Building Blocks for Leveraging In-Network Computing

Doctoral Dissertation submitted to the
Faculty of Informatics of the Università della Svizzera Italiana
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
Theo Jepsen

under the supervision of
Robert Soulé and Fernando Pedone

July 2020

Dissertation Committee

Antonio Carzaniga	Università della Svizzera italiana
Noa Zilberman	University of Oxford, UK
Edouard Bugnion	EPFL, Switzerland

Dissertation accepted on 22 July 2020

Research Advisor

Robert Soulé

Co-Advisor

Fernando Pedone

PhD Program Director

Walter Binder

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Theo Jepsen
Lugano, 22 July 2020

Abstract

With the end of Moore's law and Dennard scaling, applications no longer enjoy performance improvements by simply waiting for the next generation of CPUs. This has led to the rise of domain-specific computing. As developers try to squeeze more performance out of applications, they have offloaded application functionality to specialized hardware, such as GPUs, FPGAs and ASICs. Programming these devices presents a trade-off between generality and performance.

Recently there has been the emergence of new types of specialized hardware for networking. Similarly to other domain-specific computing devices, they are not straightforward to program and require adapting applications, but provide significant performance improvements. Although these devices were intended for network applications, they have been used for offloading other types of applications, in what is called in-network computing (INC).

INC presents many opportunities to applications because it provides high-performance computing that is centrally located in the network. However, there are many challenges for leveraging INC. Although INC devices are programmable, there are some limitations that are not present in general purpose CPUs, including computing expressiveness, resource availability (e.g., memory) and interfacing to applications.

This thesis addresses the challenge of leveraging INC for application-network co-design. Not all applications are suitable for INC; in some cases, only parts of applications can benefit from INC. The hypothesis is that application performance can be improved by moving some functionality into reusable INC building blocks. At a high level, this thesis makes the following contributions: it characterizes the types of applications that can benefit from INC; it describes the building blocks that applications can use to leverage INC; it suggests the right abstractions and level of granularity for INC; it describes INC data structures and implementation techniques; and, finally, it evaluates INC by implementing five systems for applications from different domains. Overall, this thesis revisits the separation of concerns between the application and the network, showing that co-design is not only possible, but also beneficial.

Preface

The result of this research appears in the following publications:

[1] Theo Jepsen, Masoud Moshref, Antonio Carzaniga, Nate Foster, and Robert Soulé. Life in the fast lane: A line-rate linear road. In *ACM SIGCOMM Symposium on SDN Research (SOSR)*, pages 10:1–10:7, March 2018

[2] Theo Jepsen, Leandro Pacheco de Sousa, Masoud Moshref, Fernando Pedone, and Robert Soulé. Infinite resources for optimistic concurrency control. In *Proceedings of the 2018 Morning Workshop on In-Network Computing*, pages 26–32, New York, NY, USA, August 2018

[3] Theo Jepsen, Masoud Moshref, Antonio Carzaniga, Nate Foster, and Robert Soulé. Packet subscriptions for programmable asics. In *Workshop on Hot Topics in Networks*, pages 176–183, New York, NY, USA, November 2018

[4] Theo Jepsen, Daniel Alvarez, Nate Foster, Changhoon Kim, Jeongkeun Lee, Masoud Moshref, and Robert Soulé. Fast string searching on pisa. In *ACM SIGCOMM Symposium on SDN Research (SOSR)*, pages 21–28, New York, NY, USA, April 2019

Acknowledgements

First of all, for all his guidance, encouragement and optimism that anything can be done—despite all my skepticism!—I am grateful to my adviser, Robert Soulé. My co-adviser Professor Fernando Pedone and Professor Antonio Carzaniga always had their doors open and were very patient with me. Thank you for your support.

I would like to thank my committee members, Edouard Bugnion and Noa Zilberman, for their invaluable feedback and patience.

The work in this dissertation would not have been possible without the support of my collaborators: Daniel Alvarez, Nate Foster, Changhoon Kim, Jeongkeun Lee, Alberto Lerner, and Masoud Moshref. Thank you for all the advice you gave me. It was a pleasure working together, and I hope we continue to do so in the future.

During my internship at Barefoot, everyone there became my family away from USI. For the countless hours lost under cables in the server room, discussing problems in the garage and biking across miles of CA roads—thank you.

I don't think I would have gotten through the PhD without all the hacker tips, emotional support and distractions from the people in the "lab" at USI: Pietro, Tu, Long, Enrique, Mojtaba, Leandro, Paulo, Daniel, Daniele, Daniele and Ali. I would also like to thank the people of the OpenSpace and the Decanato.

Last, but not least, I am indebted to my parents, grandfather and brother—Lisa-Anne, Frits, Ed, and Benjy—for bearing with me.

Contents

Figures

Tables

Chapter 1

Introduction

Historically, as CPUs got faster, applications experienced speed-ups "for free"; they simply upgraded to a CPU with higher clock speed and bigger caches. However, with the end of Moore's law and Dennard scaling, applications cannot rely on performance improvements from the CPU. Increasingly, application designers are turning to domain-specific computing, which uses specialized hardware to accelerate critical parts of applications.

There is a wide variety of hardware used for domain-specific computing. There is a trade-off between flexibility and performance. On one side of the spectrum are general-purpose CPUs, which can run most applications, but not efficiently. Graphics processing units (GPUs) and field-programmable gate arrays (FPGAs) provide better performance, but have more constraints. On the other side of the spectrum are application-specific integrated circuits (ASICs), which are optimized to run a single application. Recently, *programmable* ASICs have gained popularity, because they offer more flexibility, without compromising performance.

These programmable ASICs are now integrated in some network devices, including switches and network interface cards (NICs). These devices are flexible and fast, but more important is where they are located: in the network. Being positioned in the network brings a new opportunity of a global view for performing high-performance computation on application data.

Just like other domain-specific computing, the computational model on programmable network devices is not like that of a general-purpose CPU. Although these network devices can process packets at line rate, they do not offer all the same expressive functionality of a general-purpose CPU. This presents both opportunities and challenges for implementing application logic in the network. Application logic can run faster in the network and it also has a global view of processing, being between servers. However, it is not straight-forward to integrate with the application. It requires sharing application data and state with the network, as well the ability to express application logic in the network.

This raises the questions: *What types of applications can be implemented in the network? What functionality must the network provide to support these applications? Is this compatible and interoperable with existing solutions?* In some cases, only parts of the application can or should belong in the network. This forces us to revisit the separation of concerns between the application and network layer.

1.1 This Dissertation

This thesis addresses the problem of how applications can scale using the in-network computing capabilities of new hardware devices. The hypothesis is that **by exposing application-level information to the network, we can leverage in-network computing to dramatically improve application performance.**

To support the hypothesis, this work characterizes the types of applications that can benefit from in-network computing (INC); describes the building blocks that the network can provide to applications; and evaluates the performance impact of executing application logic in the network.

There are a variety of devices that support INC, including programmable switches, networked FPGAs and smart NICs. This dissertation focuses on one type of device: the programmable switch.

There are at least two ways programmable switches can be leveraged by applications. First, switches can be used as a dedicated appliance, roughly how other hardware accelerators like FPGAs and GPUs are used: offloading compute-intensive parts of the workload to the device to take advantage of parallelism and fast memory. A second way we may use the programmable switch is to operate on data as it flows through the network, not only exploiting the switch performance characteristics, but also its central location in the network.

To understand how applications should use programmable switches, we must first characterize the applications. Some applications are distributed by nature (e.g., messaging systems, distributed key-value stores, caches), while other applications are more computational (e.g., batch jobs, databases). One can understand how distributed applications—they rely heavily on network communication—can benefit from INC. It is less obvious how applications that make light use of communication can harness INC. It turns out that both communication-heavy and computationally heavy (i.e. CPU and I/O bound) workloads can benefit from INC.

INC provides different functionality for different types of applications. For example, in communication-intensive applications like pub/sub messaging, forwarding is an essential functionality. In analytics applications, matching data in packets can be more useful. Some applications have overlapping functionality requirements. For example, both key-value stores and caches can benefit from storing data in the network.

Applications can leverage INC through basic building blocks. The functionality of these building blocks should have the right level of granularity: if the functionality is too application-specific, it will not be reusable; if the functionality is too generic, the building block may not provide benefits to the application.

To understand whether and how applications benefit from INC, there are two main criteria. First, it has to be possible to express the application logic in programmable hardware. This depends on the primitive operations supported by the hardware, as well as resource availability (e.g., memory). In some cases interoperability is a strong requirement: the in-network component must be compatible with existing solutions (e.g., network or application protocols and formats).

Second, we must consider application performance to understand how it benefits from INC. We measure performance using various metrics, including end-to-end runtime (throughput), latency and overall system resource utilization. Furthermore, we must also evaluate the cost of implementing this in the network (e.g., How much device resources does it require? Does this come at the loss of other network functionality?).

1.2 Evaluation

To evaluate the hypothesis, we answer the following questions:

1. What types of applications can benefit from INC?
2. What is the right level of granularity for INC building blocks?
3. What and how much application-level information should be exposed to the network?
4. What are the potential performance benefits?
5. What must the application trade-off, in terms of features or expressiveness?

To be able to generalize the results, we choose applications from different domains. Not all application domains are suitable for INC, however. As we explain in Chapter 3, the most suitable applications are those that have a high ratio of I/O to processing expressiveness and that can easily interface their logic to the network.

We identified several applications from different domains: transactional databases, messaging systems, stream processing, and analytics. Some of these domains, like messaging systems, are naturally suited for INC, because they are inherently distributed, and must use the network. For other domains, like databases, it is not obvious how they can benefit from INC. In our evaluation, we show that even applications that are not network-centric can benefit from INC.

To evaluate INC with these applications, we did not implement the entire application in the network. Instead, we abstracted some application functionality in INC *building blocks*. We implemented several systems with these INC building blocks:

- A high throughput publish/subscribe messaging system with expressive forwarding semantics.
- A stream processing system that includes a variety of streaming operators: filter, aggregate, join and windowing.
- A string search system, which filters data using a state machine.
- An optimistic concurrency control system for a key-value store that supports speculative execution at the clients.
- An in-memory database that lets the network manage the delivery (including ordering) of transactions.

For each system, the implementation in itself demonstrates that it is possible for the application to leverage INC. In some cases, it requires the network implementation to be compatible with an existing application. For example, our publish/subscribe system is compatible with the format used by the Nasdaq ITCH protocol. There is also the question of expressiveness—does the in-network implementation provide the same functionality as the software alternative?

The implementation is also used to evaluate performance, which includes: application performance metrics (throughput and latency), system resource utilization, and cost. For some of the implementations, like the database and key-value systems, we measure throughput in transactions per second; for string search, the ingest rate; for stream processing, operations per second. Some applications are more latency sensitive, like those that use publish/subscribe, in which case latency is a more important metric. We also evaluate the overall system utilization, which has an impact on cost. This includes the resource utilization on the network devices, as well as potential savings from offloading computation from CPUs to the network.

1.3 Research Contributions

Overall, this dissertation makes the following contributions:

1. It characterizes the types of applications that can benefit from in-network computing (INC). Conversely, it also characterizes applications (or functionality) unsuitable for INC. It shows that even applications that are not network-centric can use INC.

2. It describes the basic building blocks network can provide to applications for a higher level of service. Since applications have overlapping requirements, it is possible to decompose them into the common functionality that can be provided by the network.

3. It revisits the separation of concerns between the application and the network, suggesting the right level of abstraction. The application layer is typically opaque to the underlying network. We argue that by exposing application information to the network, the network can provide more services and optimizations to the application.

4. It describes the techniques for implementing applications using INC. Because the network devices use a different architecture than general purpose CPU, we developed novel techniques for adapting application logic to these devices.

5. It provides an evaluation of INC by implementing five different systems for applications from various domains. Not only does this demonstrate the viability of INC, but also the performance benefits. It offloads compute and I/O from servers, reducing system utilization, and in turn overall costs.

The rest of this dissertation is organized as follows. We begin by providing some background on advanced network technologies in Chapter 2. Then, Chapter 3 discusses the INC building blocks that can be leveraged by applications.

The rest of the chapters describe our experiences building systems that use each building block: publish/subscribe messaging (Chapter 4); analytics for stream processing (Chapter 5) and string search (Chapter 6); coordination (Chapter 7); and load balancing (Chapter 8). Finally, Chapter 10 concludes by summarizing our findings and providing an outlook for future work.

Chapter 2

Background

This chapter provides background on networking technologies and the recent developments that have enabled in-network computing (INC). We begin with an overview of how traditional networks forward packets (§ 2.1) and describe some advanced techniques used at end hosts (§ 2.2). Then, we provide a brief history of software defined networking (§ 2.3), and how it led to programmable networks (§ 2.4), the foundation of INC.

2.1 Basic Networking

A traditional network is responsible for delivering packets to applications. The lifetime of a packet begins when an application process sends a packet by making a call to the OS. The OS schedules the packet to be sent by the network interface card (NIC). The NIC transmits the packet to the network, where it will travel through one or more switches, before reaching the destination host. The NIC on the destination host receives the packet and buffers it in memory. The NIC sends an interrupt, alerting the OS that a packet has been received. In turn, the OS passes the packet to the receiving process.

Conceptually, networking can be divided into two main parts: the data plane and the control plane. The data plane, also called the forwarding plane, is responsible for forwarding packets between devices. It makes local decisions, like deciding the port out of which a packet should be sent. The control plane, on the other hand, uses global policy to configure the data plane. It makes high level decisions, like choosing routes between devices. The control plane generates configuration (i.e. rules) upon which the data plane acts.

Advanced network technologies strive make the data plane faster and the control plane more flexible. There are various approaches that target different parts of the path through the network, including hardware devices like switches and NICs, as well as the software at the end host (e.g., the OS) that interacts with them.

2.2 Kernel Bypass

There have been efforts to reduce the time spent at each step in the transmission of packets on end hosts. In some cases, they remove steps all together. One such approach is kernel-bypass, which enables the application to interact directly with the NIC, without the OS.

