides one tuple for each packet at each queue with the following fields.

```
(router, qid, hdrs, uid, tin, tout, qsize)
```

`router` and `qid` denote the router and queue at which the packet was observed. A packet may traverse multiple queues even within a single router, so we provide distinct fields. The regular packet headers (Ethernet, IP, TCP, *etc.*) are available in the `hdrs` set of fields, with a `uid` that uniquely determines a packet.[1]

The packet performance stream provides access to a variety of performance metadata: `tin` and `tout` denote the enqueue and dequeue timestamps of a packet, while `qsize` denotes the queue depth

[1] It is usually possible to use a combination of the 5-tuple and IP ID field as the `uid`.

when a packet is enqueued. It is beneficial to have two timestamps to detect co-habitation of the queue by packets belonging to different flows. Additionally, it is beneficial to have a queue size, since we cannot always determine the queue size from the two timestamps: a link may service multiple queues, and the speed at which any specific queue drains may not be known.

Tuples in `pktstream` are processed in order of packet dequeue time (`tout`), because this is the earliest time at which all tuple fields in `pktstream` are known.[2] If a packet is dropped, `tout` and `qsize` are `infinity`. Tuples corresponding to dropped packets may be processed in an arbitrary order.

6.1.2 Restricting packet performance metadata of interest

Consider the example of tracking packets that experience high queueing latencies at a specific queue (`Q`) and router (`S`). This is expressed by the query:

```
result = filter(pktstream, qid == Q and router == S
                and tout - tin > 1ms)
```

As shown above, the `filter` operator restricts the user's attention to those tuples with the relevant performance metadata. A filter has the form `filter(R, pred)` where `R` is some stream containing performance metadata (*e.g.,* `pktstream`), and the filter predicate `pred` may involve packet headers, performance metadata, or both. The result of a `filter` is another stream that contains only tuples satisfying the predicate.

6.1.3 Computing stateless functions over packets

Marple lets users compute functions of the fields available in the incoming stream, to express new quantities of interest. A simple example is rounding packet timestamps to an 'epoch':

```
result = map(pktstream, [tin/epoch_size], [epoch]);
```

The `map` operator evaluates the expression `tin/epoch_size`, written over the fields available in the tuple stream, and produces a new field `epoch`. The general form of this construct is `map(R, [expression], [field])` where a list of `expressions` over fields in the input stream `R` creates a list of new `fields` in the `map` output stream.

6.1.4 Aggregating statefully over multiple packets

Marple allows aggregating statistics over multiple tuples at user-specified granularities. For example, the following query counts packets belonging to each transport-level flow (*i.e.*, 5-tuple):

```
result = groupby(pktstream, [5tuple], count)
def count([c], []):
  c = c + 1;
```

[2]We assume clock synchronization to let us compare `tin` and `tout` values from different routers. Without synchronization, the programmer can still write queries that do not compare time-valued fields `tin` and `tout` across routers.

Here, the `groupby` partitions the incoming `pktstream` into substreams based on the transport 5-tuple, and then applies the aggregation function `count` to count the number of tuples in each substream.

Marple allows users to write flexible order-dependent aggregation functions over the tuples of each substream. For example, a user can track latency spikes for each connection by maintaining an exponentially weighted moving average (EWMA) of queueing latencies:

```
result = groupby(pktstream, [5tuple, router], ewma);
def ewma([avg], [tin, tout]):
  avg = ((1-alpha)*avg) + (alpha*(tout-tin));
```

Here the aggregation function `ewma` evolves an EWMA `avg` using the current value of `avg` and incoming packet timestamps. Unlike the previous `count` example, the EWMA aggregation function depends on the order of packets being processed, *i.e.*, the order of their $tout$ values.

`groupby`s take the general form `groupby(R, [aggFields], fun)`, where the aggregation function `fun` operates over tuples sharing attributes in a list `aggFields` of headers and performance metadata. This construct is inspired by folds in functional programming [132]. Such order-dependent folds are challenging to express in existing query languages. For instance, SQL only allows order-independent commutative aggregations (*e.g.,* count, average, sum).

The aggregation function `fun` is written in an imperative form, with two arguments: (1) a list of state variables that represent the value(s) being aggregated across packets and (2) a list of relevant incoming tuple fields that are used in the aggregation function. Each statement in `fun` can be an assignment to an expression (`x = ...`), a branching statement (`if pred {...} else {...}`), or a special `emit()` statement that controls the output stream of the `groupby`. Below, we show an example of an aggregation that detects a new connection:

```
result = groupby(pktstream, [5tuple], new_flow);
def new_flow([fcount], []):
  if fcount == 0:
    fcount = 1
    emit()
```

The output of a `groupby` is a stream containing the aggregation fields (*e.g.,* 5-tuple) and the aggregated values (*e.g.,* `fcount`). The output stream contains only tuples for which the `emit()` statement is encountered when executing the aggregation function. For example, the output stream of `new_flow` consists of the first packet of every new transport-level connection. If the function has no `emit()`s, the user can still read the aggregated fields and their current aggregated state values as a table.

6.1.5 Chaining together multiple queries

Because all Marple constructs produce and consume streams, Marple allows users to write queries that take in the results of previous queries as inputs. A stream of tuples flows from one query to the next, and each query may modify information in the incoming tuple or even drop the tuple entirely. For example, the program below tracks the size distribution of flowlets, *i.e.*, bursts of packets from the same 5-tuple separated by more than a fixed time amount `delta`.

```
fl_track = groupby(pktstream, [5tuple], fl_detect);
def fl_detect([last_time, size], [tin]):
  if (tin - last_time > delta):
    emit()
    size = 1
  else:
    size = size + 1
  last_time = tin
```

The function `fl_detect` detects new flowlets using the last time a packet from the same flow was seen. Because of the `emit()` statement's location, the flowlet size from `fl_track` is only streamed out to other operators when the first packet of a new flowlet is encountered.

```
fl_bkts  = map(fl_track, [size/16], [bucket]);
fl_hist  = groupby(fl_bkts, [bucket], count);
```

The map `fl_bkts` bins the flowlet size emitted by `fl_track` into a bucket index, which is used to count the number of flowlets in the corresponding bucket in `fl_hist`.

6.1.6 Joining results across queries

Marple provides a `zip` operator that "joins" (in the database sense of the word) the results of two queries to check whether two conditions hold simultaneously. Consider the example of detecting the fan-in of packets from many connections into a single queue, characteristic of TCP incast [202]. This can be checked by combining two distinct conditions: (1) the number of active flows in a queue over a short interval of time is high and (2) the queue occupancy is large.

A user can first compute the number of active flows over the current epoch using two aggregations:

```
R1 = map(pktstream, [tin/epoch_size], [epoch]);
R2 = groupby(R1, [5tuple, epoch], new_flow);
R3 = groupby(R2, [epoch], count);
```

The number of active flows in this epoch can be combined with the queue occupancy information in the original packet stream through the `zip` operator:

```
R4 = zip(R3, pktstream);
result = filter(R4, qsize > 100 and count > 25);
```

The result of a `zip` operation over two input streams is a single stream containing tuples that are a concatenation of all the fields in the two streams, whenever both input streams contain valid tuples processed from the same original packet tuple. A `zip` is a special kind of stream join where the result can be computed without having to synchronize the two streams, because tuples of both streams originate from `pktstream`. The result of the `zip` can be processed like any other stream: the `filter` in the `result` query checks the two incast conditions above.

We did not find a need for more general joins akin to joins in streaming query languages like CQL [72]. Streaming joins have semantics that can be quite complex and may produce large results, *i.e.*, $O(\#pkts^2)$. Hence, Marple restricts users to simple `zip` joins.

We show several examples of Marple queries in Table 6.2. For instance, Marple can express measurements of simple counters, TCP reordering of various forms, high-loss connections, flows with high end-to-end network latencies, and TCP fan-in.

6.1.7 Restrictions on Marple queries

Some aggregations are challenging to implement over a network-wide stream. For example, consider an EWMA over some packet field across all packets seen anywhere in the entire network, while processing packets in the order of their `tout` values. Even with clock synchronization, this aggregation is hard to implement because it requires us to either (1) coordinate between routers or (2) stream all packets to a central location. Both of these alternatives impose significant overheads; hence, we disallow such queries.

Marple's compiler rejects queries with aggregations that need to process *multiple packets* at *multiple routers* in *order of their* `tout` *values*. Concretely, we only allow aggregations that relax one of these three conditions, and thus either

1. operate independently on each router, in which case we naturally partition queries by router (*e.g.,* a per-flow EWMA of queueing latencies on a particular queue belonging to a particular router), or
2. operate independently on each packet, in which case we have the packet perform the coordination by carrying the aggregated state to the next router on its path (*e.g.,* a rolling average link utilization seen by the packet along its path), or
3. are associative and commutative, in which case independent router-local results can be combined in any order to produce a correct overall result for the network, *e.g.,* a count of how many times packets from a flow appeared throughout the network. In this case, we rely on the programmer to annotate the aggregation function with the `assoc` and `comm` keywords.

6.2 Scalable Aggregation at a Router's Line Rate

How should routers implement Marple's language constructs? We require instructions on routers that can aggregate packets into per-flow state (`groupby`), transform packet fields (`map`), stream only packets matching a predicate (`filter`), or merge packets that satisfy two previous queries (`zip`).

Of these four language constructs, three (`map`, `filter`, and `zip`) are *stateless*: they operate on packet fields alone and do not modify router state. Such stateless manipulations are already supported by emerging programmable routers that support programmable packet header processing [86, 60, 25, 3]. On the other hand, the `groupby` construct needs to maintain and update router state on a per-flow basis.

Stateful manipulation on a router for a `groupby` is challenging for two reasons. First, it needs to be fast: the time budget to update state before the next packet arrives at a router can be as low as 1 ns (Chapter 4). Second, it needs large memories: the router needs to maintain state proportional to the number of aggregated records (*e.g.,* per flow), which may grow unbounded with time.

However, memories can either be fast and small (SRAM) or slow and large (DRAM). To address this mismatch, we turn to the standard technique of caching. We build a two-level programmable

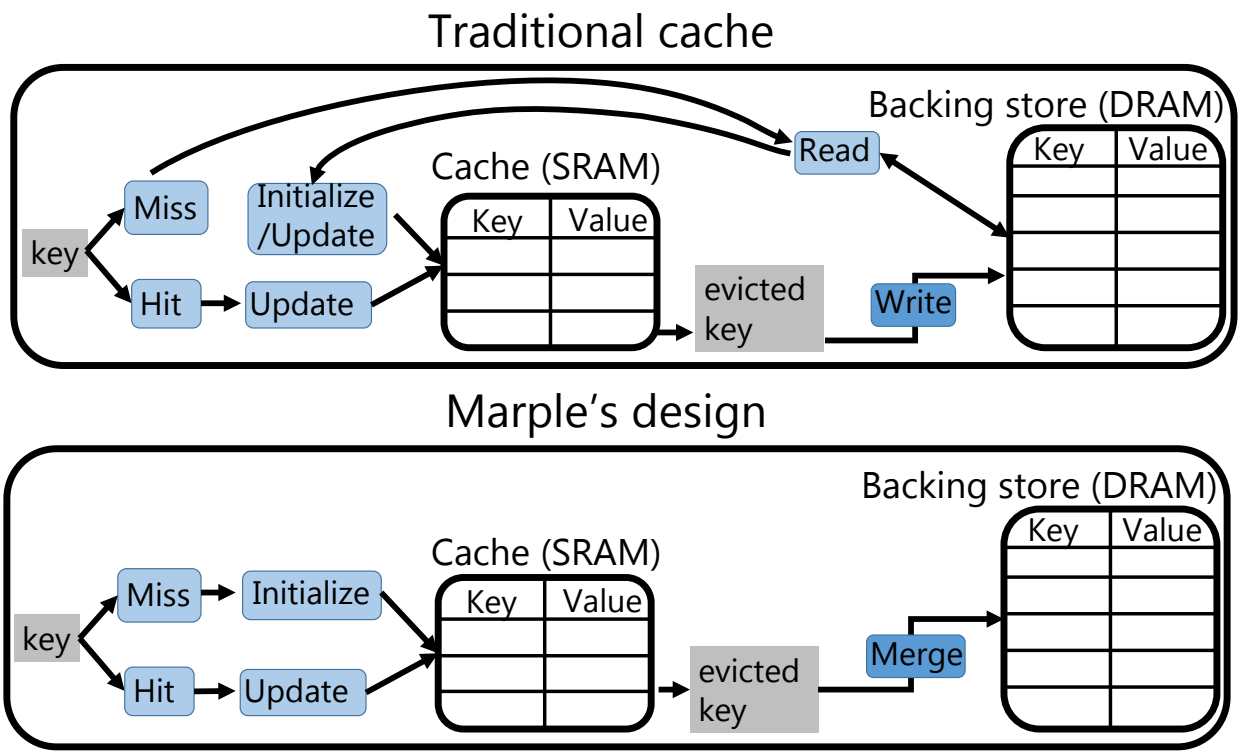


Figure 6-2: Marple's key-value store vs. a traditional cache

key-value store in hardware, where the keys represent aggregation fields (*e.g.,* the five tuple) and the values represent the state being updated by the aggregation function (*e.g.,* a packet or byte counter). To address both the fast and large requirements, our key-value store has a 'split' design. A small and fast on-chip key-value store on the router's on-chip SRAM processes packets at the router's line rate. This on-chip key-value store serves as a cache for a large and slow off-chip backing store in DRAM that can hold a large number of flows.

In traditional cache designs, cache misses require accessing off-chip DRAM with non-deterministic latencies [102] to read out the stored state. Because the aggregation operation requires us to read the value in order to update it, the entire state update operation incurs non-deterministic latencies in the process. This results in stalls in the router pipeline. Unfortunately, pipeline stalls affect the ability to provide guarantees on line-rate packet processing (10–100 Gbit/s) on all router ports.

To avoid such non-determinism, we design our key-value store to process packets at the router's line rate even on cache misses (Figure 6-2). Instead of stalling the pipeline waiting for a result from DRAM, we treat the incoming packet as the first packet from a new flow and initialize the flow's state to an initial value. Subsequent packets from the same flow are aggregated within the newly created flow entry in the key-value store, until the flow is evicted. When the flow is eventually evicted, we merge the flow's value just before eviction with its value in the backing store using a *merge function*, and write the merged result to the backing store.

In our design, the router only writes to the backing store during evictions, which is off the critical path of packet processing. Importantly, it never reads from the backing store during cache

misses, which is on the critical path of packet processing. This helps us avoid non-deterministic latencies. The backing store may be stale relative to the on-chip cache if there have been no recent evictions. We remedy this by forcing periodic evictions.

To merge a flow's new aggregated value in the cache with its old value in the backing store, the cache needs to maintain and send *auxiliary state* to the backing store during evictions. A naive usage of auxiliary state is to store relevant fields from every packet of a flow, so that the backing store can simply run the aggregation function over all the newly received packets when merging. However, in a practical implementation, the auxiliary state should be bounded in size and not grow with the number of packets in the flow.

Over the next four subsections, we describe two classes of queries that are *mergeable* with a small amount of auxiliary state (§6.2.1 and §6.2.2), discuss queries that are not mergeable (§6.2.4), and provide a general condition for mergeability that unifies the two classes of mergeable queries and separates them from non-mergeable queries (§6.2.5).

6.2.1 The associative condition

A simple class of mergeable aggregations is the class of associative functions. Suppose the aggregation function on state s is $s = op(s, f)$, where op is an associative operation and f is a packet field. Then, if op has an identity element I[3] and a flow's default value on insertion is $s_0 = I$, it is easy to show that this function can be merged using the function $op(s_{backing}, s_{cache})$, where $s_{backing}$ and s_{cache} are the value in the backing store and the value just evicted from the cache, respectively. The associative condition allows us to merge aggregation functions like addition, max, min, set union, and set intersection.

6.2.2 The linear-in-state condition

Consider the EWMA aggregation function, which maintains a moving average of queueing latencies across all packets within a flow. The aggregation function updates the EWMA s as follows:

$$s = (1 - \alpha) \cdot s + \alpha \cdot (t_{out} - t_{in})$$

We initialize s to s_0. Suppose a flow F is evicted from the on-chip cache for the first time and written to the backing store with an EWMA of $s_{backing}$.[4]

The first packet from F after F's first eviction is processed like a packet from a new flow in the on-chip cache, starting with the state s_0. Assume that N packets from F then hit the on-chip cache, resulting in the EWMA going from s_0 to s_{cache}. Then, the correct EWMA $s_{correct}$ (*i.e.*, for all packets seen up to this point) satisfies:

$$\begin{aligned} s_{correct} - (1-\alpha)^N s_{backing} &= s_{cache} - (1-\alpha)^N s_0 \\ s_{correct} &= s_{cache} + (1-\alpha)^N (s_{backing} - s_0) \end{aligned}$$

[3] An identity element I for an operation op is an element such that $op(s, I) = s \forall s$.

[4] When a flow is first evicted, it does not need to be merged.

Therefore, the correct EWMA can be obtained by: (1) having the on-chip cache store $(1-\alpha)^N$ as auxiliary state for each flow after each update, and (2) adding $(1-\alpha)^N(s_{backing} - s_0)$ to s_{cache} when merging s_{cache} with $s_{backing}$.

We can generalize this example. Let $\mathbf{p}$ be a vector with the headers and performance metadata from the last k packets of a flow, where k is an integer determined at query compile time (§6.3.3). We can merge any aggregation function with state updates of the form $\boldsymbol{S} = \boldsymbol{A}(\mathbf{p}) \cdot \boldsymbol{S} + \boldsymbol{B}(\mathbf{p})$, where $\boldsymbol{S}$ is the state, and $\boldsymbol{A}(\mathbf{p})$ and $\boldsymbol{B}(\mathbf{p})$ are functions of the last k packets. We call this condition the *linear-in-state* condition and say that $\boldsymbol{A}(\mathbf{p})$ and $\boldsymbol{B}(\mathbf{p})$ are functions of *bounded packet history.*

We say that an aggregation function is linear-in-state if, for every variable in the function, the state update satisfies the linear-in-state condition. A query is linear-in-state if all its aggregation functions are linear-in-state. Merging queries that are linear-in-state requires the router to store the first k and most recent k packets for the key since it (re)appeared in the key-value store; Appendix A describes the merge procedure for such queries in detail.

We now explain why the requirement of bounded packet history is important using real queries. Consider the TCP non-monotonic query from Table 6.2, which counts the number of packets with sequence numbers smaller than the maximum sequence number seen so far. The aggregation can be expressed as:

```
count = count + (maxseq > tcpseq) ? 1 : 0
```

While the update superficially resembles $\boldsymbol{A}(\mathbf{p}) \cdot \boldsymbol{S} + \boldsymbol{B}(\mathbf{p})$, the coefficient $\boldsymbol{B}(\mathbf{p})$ is a function of `maxseq`, the maximum sequence number so far. This could be arbitrarily far back in the stream, and hence $\boldsymbol{B}$ here is not a function of bounded packet history. Intuitively, since $\boldsymbol{B}(\mathbf{p})$ is not a function of bounded packet history, the auxiliary state required to merge `count` is large. We formalize this intuition in §6.2.5.

In contrast, the slightly modified TCP out-of-sequence query from Table 6.2 *is* linear-in-state because it can be written as:

```
count = count + (lastseq > tcpseq) ? 1 : 0
```

`lastseq`, the previous packet's sequence number, depends only on the last 2 packets: the current and the previous packet. Here, $\boldsymbol{A}(\mathbf{p})$ and $\boldsymbol{B}(\mathbf{p})$ are functions of bounded packet history, with $k = 2$. This is in contrast to `maxseq`, which depends on all packets seen so far.

6.2.3 Scalable aggregation functions

A `groupby` with no `emit()` and a linear-in-state (or associative) aggregation function can be implemented scalably without losing accuracy. Examples of such aggregations from Table 6.2 include tracking successive packets within a TCP connection that are out-of-sequence and counting the number of TCP timeouts per connection.

If a `groupby` uses an `emit()` to pass tuples to another query, it cannot be implemented scalably even if its aggregation function is linear-in-state or associative. An `emit()` outputs the current state of the aggregation function, which assumes the current state is always available in the router's on-chip cache. This is only possible if flows are never evicted, effectively shrinking the key-value store to its on-chip cache alone.

6.2.4 Handling non-scalable aggregations

While the linear-in-state and associative conditions capture several aggregation functions and enable a scalable implementation, there are two practical classes of queries that we cannot scale: (1) queries with aggregation functions that are neither associative nor linear-in-state and (2) queries where the groupby has an `emit()` statement.

An example of the first class is the TCP non-monotonic query discussed earlier. An example of the second class is the flowlet size histogram query from Table 6.2, where the first `groupby` emits flowlet sizes, which are grouped into buckets by the second `groupby`.

There are workarounds for non-scalable queries. One is to rewrite queries to remove `emit()`s. For instance, we can rewrite the loss rate query (Table 6.2) to independently record the per-flow counts for dropped packets and total number of packets in separate key-value stores. An operator can then consult both key-value stores every time they need the loss rate. Each key-value store can be scaled, but the implementation comes at a transient loss of accuracy relative to precisely tracking the loss rate after every packet using a `zip`. Second, an operator may be content with flow values that are accurate for each time period between two evictions, but not across evictions (Figure 6-9b). Third, an operator may want to run a query to collect data until the on-chip cache fills up and then stop data collection. Fourth, if the number of keys is small enough to fit in the cache (*e.g.,* if the key is an application type), the system can provide accurate results without evicting any keys.

6.2.5 A unified condition for mergeability

We now present a unified condition that separates mergeable functions from non-mergeable ones. Informally, mergeable aggregation functions are those that maintain auxiliary state linear in the size of the function's programmer-supplied state. This characterization also has the benefit of unifying the associative and linear-in-state conditions. We introduce some notation (§6.2.5.1, §6.2.5.2, and Figure 6-3) before presenting our formal results (§6.2.5.3).

6.2.5.1 Notation

A programmer supplies an aggregation function f. f takes as inputs an n-bit *state* vector and a p-bit *packet header* vector and returns an updated n-bit *state* vector. f captures incremental computations over packet streams, such as a count of packets or an EWMA of packet latencies. f can only represent incremental computations where the size of state is fixed and does not grow with the length of the packet stream. For example, f can not represent a median over the packet stream, which requires the aggregation function to maintain a log of all packets seen so far.

Given f, we want two implementation functions.

1. An incremental implementation function g, which stores and updates both the state of f and any auxiliary state required to carry out the merge. Suppose that the state used by g is $n' \geq n$ bits. $n' - n$ is the size of auxiliary state.
2. A merge implementation function m that can merge results from running g over separate packet streams.

We are interested in the question: can we merge two results from running g on two packet

$$
\begin{aligned}
f : \{0,1\}^n \times \{0,1\}^p &\to \{0,1\}^n \text{(programmer's incremental computation)} \\
g : \{0,1\}^{n'} \times \{0,1\}^p &\to \{0,1\}^{n'} \text{(implementation's incremental computation)} \\
m : \{0,1\}^{n'} \times \{0,1\}^{n'} &\to \{0,1\}^{n'} \text{(implementation's merge operation)} \\
h : \{0,1\}^{n'} &\to \{0,1\}^n \text{(transform implementation state to programmer state)} \\
s_0 &\in \{0,1\}^n \text{(programmer's start state)} \\
s'_0 &\in \{0,1\}^{n'} \text{(implementation's start state)} \\
m(s'_0, s') &= s' \ \ \forall s \in \{0,1\}^n (s'_0 \text{ is an identity for } m) \\
(g, m, h, s'_0) &\text{ merges } (f, s_0) \text{ if the 3 conditions below hold} \\
s_0 &= h(s'_0) \text{(implementation's start equals programmer's start)}
\end{aligned}
\tag{6.1}
$$

$$
f(f(f(s_0, P_1), P_2), \ldots, P_n) = h(g(g(g(s'_0, P_1), P_2), \ldots, P_n)) \; \forall n \in \mathbb{N} \;\; P_1, P_2, \ldots, P_n \in \{0,1\}^p \tag{6.2}
$$

$$
\begin{aligned}
h(g(g(g(s'_0, P_1), P_2), \ldots, P_{i+j})) = h(m(&g(g(g(s'_0, P_1), P_2), \ldots, P_i), \\
&g(g(g(s'_0, P_{i+1}), P_{i+2}), \ldots, P_{i+j}))) \\
\forall i, j \in \mathbb{N} \;\; P_1, P_2, \ldots, P_{i+j} &\in \{0,1\}^p
\end{aligned}
\tag{6.3}
$$

Figure 6-3: Formal notation for understanding mergeability

streams into one result, so that it is equivalent to running f on the concatenated stream, while keeping the size of the auxiliary state small?

We say that a merge function successfully merges an aggregation function if Equations 6.1, 6.2, and 6.3 hold. We use the notation $f(s, \{p_1, \ldots p_k\})$ to denote the composition of f over a sequence of packets.

6.2.5.2 The closure graph

We introduce the directed *closure graph* of f, denoted $G(f)$. Suppose that at some point during the aggregation, the programmer specified state is s. After running through some additional packets, that state is updated to s'. If s is known, then computing s' is straightforward: simply execute the aggregation f on all the packets seen in the interim. However, if s is *not* known, we can instead use the packets seen to compute a *function* that tells us how to get from *a symbolic* state s to the final desired state s'. That is, given a sequence of packets $\{p_k\}$, we can compute an *iterated* function $f_{iter} \in \{0,1\}^n \to \{0,1\}^n$, such that for any state s, $f_{iter}(s) = f(s, \{p_k\})$.

For an empty sequence, the corresponding iterated function is the identity function. Our goal is to store some compact representation of the iterated function in the on-chip key-value store as part of the auxiliary state. Conceptually, this iterated function is updated with each new packet that is seen, so that when a key is evicted from the on-chip key-value store, an identifier for the updated iterated function is sent to the backing store. The backing store then applies this iterated function to

the state $s_{backing}$ stored from a previous eviction to get the new value.

How does one update the iterated function stored on the router? The vertices of the closure graph of f are iterated functions f_{iter}, and we let $|G(f)|$ indicate the number of vertices. Since there are a finite number of functions $\{0,1\}^n \to \{0,1\}^n$, the size of the graph is bounded. Each vertex has an outgoing edge corresponding to every possible packet. For a given packet p, the iterated function f_i has an edge to the iterated function f_j satisfying $f(f_i(s), p) = f_j(s)$ for all s.

The closure graph thus has the following property: given a sequence of packets $\{p_1, \ldots p_k\}$, start at the identity function and, on the packet # i in the sequence, follow the edge corresponding to packet p_i. The ending vertex of this process is the iterated function that captures the effect of this packet sequence on any starting state. We say that this ending vertex is the result of *updating the iterated function* by the given sequence of packets. Conceptually, to update the iterated function stored on the router given a new packet, we simply follow the edge labeled with that packet.

The size of the closure graph (*i.e.*, the number of vertices) indicates the number of bits of auxiliary state that are needed for a merge. For efficiently mergeable functions, the size of the closure graph is small. This is because the router needs to tell the backing store what update to perform on the value it currently has. Each iterated function in the closure graph is a possible update, so the router needs to convey at least $\log |G(f)|$ auxiliary bits for the merge to be possible.

As an example, consider a simple counter, where $f(s, p) = s + 1$ and $s_0 = 0$. If the counter has n bits, the closure graph is a ring of size 2^n, where vertex i represents the iterated function $f_{iter}(s) = s + i$. Note that all edges from vertex i point to vertex $i + 1$. For every observed packet, the router walks one step further around the ring. This is implemented efficiently via adding 1 to an n-bit counter. After K packets, the iterated function on the router is $s + k$, where $k = K \mod 2^n$. Upon merging, this is sent to the backing store, which then knows to add k to the existing backing store value.

6.2.5.3 Proofs of theorems

Theorem 6.2.1. *A merge function exists to successfully merge an aggregation function f, provided it can use up to $n2^n$ auxiliary bits.*

Proof. We can represent each iterated function f_{iter} in $n2^n$ bits, using a list of 2^n $n-$bit numbers. This list enumerates the output of the iterated function on every possible input, *i.e.*, the kth item in the list is $f_{iter}(k)$. The router stores this representation. For each new packet p, the router computes the update to each item k in the list: $f_{iter}(k) \to f(f_{iter}(k), p)$. Upon a merge, the router sends this representation to the backing store, which can then evaluate $f_{iter}(s_{backing})$, where $s_{backing}$ is the value stored in the backing store. □

In general, requiring $n2^n$ bits is typically infeasible for even moderate values of n. This is where the linear-in-state and associative conditions come in. We show that either of these conditions requires only $O(n)$ auxiliary bits of space.

Theorem 6.2.2. *Aggregation functions that are either linear-in-state or associative have a merge function that uses only $O(n)$ auxiliary bits.*

Proof. The associative condition is trivial: by definition, no auxiliary bits are required because the merge function simply applies the original aggregation function on the values in the cache and the backing store. Hence, we focus on the linear-in-state condition. Suppose f is an aggregation function that is linear-in-state, *i.e.*, $f(\mathbf{S}, p) = \mathbf{A}(p) \cdot \mathbf{S} + \mathbf{B}(p)$, where $\mathbf{S}$ is a state vector, and $\mathbf{A}, \mathbf{B}$ are matrices depending only on the incoming packet. Then:

$$f(\mathbf{S}_0, \{p_1, \ldots, p_k\}) = \mathbf{A}(p_k) \ldots \mathbf{A}(p_1) \cdot \mathbf{S}_0 + \mathbf{C}(p_1, p_2, \ldots, p_k) \equiv \mathbf{A}' \cdot \mathbf{S}_0 + \mathbf{C}(p_1, \ldots, p_k)$$

Here, $\mathbf{C}$ is some function of only the packets p_1 through p_k. $\mathbf{A}'$ is a composition of the $\mathbf{A}$ matrices for the observed packets. The router keeps track of $\mathbf{A}'$ and $\mathbf{C}$, which requires $O(nd^2)$ space, where d is the dimension of the state vector, and each dimension requires n bits of storage. These matrices are sent to the backing store upon eviction, at which point the merge operation computes the new backing store value:

$$\mathbf{S}_d \leftarrow \mathbf{C}(p_1, \ldots, p_k) + \mathbf{A}' \cdot \mathbf{S}_d$$

It's easy to verify that this is the proper merge procedure. □

However, there are some functions that are not mergeable with so few auxiliary bits. One example is the TCP-non-monotonic aggregation function:

```
def nonmt(maxseq, count, tcpseq):
  if maxseq > tcpseq:
    count = count + 1
  else:
    maxseq = tcpseq
nm_q = groupby(pktstream, 5tuple, nonmt);
```

Theorem 6.2.3. *The TCP-non-monotonic function requires at least $n2^n$ auxiliary bits to merge.*

Proof. Two pieces of state must be stored: the max sequence number and the count. For simplicity, we assume each piece of state requires n bits. We present a family of packet sequences, such that there is an injection from the packet sequences to the vertices in the closure graph of f. In other words, starting from the identity, updating the identity iterated function by each sequence in this family will result in a distinct iterated function.

Consider a family of packet sequences parameterized by a tuple $(a_0, a_1, \ldots a_{2^n-1})$: the sequence consists of a_i packets with sequence number i, in increasing order of i. For each such tuple, performing an update U of the identity iterated function by the sequence for that tuple results in the following final iterated function:

$$U(a_0, \ldots, a_{2^n-1}) = f_{iter} \qquad \text{such that} \qquad f_{iter}(x, y) = \left(2^n - 1, (y + \sum_{i=0}^{x-1} a_i) \mod 2^n\right)$$

where x and y are the max sequence number and count, respectively. To see why this is the case, recall that the iterated function is taking some *previous* max seq. number and count and trying to

update it by the packet sequence. If the max sequence number was $x = k$ before, then the count will increase by the number of packets with sequence number $< k$, which is $\sum_0^{k-1} a_i$.

Lemma 6.2.3.1. *U is an injection.*

Proof. If $(a_0, \ldots, a_{2^n-1}) \neq (a'_0, \ldots, a'_{2^n-1})$, then there is some k (including $k = 0$) such that $a_k \neq a'_k$ but $a_j = a'_j$ for all $j < k$. Let $U(a_0, \ldots, a_{2^n-1}) = f_{iter}$ and $U(a'_0, \ldots, a'_{2^n-1}) = f'_{iter}$. Then, for any y:

$$f_{iter}(k+1, y) - f'_{iter}(k+1, y) = \left(0, \sum_{i=0}^{k} a_i - \sum_{i=0}^{k} a'_i\right) = (0, a_k - a'_k) \neq (0, 0)$$

Hence, $f_{iter} \neq f'_{iter}$, making U an injection. □

There are 2^{n2^n} such packet sequences because each packet sequence is represented by a 2^n element tuple, where each element is n bits. This means that the closure graph has at least 2^{n2^n} distinct vertices, and thus the number of auxiliary bits needed is at least $\log 2^{n2^n} = n2^n$. □

Now we turn to the question: given an aggregation function, how easy is it to compute a merge function that uses the smallest number of auxiliary bits? Since we know that the smallest number of bits is $\log |G(f)|$, we can construct $G(f)$. Constructing $G(f)$ is extremely inefficient, but is possible.

Theorem 6.2.4. *Given an aggregation function f, we can demonstrate a algorithm to compute a merge function using the smallest number of auxiliary bits.*

Proof. To construct the closure graph, we start with a single node representing the identity function. Starting at this node f_0, we enumerate all p-bit values for a single packet P, and compute the updated iterated function f_P satisfying $f_P(s) = f(f_0(s), P)$ for all s. If this iterated function has not been seen yet, we create a new node for it. We join two nodes by an edge labeled with the packet value that causes the transition between those two nodes and repeat this process from each newly created node until all edges out of a node lead back to existing nodes[5].