2.2.1 DPDK

Some NICs support the Data Plane Development Kit (DPDK) [5] for kernel-bypass. With DPDK, an application process running in user space can send and receive packets directly, without interacting with the OS. This reduces the overhead of interrupts and OS context switches, dramatically increasing performance. Furthermore, DPDK provides raw access to the packet, which means that the application is not limited to traditional protocols like TCP/IP. This allows for a greater degree of flexibility, as the application can send packets with arbitrary formats, and not waste time/memory for managing protocol state (e.g., managing a TCP connection).

2.2.2 RDMA

Remote Direct Memory Access (RDMA) [6] takes bypass even further: it not only bypasses the OS, but also the CPU core. The RDMA NIC writes packet data directly into memory regions owned by the application process. This enables packets to be sent and received without involving the CPU core.

There are two main RDMA communication patterns: single-sided and two-sided. With single-sided, a process writes to a memory address on a remote host, without notifying the remote process. The remote process has to poll the memory region to detect that it was updated. With two-sided, the receiving process has to explicitly request the data. Once the data is received, the receiving process is notified. The advantage is that the receiver is notified, but it requires the receiver's CPU core.

2.2.3 NIC Protocol Offload

The packet handoff between the network and an application is a sophisticated process mediated by the network card and the operating system. Modern NICs can perform several operations on hardware on behalf of the OS [7]. For instance, they can verify the received packets' CRCs, saving many CPU cycles. Of particular interest to our work is a mechanism called Receive-Side Scaling (RSS) that is present in most modern network cards [8]. When a multi-core server receives a packet, the NIC generates an interrupt against a given core. In a fast network, this task can easily overwhelm one (or more) cores. RSS load balances the interrupts across the cores. It instructs the NIC to decide which core to interrupt based on a hash over some packet fields.

2.3 Software Defined Networking

Until fairly recently, it was difficult to configure the forwarding elements of the network. Network administrators used ad hoc scripts to enact policy in the network, such as routing and access control lists (ACLs). Furthermore, there was no separation between the layer for managing the network (control plane) and the layer responsible for forwarding packets (data plane). This was mainly due to the lack of a common interface for managing the devices [9].

Software defined networking (SDN) addresses this problem by separating the data plane from the control plane. The OpenFlow [10] project was instrumental in creating a standard interface for configuring switches. This enabled network administrators to manage the network just the way they would write software.

The OpenFlow standard introduced the match+action table abstraction. An OpenFlow switch matches (looks up) certain packet header fields (e.g., IP address, TCP port, etc.) in tables, and then executes the corresponding actions (e.g., forward, drop, etc.). As the standard grew in popularity, it added support for additional header fields and actions. However, it was still not flexible enough because it did not support arbitrary packet formats or custom actions.

2.4 Programmable Data Plane

To further increase the flexibility and programmability of SDN networks, the RMT (reconfigurable match tables) model was proposed [11]. Instead of matching on a fixed set of packet header fields and executing pre-defined actions, it provides a lower-level abstraction of primitives that support arbitrary packet formats and actions. This has given rise to new hardware to support this abstraction.

2.4.1 Programmable Hardware

Modern networks are connected by links that transmit data at ever increasing rates. The line rate has progressed from 10Gb/s to 100Gb/s, and soon 400Gb/s. To process packets at line rate, network devices must be fast. That is why they are mainly built using ASICs, which provide low latency, predictable processing.

Although the ASICs in network devices are performant, they are not flexible. They have fixed functionality: the processing logic and supported protocols are baked into hardware, and cannot be modified after manufacturing. To add a new protocol or feature, the ASIC circuits must be updated and manufactured, a long and costly process.

Recently, there has been an emergence of programmable network devices. These include smart NICs, networked FPGAs, and programmable switches. Smart NICs (e.g., Xilinx Alveo [12] and Netronome Agilio [13]) use a System-on-Chip (SoC) with some accelerators, in contrast to the more flexible design of the NetFPGA SUME [14]. Programmable switches (e.g., Barefoot Tofino [15] and Broadcom Trident 3 [16]) use special ASICs, which, unlike fixed-function devices, can be programmed after manufacturing. They can be programmed much like one writes software, but provide the same line rate processing of the fixed-function devices.

2.4.2 Language Support for Programmable Network Hardware

Various frameworks and languages have been proposed to program these devices, including high-level languages like C# [17], as well as domain-specific languages (DSLs) like PX [18] and Programming Protocol-Independent Packet Processors (P4) [19]. P4 has gained traction and is the most widely used language.

P4 describes the abstractions of a pipeline architecture based on PISA (Protocol Independent Switch Architecture). This architecture allows most network functionality to be expressed in a pipeline of stages that is independent of the underlying hardware. The architecture is abstract, and is agnostic of the underlying target, which includes software switches [20], smart NICs, FPGAs and programmable switch ASICs.

In PISA, when a packet arrives, it is first parsed, then flows through a pipeline of stages. In each stage, the match+action abstraction is used to lookup packet header fields in tables, and execute actions. These are the main constructs of the P4 language:

- Parser. This defines how the bits that arrive on the wire should be parsed into headers.
- Headers. The header defines the format of the packets, including the order of fields and their sizes.
- Control. This provides the overall structure for the pipeline, specifying the order in which tables are applied.
- Tables. The match+action abstraction is represented in the table. The table looks-up fields from the packet and executes the corresponding action. The table is populated by the control plane. Depending on the target, different types of look-ups can be performed, including: exact match, longest-prefix match (LPM), ternary and range match.
- Actions. These define the manipulations on the packet headers and metadata. They execute ALU operations or special operations (e.g., hash functions) supported by the target. They can also receive arguments (data) from the control plane.

Metadata flows through the pipeline together with the packet. It is represented as a header and is used to store per-packet state. Special metadata is used to pass information between the pipeline and the device. For example, metadata contains the port on which the packet arrived. Stateful processing is used to store state across packets. State is generally stored in registers, counters and meters.

2.4.3 Challenges of Data Plane Programming

Using these devices presents the opportunity to run custom logic in the network at line rate speeds. However, programming them is not straight-forward and presents many challenges. Broadly, there are two main types of challenges: constraints on expressiveness (e.g., types of ALU operations) and on resources (e.g., memory).

In the PISA architecture there is a fixed number of pipeline stages. The P4 program controls the logic in each stage, which, depending on the target, is limited to a certain number (and types) of operations. Some targets can only execute simple ALU operations, like add and subtract, while others support more complex operations like multiplication and division. Furthermore, some targets can provide externs that expose other types of operations, like computing a hash function on header fields.

Some targets have limitations on which stages can access memory. Packets only flow one way through the pipeline, so operations in one stage cannot write to memory in previous stages. This means that updates to data structures must be atomic: in a single stage the P4 program must read and update the memory. This limitation can be overcome by recirculating the packet through the pipeline, so the same packet will reach each stage twice. Recirculation, however, halves the throughput of the program.

P4 does not include some programming paradigms that are taken for granted in high level languages like C. For example, since there is a fixed number of pipeline stages, P4 does not support iteration. As we will explain, iteration can be achieved with recirculation. Since all the operations in a stage are executed in parallel, to execute a sequence of operations, they must be split across several stages. This may be problematic if the operations must access memory from different stages, as described in the previous paragraph.

Since these devices have a fixed amount of memory and parsing states, there is also a limit to how much state a P4 program can store, and how many packet header fields it can access. Furthermore, the devices have different types of memory. For example, the Tofino has at least two types of memory: SRAM for storing tables and stateful memory, and TCAM for tables that perform LPM, ternary and range matches. Some of these memory limitations can be sidestepped by using memory efficiently and carefully engineering packet header formats.

Chapter 3

Building Blocks for INC

There are a variety of devices that support INC, including: programmable switches, networked FPGAs and smart NICs [21]. This dissertation focuses on one type of device: the programmable switch. The premise is that we want to move application functionality into these programmable switches in the network.

Building blocks for INC bridge the gap between hardware capabilities and application functionality. Finding the right level of granularity for these building blocks will ensure that they are reusable across applications, while also providing performance benefits to the applications.

This chapter begins with an overview of INC capabilities, in order to characterize the application functionality that switches can support (Section 3.1). Then, Section 3.2 identifies some building blocks that applications can use to move their functionality into the network. Finally, this chapter describes the INC techniques for implementing these building blocks (Section 3.3).

3.1 INC Capabilities

Broadly speaking, we can compare INC devices along four dimensions as shown in Table 3.1: 1) network location and fanout; 2) throughput performance (I/O); 3) memory for storage (state); and 4) processing flexibility (expressiveness). Programmable switches are centrally located in the network with a high degree of fanout. This vantage point enables them to process traffic flowing between many hosts in the network. On the other hand, FPGAs and smart NICs are installed on end-hosts at the edge of the network, and usually only have a couple of ports, which are connected to edge switches. Unlike the switches, NICs only see traffic local to the end-host.

Smart NICs generally allow for a moderate degree of flexibility. Networked

	Network Location	I/O	Parsing Depth	State	Expressiveness
Programmable Switch	Center	Tb/s	Fixed	MB	Moderate
Networked FPGA	Edge	Gb/s	Flexible	MB + host DRAM	High
Smart NIC	Edge	Gb/s	Fixed	MB + host DRAM	Moderate/High

Table 3.1. Characteristics of various INC hardware devices.

FPGAs (like the NetFGPA) are more fleixble, because they can be programmed with arbitrary circuits. However, flexibility and storage is traded-off for throughput performance. Programmable switches have a high fanout and process packets at rates an order of magnitude higher than those of smart NICs. This comes at the cost of expressiveness. Although the memory on switches is fast, it is not as abundant as that found on a typical server. The computation on switches is flexible, but there is a limit to both the *number* and *types* of operations that can be performed.

Based on the hardware capabilities described above, we observe that applications best suited for INC on switches are those that require: a global vantage point in the network; high I/O throughput; and a moderate amount of state and expressiveness. Moreover, the ratio of I/O to expressiveness is an important indicator of application suitability for INC: the most suitable applications are those which are I/O bound.

The capabilities are not equally important to all the applications; some capabilities are more important to some applications than others. We rate the importance of a capability to an application using a subjective scale of low, medium or high. Below we briefly describe each type of capability, and provide the definitions for low, medium and high used for classifying the functionality provided by building blocks in the next section.

Centrally Located. This captures the value the application logic has from being in a position in the network that lets it see traffic between many hosts.

- Low: the switch need not be positioned between servers. It is only for offloading compute, similarly to how an FPGA or GPU is used for acceleration.
- Medium: it is important to be on the path between servers, but it does not necessarily need to see all packets in the system.
- High: it is essential that all packets travel through the switch.

Throughput. This characterizes the rate at which data travels through the device, measured in bits/second. Under certain conditions (e.g. with smaller packet sizes), the throughput may change.

- Low: messages (packets) travel through the device at a relatively low rate.
- Medium: packets pass through at a higher rate (on the order of Gb/s).
- High: packets pass through at rates on the order of Tb/s.

Parsing Depth. This is the amount of bytes in the packet that can be accessed by the device. Programmable switches and smart NICs can access a fixed maximum amount of bytes, while networked FPGAs can read deeper into the packet at the expense of other functionality.

- Low: only the beginning of the packet is inspected to make forwarding decisions.
- Medium: some application payload data is accessed.
- High: most of the data in the packet is accessed.

State. This is the amount of memory the application uses on the switch. Implementing some data structures may require more memory (see Section 3.3.1).

- Low: very little state is stored on the switch. A meter may be used to keep track of how many packets the switch has seen.
- Medium: some high-level application state is stored, like counters or events.
- High: application payload data is stored on the switch.

Expressiveness. This captures the logic being executed on the switch: how many operations are performed, the types of operations and some advanced techniques (see Section 3.3.2).

- Low: only table lookups are performed, matching a few packet header fields.
- Medium: more operations are executed, but mostly simple (e.g., add, subtract). The data structures are primitives like counters of boolean flags.
- High: enables programming techniques expected from higher-level languages (e.g., iteration) and manipulation of more complex data structures.

3.2 Building Blocks

Based on the hardware capabilities outlined in the previous section, we have identified building blocks for applications to leverage INC. These building blocks are possible to implement with INC (Table 3.2 lists examples of systems that implement each type of building block) and exploit the comparative advantage of the hardware (Table 3.3 shows how important the hardware capabilities are for each building block, using the scale described in the previous section). Other classifications of INC have been proposed, with different categories [22], as well as lower-level primitives [23].

Below, for each building block, we explain: why it is suitable for INC; why its level of granularity is appropriate; and the application-level information in requires.

Messaging. Programmable switches were initially designed to be flexible forwarding devices. This seems like an obvious match for messaging systems, which require expressive forwarding. Also, they do not require too much state to be stored on the switch, but some expressiveness to match the message to the intended destination. To be application-agnostic the switch should not need to parse deep into the data in messages. Instead, the switch forwards based on message metadata that indicates the content of the message.

Performance Management. Since switches have a global perspective of data flow, they are ideal for network measurements. In-band Network Telemetry (INT) collects information about traffic, including congestion (queue occupancy), latency and throughput. This information can help make decisions on managing resources and detecting problems. Programmable switches are suitable because they can collect information about every single packet. Since this can be done transparently, it does not require modifications to application; the INT information can be collected out-of-band by the application.

Load Balancing/Partitioning. Because switches have a high degree of fanout, they are in a good position to direct traffic for load balancing. Partitioning the workload generally involves inspecting packets for application-level information, which can be performed by programmable switches. Like messaging systems, the application-level information is exposed to switch through metadata in the packet. This makes the building block reusable for different applications.

Analytics. These workloads typically have to process large amounts of data. This is a good fit for programmable switches, which are I/O machines. The main limitation for implementing an analytics query on the switch is expressiveness. Because applications execute different queries, this building block needs to be tailored to the application. It also has to be tightly integrated to the application to know how to interpret the data in the packets.

Caching. Because switches are on the path to backend servers, they can reply to data requests before they even reach the servers. This reduces the latency experienced by clients, while reducing load on servers. Furthermore, looking-up values is fairly straight-forward. The main limitation is the amount of memory for storage available on the switch. Since caching just stores data and does not need to perform complex operations, it does not need to be tightly integrated

Building Block	Example System
Messaging	**Packet Subscriptions**(§ 4), R2P2 [24]
Perf. Management	INT [25], Dapper [26], SwitchPointer [27]
Load Balancing	SilkRoad [28], **Transaction Triaging** (§ 8)
Analytics	Marple [29], Sonata [30], **Linear Road** (§ 5), **PPS** (§ 6)
Caching	SwitchKV [31], NetCache [32]
Coordination	NetPaxos [33], NetChain [34], Eris [35], **NOCC** (§ 7), HovercRaft [36]
Aggregation	DAIET [37], NetAccel [38]

Table 3.2. Examples of systems that implement each building block. The systems described in this dissertation are in bold.

with the application. In fact, in many cases caching can be done transparently.

Coordination. Components of distributed systems communicate with each other to make decisions. Programmable switches are in the network where they are able to see this communication and accelerate it. This is not typically a high throughput workload, but requires low, predictable latency, which the switch ASIC provides. Coordination can provide a service that is application-agnostic (e.g., locks, sequencing), so it does not necessarily have to be tightly integrated with the application.

Aggregation. Switches have a high degree of fan-in, which makes them ideal for collecting and merging data from many sources. Aggregation involves merging and accumulating data. Depending on the type of merge operation and the size of the accumulated state, the expressiveness of switch operators and memory available can be the most important requirements. Compared to some other building blocks, aggregation requires tighter integration with the application, because it needs to execute aggregation operators on application-specific data.

The building blocks described above are suitable for INC because it is possible to implement them with INC, and because there is a comparative advantage of running them in the network hardware. There is, however, some application functionality that is unsuitable for INC. This is either because of inadequate INC capabilities (e.g., the application needs more storage than available on switches), or because there is no comparative advantage (e.g., the application does not fully leverage the switch I/O). To justify the selection of building blocks above, we provide some counter examples of building blocks which are *not* suitable for INC.

	Centrally Located	Throughput	Parsing Depth	State	Expressiveness
Messaging	high	low	low	low	low
Perf. Management	high	high	low	medium	low
Load Balancing	low	low	low	low	low
Analytics	low	high	high	medium	high
Caching	high	medium	medium	high	low
Coordination	low	low	low	medium	high
Aggregation	high	medium	medium	high	high

Table 3.3. Importance of INC capabilities for building blocks. The scale is explained in Section 3.1.

Cryptography. A switch could be used to perform encryption/decryption for applications running on servers. There are two challenges, however. First, the device would have to implement to cryptographic functions. This would either require specialized circuits in the hardware, or programming them, which would be inefficient and consume all the switch resources. Second, there are security concerns of configuring and managing encryption keys in the network. An alternative which does not require specialized hardware is random linear network coding [39].

Block Storage. Although INC can be used to accelerate caching and coordination for storage systems, it would be unrealistic to store persistent data on switches. Current switches do not have enough memory, and that memory is not persistent.

Serverless Lambdas. Microservices may seem like a good fit for INC because they perform short-lived computation that requires little state. However, they tend to be application-specific, with a frequently changing codebase that would be too large to fit on a switch. Furthermore, it may be unjustifiable economically if the lambdas underutilize the throughput capability of the switch.

3.2.1 The Building Blocks in this Dissertation

To support the research hypothesis of this dissertation, we chose to narrow our focus to just some of the building blocks. We had two main criteria for choosing them. Firstly, we chose building blocks types that push INC to its limits. One of the most difficult requirements to satisfy is expressiveness. Second, we chose ones that have not been fully explored by other researchers.

Thus, we chose to focus on: messaging (Chapter 4), analytics (Chapters 5 and 6), coordination (Chapter 7) and load balancing (Chapter 8). The variety of this selection is meant to demonstrate how different application requirements can be satisfied by INC. Furthermore, it demonstrates how these systems can be built using INC data structures and implementation techniques, which we describe in the next section.

3.3 INC Techniques

The diverse INC building blocks can be implemented on the switch using a set of common techniques. Below we outline the main data structures and implementation techniques, and how they map onto the underlying architecture.

3.3.1 Data Structures

Maps. Some applications need an associative data structure to store data (e.g., key-value mapping for a key-value store). The most straight-forward way of implementing this is indexing into an array (e.g., see § 7.3). It can also be implemented as a hashmap using the switch's hashing functionality.

Counters. Counters build on the map data structure. Application-level information is used to access a counter in a map, which can be read and/or incremented. For example, our Linear Road implementation counts the number of vehicles in a road segment (see § 5.3.1).

Sets. Applications that need to represent set membership use bit fields. This is easy to implement using ALU bit arithmetic. However, they are limited by the width of the bit field. Applications that tolerate approximation can use bloom filters.

Automatas. Some applications have complex processing logic with many steps. Automatas, such as finite-state machines (FSMs) can be used to represent this logic. This is done by storing state transitions as tables. The advantage of this representation is the state machine can be quickly updated from the control plane, without re-compiling the program. Chapter 4 describes this in more detail.

Queues. There are two main places to store queues: in switch memory; or in the packet. On the switch, queue elements can be laid-out along the pipeline, starting with the tail in the first stage, and growing the tail in subsequent stages (see § 8.3). On the switch we are using, the size is limited by the number of stages. Queues can also be stored in the packet, using the P4 abstraction of header stacks: headers can be pushed or popped from the packet (see § 7.3).

3.3.2 Implementation Techniques

Recirculation. The PISA pipelined architecture does not provide any natural constructs for iteration. So, to simulate iteration, switch programs can recirculate packets through the pipeline (see § 5.3.1). This comes at the cost of switch bandwidth, which is explained in Section 6.5.

Loop unrolling. This is an alternative to recirculation for simulating iteration. The iterations of a loop are laid-out along the stages of the pipeline. This has better throughput than recirculation, but the number of iterations is limited by the number of stages in the pipeline (see § 5.3.1).

Threading state. Programs need to keep per-packet state as the packet travels through the pipeline. There main ways of doing this. State can be stored in metadata which travels through the pipeline alongside the packet. This uses additional resources because the packet data, as well as metadata, must move through the pipeline. An alternative is to store state directly in the packet. For example, assuming the switch only receives IPv4 packets, the Ethernet type field will always have the same value, so it can be used temporarily to store state, and then restored at the end of the pipeline.

3.4 Summary

There are various types of INC hardware devices. In this dissertation we narrow our focus to only one type of device: programmable switches. In this chapter we characterized the types of services that can benefit from INC, and outlined the building blocks necessary for building these services on programmable switches.

In the next chapters we present the INC services we built. For each service, we describe how it uses the building blocks described in this chapter, as well the implementation techniques. We then explain how applications use each service. Finally, we evaluate each service with a combination of microbenchmarks and real-world workloads.

Chapter 4

Publish/Subscribe

In this chapter, we explore how programmable data planes can provide a higher-level of service to user applications via a new abstraction called packet subscriptions. Packet subscriptions generalize forwarding rules, and can be used to express both traditional routing and more esoteric, content-based approaches. We describe a compiler, called Camus, for packet subscriptions that uses a novel algorithm with binary decision diagrams (BDD) to efficiently translate predicates into P4 tables that can support O(100K) expressions. Using our compiler, we built a proof-of-concept pub/sub financial application for splitting market feeds (e.g., Nasdaq's ITCH protocol) with line-rate message processing.

This service demonstrates usage of automatas (the BDD), as well as the importance of being centrally located in the network: it is able to filter messages early, reducing the load on down-stream switches and end-hosts.

4.1 Background

While the Internet is based on a well-motivated design [40], classic protocols such as TCP/IP provide a lower level of abstraction than modern distributed applications expect, especially in networks managed by a single entity, such as data centers. As a case in point, today it is common to deploy services in lightweight containers. Address-based routing for containerized services is difficult, because containers deployed on the same host may share an address, and because containers may move, causing its address to change. To cope with these networking challenges, operators are deploying identifier-based routing, such as Identifier Locator Addressing (ILA) [41]. These schemes require that name resolution be performed as an intermediate step. Another example is load balancing: to improve application performance and reduce server load, data centers rely on complex software systems to map incoming IP packets to one of a set of possible service end-points. Today, this service layer is largely provided by dedicated middleboxes. Examples include Google's Maglev [42] and Facebook's Katran [43]. A third example occurs in big data processing systems, which typically rely on message-oriented middleware, such as TIBCO Rendezvous [44], Apache Kafka [45], or IBM's MQ [46]. This middleware allows for a greater decoupling of distributed components, which in turn helps with fault tolerance and elastic scaling of services [47].

Although the current approach provides the necessary functionality—the middleboxes and middleware abstracts away the address-based communication fabric from the application—the impedance mismatch between the abstraction that networks offer and the abstraction that applications need adds complexity to the network infrastructure. Using middleboxes to implement this higher-level of network service limits performance, in terms of throughput and latency, as servers process traffic at gigabits per second, while ASICs can process traffic at terabits per second. Moreover, middleboxes increase operational costs and are a frequent source of network failures [48, 49].

4.2 Example: ITCH Market Feed

To motivate the design of packet subscriptions, we introduce a running example from the financial domain. Nasdaq publishes market data feeds using the ITCH format [50]. ITCH data is delivered to subscribers as a stream of IP multicast packets, each containing a UDP datagram. Inside each UDP datagram is a MoldUDP header containing a sequence number, a session ID, and a count of the number of ITCH messages inside the packet. There are several ITCH message types. We focus on *add-order* messages, which indicate a new order accepted by Nasdaq. It includes the stock symbol, number of shares, price, message length, and a buy/sell indicator.

Financial companies subscribe to the Nasdaq feed and broadcast it to all of their servers in order to execute trading strategies. Each server, in turn, may also publish a new feed expressing its strategy, letting other servers react. Typically, each server is only interested in a very small subset of stocks. For example, one trading strategy might only depend on Google stock data, while another might depend on Apple.

To be concrete, let's imagine that one trading strategy is only interested in ITCH messages that match a certain criteria:

I'd like only stock buy orders for Google with a bidding price larger *than $2.00.*

It almost goes without saying, but in this domain, time is money, as high-frequency trading strategies depend on speed to gain an advantage in arbitraging price discrepancies. How could we perform this filtering while ensuring low latency?

Conventional Approach 1: Multicast. One approach to implementing the market data feed filter would be to use IP Multicast. A multicast group would be associated with a certain portion of the data space (e.g., Google stocks) and would effectively represent a rendezvous point between publishers and subscribers. A publisher would send a packet to all the groups that the packet's content belongs to. A subscriber would join all the groups that express the subscriber's interests. However, this approach has fundamental limitations. Minimizing traffic would require many groups, which are a precious resource, and would in any case overwhelm switches. Also, this solution places a burden of evaluation logic on publishers (which may not be in the same administrative domain) to choose between many fine-grained groups, or on subscribers to filter out uninteresting packets from a few coarse-grained groups, or on both. In any case, finding a good allocation of groups (or "channelization") is a well-known NP-hard problem [51].

Conventional Approach 2: Kernel Bypass or Smart NIC. A second approach is to multicast messages to the end-hosts, which would have to evaluate subscription conditions. This processing overhead can be reduced, typically by using kernel-bypass technology or a smart NIC. With further careful engineering, this solution allows for the processing of market data with extremely low latency. In fact, it is more or less what is deployed by algorithmic traders [52]. Putting aside the cost of the engineering effort (or of the smart NIC hardware), this technique places a significant burden on subscribers, where a processing core must be dedicated to filtering. Also, at high throughput, there will be congestion both at the queue in the NIC and within the network, where it is particularly wasteful since most packets are later filtered anyway. This congestion increases latency and the chances of packet drops.

To demonstrate, we conducted an experiment with filtering performed at an end-host using DPDK. We measure the end-to-end message latency for identifying messages with the stock symbol `GOOGL`. For accuracy, the producer and consumer ran on the same server but isolated using separate CPUs and with reserved hugepages and NICs. When the producer sent messages at 1Mpps, the application experienced low latency. This is because the receive queue size at the end-host remained small. Then, we increased the throughput to 8.25 Mpps, which is 90% of the application's maximum throughput. As the CDF in Figure 4.1 shows, the tail latency increases when sending at the higher rate because the end-

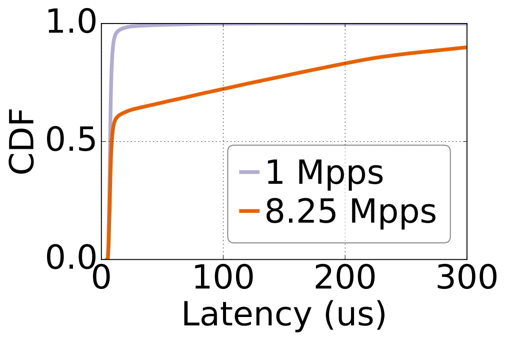


Figure 4.1. CDF of DPDK ITCH filtering.

host receive queue grows. Again, one could address this problem by allocating more cores to the task, at the expense of these additional hardware resources.

4.3 Packet Subscriptions

A packet subscription is a filter that determines whether a packet is of interest, and therefore whether it should be forwarded to an application. So, when end-points submit a packet subscription to the global controller, they are effectively saying “send me the packets that match this filter”. The following is an example of a filter:

```
ip.dst == 192.168.0.1
```

It indicates that packets with the IP destination address `192.168.0.1` should be forwarded to the end-point that submitted this filter.

One can interpret this filter the traditional way: each host is assigned an IP address, and the switches forward packets toward their destinations. However, in this traditional interpretation, the network is responsible for assigning IP addresses to end-points. Instead, with packet subscriptions it is the application that assigns IP addresses. In other words, packet subscriptions empower applications with the ability to define the routing structure for the network.

Another interpretation is that the subscription is equivalent to joining a multicast group with a given IP address. However, with packet subscriptions, the IP address has no particular global meaning, and instead it is just another attribute of the packet. Applications can use other attributes for routing, and in particular they can express their interests by combining multiple conditions on one or more attributes.