There are 2^{n2^n} possible iterated functions. We assume that given an iterated function, it is possible to find that function in the existing partial closure graph in time $O(n2^n)$, the number of bits needed to represent the number of iterated functions in the worst case. Then, since we must enumerate every possible p-bit packet the total runtime is:

$$O(\text{\# iters} \cdot \text{\# packets} \cdot \text{\# time to test if function has been seen}) = O\left(2^{n2^n} \cdot 2^p \cdot n2^n\right)$$

□

This algorithm is hopeless in practice. However, a polynomial time algorithm is unlikely. We demonstrate a hardness result by considering a decision version of this problem: given an

[5]We need a way to check node equality, *i.e.*, function equality. We do this by a brute force check on all inputs to both nodes/functions.

aggregation function f and merge function m, does m successfully merge f for all possible packet inputs? This problem is probably simpler than finding a merge function with the smallest number of auxiliary bits because it supplies a candidate merge function m as an input, instead of searching for the best merge function. Yet, even this problem turns out to be co-NP-hard.

Before we launch into the hardness proof, we must limit the scope of the problem. "All possible packet inputs" includes packet sequences of arbitrary length. However, we can restrict ourselves to sequences up to 2^n packets.

Theorem 6.2.5. *To determine whether m merges f on every packet sequence, it is sufficient to consider packet sequences up to length $2^{max(n,n')}$.*

Proof. We show it is sufficient to check equation 6.2 on packet streams of length $L <= M = 2^{max(n,n')}$. This check fails only if there is a stream P_{false} of length $L > M$ that falsifies equation 6.2. However, there must then also be a smaller stream P_{small} of length $L' <= M$ that falsifies equation 6.2.

To see why, let's say packet A within P_{false} first falsifies equation 6.2. If $A's$ position within P_{false} is less than or equal to M, the substream up to and including A is P_{small}. If not, running f repeatedly on the substream of packets until A will result in some state value s being repeated twice, after seeing (say) packets P_1 and P_2. We can remove all packets after P_1 and before P_2, creating a shorter stream that still falsifies 6.2. This process can be repeated until the stream length is less than or equal to M. This will eventually happen because there will always be a repeated state value by the pigeon-hole principle because the number of packets in a sequence of length M is greater than the number of distinct states. □

Now we know that checking whether m successfully merges f is decidable because we can check packet sequences up to length M, instead of the infinite set of all possible packet sequences. However, it is intractable unless $P = NP$.

Theorem 6.2.6. *Given a merge function m and aggregation function f, verifying that m successfully merges f is co-NP hard.*[6]

Proof. Given (f, s_0) and (g, m, h, s'_0), we define the decision problem $MERGE(f, s_0, g, m, h, s'_0)$, which outputs whether (g, m, h, s'_0) merges (f, s_0). Recall that (g, m, h, s'_0) merges (f, s_0) if Equations 6.1, 6.2, and 6.3 are satisfied. We'll show that $MERGE$ is co-NP-hard by reducing $TAUTOLOGY$ (a known co-NP-complete decision problem) to $MERGE$.

An instance of $TAUTOLOGY$ is a boolean function t. The output is 1 when t is a tautology, 0 otherwise. $TAUTOLOGY(t)$ can be reduced to[7] $MERGE(s \oplus t', 0, s \oplus t', OR, I, 0)$,[8]where

1. $s \in \{0, 1\}$
2. $t : \{0, 1\}^p \rightarrow \{0, 1\}$
3. OR is the boolean OR function on two bools.
4. I is the identity function.

[6]co-NP is made up of problems that are complements of problems in NP.

[7]This is a polynomial many-to-one reduction or a Karp reduction.

[8]t'(P) = (NOT t(P))

First, we observe that $MERGE(s \oplus t', 0, s \oplus t', OR, I, 0)$ decides whether statement 6.4 is true:

$$t'(P_1) \oplus t'(P_2) \oplus \ldots t'(P_{i+j}) = (t'(P_1) \oplus t'(P_2) \ldots t'(P_i)) OR (t'(P_{i+1}) \oplus t'(P_{i+2}) \ldots t'(P_{i+j})) \tag{6.4}$$

$$\forall i, j \in \mathbb{N} \;\; P_1, P_2, \ldots, P_{i+j} \in \{0,1\}^p$$

Next, we prove a lemma that shows the reduction from $TAUTOLOGY$ to $MERGE$.

Lemma 6.2.6.1. *Statement 6.4 is true iff* $t'(P) = 0 \; \forall P$.

Proof. Let's suppose $t'(P) = 0 \; \forall P$. It is easy to see that statement 6.4 reduces to 0 on both the LHS and RHS. If on the other hand, $t'(P^*) = 1$ for some P^*. Then, we set:

1. $i = 1$
2. $j = 2k - 1$
3. $P_1 = P_2 = \cdots = P_{i+j} = P^*$

The LHS is an XOR over an even number of 1s, which is 0. The RHS is an OR of 1 and something else, which is 1. So, we have found one setting of the quantified variables that falsifies statement 6.4. □

It is straightforward to see how $TAUTOLOGY(t(P))$ reduces to $MERGE(s \oplus t', 0, s \oplus t', OR, I, 0)$, showing that $MERGE$ is co-NP-hard. □

The practical implication of this result is that there is unlikely to be a general and efficient procedure to find a merge function that can merge an arbitrary aggregation function using the smallest amount of auxiliary state. Thus, identifying specific classes of functions (*e.g.,* linear-in-state and associative) and checking if an aggregation function belongs to these classes is probably the best that we can hope to do.

6.2.6 Related work on distributed aggregations

Our linear-in-state characterization is related to *distributed aggregation* in large-scale data analytics [181, 211]. The goal of distributed aggregation is to compute an aggregate (*e.g.,* count) of a list of items in a distributed fashion, instead of processing the entire list serially. In distributed aggregation, different portions of the list are aggregated independently and the partial results are combined together, potentially improving performance relative to serial aggregation.

The connection between distributed aggregation and the linear-in-state characterization is as follows: the merge procedure can be seen as combining the results of two independent aggregations on two halves of a list. Further, this merge procedure can be performed recursively by computing the aggregation on each half of the list in a distributed fashion, *i.e.*, using a merge function that combines results from partial aggregations on quarters of the list.

The simplest class of aggregations that permit efficient distributed aggregation are those that are both commutative and associative [211]. The associative condition is consistent with the associative

class of merge functions above. In addition, the commutative condition is required for distributed aggregation because there is no guarantee on the order in which the two halves of the list will be processed, unlike our merge procedure, which still requires the packet sequence to be processed serially.

The SYMPLE system [181] extends distributed aggregation to include functions that are neither commutative nor associative. SYMPLE works by dividing the list to be aggregated into multiple segments. The aggregation is computed starting from the initial state on the first segment. For the second and subsequent segments, the initial state at the start of the segment is the aggregated state after aggregating the list up to the previous segment. Hence, the segment's initial state is unknown until the previous segment is aggregated.

To break this dependency on the previous segment, SYMPLE computes the aggregations on the second and subsequent lists *symbolically*. The result of this is a concrete aggregated state value for the first segment, and a set of symbolic expressions/functions for the remaining segments. These symbolic expressions/functions are conceptually identical to the iterated functions used in our proof constructs. They map an unknown and symbolic initial state at the beginning of the segment to the final state at the end of the segment, given the concrete list values seen in that segment.

To facilitate symbolic computation, SYMPLE restricts the set of operations that can be carried out in distributed aggregation. For instance, SYMPLE allows a state variable to be multiplied by a constant, but not another state variable. SYMPLE also does not allow division operations on state variables. We observe that these restrictions on operations on state variables are consistent with the restrictions on state variables in a linear-in-state aggregation function. However, we leave a precise characterization of the similarities and differences between the expressiveness of SYMPLE and linear-in-state aggregations to future work.

6.2.7 Hardware feasibility

We optimize our stateful hardware design for linear-in-state queries and break it down into five components. Each component is well-known; our main contribution is putting them together to implement stateful queries. One important design choice for a router designer is how much memory to provision for the on-chip cache, which we evaluate in §6.4. We now discuss each component in detail.

The on-chip cache is a hash table where each row in the hash table stores keys and values for a certain number of flows. If a packet from a new flow hashes into a row that is full, the least recently used flow within that row is evicted. Each row has 8 flows and each flow stores both its key and value.[9] Our choice of 8 flows is based on 8-way L1 caches, which are very common in processors [28]. This cache eviction policy is close to an ideal but impractical policy that evicts the least recently used (LRU) flow across the whole table (§6.4).

Within a router pipeline stage, the on-chip cache has a logical interface similar to an on-chip hash table used for counters: each packet matches entries in the table using a key extracted from the packet header, and the corresponding action (*i.e.*, increment) is executed by the router. An on-chip

[9]The LRU policy is actually implemented across 3-bit pointers that point to the keys and values in a separate memory. So we shuffle only the 3-bit pointers to implement the LRU policy, not the entire key and value.

hash table may be used as a path to incrementally deploying a router cache for specific aggregations (*e.g.,* increments), on the way to supporting more general actions and cache eviction logic in the future.

The off-chip backing store is a scale-out key-value store such as Redis [43] running on dedicated collection servers within the network. As §6.4 shows, the number of measurement servers required to support typical eviction rates from the router's on-chip cache is small, even for a 64×100-Gbit/s router.

Maintaining packet history. Before a packet reaches the pipeline stage with the on-chip cache, we use the preceding stages to precompute $\boldsymbol{A}(\mathbf{p})$ and $\boldsymbol{B}(\mathbf{p})$ (the functions of bounded packet history) in the state-update operation $\boldsymbol{S} = \boldsymbol{A}(\mathbf{p}) \cdot \boldsymbol{S} + \boldsymbol{B}(\mathbf{p})$. Our current design only handles the case where $\boldsymbol{S}$, $\boldsymbol{A}(\mathbf{p})$, and $\boldsymbol{B}(\mathbf{p})$ are scalars.

Say $\boldsymbol{A}(\mathbf{p})$ and $\boldsymbol{B}(\mathbf{p})$ depend on packet fields from the last k packets. Then, these preceding pipeline stages act like a shift register and store fields from the last k packets. Each stage contains a read/write register, which is read by a packet arriving at that stage, carried by the packet as a header, and written into the next stage's register. Once values from the last k packets have been read into packet fields, $\boldsymbol{A}(\mathbf{p})$ and $\boldsymbol{B}(\mathbf{p})$ can be computed with stateless instructions provided by programmable router architectures [86, 188].

Carrying out the linear-in-state operation. Once $\boldsymbol{A}(\mathbf{p})$ and $\boldsymbol{B}(\mathbf{p})$ are known, we use a multiply-accumulate (MAC) instruction [32] to compute $\boldsymbol{A}(\mathbf{p}) \cdot \boldsymbol{S} + \boldsymbol{B}(\mathbf{p})$. This instruction is very cheap to implement: our circuit synthesis experiments show that a MAC instruction meets timing at 1 GHz and occupies about 2000 $\mu\mathrm{m}^2$ in a recent 32 nm transistor library. A router chip with an area of a few hundred mm^2 [116] can easily support a few hundred MAC instructions.

Queries that are not linear-in-state. We use the set of stateful instructions developed in Domino (Table 4.4) for queries that are not linear-in-state. Our evaluations show that these instructions are sufficient for our example queries that are not linear-in-state (Table 6.2).

6.3 Query compiler

We compile Marple queries to two targets: the P4 behavioral model [37], configured by emitting P4 code [36], and the Banzai machine model, configured by emitting Domino code (Chapter 4). In both cases, the emitted code configures a router pipeline, where each stage is a match-action table or our key-value store. Our compiler does not consider the problem of reconfiguring the router pipeline on the fly as queries change.

A preliminary pass of the compiler over the input query converts the query to an abstract syntax tree (AST) of functional operators (Figure 6-5a). The compiler then:

1. produces router-local ASTs from a global AST (§6.3.1);
2. produces P4 and Domino pipeline configurations from router-local ASTs (§6.3.2); and
3. recognizes linear-in-state aggregation functions and sets up auxiliary state required to merge such functions for a scalable implementation (§6.3.3). To scalably implement associative aggregation functions (§6.2.1), we use the programmer annotation `assoc` to determine if an aggregation is associative.

We use the query shown in Figure 6-4 as a running example to illustrate the details in the

```
def oos_count([count, lastseq], [tcpseq, payload_len]):
  if lastseq != tcpseq:
    count = count + 1
    emit()
  lastseq = tcpseq + payload_len

tcps    = filter(pktstream, proto == TCP
                 and (router == S1 or router == S2));
tslots  = map(pktstream, [tin/epoch_size], [epoch]);
joined  = zip(tcps, tslots);
oos     = groupby(joined,
                  [5tuple, router, epoch],
                  oos_count);
```

Figure 6-4: Running example for Marple compiler (§6.3).

compiler. The query counts the number of out-of-sequence TCP packets over each time epoch, measured independently at two routers `S1` and `S2` in the network.

6.3.1 Network-wide to router-local queries

The compiler partitions a network-wide query written over all packets at all queues in the network (§6.1) into router-local queries, which are then transformed into router-specific configurations. We achieve this in two steps. First, we determine the *stream location*, *i.e.*, the set of routers that contribute tuples to a stream, for the final output stream of query. For instance, the output stream of a query that filters by router id s has a stream location equal to the singleton set s. Second, we determine how to partition queries with aggregation functions written over the entire network into router-local queries.

Determining stream location for the final output stream. To start with, the stream location of `pktstream` is the set of all routers in the network. The stream location of the output of a `filter` is the set of routers implied by the filter's predicate. We evaluate the set of routers contributing tuples to the output of a `filter` operation through basic syntactic checks of the form `router == X` on the `filter` predicate. We combine router sets for boolean combinators (`or` and `and`) inside filter predicates using set union and intersection respectively. The stream location of the output of a `zip` operator is the intersection of the stream locations of the two inputs. Stream locations are unchanged by the `map` and `groupby` operators.

The stream locations for the running example are shown in Figure 6-5b. The stream location of `pktstream` is the set of all network routers, but is restricted to just `S1` and `S2` by the `filter` in the query (left branch). This location is then propagated to the root of the AST through the `zip` operator in the query.

Partitioning network-wide aggregations. As described in §6.1, we only permit aggregations that satisfy one of three conditions: (1) they operate independently on each router, (2) operate

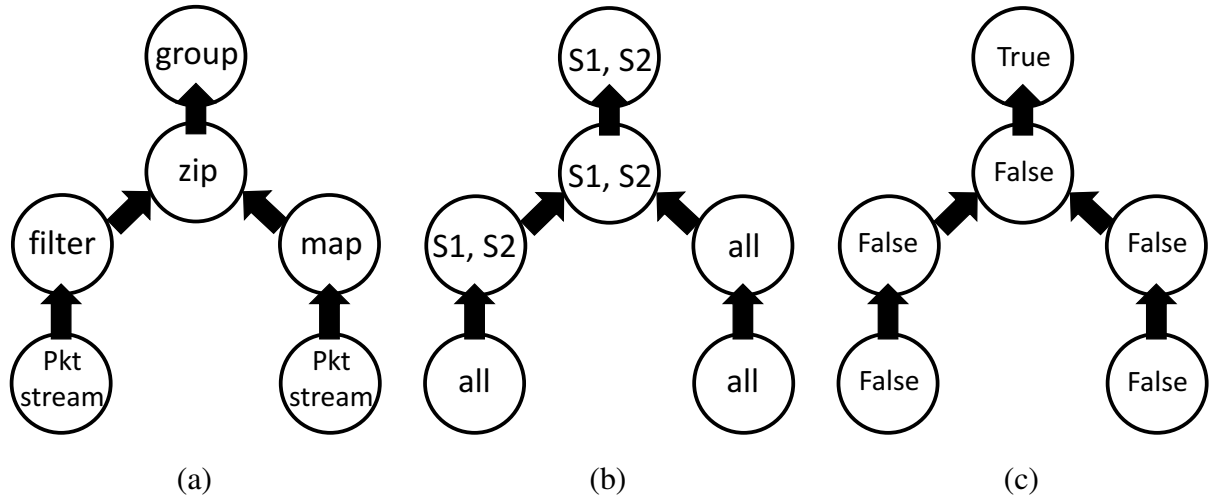


Figure 6-5: Abstract Syntax Tree (AST) manipulations for the running example (§6.3). (a) Operator AST. (b) Stream location (set of routers). (c) Stream router-partitioned (boolean).

independently on each packet, or (3) are associative and commutative. We describe below how we check the first condition, failing which we check the last two conditions syntactically: either the `groupby` aggregates by `uid` (condition 2) or contains programmer annotations `assoc` and `comm` (condition 3).

To check if an aggregation operates independently on each router, we label each AST node with a boolean attribute, *router-partitioned*, corresponding to whether the output stream has been partitioned by the router at which it appears. Intuitively, if a stream is router-partitioned, we allow packet-order-dependent aggregations over multiple packets of that stream; otherwise, we do not.

Determining and propagating *router-partitioned* through an AST is straightforward. The base `pktstream` is not router-partitioned. The `filter` and `zip` operators produce a router-partitioned stream if their output only appears at a single router. The `groupby` produces a router-partitioned stream if it aggregates by `router`. In all other cases, the operators retain the operands' router-partitioned attribute.

The router-partitioned attributes for our running example are shown in Figure 6-5c. The `filter` produces output streams at two routers, hence is not router-partitioned. The `groupby` aggregates by `router` and hence is router-partitioned.

After the partitioning checks have succeeded, we are left with a set of independent router-local ASTs corresponding to each router location that the AST root operator appears in, *i.e.*, `S1, S2`.

6.3.2 Query AST to pipeline configuration

This compiler pass first generates a sequence of operators from the router-local query AST of §6.3.1. This sequence of operators is then used in the same order to generate a router pipeline configuration. There are two aspects that require care when constructing a pipeline structure: (1) the pipeline

should respect read-write dependencies between different streams, and (2) repeated subqueries should not be re-executed by creating additional pipeline stages.

We generate a sequence of operators through a post-order traversal of the query AST, which guarantees that the operands of a node are added into the pipeline before the operator in the node, thereby respecting read-write dependencies. Further, we deduplicate subquery results from the pipeline to avoid repeating stages in the final output. For the running example, the algorithm produces the sequence of operators: `tcps` (`filter`) $\rightarrow$ `tslots` (`map`) $\rightarrow$ `joined` (`zip`) $\rightarrow$ `oos` (`groupby`).

Next, the compiler emits P4 code for a router pipeline from the operator sequence. The `filter` and `zip` configuration just involves checking a predicate and setting a "valid" bit on the packet metadata. The `map` configuration assigns a packet metadata field to the computed expression. The `groupby` configuration uses a P4 register that is indexed by the aggregation fields. The state located at a particular register index is updated through a P4 action representing the aggregation function.

To target the Banzai machine model (§4.1), the Marple compiler emits imperative code fragments for each pipeline stage and then concatenates these fragments together into a single Domino program. The Domino compiler then takes this program and compiles it to a pipeline of Banzai atoms.

6.3.3 Handling linear-in-state aggregations

We now consider the problems of detecting if an aggregation function is linear-in-state and setting up auxiliary state for such linear-in-state aggregation functions (Figure 6-6). Recall that an aggregation function is linear-in-state if the updates to all state variables within the aggregation function can be written as $S = \boldsymbol{A}(\mathbf{p}) \cdot S + \boldsymbol{B}(\mathbf{p})$). A general solution to this problem is challenging because the aggregation function can take varied forms. For instance, the assignment $S = \frac{S^2-1}{S-1}$ is linear-in-state, but detecting that it is linear-in-state needs the compiler to perform algebraic simplifications.

We take a pragmatic approach and sacrifice completeness, but still cover useful functions. Specifically, we only detect linear-in-state state updates through simple syntactic pattern matching in the compiler (*i.e.*, without any algebraic transformations). Despite these simplifications, the Marple compiler correctly identifies all the linear-in-state aggregations in Table 6.2 and targets the multiply-accumulate atom that we added to the Banzai pipeline.

To describe how linear-in-state detection works, we introduce some terminology. Recall that an aggregation function takes two arguments (§6.1): a list of state variables (*e.g.,* a counter) and a list of tuple fields (*e.g.,* the TCP sequence number). We use the term variable in this subsection to refer to either a state variable or a tuple field. These are the only variables that can appear within the body of the aggregation function.[10]

We carry out a three-step procedure, summarized in Figure 6-6. First, for each variable in an aggregation function, we assign a *history*. This history tells us how many previous packets we need to look at to determine a variable's value accurately (history = 1 means the current packet). For instance, for the value of a byte counter, we need to look back to the beginning of the packet stream (history = ∞), while for a variable that tracks the last TCP sequence number we need to only look

[10]Marple supports local variables within the function body, but the more general algorithm is not materially different from the simpler version we present here.

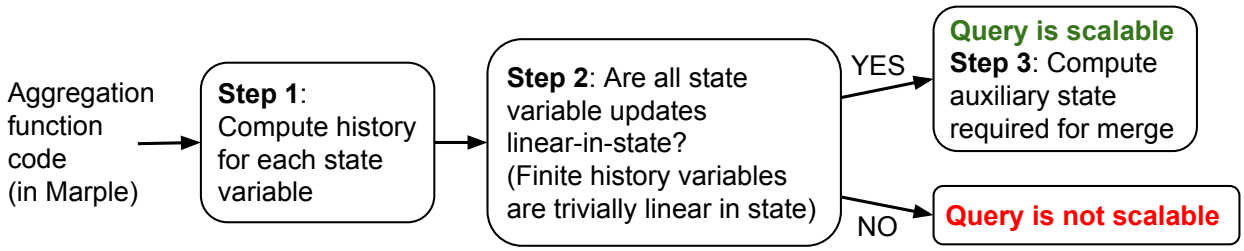


Figure 6-6: Steps for compiling linear-in-state updates.

back to the previous packet (history = 2). Consistent with the definition of history, constants are assigned a history of 0, and variables in the tuple field list are assigned a history of 1. For state variables, we use Algorithm 1 to determine each variable's history.

Second, once each variable has a history, we look at the history of each state variable s. If the history of s is a finite number k, then s only depends on the last k packets. In this case, the state update for s is trivially linear-in-state, by setting A to 0 and B to the aggregation function itself.[11] If s has an infinite history, we use syntactic pattern matching to check if the update to s is linear-in-state.

Third, if all state variables have linear-in-state state updates, the aggregation function is linear-in-state. For such linear-in-state aggregations, we generate the auxiliary state that permits merging of the aggregation function (§6.2). If not, we use the set of stateful atoms developed in Domino (Table 4.4) to implement the aggregation function. We now describe each of the three steps in detail.

Determining history of state variables. We now describe Algorithm 1 that assigns a history to every state variable. To motivate this algorithm, observe that if all assignments to a state variable only use variables that have a finite history, then the state variable itself has a finite history. For instance, in Figure 6-4, right after it is assigned, `lastseq` has a history of 1 because it only depends on the current packet's fields `tcpseq` and `payload_len`. To handle branching in the code, *i.e.*, `if (predicate) {...}` statements, we generalize this observation. A state variable has finite history if (1) it has finite history in all its assignments in all branches of the program, and (2) each branching condition `predicate` itself only depends on variables with a finite history.

Concretely, COMPUTEHISTORY (line 2) assigns each variable a history corresponding to an upper bound on the number of past packets that the state variable depends on. We track the history separately for each branching context, *i.e.*, the sequence of branches enclosing any statement.[12] The algorithm starts with a default large pessimistic history (*i.e.*, an approximation to ∞) for each state variable (line 1), and performs a fixed-point computation (lines 3–20), repeatedly iterating over the statements in the aggregation function (line 7–16).

For each assignment to a state variable in the aggregation function, the algorithm updates the history of that state variable in the current branching context (lines 7–9). For each branch in the aggregation function, the algorithm maintains a new branching context and a history for the

[11] More precisely, the parts of the aggregation function that update s.

[12] Currently, Marple forbids multiple `if ... else` statements at the same nesting level; hence, the enclosing branches uniquely identify a code path through the function. This restriction is not fundamental; the more general form can be transformed into this form.

branching context itself (lines 10–14). At the end of each iteration, the algorithm increments each variable's history to denote that the variable is one packet older (line 18). The algorithm returns a conservative history k for each state variable, including possibly max_bound (line 1) to reflect an infinite history.

Algorithm 1 Determining history of all state variables

```
1:  hist = {state = {true: max_bound}}                    ▷ Init. hist. for all state vars.
2:  function COMPUTEHISTORY(fun)
3:      while hist is still changing do                    ▷ Run to fixed point.
4:          hist ← {}
5:          ctx ← true                                     ▷ Set up outermost context.
6:          ctxHist ← 0                                    ▷ History value of ctx.
7:          for stmt in fun do
8:              if stmt == state = expr then
9:                  hist[state][ctx] ← GETHIST(ctx, expr, ctxHist)
10:             else if stmt == if predicate then
11:                 save context info (restore on branch exit)
12:                 newCtx ← ctx and predicate
13:                 ctxHist ← GETHIST(ctx, newCtx, ctxHist)
14:                 ctx ← newCtx
15:             end if
16:         end for
17:         for ctx, var in hist do                        ▷ Make history one pkt older.
18:             hist[var][ctx] ← min(hist[var][ctx] + 1, max_bound)
19:         end for
20:     end while
21: end function
22: function GETHIST(ctx, ast, ctxHist)
23:     for xi ∈ LEAFNODES(ast) do
24:         hi = hist[xi][ctx]
25:     end for
26:     return max(h1, ... , hn, ctxHist)
27: end function
```

Determining if a state variable's update is linear-in-state. For each state variable S with an infinite history, we check whether the state updates are linear-in-state as follows: (1) each update to S is syntactically affine, *i.e.*, $S \leftarrow A \cdot S + B$ with either A or B possibly zero; and (2) A, B and every branch predicate depend on variables with a finite history. This approach is sound, but incomplete: it misses linear-in-state updates such as $S = \frac{S^2-1}{S-1}$.

Determining auxiliary state. For each state variable with a linear-in-state update, we initialize four pieces of auxiliary state for a newly inserted key:[13] (1) a running product $S_A = 1$; (2) a packet

[13]This can happen either when a key first appears or reappears following an eviction.

counter $c = 0$; (3) an entry log, consisting of relevant fields from the first k packets following insertion; and (4) an exit log, consisting of relevant fields from the last k packets seen so far. After the counter c crosses the packet history bound k, we update S_A to $A \cdot S_A$ each time S is updated. This update to S_A can also be implemented using the same multiply-accumulate atom. When the key is evicted, we send S_A along with the entry and exit logs to the backing store for merging (see Appendix A).

6.4 Evaluation

We evaluate Marple in three ways. In §6.4.1, we quantify the number of Banzai atoms required for Marple queries. In §6.4.2, we quantify the memory size vs. eviction rate tradeoff for the key-value store. In §6.4.3 and §6.4.4, we describe two case studies that use Marple compiled to the P4 behavioral model running on Mininet: debugging microbursts [130] and computing flowlet size distributions.

6.4.1 Hardware compute resources

Table 6.2 shows several Marple queries. Next to each query, we show (1) whether all its aggregations are linear-in-state, (2) whether it can be scaled by merging correctly with a backing store, and (3) the router resources required, measured through the pipeline depth (number of stages), pipeline width (maximum number of parallel computations per stage), and the number of Banzai atoms (total number of computations).

Table 6.2 shows that many useful queries contain only linear-in-state aggregations, and many of them can be implemented scalably (§6.2.2). Notably, the flowlet size histogram and lossy connection queries are not scalable despite being linear-in-state, since they contain `emit()` statements. In §6.2.4, we showed how to rewrite some of these queries (*e.g.,* lossy connections) to scale, at the cost of losing some accuracy.

We compute the pipeline's depth and width by compiling each query to Banzai using the Domino compiler. When compiling each query, Banzai is supplied with a single stateless atom type, which perform binary operations (arithmetic, logic, and relational) on pairs of packet fields, and a stateful atom type depending on the type of the query. For the linear-in-state queries, we use the multiply-accumulate atom as the stateful atom, while for the other operations, we use the NestedIf atom (Table 4.4). As expected, all the linear-in-state queries compile to a pipeline with the multiply-accumulate atom; for all the queries that are not linear-in-state, the NestedIf atom turns out to be sufficiently expressive.

The computational resources required for Marple queries are modest. All queries in Table 6.2 require a pipeline shorter than 11 stages. This is feasible, *e.g.,* the RMT architecture offers 32 stages [86]. Further, functionality other than measurement can run in parallel in each stage because the number of atoms required per stage is at most 6, while programmable routers provide a few 100 parallel instructions per stage (*e.g.,* RMT provides 224 [86]).

Example	Query code	Description	Linear in state?	Scales?	Pipe depth	Pipe width	# of atoms
Packet counts	`def count([cnt], []): cnt = cnt + 1; emit()` `result = groupby(pktstream, [srcip], count);`	Count packets per source IP.	Yes	Yes	5	2	7

EWMA over latencies	`def ewma([avg], [tin, tout]):`
	`  avg = (1-alpha)*avg + (alpha)*(tout-tin)`
	`ewma_q = groupby(pktstream, [5tuple], ewma);`
		Maintain a moving EWMA over packet latencies per flow.	Yes	Yes	6	4	11

TCP out-of-sequence	`def oos([lastseq, cnt], [tcpseq, payload_len]):`
	`  if lastseq != tcpseq: cnt = cnt + 1`
	`  lastseq = tcpseq + payload_len`
	`oos_q = groupby(pktstream, [5tuple], oos);`
		Count the number of packets per connection arriving with a sequence number that is non-consecutive with the last packet.	Yes	Yes	7	4	14

TCP non-monotonic	`def nonmt([maxseq, cnt], [tcpseq]):`
	`  if maxseq > tcpseq: cnt = cnt + 1`
	`  else: maxseq = tcpseq`
	`nm_q = groupby(pktstream, [5tuple], nonmt);`
		Count the number of packets per connection with sequence numbers lower than the maximum so far.	No	No	5	2	6

Flowlet size histogram	`def fl_detect([last_time, size], [tin]):`
	`  if tin - last_time > delta:`
	`  emit(); size = 1`
	`  else: size = size + 1`
	`  last_time = tin`
	`R1 = groupby(pktstream, [5tuple], fl_detect);`
	`fl_hist = groupby(R1, [size], count);`
		Compute a histogram over the lengths of flowlets. This statistic is useful to evaluate network load balancing schemes, *e.g.,* [65].	Yes	No	11	6	31

High E2E latency	`def sum_lat([e2e_lat], [tin, tout]):`
	`  e2e_lat = e2e_lat + tout - tin`
	`e2e = groupby(pktstream, [uid], sum_lat);`
	`high_e2e = filter(e2e, e2e_lat > 10);`
		Capture packets experiencing high end-to-end queueing latency, by adding time spent in the queue at each hop.	Yes	Yes	5	3	8

Count concurrently active connections	`def new_flow([cnt], []):`
	`  if cnt == 0: emit(); cnt = 1`
	`R1 = map(pktstream, [tin/128], [epoch]);`
	`R2 = groupby(R1, [5tuple, epoch], new_flow);`
	`num_conns = groupby(R2, [epoch], count);`
		Count the number of active connections in a queue over a period of time ("epoch").	No	No	4	3	10

TCP incast	`R3 = zip(num_conns, pktstream);` `ic_q = filter(R3, qin > 100 and cnt < 25);`	Detect when many connections use a long queue. Uses the query above.	No	No	7	4	14

Lossy connections	`total = groupby(pktstream, [5tuple], count);`
	`R1 = filter(pktstream, tout == infinity);`
	`lost = groupby(R1, [5tuple], count);`
	`Z = zip(total, lost);`
	`lc_q = filter(Z, lost.cnt > p*total.cnt);`
		Compute packet loss rate per connection, reporting connections with packet drop rate higher than a threshold `p`.	Yes	No	8	4	19

TCP timeouts	`def timeout([cnt], [last_time, tin]):`
	`  timediff = tin - last_time`
	`  if timediff > 280ms and timediff < 320ms:`
	`  cnt = cnt + 1`
	`  last_time = tin`
	`to_q = groupby(pktstream, [5tuple], timeout);`
		Count the number of timeouts for each TCP connection, by checking for packet inter-arrival times around 300 ms (retransmission timer).	Yes	Yes	8	3	15

Table 6.2: Examples of performance queries. We report that a query *scales* to a large number of keys either if (1) there are no stateful updates involved, or (2) all its stateful updates are linear-in-state *and* there are no `emit()`s. We use Domino [188] to report the hardware resources, *i.e.*, atom count and pipeline depth and width. Linear-in-state queries use the multiply-accumulate atom (§6.2); others use a NestedIf atom [188] that supports updates predicated on the state value itself.

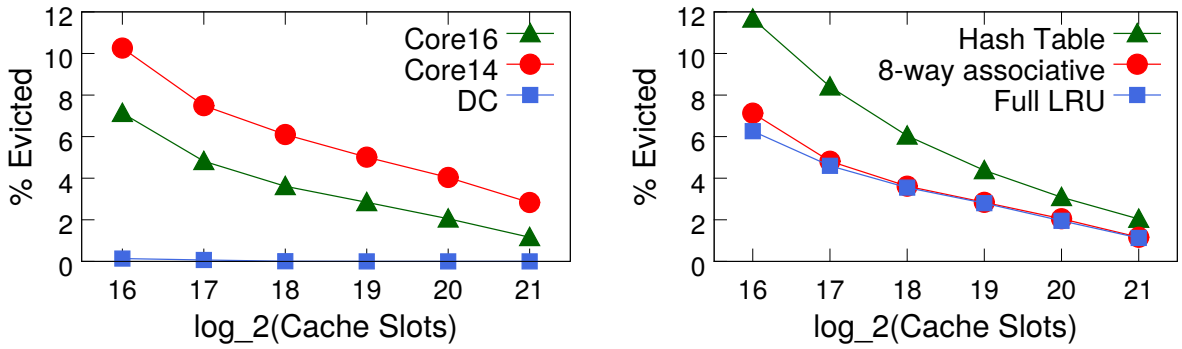
6.4.2 Memory and bandwidth overheads

In this section, we answer the following questions:

1. What is a good size for the on-chip key value store?
2. What are the eviction rates to the backing store?
3. How accurate are queries that are not mergeable?