$h \in$ *Packet headers*	$n \in$ *Numbers*
$f \in$ *Header fields*	$s \in$ *Strings*
$v \in$ *State variables* (e.g., `my_counter`, see Figure 4.3)	
$g \in$ *State aggregation functions* (e.g., `avg`)	
$c ::= c_1 \wedge c_2 \mid c_1 \vee c_2 \mid !\ c \mid \mathrm{e}$	Filter: logical expression
$e ::= a > n \mid a < n \mid a == n \mid \ldots$	Numeric constraint
$\mid a\ \textit{prefix}\ s \mid a == s \mid \ldots$	String constraint
$a ::= h.f \mid v \mid g(v_0 \ldots v_n)$	Attributes

Figure 4.2. Packet subscription language abstract syntax.

For example, suppose that a trading application is interested in ITCH messages about Google stock. The following filter matches ITCH messages where the `stock` field is the constant `GOOGL` and the `price` field is greater than 50:

```
stock == GOOGL  ∧ price > 50
```

The filters we have seen so far are stateless: the condition does not depend on previously processed data packets. However, packet subscriptions may also be stateful. A stateful filter may read or write variables inside the switch data plane:

```
stock == GOOGL ∧ avg(price) > 60
```

In addition to checking equality on the `stock` field, this filter requires that the moving average of the `price` field exceeds the threshold value 60. The macro `avg` stores the current average, which is updated when the rest of the filter matches.

In general, a packet subscription is a logical expression of constraints on individual attributes of a packet or on state variables (see Figure 4.2). Each constraint compares the value of an attribute or a state variable (or an aggregate thereof) with a constant, using a specified relation. The Camus subscription language supports basic relations over numbers (e.g., equality and ordering) and over strings (e.g., equality and prefix).

Camus also supports stateful predicates, but to a limited extent. First, Camus can only evaluate predicates that reason about local state. It cannot filter on global state (e.g., the sum of values at more than one device). Second, reevaluating stateful predicates on multiple devices can lead to unexpected results (e.g., the average of the average of the average). Therefore, Camus only evaluates stateful functions at the last hop switch before a subscriber. And, third, due to the underlying hardware constraints, the types of computations Camus can perform is limited. For these reasons, the stateful functions that Camus supports are restricted to basic aggregations over tumbling windows, including count, sum, and average. This is similar to systems such as our Linear Road implementation (§ 5) and Sonata [30].

4.4 Compiling Subscriptions

Compilation is divided into two steps: *static* and *dynamic*. The *static* step is performed once per application, and generates the packet processing pipeline (i.e., packet parsers and a sequence of match-action tables) deployed on the switch. The *dynamic* compilation step is performed whenever the subscription rules are updated, and generates the control-plane entries that populate the tables in the pipeline.

Note that this compilation strategy assumes long-running, mostly stable queries. Highly dynamic queries would require an incremental algorithm, both to reduce compilation time and to minimize the number of state updates in the network. Prior work has demonstrated that such incremental algorithms are feasible. BDDs—our primary internal data structure—can leverage memoization [53], and state updates can benefit from table entry re-use [54].

4.4.1 Compiling the Static Pipeline

In general, a packet processing pipeline includes a packet parsing stage followed by a sequence of match-action tables. The compiler installs a different pipeline for each application, as different applications require different protocol headers, packet parser, and tables to match on header fields.

To generate the static plane, users must provide a message format specification. The specification is based on data packets structured as a set of named attributes. Each attribute has a typed atomic value. For example, a particular ITCH data packet representing a financial trade would have a string attribute called `stock`, and two numeric attributes called `shares` and `price`.

Figure 4.3 shows the specification for the ITCH application. The message format specification extends a P4 header specification with annotations that indicate state variables and fields that will be used by the filters. In the figure, lines 12-13 contain annotations indicating that the fields `shares`, `price`, and `stock` from the `add_order` header will be used in subscriptions. Thus, the compiler should generate P4 code that matches on those fields. As an optimization, users may specify the match type. The annotation on line 13 specifies that the match should be exact by appending the suffix `_exact`. The annotation on line 14 declares a counter state variable. The first argument is the name of the counter

```
header itch_order {
    bit<16> stock_locate;
    ...
    bit<32> shares;
    bit<64> stock;
    bit<32> price;
}
@pragma query_field(itch_order.shares)
@pragma query_field(itch_order.price)
@pragma query_field_exact(itch_order.stock)
@pragma query_counter(my_counter, 100, 1024)
```

Figure 4.3. Specification for ITCH message format.

(`my_counter`) and the second is its window size (100us).

To support state variables, the compiler statically pre-allocates a block of registers that are then assigned to specific variables dynamically. The compiler also outputs the code to update state variables in response to subscription actions at periodic intervals—e.g., to implement the tumbling window used on line 14 in Figure 4.3. Notice that the use (read/write) of state variable is determined by subscription rules, which are not known statically. Therefore, the static compiler outputs generic code for various update functions, and the dynamic compiler effectively links subscription actions to that code. In particular, the dynamic compiler links an update action of the general form $v \leftarrow f(args)$ with a subscription action by associating that action to what amounts to pointers to v, f, and *args*. However, the dynamic compiler implemented in our current prototype only supports actions without arguments.

4.4.2 Compiling Dynamic Filters

A naïve approach for representing subscription rules would use one big match-action table containing all the rules—each rule would be encoded using a single table entry. However, this approach would be incredibly inefficient because the table would require a wide TCAM covering all headers but containing only a few unique entries per header. Furthermore, programmable switch ASICs only support matching a single entry in a table, but a packet might satisfy multiple rules. Hence, we would require a table entry for every possible *combination* of rules, resulting in an exponential number of entries in the worst case.

Instead, our compiler generates a pipeline with multiple tables to effectively compress the large but sparse logical table used by the program. To do this,

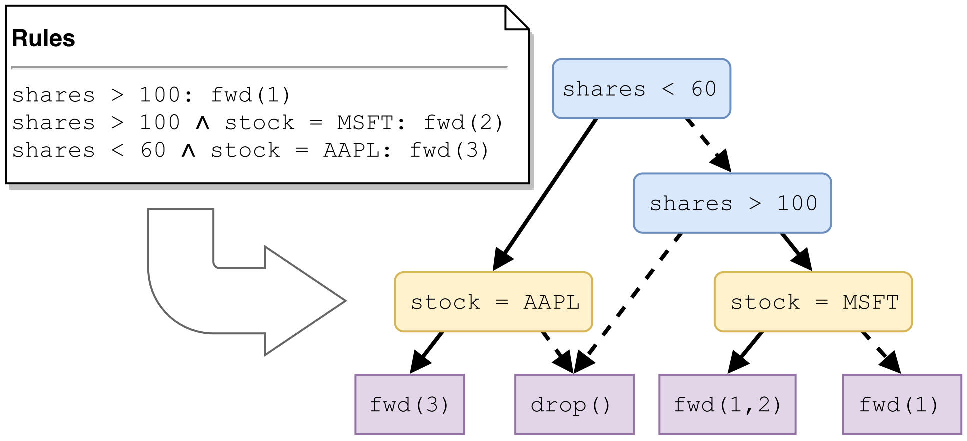


Figure 4.4. BDD for three rules. Solid and dashed arrows represent true and false branches, respectively.

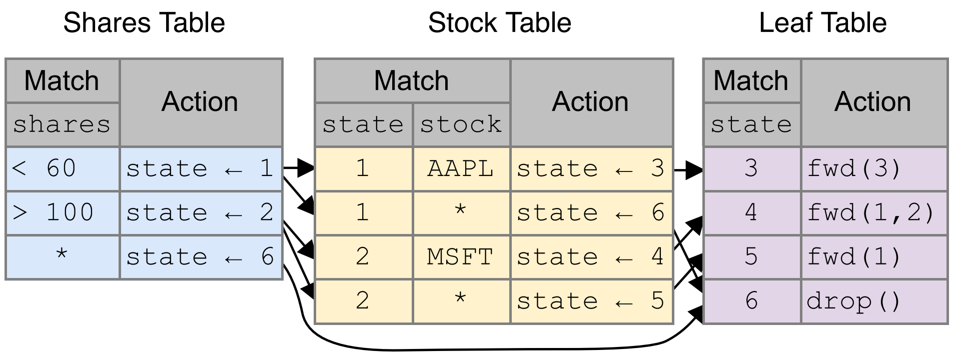
the compiler represents the subscription rules using a binary decision diagram (BDD) [55, 56]. BDDs are often used to obtain compact representations of functions on a wide input domain for which a single table would be too large. A BDD is a rooted acyclic graph in which non-terminal nodes encode conditions on the input (i.e., the packet headers), and terminal nodes encode the result (i.e., a set of actions). See the example in Figure 4.4.

The evaluation of the overall function of the BDD that encodes all subscription rules starts at the root node and recursively evaluates the conditions (if) at each node, proceeding to the true (then) or false (else) branch as appropriate. Evaluation terminates when it reaches a terminal node (actions).

We now briefly describe the algorithm for building a BDD out of subscriptions rules. What is important for our purposes is to define the structure of the BDD, so we can implement the BDD evaluation as a sequence of table lookups.

Representing Rules with a BDD. The subscription rules are first normalized into disjunctive form, yielding a set of independent rules in which the condition in each rule consists of a conjunction of atomic predicates. An atomic predicate is defined by an equals, greater-than, or less-than relationship between a field and a constant. For example, the rules in Figure 4.4 are in disjunctive normal form. The compiler then builds the BDD incrementally by evaluating the condition at each node using the Shannon expansion and adding nodes for the predicates in the condition as needed.

The compiler reduces the BDD using a combination of standard and domain-specific transformations. (i) If two nodes are isomorphic, one is deleted. The incoming edges of the deleted node are updated to point to the remaining copy. (ii) If both outgoing edges of a node point to the same successor, then that node



Shares Table

Match	Action
shares	
< 60	state ← 1
> 100	state ← 2
*	state ← 6

Stock Table

Match		Action
state	stock	
1	AAPL	state ← 3
1	*	state ← 6
2	MSFT	state ← 4
2	*	state ← 5

Leaf Table

Match	Action
state	
3	fwd(3)
4	fwd(1,2)
5	fwd(1)
6	drop()

Figure 4.5. Table representation of the BDD in Figure 4.4.

is replaced with the successor. (iii) If any ancestor n' of a new node n implies that n is always true or always false, then n is not added; instead, it reduces to a direct connection to its true or false branch, respectively. The overall effect is to share common structure and remove redundant nodes and unsatisfiable paths [57].

As is standard in ordered BDDs, the conditions in the BDD are arranged in a fixed order. For example, every path in the BDD of Figure 4.4 consists of a sequence of atomic predicates such that the conditions on field `shares` precede the conditions on field `stock`. This is essential for the representation and evaluation of the BDD as a sequence of table lookups, as we discuss next. The choice of an order can significantly impact the size of a BDD. Determining an optimal field order is NP-hard, but simple heuristics often work well in practice.

BDDs to Tables. The BDD can be seen as a state machine, where each state corresponds to a predicate, and the transition function is the evaluation of the predicate on the input packet. However, this naïve evaluation would require an excessively long sequence of evaluation steps. We instead implement BDD evaluation using a fixed-length pipeline.

Since every path in the BDD traverses predicates that consider fields in order, and that order is the same for every path, we use that ordering to effectively slice the BDD into a fixed number of field-specific components. Each component is a subgraph of the BDD that contains all and only those nodes that predicate on a particular field. By extension, we also consider the set of terminal nodes as a component. For example, the BDD in Figure 4.4 has three components consisting of the blue, yellow, and red nodes, corresponding to the `shares` and `stock` fields, and to actions, respectively.

We can now consider the evaluation of the BDD as a state-machine at the level of the field-specific components. Thus the transition function out of the component of field f depends on the value of field f in the packet. However, since the component of field f is a macro-state corresponding to potentially many

Algorithm 1: Translating BDD to Tables

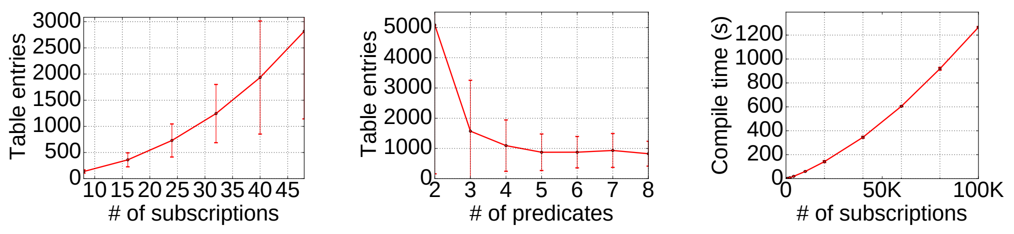
```
Input: The BDD graph, $G$
Output: A set of tables $T_f$ : $state \times dom(f) \rightarrow state$
1 foreach field $f$ do
2     $C_f \leftarrow$ subgraph of $G$ predicating on field $f$
3     $In \leftarrow \{n \in C_f$ with in-edges from outside $C_f\}$
4     $Out \leftarrow \{n \notin C_f$ with in-edges from $C_f\}$
5     foreach path $p = (u \in In, \ldots, v \in Out)$ in $C_f$ do
6         $range \leftarrow \top$                ▷ all allowable values for field $f$
7         foreach node $n \in p$ do
8             $range \leftarrow range \cap$ predicate($n$)
9         $T_f \leftarrow T_f \cup \{(u, range) \mapsto v\}$
```

states of the BDD, the transition function must also depend on the BDD state in which we enter the component. This entry BDD state and the value of field f are necessary and sufficient to determine the path through the component of field f and therefore the transition function for that component. We represent this transition function as a match-action table where we match on the entry state and on the value of field f, and where the action points to the next component and BDD state.