Experimental setup. We simulate a Marple query over three unsampled packet traces: two traces from 10 Gbit/s core Internet routers, one from Chicago (~150M packets) from 2016 [54] and one from San Jose (~189M packets) from 2014 [53]; and a 2.5 hour university data-center trace (~100M packets) from 2010 [77]. We refer to these traces as Core16, Core14, and DC respectively.

We evaluate the impact of memory size on cache evictions for a Marple query that aggregates by



(a) By trace

(b) By cache geometry (Core16)

Figure 6-7: Eviction ratios to the backing store.

5-tuple. As discussed in §6.2.7, our hardware design uses an 8-way LRU cache. We also evaluate two other geometries for the cache: a hash table, which evicts the incumbent key upon a collision, and a fully associative LRU. Comparing our 8-way LRU with other hardware designs demonstrates the tradeoff between hardware complexity and eviction rate.

Eviction ratios. Each evicted key-value pair is streamed to a backing store. This requires the backing store to process packets as quickly as they are evicted, which depends on the incoming packet rate and the eviction ratio, *i.e.*, the ratio of evicted packets to incoming packets. The eviction ratio depends on the geometry of the on-chip cache, the packet trace, and the cache size measured by the number of key-value pairs the cache can store. Hence, we measure eviction ratios over (1) the three geometries for the Core16 trace (Figure 6-7b), (2) the three traces using the 8-way LRU geometry (Figure 6-7a), and (3) for caches sizes between 2^{16} (65K) and 2^{21} (2M) key-value pairs.

Figure 6-7b shows that a full LRU has the lowest eviction ratios, since the entire LRU must be filled before an eviction occurs. However, the 8-way associative cache is a good compromise. It avoids the hardware complexity of a full LRU while coming within 2% of its eviction ratio. Figure 6-7a shows that the DC trace has the lowest eviction ratios. This is because it has much fewer unique keys than the other two traces and these keys are less likely to be evicted.

The reciprocal of the eviction ratio is the reduction in server data collection load relative to a *per-packet collector* that processes per-packet information from routers. For example, for the Core14 trace with a 2^{19} key-value pair cache, the server load reduction is 25×, corresponding to an eviction ratio of 4%.

Eviction rates. Eviction ratios are agnostic to specifics of the router, such as link speed, link utilization, and on-chip cache size in bits. To translate eviction ratios in evictions per packet to eviction rates in evictions per second, we first compute the average packet size (700 Bytes) and link utilization (30%) from the Core16 trace.

Next, we estimate the on-chip cache size for a 64×10-Gbit/s router and a 64×100-Gbit/s router. On a 64×10-Gbit/s router, SRAM densities are ≈ 3–4 $^{Mbit}/_{mm^2}$ [1], and the smallest router chips occupy 200 mm^2 [115]. Therefore, a 64 Mbit cache in SRAM costs around 10% additional area, which we believe is reasonable. For recent 64×100-Gbit/s routers [5], SRAM densities are

Query	State size (bits)	Eviction rate at 64×10-Gbit/s (packets/s)	Eviction rate at 64×100-Gbit/s (packets/s)
Packet count	32	1.0M (34×)	4.29M (81×)
Lossy connections	64	1.08M (32×)	5.18M (66×)
TCP out-of-sequence	128	1.21M (28×)	6.72M (52×)
Flowlet size histogram (Stage 1)	160	1.26M (27×)	7.17M (48×)

Figure 6-8: Eviction rates and reduction in collection server load for queries from Table 6.2. Each key-value pair occupies the listed state size plus 104 bits for a 5-tuple key. The 10-Gbit/s and 100-Gbit/s routers have a 64 Mbit and 256 Mbit cache, respectively.

$\approx 7^{Mbit}/_{mm^2}$ [50], and the routers occupy $\approx 500\ \mathrm{mm}^2$,[14] making a 256 Mbit cache (7.3% area overhead) a reasonable target.

Given a query with an aggregation, we divide these cache sizes in bits by the size of the aggregation's key-value pair to get the cache size in terms of the number of key-value pairs. We then look up this number in Figure 6-7 to get the eviction ratio for that query. We then translate this eviction ratio to an eviction rate using the network utilization and packet size mentioned earlier.

Eviction rates for some sample queries are shown in Figure 6-8. For a 64×10-Gbit/s router with a 64 Mbit cache, we observe eviction rates of ≈ 1M packets per second. For a 64×100-Gbit/s router with a 256 Mbit cache and the same average packet size and utilization, the eviction rates can reach 7.17M packets per second. Relative to a per-packet collector, Marple reduces the server load by 25–80×.

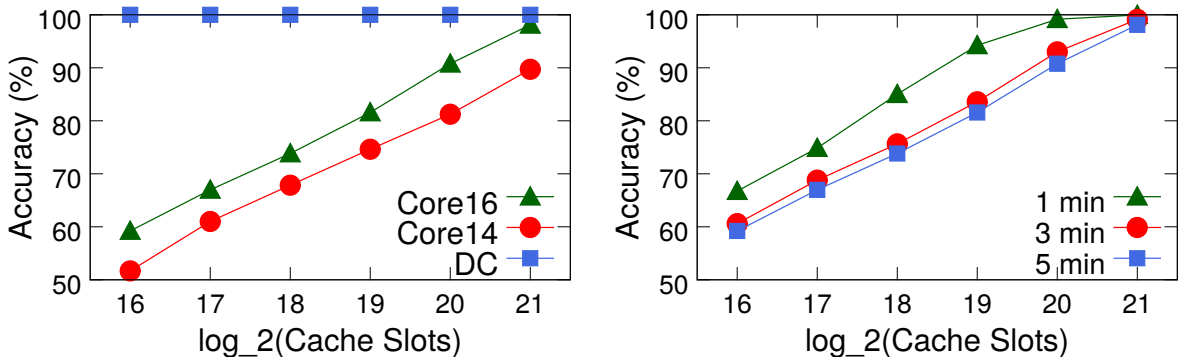
The eviction rates for both the 10 Gbit/s and 100 Gbit/s routers are under 10M packets per second, well within the capabilities of multi-core scale-out key-value stores [21, 125, 2], which typically handle 100K–1M operations per second per core. For instance, for a single 64×10-Gbit/s router running an aggregation with a 64-bit state size, a single 8-core server is sufficient to handle the eviction rate of 1.08M packets per second. For a single 64×100-Gbit/s router running the same aggregation, the eviction rate goes up to 5.18M packets per second, requiring four such servers.

Generalizing to other scenarios. Figure 6-7 also generalizes to multiple aggregations and aggregations of different state sizes. First, coarsening the aggregation key by picking a subset of the 5-tuple reduces the eviction ratio, since there are fewer keys in the working set. We believe that the 5-tuple may well be the most fine-grained and still practically useful aggregation level; hence, our results show the worst-case eviction ratios for a single `groupby`. Second, variations in the size of the `groupby` value simply result in a different number of key-value pairs for a given memory size. Third, running multiple `groupby` queries with the same number of key-value pairs, and aggregating by the same key, results in synchronized evictions across all queries. Hence, the eviction ratios can be read off Figure 6-7 at the correspondingly reduced memory size.

Accuracy of non-mergeable queries. Queries that are neither linear-in-state nor associative cannot be merged in the backing store. If a key from such a query is evicted multiple times, Marple cannot guarantee its correctness and marks it as invalid. However, these keys' values are still valid if they are either never evicted or are evicted once and never reappear. Hence, we define a query's accuracy as the fraction of keys with valid values over the query's lifetime.

[14] S. Chole. Cisco Systems. Private communication. June 2017.

Figure 6-9a shows query accuracy using the three traces, with the DC trace being near-perfect since it has fewer unique keys, and hence, evictions. If the query is run over a shorter time interval, its accuracy is typically higher, since the cache may not be full and a smaller fraction of keys are evicted. Figure 6-9b shows this tradeoff for a range of cache sizes and geometries using the Core16 trace. Shortening the query from 5 minutes to 1 minute boosts accuracy by 10%.



(a) By trace

(b) By query duration (Core16)

Figure 6-9: Query accuracy for non-mergeable aggregations.

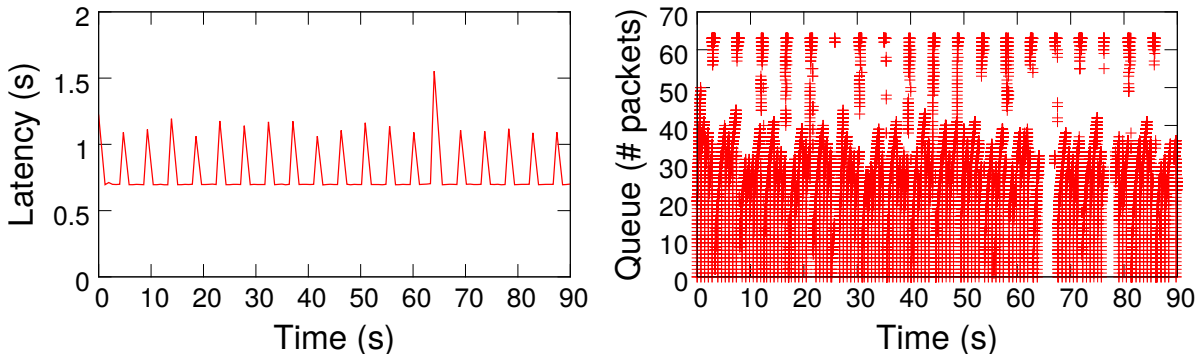
6.4.3 Case study #1: Debugging microbursts

To demonstrate Marple in practice, we present a case study of diagnosing microbursts, which are spikes in latency caused by bursty transmissions of packets into a queue. Our setup in Mininet [149] consists of four hosts (`h1, h2, h3, h4`) and two routers (`S1, S2`) in a dumbbell topology. Router `S1` is connected to `h1` and `h3`, and `S2` to `h2` and `h4`. The routers are connected via a single link and programmed in P4 [37] with queries compiled by Marple.

Host `h2` repeatedly downloads a 1MB object over TCP from `h1`. Meanwhile, `h3` sends `h4` bursts of UDP traffic, which `h4` acks. Suppose a network operator notices the irregular latency spikes for the downloads (Figure 6-10a) caused by these bursts of UDP traffic. However, the operator does not know what is causing these bursts. To diagnose the problem, she suspects a queue buildup in the routers and measures the queue depths seen by the traffic by writing: `result = filter(pktstream, srcip == h1 and dstip == h2).`

The results are streamed out on each packet to a collection server. After plotting the queue sizes, she notices spikes in queue size at egress port 3 on the router (Figure 6-10b) matching the periodicity of the latency spikes. To isolate the responsible flow(s), she divides the traffic into "bursts," which she defines as a series of packets separated by a gap of at least 800ms, as determined from the gap between latency spikes. She issues the following Marple query:

```
def burst_stats([last_time, nburst, time], [pkts, tin]):
    if tin - last_time > 800000:
        nbursts++;
        emit();
    else:
```



(a) TCP request latency

(b) Queue depth at egress port 2

Figure 6-10: Microburst case study measurements

src → dst	protocol	# Bursts	Time (μs)	# Packets
h3:34573 → h4:4888	UDP	19	8969085	6090
h4:4888 → h3:34573	UDP	18	10558176	5820
h1:1777 → h2:58439	TCP	1	72196926	61584
h2:58439 → h1:1777	TCP	1	72248749	33074

Figure 6-11: Per-flow burst statistics from Marple

```
        time = time + tin - last_time;
    pkts = pkts + 1;
    last_time = tin;
result = groupby(R1, 5tuple, burst_stats)
```

She runs the query for 72 seconds and sees the result in Figure 6-11. She concludes, correctly, that UDP traffic between `h3` and `h4` is responsible for the latency spikes. There are 18 UDP bursts, with an average packet size of 320 packets and average duration of 472 ms, which matches our emulation setup.

Marple's flexibility makes this diagnosis simple. By contrast, localizing the root cause of microbursts with existing monitoring approaches is challenging. Packet sampling and packet captures would miss microbursts because of heavy undersampling. Packet counters from routers would have disguised the bursty nature of the offending UDP traffic. Finally, probing from end hosts would not be able to localize queues with bursty contending flows.

Marple builds on In-band Network Telemetry [23, 61] (INT) that exposes queue lengths at routers. However, Marple's on-router aggregation goes beyond INT's ability to report queue sizes and latencies. For instance, Marple can determine contending flows at the problematic queue through a customized query to measure bursts, without prior knowledge of those flows. In comparison, a pure INT-based solution may require a network operator to manually identify flows that could contend at the problematic queue, and then collect data from the INT end hosts for those flows.

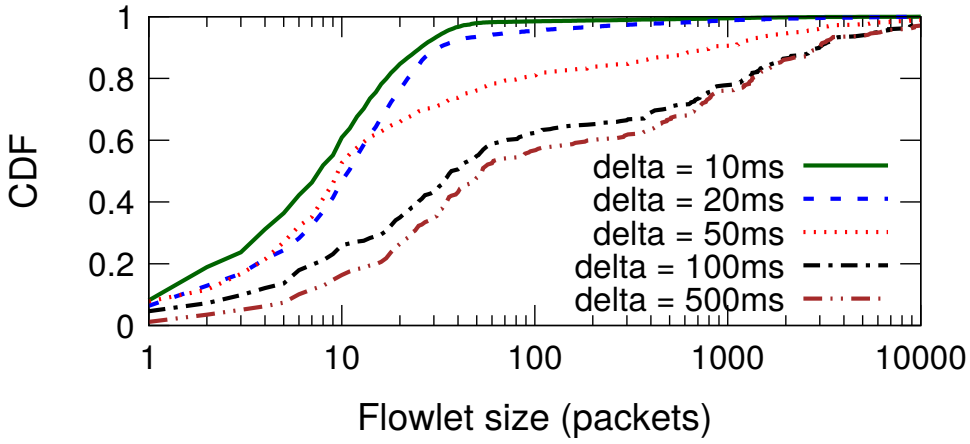


Figure 6-12: CDF of flowlet sizes for different flowlet thresholds.

6.4.4 Case study #2: Flowlet size distributions

We demonstrate another use case for Marple: computing the flowlet size distribution as a function of the flowlet *threshold*, the time gap above which consecutive packets are considered to be in different flowlets. This analysis has many practical uses, *e.g.,* for configuring flowlet-based load balancing strategies [65, 201]. In particular, the performance of the LetFlow [201] load balancing algorithm depends heavily on the distribution of flowlet sizes.

Our setup uses Mininet with a single router connecting five hosts: a single client and four servers. Flow sizes are drawn from an empirical distribution computed from a trace of a real datacenter [69]. The router runs the "flowlet size histogram" query from Table 6.2 for six values of `delta`, the flowlet threshold.

Figure 6-12 shows the CDF of flowlet sizes for various values of `delta`. Note that the actual values of `delta` are a consequence of the bandwidth allowed by the Mininet setup; a datacenter deployment would likely use much lower `delta` values.

6.5 Summary

Performance monitoring is a crucial part of the upkeep of any large system. Marple's whole network visibility and query language demystify network performance for applications, enabling operators to quickly diagnose problems. We want network operators to query for whatever they need, and not be constrained by limited router feature sets. Marple presents a step forward in enabling fine-grained and programmable network monitoring, putting the network operator—and not the router vendor—in control of the network.

Chapter 7

Limitations

We now discuss limitations of the systems presented in this dissertation. We begin with limitations that are common to all three systems. We then discuss limitations of each individual system.

7.1 Limitations common to all three systems

7.1.1 Lack of silicon implementations

None of our three systems has a hardware implementation in silicon. This is a problem that arises whenever a new system requiring hardware modifications needs to be evaluated. Designing a new chip can take a few years, and it requires both a significant financial investment and access to a large team of engineers.

As a result, we evaluated the three systems in this dissertation using a combination of simulation to check correctness and synthesis experiments to evaluate area overheads of new hardware designs. We briefly considered an FPGA implementation on the NetFPGA platform [216], but decided against it because the relative areas consumed by router subsystems on an FPGA may not accurately reflect the relative area consumptions on a router chip.

Instead, we paid careful attention to keeping our hardware designs simple, and reused standard hardware designs (*e.g.,* LRU caches and First In First Out Queues) wherever possible. To allow a router chip designer to leverage our results, we clearly specified the interfaces between the different hardware blocks in our design.