Figure 4.5 shows all the component-specific match-action tables corresponding to the transition functions for the BDD of Figure 4.4. The three tables also define the three-stage processing pipeline. The evaluation through the pipeline stores the current BDD state in metadata. The initial state is set to 0 and can be omitted from the first table. The actions set the entry state for the following stage, except for the *Leaf* table where the action corresponds to the overall BDD evaluation. For example, the rightmost path through the BDD in Figure 4.4 corresponds to the path through the 2nd, 4th, and 3rd rows of the *Shares*, *Stock*, and *Leaf* tables in Figure 4.5, respectively.

It is possible for multiple rules to match the same packet. For example, in Figure 4.4, the first two rules could match the same packet, so the actions `fwd(1)` and `fwd(2)` are merged into one action: `fwd(1,2)`. The compiler translates this to forwarding to a multicast group with ports 1 and 2.

We compute the transition tables with Algorithm 1. In essence, for each field-specific component C_f in the BDD, Algorithm 1 identifies a set of *In* nodes within C_f that are the destinations of all the edges that enter C_f from components of preceding fields, and a set of *Out* nodes outside C_f that are the destinations of all



(a) Subscriptions (b) Predicates (c) Compile time

Figure 4.6. Compiler efficiency

the edges that exit from C_f to components of succeeding fields. Then Algorithm 1 computes the transition table by iterating over all the paths that connect *In* and *Out* nodes. In general, a BDD could have an exponential number of such paths. However, the domain-specific optimizations we use guarantee that there is at most one path between any pair of *In* and *Out* nodes, which in turn guarantees that the number of paths is at most quadratic.

Resource Optimizations. One of the scarce resources in switching ASICs are TCAM memories that allow matching on a subset of bits in headers but consume large area of die and high power. The compiler uses three techniques that are application-agnostic to reduce TCAM usage. First, by default the compiler generates P4 code that implements range matches, which usually require an expensive TCAM lookup. However, the user can guide the compiler by specifying a matching type for each field that may not require a TCAM lookup. Second, matching on a range in TCAM is not scalable to hundreds of thousands of ranges as each range-match requires multiple TCAM entries ($O(\#bits)$). To cope with this, the compiler uses exact matches instead of range when possible, allowing it to leverage SRAM while saving TCAM. Third, some fields, like `shares`, will probably have only a few unique range predicates. The compiler can map values for that field and the corresponding range predicates onto a lower-resolution domain (e.g., 8-bits).

4.5 Evaluation

We have implemented a prototype compiler in OCaml. The compiler parses the application specifications written in $P4_{14}$ using the P4V library [58], patched to support our custom annotations. We use our own implementation of a multi-terminal BDD library with reduction optimizations.

There are three parts to our evaluation: (i) we explore the space/time efficiency of the compiler; (ii) we demonstrate the efficacy of packet subscriptions

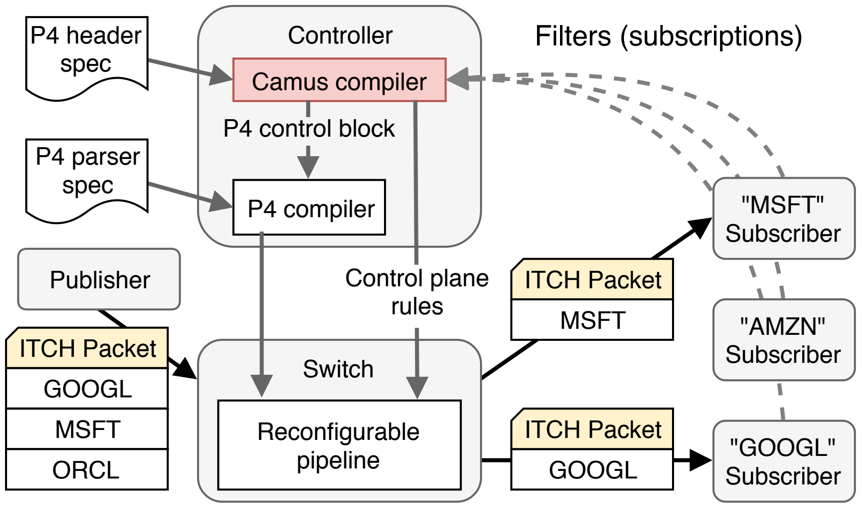


Figure 4.7. Overview of Camus.

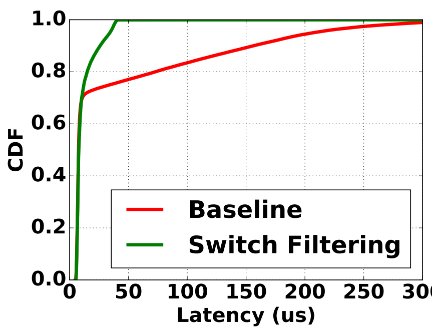
by implementing an in-network publish/subscribe system; and (iii) we compare the end-to-end system latency and throughput for in-network publish/subscribe to a baseline software implementation.

Efficiency of the compiler. To measure the space efficiency of the compiler, we generated workloads using the Siena Synthetic Benchmark Generator [59], which has been used to evaluate prior work in pub/sub systems [60]. Figure 4.6 shows the number of table entries required on the switch as we vary key parameters: (a) number of subscriptions; and (b) selectiveness of subscriptions (number of predicates).

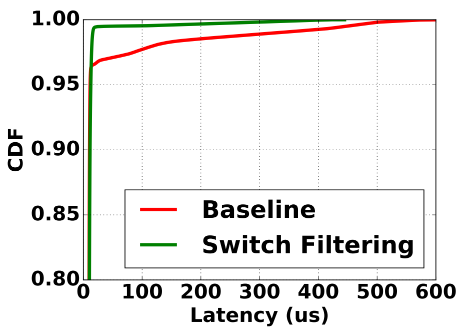
Given the low growth rate of table entries as workloads become more complex (4.6a), the experiments show that Camus uses available space effectively. Figure 4.6b shows that more selective subscription conditions (i.e. more predicates in the conjunction) require fewer table entries, which is because they result in fewer paths in the BDD.

To measure our compiler's runtime, we used a synthetic workload generator to create ITCH subscriptions of the form "`stock == S ∧ price > P: fwd(H)`", where `S` is one of a 100 stock symbols, `P` is in the range (0, 1000) and `H` is one of 200 end-hosts. Figure 4.6c shows the results. Compiling 100K subscriptions resulted in 21,401 table entries and 198 multicast groups, which can easily fit in switch memory.

Case Study: In-Network Pub/Sub. As an example use-case for packet subscriptions, we have implemented an in-network pub/sub system. Figure 4.7 illustrates the design of our pub/sub system, which we call Camus. Camus takes the subscription filters together with the message format specification, and generates



(a) Nasdaq trace



(b) Synthetic trace

Figure 4.8. ITCH experiments on hardware.

two outputs: (i) a P4 control block that specifies the control-flow and match-action tables in the pipeline, and (ii) a set of control-plane rules to populate the tables. The P4 compiler then takes the P4 parser specification (i.e., the packet format) and the control block generated by the Camus compiler to generate the switch image for the packet processing pipeline. At runtime, publishers send messages. The switches running the Camus pipeline process the messages and forward them to interested subscribers.

We used our Camus pub/sub system to do in-network filtering of market data feeds. Many financial companies subscribe to the Nasdaq feed and broadcast it to all of their servers in order to execute trading strategies. Typically, each server is only interested in a very small subset of stocks. For example, one trading strategy might only depend on data related to Google stock, while another might depend on data related to Apple. Therefore, broadcasting the feed wastes resources. Moreover, broadcasting all packets to servers builds queues at switches and servers, which increases delay and the chances of packet drops. Any increase in latency can have a significant impact on the user, as trading strategies depend on speed to gain an advantage in arbitraging price discrepancies.

Throughput and Latency. Our experimental setup resembles Figure 4.7, except for that the publisher and subscriber are collocated for accurate timestamping. We ran our packet subscription pipeline on a 32-port Barefoot Tofino switch, which can process packets at 3.25Tbps (on the 64-port version of the switch, we would support 6.5Tbps). The ITCH publisher and subscriber are implemented with DPDK [5], running on a server with an 8-core Intel Xeon E5-2620 v4 @ 2.10GHz CPUs, 256GB DDR4-2133 RAM and 25Gb/s NICs (Mellanox ConnectX-4 Lx and Intel XXV710-DA2). We ran the same workloads under two configurations. In the baseline configuration, the subscriber filters the feed for `add-order` messages with stock symbol `GOOGL`. In the second configuration, the filtering is done with Camus.

We used two workloads: a Nasdaq trace from August 30th 2017 and a synthetic feed. The number of messages of interest (i.e. for `GOOGL`) is 0.5% of the Nasdaq trace, and 5% of the synthetic feed. We measured the latency between the publisher sending a message with `GOOGL`, and it being received by the subscriber. Figure 4.8 shows the latency CDF for both workloads. For the Nasdaq trace, all messages arrived within 50us with Camus, compared to 300us for the baseline. For the synthetic workload, 99.5% of the messages arrived within 20us with Camus, compared to 96.5% with the baseline. Overall, filtering messages on the switch with Camus reduces the tail latency, allowing applications to meet their latency requirements under high throughput.

Other applications. We have focused on one running example for pub/sub, financial market data. We chose this application because it demonstrates many of the requirements that motivate packet subscriptions: routing based on application-specific content, stringent latency demands, and filtering based on expressive predicates. Because packet subscriptions can be used to implement pub/sub communication, they could be used as an alternative for frameworks like Kafka or ActiveMQ [61]. However, packet subscriptions are not a one-for-one replacement. They are limited, in that they do not provide features such as reliable communication and persistence. Rather, packet subscriptions are best suited for application domains with high-throughput workloads that can tolerate some loss, such as streaming analytics. Packet subscriptions would also be a useful abstraction for applications that require routing based on content identifier, including in-network caching (e.g., NetCache [32]), load-balancing, elastic scaling of services, and security.

4.6 Conclusion

Today, networks provide a lower level of abstraction than what is expected by modern distributed applications. The main argument of this chapter is that the emergence of programmable data planes has created an opportunity to resolve this incongruity, by allowing the network to offer a more expressive interface. The core technical contribution of this work is a set of algorithms for compiling complex filter expressions to reconfigurable network hardware using BDDs. These techniques are widely applicable to a range of network services. As a systems artifact, we have used these techniques to build an in-network publish-subscribe system that demonstrates predictable, low-latency packet processing using the full switch bandwidth.

Chapter 5

Stream Processing

This chapter explores the question: what abstractions are needed for INC to support a more general form of stateful processing? To address this question, we look for clues from the domain of stream processing. As a case study, we describe an implementation of the Linear Road benchmark for stream processing systems written in P4. The artifact of our implementation, which runs on a programmable ASIC, provides a version of the benchmark that far exceeds the throughput of any prior work. More importantly, the experience provides perspective on the challenges for implementing stateful abstractions in P4.

5.1 Background

In stream processing, also known as dataflow programming, an application is represented by a directed graph where the vertices are operators and the edges are streams. Streams are a continuous sequence of individual data items that contain attributes. When a vertex receives a data item, an operator is fired. An operator can merge multiple input streams, and split its input to multiple output streams. A stateful operator stores data between firings.

The stream processing abstraction has been used to build various systems, including the STREAM data stream management system [62] from Stanford University; the Aurora system [63] from Brandeis University, Brown University and MIT; and IBM's Infosphere Streams [64]. More recent systems include Apache Flink [65].

5.2 The Linear Road Benchmark

Linear Road is a simulation of a hypothetical application that computes tolls for vehicles on a highway system that consists of *L* expressways traveling from east to west. Each expressway is divided into 100 segments, each with five lanes, including an entrance and an exit ramp. Vehicles pay a toll when they drive on a congested highway (where the average speed of all vehicles is under 40 mph in a 5 minute span).

The benchmark queries receive input from both a set of continuous streams of data, and some pre-loaded, historical data referred to as relations. The four input data streams are:

- *Position Report*: a message periodically emitted by each vehicle with its current speed and location.
- *Account Balance Request*: a request for the sum of tolls assessed to a vehicle since the start of the simulation.
- *Daily Expenditure Request*: a request for the sum of tolls on a specific day from the historical data.
- *Travel Time Request*: a request for the estimated travel time between two segments.

The historical data includes the following two relations:

- *TollHistory*: for each day, for each vehicle, for each expressway, the total of tolls assessed.
- *SegmentHistory*: for every minute, the average speed and sum of tolls assessed in each segment of each expressway.

Each stream or relation has a schema that specifies the names of the attributes. For example, a *PositionReport* stream has the following schema:

```
(Time, VID, Spd, XWay, Seg, Lane, Dir)
```

where `Time` is the number of seconds since the start of the simulation, `VID` is the vehicle's identifier, `Spd` is the vehicle's current speed, and `XWay`, `Seg`, `Lane`, and `Dir` indicate the vehicles location (expressway, segment, lane, and direction, respectively). Arasu et al. provide a complete benchmark specification [66].

The benchmark defines five queries that compute the outputs from the input streams and relations. In prose, these queries are as follows:

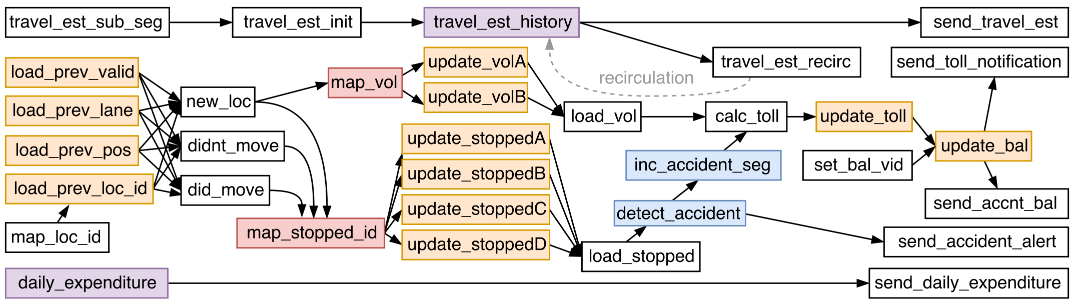


Figure 5.1. Tables and control flow of P4 Linear Road. Colors indicate a particular implementation technique.

- *Toll Notification:* Upon entering a segment of an expressway, a vehicle should be notified of the toll for that segment, which is based on the segment's level of congestion.
- *Accident Alert:* A vehicle travelling up to 4 segments upstream from an accident (detected as two or more vehicles stopped in the same lane) should be notified.
- *Account Balance:* Upon requesting an account balance, a vehicle should receive a response with the sum of tolls for that vehicle since the beginning of the simulation.
- *Daily Expenditures:* A request for the sum of tolls for a vehicle on a given day on a given expressway. This should be computed from the *TollHistory* historical data.
- *Travel Time Estimation:* Given a time of day and day of week, calculate the estimated travel time between two segments (computed from *SegmentHistory* historical data).

As a workload, we use sample input data available on the benchmark website [67].

5.3 P4 Linear Road

Using the P4 language, we implemented two versions of the benchmark: one that runs in the software Behavioral Model, and one that runs on the Tofino ASIC. Below, we describe the key implementation techniques, as well as some limitations in our implementation.

Our forwarding-plane version of Linear Road depends on the abstractions offered by the P4 language, which reflects the structure of the target hardware—i.e., the RMT architecture [11]. A RMT architecture has a pipeline of logical match-action units with local memory. Each match-action unit imposes a strict ordering on operations; all data reads must occur before all writes. There are also a number of physical constraints, e.g., a fixed number of match units in a pipeline; a limited amount of available SRAM and TCAM; and each TCAM can only return a single result from a match (i.e., the highest priority match).

Our implementation runs on a single switch which receives and emits streams of UDP packets. A stream tuple is encoded as a P4 header with fixed-width fields, including a field that specifies the tuple type (e.g. a *PositionReport* or *AccidentAlert*). Figure 5.1 illustrates the tables and control flow of our P4 program. The arrows indicate the direction packets flow through the pipeline. The colors indicate locations in the pipeline where we use different implementation techniques.

5.3.1 Implementation Techniques

To cope with the above constraints, our implementation relies on several techniques, which we describe below.

Incremental Operator Computations All queries must perform their computations incrementally. That is, for every input tuple (i.e., packet), the operator computes the differences in state relative to the previous operator invocation. This approach reduces memory usage, as the query only maintains a limited amount of incremental state. This technique is used in the orange tables in Figure 5.1.

For example, the *Toll Notification* query checks the number of vehicles in a segment, as well as their average velocity. This requires storing two state aggregates per segment: a counter for the number of vehicles, and an average of their speeds. If a *Position Report* indicates that a car crossed into a segment, then the previous segment's counter is decremented and the next segment's is incremented.

Explicit Loop Unrolling P4 excludes looping constructs, which are undesirable in hardware pipelines. Therefore, all loops in our queries must be explicitly unrolled. For example, the *Accident Alert* and *Toll Notification* queries must explicitly check the next four segments for stopped cars, which is done in the blue tables in Figure 5.1.

Multiple Register Arrays Based on the report from Sharma et al. [68], we assume that a single index of the same register can be accessed (first a read, then a write) in a stage. Therefore, to implement queries that need to access multiple indexes, we partitioned the data into multiple registers (highlighted red in Figure 5.1). For example, when a car crosses from segment 5 to 6, the query must decrement the volume of cars in the previous segment, and increment the next. Rather than express this as `segment[5]--; segment[6]++`, we keep two register arrays, one for even and one for odd segments: `segmentOdd[5/2]--; segmentEven[6/2]++`.

Pre-computed Historical State The purple tables in Figure 5.1 store data for answering historical requests. Using tables, rather than registers, simplifies lookups, since the match already implements the logic for reading by keys. The controller inserts the tuples from *TollHistory* and *SegmentHistory* into two separate tables. When the switch evaluates a historical query, the row matching the query is selected.

The *Travel Time Estimation* query selects multiple rows (one for each segment in a path). However, the match-action paradigm only returns one table entry per lookup. To find all the entries, our implementation recirculates the packet through the egress pipeline: on each recirculation the sum of estimated travel time is incremented with the next entry in the table, and finally the sum is sent in the output packet. This is indicated by the gray dashed arrow in Figure 5.1.

Another technique for selecting all the entries could be to create multiple replicas of the packet, assigning each a different key, and multicasting them to the egress pipeline. Each replica would match a different entry in the table, and update a common register. The final replica would read and output the accumulated value from that register. This technique would use the same amount of bandwidth as the one we implemented, but have lower latency, since the replicas could be processed in parallel. However, this would require an additional stage and a register to accumulate the partial result from each replica. It is only applicable if the accumulation operation is commutative (due to parallelization).

We assume memory is local to a pipe and not shared between pipes. To increase the amount of memory available, values can be partitioned among multiple pipes; if a query arrives in a certain pipe, but requires values stored in another pipe, the query can be recirculated to that pipe, as done in NetCache [32]. Of course, this requires re-circulation, and would therefore reduce throughput.

Over-allocation of Resources Our implementation stores vehicle state in registers. To lookup the state of a vehicle, the VID is used as an index into the registers. If VIDs are sparse, then register space will be wasted.

Passing State Through Stages A register array is stored in a specific pipeline stage, and thus can only be accessed in that stage. Since computation in a stage may require state that is read in a previous stage, P4 metadata is used to pass state between stages. After reading/writing state to a register, the state is also written to a metadata field, so that it can be used in a subsequent stage (e.g. for determining whether a query evaluation has been triggered).

5.3.2 Deviations from Specification

Our hardware implementation diverges from the original Linear Road specification in some relatively minor aspects:

Lane detection The original specification requires that an accident should be detected when two or more cars are stopped in the same *lane*; our implementation checks whether they are stopped in the same *segment*. This is due to the restriction on the number of stages in a RMT machine.

Time-based Average The original specification requires that the queries calculate the average speed in a 5 minute window. Doing so would require us to maintain values for 5 minutes. Maintaining a sliding window on an ASIC is difficult. So, we use a hardware-supported low-pass filter (LPF) to calculate the average using single exponential smoothing [1]. Since a LPF is not window-based, we cannot compare it to the specification's window approach for accuracy.

5.4 Towards a General Query Language

In the previous section, we described an implementation of the Linear Road benchmark in P4. We chose to focus on Linear Road for several reasons: (i) it has small scale in input data stream and queries, making a switch-based deployment feasible; (ii) it has a clearly defined semantics [66], allowing us to verify the correctness of our implementations; and (iii) it has been generally adapted

[1] $\text{avg}_n = \alpha \text{avg}_{n-1} + (1-\alpha)x_n$, where weight $0 < \alpha < 1$ for the nth observation x_n.

to a number of streaming engines (e.g., [62, 63, 69, 64]), indicating that it is representative of stream processing applications.

However, the more general question this work addresses is: *how feasible would it be to implement general abstractions from streaming languages targeting the RMT architecture via P4*? In this section, we expand our discussion to more general abstractions from the domain of stream processing systems that could be adapted for use in a programmable data plane.

There are, of course, many stream processing languages (e.g., [70, 71, 72, 69, 73, 74]). Although they are all different, we make two broad generalizations that we believe are useful. First, many stream languages distinguish between two types of inputs: data from time varying streams (which is updated continuously) and data from relations (which is mostly static). Second, many of them are based on streaming-specific extensions to SQL, which in term is based on relational algebra [75]. Below, we organize our discussion along these lines.

5.4.1 Input Data

Because the handling and storage requirements for transient, continuous data may differ significantly from historical data, it is useful to distinguish between two types of inputs: time varying streams and static relations.

Time Varying Streams Time-varying streams are data that arrives continuously in online fashion. Stream processing systems try to process this data with high throughput and low latency (average and tail). Programmable data planes are well-suited to processing this type of data, as it is similar to processing network packets. This data is typically associated with a timestamp, which must be either an explicit attribute (i.e., packet header field), or acquired from the hardware. P4 does not provide a built-in function for accessing a hardware timestamp, but this can be exposed by the architecture, such as Portable Switch Architecture (PSA).

Static Relations Static Relations are used to store historical data. Depending on the needs of the query, this data may or may not be pre-aggregated. There are two methods for storing static data with different trade-offs: (i) as pre-loaded data in action parameters stored in SRAM/TCAM match tables, or (ii) in registers, index by a key coming from a match table or a hash.

Match tables can only be programmed from the control-plane which has limited throughput. Although they cannot be used for instantaneous data, they perfectly suit the historical data where the pre-loading latency is not critical.

Registers are more general and can be programmed and queried from data-plane and control-plane. However, the number of registers and the bitwidth that can be read in each stage is an order of magnitude smaller than tables.

Beyond storing the data, there must be ways to quickly access the data with one or more *keys*. For both methods, keys can be generated from a match table or a hash function. Match tables do not need to save all possible combinations for historical data. Since the control-plane can pre-process the data, it can program only the keys used in the match tables, thus reducing memory consumption. On the other hand, if we use the hash for key, we do not need the match table. But, if key indexes are sparse, then memory might be over-allocated. Depending on the hash function and key space, there may be hash collisions. Depending on the needs of the application, such collisions may or may not be tolerable.

Note that the ASIC cannot guarantee that historical data is persistent. However, in most streaming systems, the application needs only to consult historical data when answering a query (i.e., persistent storage is not a requirement). So, data can be re-loaded into the device in the event of a memory failure or device restart.

5.4.2 Query Operators

Many streaming languages are based on SQL, which is based on relational algebra [75]. The standard relational operators include *set operations*, such as union and difference; *join operations* to correlate data; *filter operations*, such as selection and projection, and *aggregations*. Streaming languages extend these operators with *window operators*, which convert an input stream into a relation.

Set Operators Some operators perform set operations, such as union, difference or Cartesian product. Implementing these set-based operators would require some form of iteration. For example, one naïve implementation strategy would be to store data as “tables” in an array of registers. In this strategy, each “row” could be implemented as a P4 metadata structure, and operators would iterate over these structures to perform their computations. However, this design is impractical on a switch for two reasons. First, storing the individual tuples of the input stream requires a prohibitive amount of memory (SRAM). Second, iterating over the table rows would be difficult to express in P4 and could not be done at line-rate, as it would likely require re-circulation through the pipeline.

	Time-based	Count-based	Event-based
Sliding		✓	
Tumbling	✓	✓	✓

Table 5.1. Categories of windowed operators. Checks indicate windowing that is implementable in P4.

Join Operators Performing a join in hardware can be implemented if the sets being joined have constant sizes. In our Linear Road implementation, the vehicle state registers are joined on the VID. In this case, each register contains exactly one entry for each VID, so this operation can be implemented with a fixed amount of memory and without iterating.

Filtering Operators Selections and projections are straight-forward to implement as a single transformation. A selection simply matches on a field, for which switch hardware is specialized. Projections can be implemented by modifying fields, or adding or removing new fields (i.e. with the P4 `add_header()` primitive). Depending on the computational resources of the hardware (e.g. arithmetic operations supported by the ALU), some projections may not be possible.

Aggregation Operators Aggregates, such as average, count, sum and min/max, are computed over a window. The aggregate operators that require a fixed-size window can be implemented incrementally by updating the latest value, which is stored in a register array. To index into the register array, the input tuple's key (or hash of the key) can be used as an index. If the queries can tolerate bounded error, an approximate data structure (sketch) can be used. For example, to count the number of unique items, a cardinality estimator such as linear counting [76] or hyperloglog [77] can be implemented using hashing and registers in data plane [78, 79].

Window Operators Many streaming queries require window operations, which intuitively convert a stream into a relation. In general, there are many types of windows. Table 5.1 provides a summary of the main categories. A window may be *sliding* or *tumbling*. A sliding window is a series of fixed-sized, overlapping, contiguous time slices. The window advances by a *slide size*, and events outside of the slide are evicted, while new events are added. A tumbling window is a series of fixed-sized, non-overlapping, contiguous time slices. At the end of each interval, the window “tumbles” and all data items are evicted. Both sliding and tumbling windows may be time-based, count-based, or event based.

In P4/Tofino, it is feasible to implement count-based and tumbling windows as they both have a static size. Sliding time-based and event-based windows, however, require a dynamic window size, because the window can grow to an arbitrary size during the interval or before the event. They are thus impractical to implement, because of the memory and iteration constraints already described.

Linear Road makes use of sliding time-based windows (e.g. for calculating average speed in a 5 minute interval), and count-based tumbling windows (e.g. counting vehicles in the same segment). To approximate the sliding time-based window for average speed, in our implementation we use single exponential smoothing.

5.4.3 Summary

P4 primitives and data structures can be used to express many operators, albeit some more cumbersomely (e.g. with loop unrolling). We found that data planes are well-suited for streaming operations that do not require looping and that require a bounded amount of state. Overall, the stream processing model maps surprisingly well onto the P4 programmable switch hardware abstraction. That we were able to implement the whole Linear Road benchmark demonstrates the expressive power of this new programmable substrate.

5.5 Evaluation

We implemented two versions of the benchmark using $P4_{14}$: one that runs in the software Behavioral Model (BMv2) with 1,263 lines of code (LoC), and one that runs on the Tofino ASIC with 1,335 LoC. The code could be easily converted to $P4_{16}$. We created a Python library (560 LoC) that parses the sample workload data available on the Linear Road website [67], and outputs packets to be sent to the switch. The BMv2 version of the code is publicly available[2].

We deployed our Linear Road implementation on a 64-port ToR switch, with Barefoot Network’s Tofino ASIC [11]. As per standard practice in industry for benchmarking switch performance, we used a snake test: each port is looped-back to the next port, so a packet passes through every port before being sent out the last port. This is equivalent to receiving 64 replicas of the same packet. To generate and receive traffic, we used an Ixia XGS12-H hardware packet tester,

[2]`https://github.com/usi-systems/p4linearroad`

connected to the switch with 100G QSFP+ direct-attached copper cables. We verified that P4 Linear Road can process over 4 billion events/second. Furthermore, the P4 Linear Road packet processing pipeline has a fixed latency that is orders of magnitude lower than that of software implementations.