There is still value to an FPGA implementation, which we hope to explore in future work. First, an FPGA implementation allows us to check the correctness of the same Verilog code that we use for synthesis experiments, especially to test correctness when interfacing a router with the external world. Second, an FPGA implementation can serve as a vehicle for end-to-end evaluation of ideas, much like our Mininet evaluation (§6.4.3). Third, the development of new FPGA technology that allows clock rates of up to 1 GHz [151] suggests a way to prototype hardware at near-ASIC speeds while benefiting from the reconfigurability of an FPGA. Note that, even with such high speed FPGAs, an ASIC design of the hardware would still be required for large-scale commercial production of the hardware design. This is because, relative to an ASIC, an FPGA consumes more

power, occupies more chip area, and has a higher unit price.

7.1.2 Lack of completeness theorems

We do not have a formal characterization of what can and cannot be done by each system. While we have examples of algorithms that cannot be expressed by each system, it would be ideal to have a "completeness" theorem that states that all algorithms within a class (and within that class alone) can be expressed using a particular system. Such theorems would also give a formal assurance of a system being "future proof" so long as a new algorithm falls within a particular class of algorithms. An example of such a theorem is the equivalence of finite-state machines and regular expressions, which states that any regular expression—whether a regular expression that is known today or a regular expression that shows up in the future—can be expressed using a finite-state machine.

7.1.3 Lack of a user study

One way to empirically study the "future proofness" of a programming abstraction is to freeze the abstraction and then run a user study with new programmers who have never seen the abstraction before. We have not been able to carry out a such a user study because the concept of programming the data plane of a high speed network is relatively new and has not seen widespread adoption yet.

At the same time, some of our results suggest that our abstractions and primitives may be general. For instance, as mentioned earlier, the atoms developed in Domino (Chapter 4) supported unanticipated use cases that were developed after the atom designs were frozen (Table 4.6). Similarly, we did not anticipate the rate-controlled service disciplines (RCSD) as a use case for PIFOs when first designing PIFOs, but later found that PIFOs could express the RCSD class (§5.3). Finally, the atomic semantics of packet transactions are now part of the P4 language (§8.1), suggesting that the packet transaction abstraction might be quite general and natural.

7.1.4 Supporting routers with multiple pipelines

As discussed earlier (§4.1), throughout this dissertation, we assume a router architecture with a single ingress pipeline and a single egress pipeline shared across all ports. These pipelines can be made to run at a clock rate of up to 1 GHz, which at the minimum packet size of 64 bytes, translates into an aggregate capacity of ~640 Gbit/s, after accounting for minimum inter-packet gaps [86].

To scale beyond this aggregate capacity, a router needs multiple parallel pipelines, each devoted to a subset of the router's ports. This has implications for all our designs, which we discuss below.

Domino. In Domino, state is local to an atom, and hence to the pipeline stage containing that atom. In a single pipeline router, a pipeline stage can see all the packets arriving at the router. This allows the pipeline stage to maintain and update the state consistently, *e.g.,* a counter that counts the number of bytes received across all ports.

In a multi-pipeline router, no single pipeline sees all packets, so the state in a pipeline stage can only be updated by packets belonging to ports assigned to that pipeline. This may suffice for some applications, *e.g.,* per-port state that is maintained in each pipeline, which does not need to be

accessed from other pipelines.

For simple stateful operations like global, router-wide, counters, the associativity of addition allows us to update the state separately in each pipeline and combine it through addition later. However, we know of no general mechanism to combine pipeline-local state into a single, global, router-wide state value in a manner that emulates a single-pipeline router. Using a single state value that is accessed by multiple pipelines is a potential solution, but sharing state requires multi-ported memory and locking. Multi-ported memory is expensive, while locking degrades performance.

Performance queries. Because the `groupby` operator in Marple maintains and update state in the data plane, performance queries share the same limitations as Domino when it comes to multi-pipeline routers. The reason is the same: multi-pipeline routers have a separate instance of state for every pipeline that needs to be efficiently combined into a global router-wide state value without losing accuracy.

PIFOs. A multi-pipeline router has multiple ingress and egress pipelines that share access to the scheduler subsystem alone. The scheduler subsystem is common to all pipelines and consists of the data buffer to store packet payloads and the scheduling logic for different scheduling algorithms. To support enqueues and dequeues from multiple pipelines into the data buffer every clock cycle, the data buffer memory in a multi-pipeline router is multi-ported to support multiple writes/reads from multiple ingress/egress pipelines every clock cycle.

In multi-pipeline routers, each PIFO block needs to support multiple enqueue and dequeue operations per clock cycle (as many as the number of ingress and egress pipelines). This is because packets can be enqueued from any of the input ports every clock cycle, and each input port could reside in any of the ingress pipelines. Similarly, each egress pipeline requires a new packet every clock cycle, resulting in multiple dequeues every clock cycle.

We leave a full fledged PIFO design for multi-pipeline routers to future work, but observe that our current PIFO design facilitates a multi-pipeline implementation. A rank store supporting multiple pipelines is similar to the data buffers of multi-pipeline routers today, which already support multiple enqueues and dequeues. Building a flow scheduler to support multiple enqueues/dequeues per clock cycle is relatively easy because the flow scheduler is maintained in flip flops. A flip-flop-based implementation makes it simpler to add multiple ports, relative to an implementation using SRAM.

7.2 Domino limitations

The Domino compiler doesn't aggressively optimize, instead focusing on generating sub-optimal, but correct, pipeline configurations. For instance, it is possible to fuse two stateful codelets incrementing two independent counters into the same instance of the Pairs atom. However, by carrying out a one-to-one mapping from codelets to the atoms implementing them, our compiler precludes these optimizations. One practical implication of this is that we have had to manually rewrite programs to make them compile (Table 4.6), instead of a compiler rewriting such programs automatically.

Supporting multiple packet transactions in Domino also requires further work. This is especially

important because any real program running on a router is likely to execute multiple transactions, each on a subset of the packets seen by the router. When a router executes multiple transactions, there may be opportunities for inter-procedural analysis [63], which goes beyond compiling individual transactions and looks at multiple transactions together. For instance, the compiler could detect computations common to multiple transactions and execute them only once.

7.3 PIFO limitations

Our programming model for scheduling based on PIFOs is a scheduling tree with scheduling and shaping transactions. While a scheduling tree makes it possible to program new scheduling algorithms using PIFOs, it does not make it easy. We have found that it requires considerable effort to program new scheduling algorithms using scheduling trees. In other words, scheduling trees are still a low-level abstraction. We need to raise the level of abstraction if we hope to make programmable scheduling more broadly accessible to network operators.

Our current PIFO design scales to 2048 flows. If these 2048 flows are allocated evenly across 64 ports in a 64-port 10G router, we could program scheduling across 32 flows at each port. This permits per-port scheduling across traffic aggregates (*e.g.,* fair queueing across 32 VMs/tenants within a server), but not at a finer granularity (*e.g.,* 5-tuples). Ideally, to schedule at the finest granularity, our flow scheduler would realize the ideal PIFO where each packet in the PIFO belongs to a different flow. We currently support only 2048 flows, while we can support up to 60K packets. More design work is required to close this gap between the number of flows and the number of packets and realize an idealized PIFO.

7.4 Marple limitations

The most important limitation of performance queries is the fact that the latest value of the aggregated state is not available in the data plane, and is only accessible to software reading the backing store. This manifests itself most when using the `emit()` statement in our programs, which requires access to the latest value of the aggregated state in the data plane. Currently, we handle such queries by not evicting entries from a key-value store if the state stored in those entries is emitted to another query downstream. But not evicting keys severely limits the scalability of the key-value store because all keys must fit within the small on-chip cache. Unfortunately, this is unavoidable if we want access to the latest value of state during `emit()`s.

A similar scalability limitation affects queries that are not linear-in-state because there is no way to evict key-value pairs while still guaranteeing correctness if the query is not linear-in-state. In practice, we have found (§6.2.4) that it is possible to rewrite measurement functionality in such a way that the resulting query is linear-in-state, at some cost to measurement fidelity.

Chapter 8

Conclusion

To conclude this dissertation, we mention broader impact that this dissertation has had, outline areas for future work, and discuss the broader implications of this dissertation.

8.1 Broader impact

Several ideas from Domino have now found their way into P4 [85], an emerging language for programming network devices with considerable industry interest [38]. We describe changes to the P4 specification informed by Domino below.

1. When we began work on Domino in 2015, P4 was still relatively low-level in the sense of being close to the targets it was programming. For instance, to match the underlying router pipeline, P4 expected programmers to specify a packet-processing program as a sequence of lookups in a set of match-action tables. Each action within a match-action table was made up of a set of primitive actions, defined by the P4 language. These primitive actions within an action were expected to execute in parallel to mirror the execution model of the underlying hardware.
   This caused considerable confusion to programmers [39] because the P4 program had serial semantics between table lookups, while it had parallel semantics within the action half of a table lookup. Based on Domino's sequential semantics, P4 adopted a more uniform semantics, where primitive actions within a larger action also executed sequentially [46, 49].
2. The version of P4 released in 2016 [36] adopted many high-level language constructs introduced in Domino such as C-style expressions and the C-style conditional operator.
3. The 2016 version of P4 also allows programmers to specify atomic blocks of code [15, 13], which have the same transactional semantics as packet transactions. This is especially useful when writing P4 code for algorithms similar to the ones considered by Domino, which manipulate router state on a per-packet basis.

The impact of PIFOs and performance queries remains to be seen. We have talked to hardware engineers in industry about the possibility of adding PIFOs to a router's scheduler. Many of them seem interested in a programmable scheduler because they can now expressing scheduling algorithms as firmware, and not hardware. However, one major concern expressed by them is

whether the PIFO hardware would scale to a multi-Tbit/s router, which typically features multiple ingress and egress pipelines sharing the packet buffer and scheduling logic (§7.1.4).

We have also talked to hardware engineers about the possibility of adding a multiply-accumulate instruction to support linear-in-state measurement queries efficiently. Based on our conversations, we understand that this is well within reach, although there might be limitations on the bit width of the operands and outputs of the multiply-accumulate instruction.

8.2 Future work

Besides addressing the limitations described earlier (§7), there are a number of avenues for future work, described below.

Instruction set design for routers. The first generation of programmable router instruction sets, whether those in commercial chips [60, 25, 3, 86] or those proposed by Domino, were designed manually through a process of trial and error.

In particular, when designing atoms for Domino, we use trial and error to first guess atoms that we think might be useful and then use our compiler to check if those atoms can actually support useful algorithms. Formalizing this design process and automating it into an atom-design tool would be useful when designing router instruction sets. For instance, given a corpus of data-plane algorithms, this tool would automatically mine the corpus for recurring motifs of state and packet header modification. A router hardware engineer could then design hardware for atoms that capture these motifs.

Once programmers start writing P4 programs more broadly, we have the opportunity to design these instruction sets in a more empirical manner, optimizing instruction sets for actual use cases, as opposed to what a router engineer thinks the use cases will be. The continued evolution of the x86 instruction set over the last four decades suggests that there is considerable opportunity for workload-driven instruction set design.

Understanding the x86 tax*[1] *of networking. This dissertation has focused entirely on routers, which have historically been architected as fixed-function hardware devices. It leaves out a large number of networking devices with lower aggregate speeds that are built on top of commodity x86 processors (*e.g.,* network interface cards, the Linux kernel, middleboxes, proxies, and front-end servers). Until now, the steady improvement in x86 performance has allowed these devices to be programmable, while still providing sufficient forwarding performance for their needs.

Going forward, as processor clock frequencies and core counts stall [105], it seems natural to turn to specialized hardware [199] for such devices. Can we profile repeatedly used and relatively mature networking motifs that run on x86 chips today and measure their "x86 tax" relative to a pure silicon implementation? Such motifs (*e.g.,* SSL encryption and data compression) can be hardened as accelerators in silicon, providing energy and performance benefits over running on a general-purpose processor.

Approximate semantics for packet processing. Domino provides transactional semantics for packet processing. These semantics are straightforward, but end up rejecting programs for which trans-

[1] We thank Adam Belay for coming up with this expression.

actional semantics are not feasible while running at the router's line rate. Are weaker semantics sensible?

One possibility is approximating transactional semantics by only processing a sampled packet stream. This provides an increased time budget for each packet in the sampled stream, potentially allowing the packet to be *recirculated* through the pipeline for further packet processing. An example of such a use case is a measurement algorithm that needs a considerable amount of processing for each packet, but can still function if run on a sampled stream of packets.

A middle plane for networking. There will always be algorithms that cannot run in the data plane because the algorithms cannot sustain packet processing at the router's line rate. Today, such algorithms can only run on the much slower control plane, leading to a performance cliff once an algorithm crosses a threshold of complexity. Can we develop a programmable "middle plane" that sits between the control and data planes? This middle plane would provide greater programmability than the data plane, but at lower performance. It could be used to run algorithms either on a sample of packets across all ports or at full line rate on a subset of ports. For instance, an operator might require programmable measurement on a sample of all packets or line-rate scheduling support only on the congested ports.

Average case designs for routers. Routers have historically been designed to handle worst-case traffic patterns, *i.e.*, the smallest packet size at 100% utilization on all ports. This worst-case design mindset has several advantages. For instance, it obviates the need for performance profiling because routers guarantee line-rate performance regardless of packet size or utilization. It also guards routers against denial-of-service attacks that flood the router with small packets.

But it also leads to overengineering. A recent study found that the average packet size in datacenters is around 850 bytes (relative to a minimum packet size of 64 bytes) [78], while the utilization can be as low as 30% [78]. Factoring both average packet size and utilization, this implies that routers are designed to handle as much as 44 $\times$ more traffic than the average. Clearly, there are substantial gains to be had from adopting an average case mindset.

Software engineering for programmable routers. The development environment for programmable router chips is still quite rudimentary. This gives us an opportunity to revisit many traditional software engineering questions in the context of programmable routers. To name a few, developing better verifiers, debuggers, test generators, and exception handling mechanisms.

Evaluating ideas end-to-end on programmable router chips. Programmable router chips are just becoming available [3]. While they do not directly support the kinds of programmability described in this dissertation, they provide a platform to program different router algorithms on a physical router within a real network at aggregate speeds exceeding a Tbit/s. They also allow us to evaluate the benefits of our ideas end-to-end. For instance, if we could program a new active queue management scheme on a programmable router, what would be the effect on the completion time of a data analytics job?

Design-space exploration of router architectures. To evaluate the additional area of our hardware designs in this dissertation, we have relied on anecdotal estimates of the area of a baseline router chip based on private conversations with engineers in industry [115]. Ideally, we would have an open-source hardware design for a router chip that is competitive with industry-standard router

chips. We could then run synthesis experiments on this open-source hardware design to provide a rigorous baseline area estimate for our evaluations. When adding new hardware functionality, we could add it to this open-source chip, and rerun synthesis to estimate the new area of the chip. An open-source chip would also enable design-space exploration of router architectures. For instance, given an overall silicon budget, what is the Pareto frontier that trades off exact-match table capacity for ternary-match table capacity? How does this Pareto frontier move with changes in transistor size (*e.g.,* moving from 32 nm to 16 nm)?

Network architecture in the age of programmable routers. We have shown that it is technically feasible to build fast and programmable routers, allowing us to program routers with functionality that was traditionally on end hosts (*e.g.,* congestion control, measurement, and load balancing). Now that we have this capability to program the network's internals, and assuming we want to push our networks to the limits of their performance, what functionality belongs on the end hosts and what belongs within the network? Does this answer change if we only want basic correctness (and not optimal performance) from our networks?

8.3 Towards a world of programmable networks

The results in this dissertation point towards a world of programmable networks, where network operators *tell* routers what to do, without being *told* that this is all that the router can do. In such a setting, operators could customize networks as they see fit. They could program additional features that give them performance benefits or better visibility into their networks. More interestingly, they could remove needless features from their router, allowing them to simplify their routers' feature sets, which in turn could ease troubleshooting when things go wrong.

Besides the benefits to network operators, a programmable router chip has benefits for router vendors as well. Router vendors (*e.g.,* Arista and Dell) can now add new features in firmware and sell different versions of the firmware to different market segments. Further, when bugs arise, it is much easier to fix these bugs in firmware, as opposed to redoing the hardware design for the router, which could easily take years. It also allows vendors to respond to new requests from network operators in a period of days instead of years.

Whether the classes of router programmability described here will be sufficient remains to be seen, but we are encouraged by the fact that the hardware designs proposed here support a wide range of existing use cases and some new use cases that we had not anticipated initially. We hope the results of this dissertation provide guidance to router chip manufacturers when designing hardware for programmable routers.

Appendix A

The Merge Procedure for Linear-in-state Functions

For queries that are linear in state, the state update takes the form $\mathbf{S} = A(\mathbf{p}) \cdot \mathbf{S} + B(\mathbf{p})$, where $\mathbf{S}$ is a vector of state, and A and B are functions of the previous k packets (denoted by $\mathbf{p}$). Here k is an arbitrary integer.