For comparison, the most recent published implementation of Linear Road [64] running on a single node (dual-core 3GHz Xeon CPU with 2GB RAM) could handle around 2M events/second with 1.67 seconds of latency. A more recent streaming engine, Drizzle [80], does not evaluate Linear Road, but reports 100M events/second for the Yahoo streaming benchmark, using 128 `r3.xlarge` Amazon EC2 instances. We report these numbers simply to give context for our performance; this is not an apples-to-apples comparison because these other systems have different capabilities (e.g. persistent memory, complex transformations, and fault-tolerance).

5.6 Conclusion

This chapter argues for using stream processing as a model for the types of abstractions we will need to support general stateful computations in programmable network hardware. This exercise not only provides a line-rate implementation of Linear Road, but also helps to identify constraints and challenges for stateful processing. It demonstrates that INC is suitable for applications that must perform computation on large quantities of data while they are in-flight, although it trades-off some functionality (e.g., sliding windows, dynamically data structures) and portability across applications (i.e. the switch runs queries that are specific to the application). As developers and network operators continue to explore ways to leverage this new hardware to offload or accelerate services, this work highlights the pressing need for new language abstractions.

Chapter 6

String Search

This chapter describes PISA Parallel Search (PPS), a system for locating occurrences of string keywords stored in the payload of packets. It motivates the use of INC for computational streaming workloads. Moreover, it describes the techniques for implementing algorithms on the switch. The PPS compiler first converts keywords into Deterministic Finite Automata (DFA) representations, and then maps the DFA into a sequence of forwarding tables in the switch pipeline. Our design leverages several hardware primitives (e.g., TCAM, hashing, parallel tables) to achieve high throughput. Our evaluation shows that PPS demonstrates significantly higher throughput and lower latency than string searches running on CPUs, GPUs, or FPGAs.

This system is similar to the one described in the previous chapter, in that they are both provide an analytics service. However, this system is more computational and makes more extensive use of the switch I/O.

6.1 Background

String searching is one of the most common and important functions run on computers. It is estimated that 80% of the world's data is unstructured [81], meaning that it cannot be easily queried using a fixed data model. Instead, users must search through large amounts of log data, JSON files, email, web pages and other documents to find the relevant patterns.

These searches can have a significant impact on application performance. For example, Palkar et al. [82] recently showed that using string searches to pre-filter data before feeding it into Spark can provide a 9× improvement in end-to-end application completion time.

Unfortunately, data is increasingly stored on devices that cannot provide good search performance. In order to improve the utilization of resources, many data centers have turned towards a disaggregated architecture [83], in which storage devices have weak CPUs and little memory. In the storage community, these devices are commonly referred to as JBODS—just a bunch of disks.

Besides the fact that these machines are “wimpy”, performing search also requires paying the levitation cost to get data off the disk. When running search on an x86 CPU, this will be through PCI-Express, which is a known bottleneck [84, 85]. The fourth generation of this interconnect achieves 128Gbps over sixteen serial links [86].

6.1.1 String Search Algorithms

There is a long history of research on string searching algorithms, and we are unlikely to improve the performance via a purely algorithmic solution. Instead, we seek to adapt an existing algorithm to best utilize the characteristics of the new domain-specific machine: the programmable networking ASIC. Below, we discuss the most well-known solutions, focusing on the suitability for PISA.

More formally, the string search problem is defined as follows. Let Σ be an arbitrary alphabet. Given a text string $t = t_1 \ldots t_n$ and the pattern string $p = p_1 \ldots p_m$, where each t_i and p_i are characters in Σ, then output the set of all positions in t where an occurrence of p starts as a substring.

The naïve algorithm iterates over every index of t, and checks if the string starting from that index matches p, running in $\Theta(nm)$ time. Searching for multiple patterns requires iterating over the string multiple times, once for each pattern. This algorithm could be implemented on PISA, but it would be unnecessarily slow.

The Boyer-Moore algorithm [87], used by Unix `grep` matches on the tail of the pattern, rather than the head, and uses information gathered in a pre-processing step to jump ahead multiple characters, rather than advancing one index at a time. This reduces the best case running time to $\Omega(n/m)$, but the worst case time is still $\mathcal{O}(mn)$. It also requires $\Theta(k)$ space, where $k = |\Sigma|$ is the size of the alphabet. The implementation would be similar to the naïve algorithm.

The Rabin-Karp algorithm [88] speeds up the comparison of the pattern with the substring of t by using a hash function. This reduces the running time to $\Theta(m + n)$ time. However, Rabin-Karp is not a good match for a PISA for two reasons. First, it requires computing the sliding hash of the text being searched. This would require a large number of hash units, which does not scale on hardware. Second, there may be hash collisions, in which case the algorithm reverts to comparing the two strings index by index anyways.

Instead, PPS is based on the Aho-Corasick algorithm [89]. Aho-Corasick, which was the basis of the original `fgrep` command, runs in $\mathcal{O}(n+m+z)$ time, where z is the number of matches. The algorithm constructs a finite-state machine that resembles a trie, with additional edges between nodes that share a common prefix.

6.2 Design Overview

At a high-level, PPS implements a finite-state machine in the pipeline of a PISA switch to search for patterns in the payload of packets. PPS extends this basic design with optimizations to effectively utilize the switch hardware.

The PPS state machine works at byte level granularity. Thus, PPS works with binary data or ASCII strings. It does not make any assumptions about the character encoding (e.g., UTF-8, UTF-16, etc.) or length of the pattern.

Our prototype implementation assumes that input packets have Ethernet, IP, and UDP headers. It begins searching at the UDP payload. However, this is not inherent to the design, and PPS could just as easily start the search from the beginning of the packet. PPS searches for patterns throughout the entire packet, using recirculation to examine deep into the payload. Furthermore, it assumes the switch is not oversubscribed and thus no packet loss.

If PPS detects a matching packet, it emits a new packet that contains a custom header, indicating which packet matched, as well as the offset of where the match occurred.

6.2.1 Expected Deployment

PPS can be deployed in two ways: as a *dedicated appliance*, or as a network switch that also does *bump-in-the-wire* processing. However, in order to search deep into the packet payload, PPS relies on re-circulation and it must discard the initial portion of the packet at each iteration. Therefore, a bump-in-the-wire deployment would require access to some external memory architecture to buffer packets [90]. When deployed as an appliance, the ASIC program is relieved of the responsibility of forwarding packets, and more resources can be dedicated to searching.

In either case, the switch is configured to run the PPS data plane pipeline. Conceptually, the pipeline consists of a sequence of tables that contain state machine transitions. Note that the pipeline is completely static. It is compiled once when PPS is deployed, and does not change when search patterns are updated.

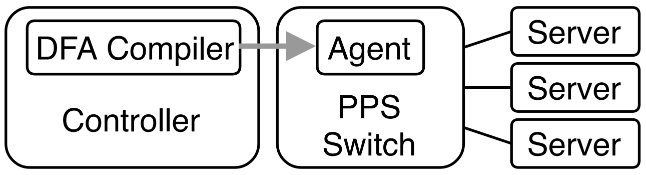


Figure 6.1. Expected deployment.

The PPS controller process compiles patterns and installs table entries on the switch. These entries are dynamic, and are re-generated whenever there is a new search pattern.

The controller process is divided into two parts: the server sends rules to the switch and the agent receives the rules and installs them via the control plane API. PPS uses a custom controller agent to reduce serialization overhead.

Applications that use PPS run on servers connected to the switch. In order to use PPS, they must send the data to be searched to the switch. Our prototype implementation includes a small library that reads a stream of data and sends it to the switch, one chunk per packet.

Figure 6.1 shows an example deployment, which is the same deployment used in the Spark experiments described in Section 6.6. There are three servers connected to the PPS switch. A fourth server acts as the controller process. The servers have multiple NIC interfaces—as is common in data center deployments—and the second interface is used to provide external network connectivity.

6.3 Pattern Compilation

The PPS compiler has two main tasks. First, it converts a set of search patterns into a *k*-stride DFA. Second, it maps the DFA into a pipeline of match action tables.

Patterns to *k*-stride DFA. PPS takes a set of search patterns as input. Patterns can be arbitrary regular expressions. However, all of our experiments use inputs that are finite strings (i.e., only concatenation operator), which are converted to DFAs. As we will discuss in Section 6.5, handling arbitrary DFAs is expensive.

PPS differs from the standard Aho-Corasick algorithm in that it uses a *k*-stride DFA. The *stride* size of a DFA refers to how many characters are read for each transition. For example, a stride size of 2 means that the implementation reads 2 character per transition.

Increasing the stride size increases the throughput. However, the throughput improvement comes at the cost of memory, as the size of the state transition table increases with the stride. With a stride size of s and an alphabet Σ, there are $|\Sigma|^s$ transitions per state.

The algorithm to convert a set of patterns to a k-stride DFA is as follows. The compiler first converts the patterns to a nondeterministic finite automata (NFA) using the Aho-Corasick algorithm [89]. It then transforms the NFA to a DFA using subset construction [91]. Finally, to convert the DFA to a k-stride DFA, the compiler computes the k-stride closure on each node in the 1-stride DFA—for each state s, a new transition is added for each state s' that is reachable in k characters from s. In Figure 6.2, the DFA on the right is the 2-stride equivalent of the DFA on the left. Both machines match the string "dog".

Patterns can occur at any offset within the input string, and patterns may not be multiples of the stride size, k. This means that some transitions need to ignore some characters to match the start and end of the string. For example, the 2-stride DFA in Figure 6.2 with pattern "dog" has start transitions `*d` and `do` ("*" matches any character). Likewise, some of the terminal transitions also include "*" and can be implemented with ternary matching.

k-stride DFA to MAU Pipeline. The DFA is then translated to the match-action abstraction. The switch has a pipeline of tables that is compiled once and can be used to execute any DFA. The DFA is installed in the tables at runtime.

The first step is to represent the DFA as a state transition table. Figure 6.3 shows the table corresponding to the 2-stride DFA in Figure 6.2. A naïve approach would be to store one big transition table in the pipeline that performs a single transition. However, this would limit the number of characters consumed per pipeline pass. Instead, we replicate the transition table on all stages. This way, multiple transitions are performed per pass, increasing throughput by the number of stages. Given the current state and the current k input characters, each stage transitions to the next state.

To store the DFA on the switch, we leverage different types of memory. DFA transitions that consume exactly k characters require an exact match, which is stored in SRAM. The start and end transitions that match less than k characters require ternary matching; these transitions are stored in TCAM. In each stage, first the exact match table is applied; if no entry matches, then the TCAM table is applied. If none of the tables match, then the default action is executed, setting the state to 0. This is an implicit transition to the start state.

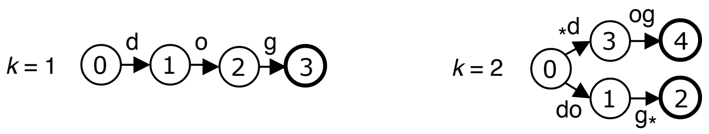


Figure 6.2. DFAs with different strides for "dog".

Match		Action
state	chars	
0	do	`set_state(1)`
3	og	`accept(4)`
1	g*	`accept(2)`
0	*d	`set_state(3)`

Figure 6.3. Table for 2-stride (k=2) DFA

6.3.1 Optimizations

Multiple DFAs. Memory is a scarce resource on switches. To reduce memory usage, we split the DFA into multiple smaller DFAs, which run in parallel on the switch. We do this by partitioning the patterns into multiple subsets, and constructing a DFA from each subset of patterns. The aggregate size of these DFAs (# transitions) is smaller than that of one large DFA containing all the patterns. This is because patterns with similarities can cause an explosion in the number of transitions. An optimal partitioning of the patterns yields a set of DFAs with the smallest aggregate size. Currently, we partition the patterns randomly multiple times and pick the best one. We chose to use 3 parallel DFAs in our pipeline because: it is the most number of splits before the returns diminish; and, coincidentally, it is the most that we can fit in the our hardware switch due to resource limitations.

Tunable pipeline. In the pipeline design outlined above, every stage in the ingress and egress pipeline performs a DFA state transition. To support more patterns, we can make two changes: (i) reduce the stride size of the DFA; and (ii) use the resources of multiple stages to perform a single transition. By reducing the stride size, the number of transitions in the DFA is reduced, producing a more compact DFA. By performing fewer transitions, we can combine the resources of multiple stages. For example, if the DFA representation does not fit within the resources of a single stage, we can split it across two stages. These optimizations come at the expense of throughput, which we explore in the evaluation.

6.3.2 Approximation

We observe that for some applications accuracy is not a strict requirement, such as with approximate streaming analytics [92]. We can trade-off accuracy for reduced memory usage by storing a hash of characters. Matching each character in the stride size, k, requires storing k bytes per transition. Instead, the switch can compute a CRC16 hash of the k characters. Some hash collisions are detected at compile time (i.e. two different transitions out of the same state have the same hash). In this case, we use perfect hashing: the compiler finds a different hash function (CRC with a different polynomial) that doesn't produce collisions, and updates the pipeline to use that hash function. Hash collisions that occur at runtime will produce false positives and false negatives.

6.4 Implementation

We have implemented a PPS prototype running on a Barefoot Tofino switch. At its core, the prototype includes a DFA compiler written in Python (365 LOC), as well as the DFA data plane program written in P4 (4754 LOC).

Controller. The Barefoot Networks switch agent offers a Thrift [93]-based API, which requires serializing and installing each table entry one-by-one. To reduce overhead, we implemented a custom agent that uses a binary serialization format and installs entries in bulk. This optimization reduces entry installation time from tens of seconds to milliseconds.

Data Levitation. Our prototype includes a client library that sends data to the switch. Because PPS searches one packet (chunk) at a time, it is most suitable for data that can be partitioned into chunks (e.g. lines, records). If the data is a continuous stream of bytes that cannot be partitioned, the client can format the data as *overlapping* chunks.

The client library is implemented in C and runs in userspace. It would be possible to use DPDK [5], which could support higher throughput and lower CPU load. Recent work by Kim et al. demonstrates that a Tofino switch can serve as an RDMA end-point [90]. One could imagine connecting the switch to JBODS via RDMA. Accessing the data from the storage servers would completely eliminate the use of the server CPU for searching, allowing it to focus on other storage tasks, e.g., error correction, de-duplication, etc.