A.1 Single packet history

Consider the case where $k = 1$. If the state is $\mathbf{S}$ at any point in time, then after N packets, $\mathbf{S_N} = A^N \cdot \mathbf{S}+$ terms independent of $\mathbf{S}$. Here, A^N is shorthand for $A(p_N) \cdot A(p_{N-1}) \cdot \ldots \cdot A(p_1)$ Therefore to merge an evicted value $\mathbf{S_N}$ with the existing value $\mathbf{S}_{backing}$ in the backing store, we need to replace $\mathbf{S}$ with $\mathbf{S}_{backing}$ in the expression for $\mathbf{S_N}$ by computing:

$$\mathbf{S_N} - A^N \cdot \mathbf{S} + A^N \cdot \mathbf{S}_{backing} = \mathbf{S_N} + A^N(\mathbf{S}_{backing} - \mathbf{S})$$

The merge procedure is straightforward: the router keeps A^N as auxiliary state and passes it with $\mathbf{S_N}$ to the backing store to complete the merge. The backing store already knows $\mathbf{S}$, since it is the default starting value for the state, which does not change.

A.2 Bounded packet history

The required auxiliary state is more complex for larger values of k. If $k = 1$, $A(p)$ and $B(p)$ are known to the router for every packet. However, for larger values of k, the router needs access to older packet fields to compute $A(\mathbf{p})$ and $B(\mathbf{p})$. This means the values of A and B are themselves incorrect for the first $k - 1$ packets after an eviction, an issue that must be addressed by the merge procedure.

Consider Figure A-1 for $k = 3$. Once the router performs an eviction at T_1, the router cannot properly update its state for the next two packets (first packets of the subsequent epoch). It thus starts updating its state from the third packet onwards and entrusts the backing store to fill the "hole"

caused by missing the first two state updates.

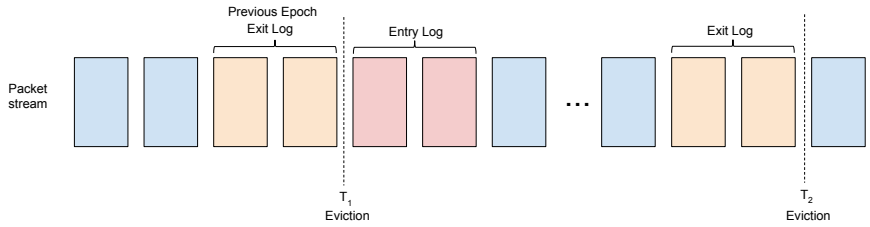


Figure A-1: Entry and exit log for an eviction, with $k = 3$.

To perform the merge successfully, the backing store needs two additional pieces of information, in addition to the query-specific state A^N discussed in the previous section:

- The last $k - 1$ packets from the *previous epoch*, called the *exit log* R of the previous epoch.
- The first $k - 1$ packets from the *current epoch*, called the *entry log* X of the current epoch.

Upon merging, the backing store receives $\mathbf{S}$, R, and X, and must merge them with $S_{backing}$. First, the backing store runs the actual aggregation function over R starting from $S_{backing}$: $S' = g(S_{backing}, R)$. Doing this requires the backing store to use the exit log of the *previous* epoch. The backing store then performs a standard merge $S'' = m(S', S)$ and stores X for use in the *next* epoch's merge.

Bibliography

[1] 45 nanometer - Wikipedia, Technology demos. https://en.wikipedia.org/wiki/45_nanometer#Technology_demos.

[2] An Update on the Memcached/Redis Benchmark. http://oldblog.antirez.com/post/update-on-memcached-redis-benchmark.html.

[3] Barefoot: The World's Fastest and Most Programmable Networks. https://barefootnetworks.com/media/white_papers/Barefoot-Worlds-Fastest-Most-Programmable-Networks.pdf.

[4] Benchmarking Apache Kafka: 2 Million Writes Per Second (On Three Cheap Machines). https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines.

[5] Broadcom First to Deliver 64 Ports of 100GE with Tomahawk II 6.4Tbps Ethernet Switch. https://www.broadcom.com/news/product-releases/broadcom-first-to-deliver-64-ports-of-100ge-with-tomahawk-ii-ethernet-switch.

[6] Cadence Encounter RTL Compiler. http://www.cadence.com/products/ld/rtl_compiler.

[7] Cavium XPliant switches and Microsoft azure networking achieve SAI routing interoperability. http://www.cavium.com/newsevents-Cavium-XPliant-Switches-and-Microsoft-Azure-Networking-Achieve-SAI-Routing-Interoperability.html.

[8] Cisco IOS NetFlow. http://www.cisco.com/c/en/us/products/ios-nx-os-software/ios-netflow/index.html.

[9] Cisco QuantumFlow Processor. https://newsroom.cisco.com/feature-content?type=webcontent&articleId=4237516.

[10] Configuring SPAN. http://www.cisco.com/c/en/us/td/docs/switches/lan/catalyst2940/software/release/12-1_19_ea1/configuration/guide/2940scg_1/swspan.html.

[11] Data center flow telemetry. http://www.cisco.com/c/en/us/products/collateral/data-center-analytics/tetration-analytics/white-paper-c11-737366.html.

[12] Design Compiler - Synopsys. http://www.synopsys.com/Tools/Implementation/RTLSynthesis/DesignCompiler/Pages/default.aspx.

[13] [design] Concurrency model for P4 . Issue #48 . p4lang/p4-spec. https://github.com/p4lang/p4-spec/issues/48.

[14] Entrepreneurial Capitalism and Innovation: A History of Computer Communications 1968-1988, 12.22: Proteon. http://www.historyofcomputercommunications.info/Book/12/12.22_Proteon.html.

[15] First cut of atomic specification by anirudhsk . Pull Request #80 . p4lang/p4-spec. https://github.com/p4lang/p4-spec/pull/80.

[16] Fixed-function - Wikipedia. https://en.wikipedia.org/wiki/Fixed-function.

[17] Flowlet Switching in P4. https://github.com/p4lang/tutorials/tree/master/SIGCOMM_2015/flowlet_switching.

[18] High Capacity StrataXGS®Trident II Ethernet Switch Series. http://www.broadcom.com/products/Switching/Data-Center/BCM56850-Series.

[19] High-Density 25/100 Gigabit Ethernet StrataXGS Tomahawk Ethernet Switch Series. http://www.broadcom.com/products/Switching/Data-Center/BCM56960-Series.

[20] Home » Open Compute Project. http://www.opencompute.org/.

[21] How Fast is Redis? http://redis.io/topics/benchmarks.

[22] How to Build a Gateway – C-Gateway: An Example. https://groups.csail.mit.edu/ana/Publications/Zhang-How-to-Build-A-Gateway-1987.pdf.

[23] "In-band Network Telemetry". https://github.com/p4lang/p4factory/tree/master/apps/int.

[24] Intel Enhances Network Processor Family with New Software Tools and Expanded Performance. http://www.intel.com/pressroom/archive/releases/2001/20010220net.htm.

[25] Intel FlexPipe. http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/ethernet-switch-fm6000-series-brief.pdf.

[26] Intel IXP2800 Network Processor. http://www.ic72.com/pdf_file/i/587106.pdf.

[27] Intel makes IXP its net processor cornerstone. http://www.eetimes.com/document.asp?doc_id=1142023.

[28] Intel64 and IA-32 Architectures Optimization Reference Manual. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

[29] IXP4XX Product Line of Network Processors. http://www.intel.com/content/www/us/en/intelligent-systems/previous-generation/intel-ixp4xx-intel-network-processor-product-line.html.

[30] LLVM Language Reference Manual - LLVM 3.8 documentation. http://llvm.org/docs/LangRef.html#abstract.

[31] Microsoft bets big on sdn. https://azure.microsoft.com/en-us/blog/microsoft-showcases-software-defined-networking-innovation-at-sigcomm-v2/.

[32] Multiply-accumulate operation. https://en.wikipedia.org/wiki/Multiply-accumulate_operation.

[33] Network Functions Virtualisation âĂŞ Introductory White Paper. https://portal.etsi.org/nfv/nfv_white_paper.pdf.

[34] Network Virtualization using Generic Routing Encapsulation. https://msdn.microsoft.com/en-us/library/windows/hardware/dn144775%28v=vs.85%29.aspx.

[35] Opening designs for 6-pack and Wedge 100. https://code.facebook.com/posts/203733993317833/opening-designs-for-6-pack-and-wedge-100/.

[36] P4-16 Language Specification. https://p4lang.github.io/p4-spec/docs/P4-16-v1.0.0-spec.html.

[37] P4 Behavioral Model. https://github.com/p4lang/behavioral-model.

[38] P4.org. http://p4.org/.

[39] P4's action-execution semantics and conditional operators. https://github.com/anirudhSK/p4-semantics/raw/master/p4-semantics.pdf.

[40] Packet Buffers. http://people.ucsc.edu/~warner/buffer.html.

[41] Place and route - Wikipedia. https://en.wikipedia.org/wiki/Place_and_route.

[42] Priority Flow Control: Build Reliable Layer 2 Infrastructure. http://www.cisco.com/en/US/prod/collateral/switches/ps9441/ps9670/white_paper_c11-542809_ns783_Networking_Solutions_White_Paper.html.

[43] Redis. http://redis.io/.

[44] RFC 2698 - A Two Rate Three Color Meter. https://tools.ietf.org/html/rfc2698.

[45] Sampled NetFlow. http://www.cisco.com/c/en/us/td/docs/ios/12_0s/feature/guide/12s_sanf.html.

[46] Sequential semantics by anirudhsk . Pull Request #21 . p4lang/p4-spec. https://github.com/p4lang/p4-spec/pull/21.

[47] sFlow. https://en.wikipedia.org/wiki/SFlow.

[48] Spark Streaming. http://spark.apache.org/streaming/.

[49] [spec] Sequential execution semantics . Issue #9 . p4lang/p4-spec. https://github.com/p4lang/p4-spec/issues/9.

[50] SRAM - ARM. https://www.arm.com/products/physical-ip/embedded-memory-ip/sram.php.

[51] System Verilog. https://en.wikipedia.org/wiki/SystemVerilog.

[52] Target Builtins - Using the GNU Compiler Collection (GCC). https://gcc.gnu.org/onlinedocs/gcc/Target-Builtins.html.

[53] The CAIDA UCSD Anonymized Internet Traces 2014 - June. http://www.caida.org/data/passive/passive_2014_dataset.xml.

[54] The CAIDA UCSD Anonymized Internet Traces 2016 - April. http://www.caida.org/data/passive/passive_2016_dataset.xml.

[55] The Juniper M40 Router | Computer History Museum. http://www.computerhistory.org/atchm/the-juniper-m40-router/.

[56] The Router: A Toast to William Yeager - CNN iReport. http://ireport.cnn.com/docs/DOC-235068.

[57] Three-address code. https://en.wikipedia.org/wiki/Three-address_code.

[58] Token Bucket. https://en.wikipedia.org/wiki/Token_bucket.

[59] TR 10: Software-Defined Networking - MIT Technology Review. http://www2.technologyreview.com/news/412194/tr10-software-defined-networking/.

[60] XPliant™ Ethernet Switch Product Family. http://www.cavium.com/XPliant-Ethernet-Switch-Product-Family.html.

[61] The Future of Network Monitoring with Barefoot Networks. https://youtu.be/Gbm7kDHXR-o, 2017.

[62] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*, 2016.

[63] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[64] D. S. Alexander, W. A. Arbaugh, M. W. Hicks, P. Kakkar, A. D. Keromytis, J. T. Moore, C. A. Gunter, S. M. Nettles, and J. M. Smith. The SwitchWare active network architecture. *IEEE Network*, 1998.

[65] 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. In *SIGCOMM*, 2014.

[66] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data Center TCP (DCTCP). In *SIGCOMM*, 2010.

[67] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda. Less Is More: Trading a Little Bandwidth for Ultra-Low Latency in the Data Center. In *NSDI*, 2012.

[68] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pFabric: Minimal Near-Optimal Datacenter Transport. In *SIGCOMM*, 2013.

[69] Alizadeh, Mohammad. Empirical Traffic Generator. https://github.com/datacenter/empirical-traffic-gen, 2017.

[70] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of Control Dependence to Data Dependence. In *POPL*, 1983.

[71] M. T. Arashloo, Y. Koral, M. Greenberg, J. Rexford, and D. Walker. SNAP: Stateful network-wide abstractions for packet processing. In *SIGCOMM*, 2016.

[72] A. Arasu, S. Babu, and J. Widom. The cql continuous query language: Semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, June 2006.

[73] B. Arzani, S. Ciraci, B. T. Loo, A. Schuster, and G. Outhred. Taking the blame game out of data centers operations with netpoirot. In *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference*, SIGCOMM '16, pages 440–453, New York, NY, USA, 2016. ACM.

[74] W. Bai, K. Chen, H. Wang, L. Chen, D. Han, and C. Tian. Information-Agnostic Flow Scheduling for Commodity Data Centers. In *NSDI*, 2015.

[75] H. Ballani, P. Costa, C. Gkantsidis, M. P. Grosvenor, T. Karagiannis, L. Koromilas, and G. O'Shea. Enabling End-Host Network Functions. In *SIGCOMM*, 2015.

[76] J. C. R. Bennett and H. Zhang. Hierarchical Packet Fair Queueing Algorithms. In *SIGCOMM*, 1996.

[77] T. Benson, A. Akella, and D. A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. *ACM International Measurement Conference*, Nov. 2010.

[78] T. Benson, A. Anand, A. Akella, and M. Zhang. Understanding Data Center Traffic Characteristics. *SIGCOMM Computer Communication Review*, Jan. 2010.

[79] R. Bhagwan and B. Lin. Design of a High-Speed Packet Switch for Fine-Grained Quality-of-Service Guarantees. In *ICC*, 2000.

[80] R. Bhagwan and B. Lin. Fast and Scalable Priority Queue Architecture for High-Speed Network Switches. In *INFOCOM*, 2000.

[81] S. Bhaumik, S. P. Chandrabose, M. K. Jataprolu, G. Kumar, A. Muralidhar, P. Polakos, V. Srinivasan, and T. Woo. CloudIQ: A Framework for Processing Base Stations in a Data Center. In *MOBICOM*, 2012.

[82] J. Bicket, D. Aguayo, S. Biswas, and R. Morris. Architecture and evaluation of an unplanned 802.11b mesh network. In *MOBICOM*, 2005.

[83] J. C. Bicket. Bit-rate selection in wireless networks. In *MasterâĂŹs thesis, MIT*, 2005.

[84] L. Bilge, E. Kirda, C. Kruegel, and M. Balduzzi. EXPOSURE: Finding Malicious Domains Using Passive DNS Analysis. In *NDSS*, 2011.

[85] 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 CCR*, July 2014.

[86] 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 *SIGCOMM*, 2013.

[87] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and J. van der Merwe. Design and Implementation of a Routing Control Platform. In *NSDI*, 2005.

[88] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking Control of the Enterprise. In *SIGCOMM*, 2007.

[89] M. Casado, T. Koponen, S. Shenker, and A. Tootoonchian. Fabric: A Retrospective on Evolving SDN. In *HotSDN*, 2012.

[90] V. Cerf and R. Kahn. A protocol for packet network intercommunication. *IEEE Transactions on Communications*, May 1974.

[91] A. Cheung, A. Solar-Lezama, and S. Madden. Using Program Synthesis for Social Recommendations. In *CIKM*, 2012.

[92] A. Cheung, A. Solar-Lezama, and S. Madden. Optimizing Database-backed Applications with Query Synthesis. In *PLDI*, 2013.

[93] A. K. Choudhury and E. L. Hahne. Dynamic Queue Length Thresholds for Shared-memory Packet Switches. *IEEE/ACM Trans. Netw.*, 6(2):130–140, Apr. 1998.

[94] S.-T. Chuang, A. Goel, N. McKeown, and B. Prabhakar. Matching Output Queueing with a Combined Input Output Queued Switch. *IEEE Journal on Selected Areas in Communications*, 17(6):1030–1039, Jun 1999.

[95] D. Clark. The Design Philosophy of the DARPA Internet Protocols. In *SIGCOMM*, 1988.

[96] G. Cormode and S. Muthukrishnan. An Improved Data Stream Summary: The Count-Min Sketch and Its Applications. *Journal of Algorithms*, April 2005.

[97] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: A Stream Database for Network Applications. In *SIGMOD*, 2003.

[98] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Language Systems*, 13(4):451–490, 1991.

[99] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. In *SIGCOMM*, 1989.

[100] D. R. Ditzel and D. A. Patterson. Retrospective on high-level language computer architecture. In *Proceedings of the 7th Annual Symposium on Computer Architecture*, ISCA '80, pages 97–104, New York, NY, USA, 1980. ACM.

[101] M. Dobrescu, K. Argyraki, and S. Ratnasamy. Toward Predictable Performance in Software Packet-Processing Platforms. In *NSDI*, 2012.

[102] M. Dobrescu, K. Argyraki, and S. Ratnasamy. Toward predictable performance in software packet-processing platforms. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 11–11, Berkeley, CA, USA, 2012. USENIX Association.

[103] 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 *SOSP*, 2009.

[104] R. Duncan and P. Jungck. packetC Language for High Performance Packet Processing. In *11th IEEE International Conference on High Performance Computing and Communications*, 2009.

[105] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark Silicon and the End of Multicore Scaling. In *ISCA*, 2011.

[106] C. Estan, G. Varghese, and M. Fisk. Bitmap Algorithms for Counting Active Flows on High-speed Links. *IEEE/ACM Trans. Netw.*, 14(5):925–937, Oct. 2006.

[107] N. Feamster, H. Balakrishnan, J. Rexford, A. Shaikh, and K. van der Merwe. The Case for Separating Routing from Routers. In *ACM SIGCOMM Workshop on Future Directions in Network Architecture (FDNA)*, 2004.

[108] N. Feamster, J. Rexford, and E. Zegura. The Road to SDN: An Intellectual History of Programmable Networks. *SIGCOMM Computer Communication Review*, 2014.

[109] W.-c. Feng, K. G. Shin, D. D. Kandlur, and D. Saha. The BLUE Active Queue Management Algorithms. *IEEE/ACM Transactions on Networking*, 2002.

[110] D. Firestone. VFP: A Virtual Switch Platform for Host SDN in the Public Cloud. In *NSDI*, 2017.

[111] S. Floyd and V. Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Trans. Netw.*, Aug. 1993.

[112] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A Network Programming Language. In *ICFP*, 2011.

[113] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. OpenNF: Enabling Innovation in Network Function Control. In *SIGCOMM*, 2014.

[114] M. Ghasemi, T. Benson, and J. Rexford. Dapper: Data Plane Performance Diagnosis of TCP. In *SOSR*, 2017.

[115] G. Gibb, G. Varghese, M. Horowitz, and N. McKeown. Design Principles for Packet Parsers. In *ANCS*, 2013.

[116] G. Gibb, G. Varghese, M. Horowitz, and N. McKeown. Design Principles for Packet Parsers. In *ANCS*, 2013.

[117] S. J. Golestani. A Stop-and-Go Queueing Framework for Congestion Management. In *SIGCOMM*, 1990.

[118] P. Goyal, H. M. Vin, and H. Chen. Start-time Fair Queueing: A Scheduling Algorithm for Integrated Services Packet Switching Networks. In *SIGCOMM*, 1996.

[119] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang. A Clean Slate 4D Approach to Network Control and Management. *SIGCOMM Computer Communication Review*, 2005.

[120] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z.-W. Lin, and V. Kurien. Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis. In *SIGCOMM*, 2015.

[121] A. Gupta, R. Birkner, M. Canini, N. Feamster, C. Mac-Stoker, and W. Willinger. Network Monitoring is a Streaming Analytics Problem. In *HOTNETS*, 2016.

[122] S. Ha, I. Rhee, and L. Xu. CUBIC: A New TCP-friendly High-speed TCP Variant. *SIGOPS Operating Systems Review*, 2008.

[123] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: A GPU-accelerated Software Router. In *SIGCOMM*, 2010.

[124] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks. In *NSDI*, 2014.

[125] S. Hart, E. Frachtenberg, and M. Berezecki. Predicting Memcached Throughput Using Simulation and Modeling. In *Symposium on Theory of Modeling and Simulation*, 2012.

[126] F. E. Heart, R. E. Kahn, S. Ornstein, W. Crowther, and D. C. Walden. The interface message processor for the ARPA computer network. In *Proceedings of the May 5-7, 1970, spring joint computer conference*, 1970.

[127] J. L. Hennessy and D. A. Patterson. *Computer Architecture: a Quantitative Approach*. Elsevier, 6^{th} edition, 2017.

[128] S. Iyer, R. Zhang, and N. McKeown. Routers with a single stage of buffering. In *SIGCOMM*, 2002.

[129] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a Globally-deployed Software Defined Wan. In *SIGCOMM*, 2013.

[130] 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 *SIGCOMM*, 2014.

[131] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, A. Greenberg, and C. Kim. EyeQ: Practical Network Performance Isolation at the Edge. In *NSDI*, 2013.

[132] S. P. Jones and P. Wadler. Comprehensive Comprehensions. In *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop*, Haskell '07, pages 61–72, New York, NY, USA, 2007. ACM.

[133] L. Jose, L. Yan, M. Alizadeh, G. Varghese, N. McKeown, and S. Katti. High Speed Networks Need Proactive Congestion Control. In *HotNets*, 2015.

[134] L. Jose, L. Yan, G. Varghese, and N. McKeown. Compiling Packet Programs to Reconfigurable Switches. In *NSDI*, 2015.

[135] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, et al. In-datacenter performance analysis of a tensor processing unit. *arXiv preprint arXiv:1704.04760*, 2017.

[136] C. R. Kalmanek, H. Kanakia, and S. Keshav. Rate Controlled Servers for Very High-Speed Networks. In *GLOBECOM*, 1990.

[137] N. Kang, Z. Liu, J. Rexford, and D. Walker. Optimizing the "One Big Switch" Abstraction in Software-defined Networks. In *CoNEXT*, 2013.

[138] M. Karol, M. Hluchyj, and S. Morgan. Input Versus Output Queueing on a Space-Division Packet Switch. *IEEE Transactions on Communications*, 1987.

[139] D. Katabi, M. Handley, and C. Rohrs. Congestion Control for High Bandwidth-Delay Product Networks. In *SIGCOMM*, 2002.

[140] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford. HULA: Scalable Load Balancing Using Programmable Data Planes. In *SOSR*, 2016.

[141] S. Katti, H. Rahul, W. Hu, D. Katabi, M. Médard, and J. Crowcroft. XORs in the air: Practical wireless network coding. In *ACM SIGCOMM computer communication review*, 2006.

[142] M. Khazraee, L. Zhang, L. Vega, and M. B. Taylor. Moonwalk: NRE Optimization in ASIC Clouds. In *ASPLOS*, 2017.

[143] C. Kim, A. Sivaraman, N. Katta, A. Bas, A. Dixit, and L. J. Wobker. In-band network telemetry via programmable dataplanes. In *ACM SIGCOMM Industrial Demo Session*, 2015.

[144] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Trans. Comput. Syst.*, 18(3):263–297, Aug. 2000.

[145] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, P. Ingram, E. Jackson, A. Lambeth, R. Lenglet, S.-H. Li, A. Padmanabhan, J. Pettit, B. Pfaff, R. Ramanathan, S. Shenker, A. Shieh, J. Stribling, P. Thakkar, D. Wendlandt, A. Yip, and R. Zhang. Network Virtualization in Multi-tenant Datacenters. In *NSDI*, 2014.

[146] S. S. Kunniyur and R. Srikant. An Adaptive Virtual Queue (AVQ) Algorithm for Active Queue Management. *IEEE/ACM Trans. Netw.*, Apr. 2004.

[147] I. Kuon and J. Rose. Measuring the Gap Between FPGAs and ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2007.

[148] M. Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *PLDI*, 1988.

[149] B. Lantz, B. Heller, and N. McKeown. A Network in a Laptop: Rapid Prototyping for Software-defined Networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, 2010.

[150] J.-T. Leung. A New Algorithm for Scheduling Periodic, Real-Time Tasks. *Algorithmica*, 4(1-4):209–219, 1989.

[151] D. Lewis, G. Chiu, J. Chromczak, D. Galloway, B. Gamsa, V. Manohararajah, I. Milton, T. Vanderhoek, and J. Van Dyken. The Stratix™10 Highly Pipelined FPGA Architecture. In *FPGA*, 2016.

[152] Y. Li, R. Miao, C. Kim, and M. Yu. FlowRadar: A Better NetFlow for Data Centers. In *NSDI*, 2016.

[153] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In *SIGCOMM*, 2016.

[154] Y. Lu, A. Montanari, B. Prabhakar, S. Dharmapurikar, and A. Kabbani. Counter Braids: A Novel Counter Architecture for Per-Flow Measurement. In *SIGMETRICS*, 2008.

[155] I. Lynce and J. Marques-silva. On Computing Minimum Unsatisfiable Cores. 2004.

[156] W. M. McKeeman. Language directed computer design. In *Proceedings of the November 14-16, 1967, fall joint computer conference*, pages 413–417. ACM, 1967.

[157] P. McKenney. Stochastic Fairness Queuing. In *INFOCOM*, 1990.

[158] 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.*, 38(2):69–74, Mar. 2008.

[159] G. D. Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1st edition, 1994.

[160] D. L. Mills. The Fuzzball. In *SIGCOMM*, 1988.

[161] R. Mittal, R. Agarwal, S. Ratnasamy, and S. Shenker. Universal Packet Scheduling. In *NSDI*, 2016.

[162] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing Software-defined Networks. In *NSDI*, 2013.

[163] J. T. Moore, M. Hicks, and S. Nettles. Practical programmable packets. In *INFOCOM*, 2001.

[164] M. Moshref, A. Bhargava, A. Gupta, M. Yu, and R. Govindan. Flow-level State Transition As a New Switch Primitive for SDN. In *SIGCOMM*, 2014.

[165] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. Dream: Dynamic resource allocation for software-defined measurement. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 419–430, New York, NY, USA, 2014. ACM.

[166] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. Trumpet: Timely and Precise Triggers in Data Centers. In *SIGCOMM*, 2016.

[167] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim. Language-Directed Hardware Design for Network Performance Monitoring. In *SIGCOMM*, 2017.

[168] S. Narayana, M. Tahmasbi, J. Rexford, and D. Walker. Compiling path queries. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 207–222, Santa Clara, CA, 2016. USENIX Association.

[169] K. Nichols and V. Jacobson. Controlling Queue Delay. *ACM Queue*, 10(5), May 2012.

[170] A. Ousterhout, J. Perry, H. Balakrishnan, and P. Lapukhov. Flexplane: An Experimentation Platform for Resource Management in Datacenters. In *NSDI*, 2017.

[171] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker. E2: A Framework for NFV Applications. In *SOSP*, 2015.

[172] R. Pan, L. Breslau, B. Prabhakar, and S. Shenker. Approximate Fairness Through Differential Dropping. *SIGCOMM Computer Communication Review*, April 2003.

[173] R. Pan, P. Natarajan, C. Piglione, M. Prabhu, V. Subramanian, F. Baker, and B. VerSteeg. PIE: A lightweight control scheme to address the bufferbloat problem. In *IEEE HPSR*, 2013.

[174] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker. NetBricks: Taking the V out of NFV. In *OSDI*, 2016.

[175] J. Perry, H. Balakrishnan, and D. Shah. Flowtune: Flowlet Control for Datacenter Networks. In *NSDI*, 2017.

[176] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. Fastpass: A Centralized "Zero-queue" Datacenter Network. In *SIGCOMM*, 2014.

[177] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. The Design and Implementation of Open vSwitch. In *NSDI*, 2015.

[178] P. M. Phothilimthana, T. Jelvis, R. Shah, N. Totla, S. Chasins, and R. Bodik. Chlorophyll: Synthesis-aided Compiler for Low-power Spatial Architectures. In *PLDI*, pages 396–407, 2014.

[179] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. FairCloud: Sharing the Network in Cloud Computing. In *SIGCOMM*, 2012.

[180] S. Ramabhadran and J. Pasquale. Stratified Round Robin: A Low Complexity Packet Scheduler with Bandwidth Fairness and Bounded Delay. In *SIGCOMM*, 2003.

[181] V. Raychev, M. Musuvathi, and T. Mytkowicz. Parallelizing User-defined Aggregations Using Symbolic Execution. In *SOSP*, 2015.

[182] H. Sariowan, R. L. Cruz, and G. C. Polyzos. SCED: A Generalized Scheduling Policy for Guaranteeing Quality-of-service. *IEEE/ACM Transactions on Networking*, Oct. 1999.

[183] L. E. Schrage and L. W. Miller. The Queue M/G/1 with the Shortest Remaining Processing Time Discipline. *Operations Research*, 14(4):670–684, 1966.

[184] D. Shah, S. Iyer, B. Prahhakar, and N. McKeown. Maintaining Statistics Counters in Router Line Cards. *IEEE Micro*, 2002.

[185] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making Middleboxes Someone else's Problem: Network Processing As a Cloud Service. In *SIGCOMM*, 2012.

[186] M. Shreedhar and G. Varghese. Efficient Fair Queuing using Deficit Round Robin. *IEEE/ACM Transactions on Networking*, 4(3):375–385, 1996.

[187] S. Sinha, S. Kandula, and D. Katabi. Harnessing TCPs Burstiness using Flowlet Switching. In *HotNets*, 2004.

[188] 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 *SIGCOMM*, 2016.

[189] A. Sivaraman, S. Subramanian, A. Agrawal, S. Chole, S.-T. Chuang, T. Edsall, M. Alizadeh, S. Katti, N. McKeown, and H. Balakrishnan. Towards Programmable Packet Scheduling. In *HotNets*, 2015.

[190] 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 *SIGCOMM*, 2016.

[191] A. Sivaraman, K. Winstein, S. Subramanian, and H. Balakrishnan. No Silver Bullet: Extending SDN to the Data Plane. In *HotNets*, 2013.

[192] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford. Heavy-Hitter Detection Entirely in the Data Plane. In *SOSR*, 2017.

[193] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial Sketching for Finite Programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 404–415, New York, NY, USA, 2006. ACM.

[194] I. Stoica, S. Shenker, and H. Zhang. Core-stateless Fair Queueing: A Scalable Architecture to Approximate Fair Bandwidth Allocations in High-speed Networks. *IEEE/ACM Transactions on Networking*, 2003.

[195] C. Tai, J. Zhu, and N. Dukkipati. Making Large Scale Deployment of RCP Practical for Real Networks. In *INFOCOM*, 2008.

[196] P. Tammana, R. Agarwal, and M. Lee. Simplifying datacenter network debugging with pathdump. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 233–248, GA, 2016. USENIX Association.

[197] K. Tan, H. Liu, J. Zhang, Y. Zhang, J. Fang, and G. M. Voelker. Sora: high-performance software radio using general-purpose multi-core processors. *Communications of the ACM*, 2011.

[198] K. Tan, J. Song, Q. Zhang, and M. Sridharan. A Compound TCP Approach for High-speed and Long Distance Networks. Technical report, July 2005.

[199] M. B. Taylor. Is Dark Silicon Useful?: Harnessing the Four Horsemen of the Coming Dark Silicon Apocalypse. In *Proceedings of the 49th Annual Design Automation Conference*, 2012.

[200] D. Ungar, R. Blau, P. Foley, D. Samples, and D. Patterson. Architecture of soar: Smalltalk on a risc. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, ISCA '84, pages 188–197, New York, NY, USA, 1984. ACM.

[201] E. Vanini, R. Pan, M. Alizadeh, P. Taheri, and T. Edsall. Let it Flow: Resilient Asymmetric Load Balancing with Flowlet Switching. In *NSDI*, 2017.

[202] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Andersen, G. R. Ganger, G. A. Gibson, and B. Mueller. Safe and Effective Fine-Grained TCP Retransmissions for Datacenter Communication. In *SIGCOMM*, 2009.

[203] D. Verma, H. Zhang, and D. Ferrari. Guaranteeing Delay Jitter Bounds in Packet Switching Networks. In *TRICOMM*, 1991.

[204] D. Wetherall. Active network vision and reality: lessons from a capsule-based system. In *Proceedings of DARPA Active Networks Conference and Exposition*, 2002.

[205] D. J. Wetherall. *Service Introduction in an Active Network*. PhD thesis, 1999.

[206] D. J. Wetherall, J. V. Guttag, and D. L. Tennenhouse. ANTS: a toolkit for building and dynamically deploying network protocols. In *1998 IEEE Open Architectures and Network Programming*, 1998.

[207] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron. Better Never Than Late: Meeting Deadlines in Datacenter Networks. In *SIGCOMM*, 2011.

[208] L. Yang, R. Dantu, T. Anderson, and R. Gopal. Forwarding and Control Element Separation (ForCES) Framework. Technical report, United States, 2004.

[209] M. Yu, A. Greenberg, D. Maltz, J. Rexford, L. Yuan, S. Kandula, and C. Kim. Profiling Network Performance for Multi-tier Data Center Applications. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 57–70, Berkeley, CA, USA, 2011. USENIX Association.

[210] M. Yu, L. Jose, and R. Miao. Software Defined Traffic Measurement with OpenSketch. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 29–42, Lombard, IL, 2013. USENIX.

[211] Y. Yu, P. K. Gunda, and M. Isard. Distributed aggregation for data-parallel computing: Interfaces and implementations. In *SOSP*, 2009.

[212] L. Yuan, C.-N. Chuah, and P. Mohapatra. Progme: Towards programmable network measurement. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '07, pages 97–108, New York, NY, USA, 2007. ACM.

[213] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz. DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks. In *SIGCOMM*, 2012.

[214] H. Zhang and D. Ferrari. Rate-Controlled Service Disciplines. *J. High Speed Networks*, 3(4):389–412, 1994.

[215] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, and H. Zheng. Packet-Level Telemetry in Large Datacenter Networks. In *SIGCOMM*, 2015.

[216] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore. NetFPGA SUME: Toward Research Commodity 100Gb/s. *IEEE Micro*, 2014.