6.5 Discussion

Our prototype provides significant performance benefits to applications as described in our evaluation. However, there are some limitations on the formatting of input data and the types of search patterns supported.

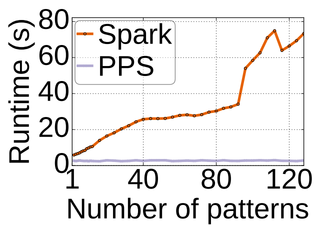
Input Alignment. Data chunks sent to PPS must be aligned to the packet. If a chunk spans multiple packets, a potential match split across the packets will not be detected. To address this, as described above, our client can generate overlapping chunks, which uses more bandwidth. To ensure that matches are detected, the overlap must be the size of the longest search pattern. Furthermore, the chunk must be less than the network Maximum Transmission Unit (MTU).

Recirculation. In one pipeline pass, PPS can search a fixed number of bytes in the packet: k times the number of pipeline stages. To search deeper, the packet is recirculated through the pipeline. At the end of each pipeline pass, the bytes that were just searched are truncated from the packet. For example, with 4 recirculations, the first pass searches the entire packet; the second $\frac{3}{4}$ of the packet; the third $\frac{2}{4}$; and the fourth $\frac{1}{4}$. Thus, one packet actually uses more bandwidth: $1 + \frac{3}{4} + \frac{2}{4} + \frac{1}{4} = \frac{10}{4}$. This is $\frac{6}{4}$ times the bandwidth for a single pass. The bandwidth overhead factor can be generalized to $\frac{n-1}{n+1}$, where n is the number of recirculations. Although this reduces the overall throughput, it is still orders of magnitude higher than that of other solutions (discussed in Section 9.5).

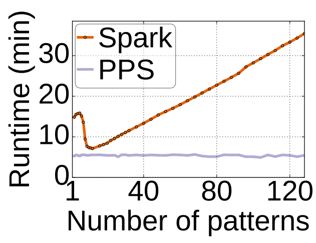
Generalizing to Regular Expressions. The technique we described for searching fixed patterns also generalizes to Regular Expression (RegEx) matching. Our compiler supports the Kleene star operator, alternation, and concatenation. It also supports character classes using alternation. However, we found that complex expressions result in a DFA state space explosion, using more switch memory. This is especially the case for character classes, which require exact transitions for all the characters in the class, which cannot leverage the ternary matching of TCAMs. To enable more efficient RegEx matching, it could be possible to translate each input byte to a symbolic value, reducing the number of table entries.

6.6 Evaluation

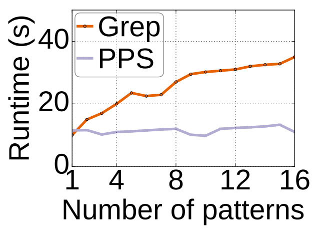
Our evaluation focuses on three questions: (i) How does PPS help with end-to-end application performance? (ii) How does PPS performance vary with the number of patterns? and (iii) How does PPS performance compare to state-of-the-art software and hardware solutions.



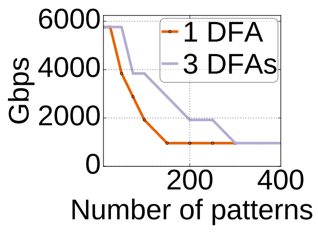
(a) Filtering e-mails.



(b) Filtering tweets.



(c) Searching a 12GB log file.



(d) Calculated tput feasibility

Figure 6.4. PPS end-to-end experiments and micro-benchmarks.

Experimental setup. We ran PPS on a 32x100G port Barefoot Tofino switch connected to a 4-node cluster with QSFP+ breakout cables. Each server has 12 cores (dual-socket 1.6GHz Intel Xeon E5-2603 CPUs), 16GB of 1600MHz DDR4 memory, and an Intel 82599ES 10Gb Ethernet controller. For our microbenchmarks, we randomly selected non-disjoint "content" patterns from the Snort [94] community ruleset.

6.6.1 End-to-end Application Performance

We used PPS to accelerate two Spark filtering jobs: scanning e-mails and filtering Twitter tweets. For the baseline, we use Spark SQL, which partitions the filtering among the worker cores. We compare this to a Spark program that pipes the data to PPS for filtering. In both cases, the Spark job executes a reduce stage that aggregates the sum of matching lines or JSON records. As we increase the number of patterns, the end-to-end runtime with PPS stays constant.

Scanning E-mails with Spark. We did a line-by-line search of the "Podesta E-mails": a collection of 50K e-mails (4.7GB) published by WikiLeaks in 2016 [95]. Figure 6.4a shows the results for searching an increasing number of patterns. Spark's execution time increases steadily and jumps above a minute after 96 (a multiple of 12 cores) because of query planning. PPS consistently searches the entire file in under 3 seconds.

Filtering Tweets with Spark. Inspired by the evaluation in Sparser [82], we filtered 212GB of JSON tweets collected using the Twitter Streaming API [96]. In addition to using unstructured patterns, we also searched for structured JSON key/values, e.g. `"lang":"ro"` and `"filter_level":"medium"`. Figure 6.4b shows the results. Spark SQL chooses a poor query plan with fewer than 12 patterns (which explains the dip); at 12 patterns, it has similar performance to PPS, but slowly reduces as the number of patterns increases. For 128 search strings, Spark takes 35.4 minutes, compared to 5.43 min for PPS (6.5x speedup).

6.6.2 Microbenchmarks

PPS vs Grep. For a direct comparison to `grep`, we measure the end-to-end time to search a 12GB log file. The file is stored in a RAM disk, because otherwise disk I/O (not compute) is the dominant factor in the search time. For PPS, the end-to-end time includes: (i) sending the patterns to the controller, (ii) compiling rules from the DFA, (iii) installing the rules and sending an acknowledgement to the client, and (iv) streaming the data to the switch. Figure 6.4c shows the search time for an increasing number of patterns. For a single pattern, both have similar performance. For multiple patterns, `grep`'s runtime increases, while PPS remains constant. This demonstrates that there are clear performance benefits, even including the overhead of sending data to the network.

Pattern Complexity vs. Throughput. Ideally, the switch would have an unlimited amount of memory that could hold as many patterns as necessary. However, in reality, to be able to process data at line rate, the switch has a fixed amount of memory. As the number (or complexity) of patterns increases, the stride size of the DFA has to be reduced to fit the DFA in the switch memory. To further reduce memory usage, we can also split the DFA into multiple DFAs (§ 6.3.1).

We calculated the throughput achievable for workloads with an increasing number of patterns. We randomly selected patterns from the Snort ruleset that are up to 32 characters long, which is the case for over 80% of the rules. We then compiled the patterns into a DFA with the largest stride size that would fit on the switch. Figure 6.4d shows the throughput using a single DFA (red) and multiple DFAs in parallel (green). Note that these are theoretical values that we calculated. It is reasonable to calculate the throughput (instead of testing it experimentally), because a compiled P4 program runs at the speed of the architecture with a fixed number of stages and bounded memory access time.

6.6.3 Comparison to State-of-the-Art

To provide context for PPS performance, we briefly report results from comparable state-of-the-art solutions. To be clear, these results are not direct benchmark comparisons and are not collected using the same workloads. Using a GPU, Hsieh et al. [97] reached 150Gbps for 20 Snort patterns. Titan IC's Helios ASIC [98] reports 100Gbps for 1 million rules. DFC [99] achieves 45Gbps using x86 servers. With a single recirculation and a stride size of 4, PPS can search 100 Snort patterns at 3.8Tbps on a 64-port Tofino.

6.7 Conclusion

PPS is inspired by the observation that PISA is well-suited for particular computing tasks. Some of the common characteristics of those tasks are: (i) the I/O to computing ratio is high, (ii) the space complexity of the computing algorithm (i.e., amount of memory required during computing) is low and independent of the size of the input workloads fed to PISA via I/O, and (iii) the computing algorithm is branch heavy. String search has these characteristics.

Searching in strings is a fundamental problem in computer science, and improving the performance of the algorithm can have significant impact on a wide variety of applications. We have described a set of implementation techniques that build on the classic Aho-Corasick algorithm, while efficiently utilizing hardware primitives (e.g., TCAM, hashing, parallel tables) to achieve high throughput string searching on a programmable network ASIC. Compared to state-of-the-art alternatives on CPUs, GPUs, and ASICs, PPS offers orders of magnitude improvements in throughput.

Chapter 7

Optimistic Concurrency Control

Optimistic concurrency control (OCC) is inefficient for high-contention workloads. When concurrent transactions conflict, an OCC system wastes CPU resources verifying transactions, only to abort them. This chapter describes a new system, called Network Optimistic Concurrency Control (NOCC), which reduces load on storage servers by identifying transactions that will abort as early as possible, and aborting them before they reach the store. NOCC leverages in-network computing to speculatively execute transaction verification logic. NOCC examines network traffic to observe and log transaction requests. If NOCC suspects that a transaction is likely to be aborted at the store, it aborts the transaction early by re-writing the packet header, and routing the packets back to the client. For high-contention workloads, NOCC improves transaction throughput, and reduces server load.

This system demonstrates why a coordination service benefits from being centrally located in the network. It also shows that INC provides enough stateful memory for simple data structure (a value cache), as well as enough expressiveness to process variable sized packets with the recirculation technique.

7.1 Background

Optimistic concurrency control (OCC) [100] is a key technique used by storage systems to ensure correctness in the presence of concurrent transactions. With optimistic concurrency control, a storage system speculatively executes a transaction without acquiring locks. Before committing a transaction, t, the storage system must verify that no other transaction has modified the data that has been read by t.

This *optimistic* approach stands in contrast to two alternative mechanisms for concurrency control: *blocking* and *immediate restart* [101]. Blocking and immediate restart are both pessimistic approaches based on locking objects before a transaction executes. The mechanisms differ in how they handle denied lock requests. In the first approach, the transaction blocks until locks are acquired. In the latter approach, the transaction is immediately aborted and must be retried.

There have been numerous studies comparing the performance of OCC to pessimistic concurrency control [102, 103, 104, 105, 102]. Many of these studies have contradictory results. Carey and Stonebraker [102] argue that blocking provides better performance than restarts. Tay [104] argues that restarts provide better performance than blocking. And Franaszek and Robinson [105] argue that optimistic methods are preferable to pessimistic approaches.

A landmark paper by Agrawal et al. [101] sheds some light on why these reports disagree. The authors identify three important assumptions on which the prior works differ: (i) the existence of infinite resources; (ii) whether or not restarted transactions are replaced with new independent transactions; and (iii) whether read operations use exclusive locks, or shared locks that may be upgraded to exclusive locks. This chapter focuses on optimistic concurrency control and the infinite resource assumption.

By "infinite resources", Agrawal et al. mean an infinite number of CPUs. Given an infinite number of CPUs, in the absence of contention, throughput should be a function of the number of concurrent transactions (since each CPU could process a separate transaction in parallel). Note that the likelihood of conflicts increases with the number of concurrent transactions. Agrawal et al. showed in simulation that, assuming the existence of infinite resources, OCC throughput continues to increase with the amount of concurrency since it never needs to acquire locks. In contrast, the throughput of blocking and immediate restart would both plateau.

However, in practice, with a bounded number of CPUs, OCC is inefficient under high contention. Its performance degrades, since OCC wastes CPU resources to verify transactions, only to abort them. In other words, the throughput of OCC is bounded by the number of non-conflicting parallel executions of transactions.

In this chapter, we argue that INC allows us to develop an OCC system that behaves as if it had infinite resources. The key idea is a new technique called *speculative verification offload*, which executes verification logic in the network. We have implemented speculative verification offload in a system named Network Optimistic Concurrency Control (NOCC). NOCC identifies transactions that are likely to abort in a Top-of-Rack switch, and aborts them before they reach the store. Thus, transactions that reach the store rarely abort, avoiding wasted server CPU, resulting in an extremely efficient optimistic concurrency control.

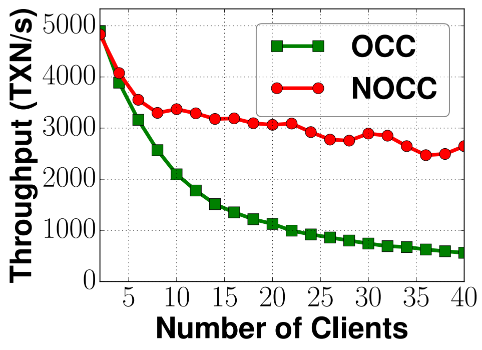


Figure 7.1. Throughput for incrementing a single counter as contention increases.

Motivation To demonstrate the above behavior, and motivate NOCC, we performed a simple experiment in which we increased contention, and measured the throughput of successful transactions. To increase contention, we increased the number of clients attempting to read and modify (increment) the same object in a key-value store. All the transactions passed through a switch that operated in one of two different modes of execution. In the first, the switch acted traditionally, and simply forwarded requests to the store. In the second configuration, which will be explained in Section 7.2, the switch executed NOCC logic to offload verification.

Figure 7.1 demonstrates that the performance of OCC degrades under high contention. With the traditional OCC store, as the number of clients increases, the store sends more aborts per transaction. In contrast, NOCC is much more efficient. Since the switch performs the validation of transactions, most conflicting transactions are aborted in the network, resulting in higher throughput.

Contributions We have implemented a prototype of NOCC using the P4 language [19], and evaluated it using Barefoot Network's Tofino ASIC [11]. Our experiments include both a set of micro-benchmarks that explore the parameter space, and an implementation of TPC-C [106] to emulate a real-world workload. Our evaluation shows that under high-contention workloads, NOCC significantly increases transaction throughput and reduces server load. Under low-contention workloads, NOCC adds no additional overhead.

7.2 Design

Before presenting the design of NOCC, we briefly provide important definitions and describe the system model (i.e., key aspects of the system and environment).

Definitions and System Model

We consider a distributed system composed of *client* processes and a *store*. Processes communicate through message passing and do not have access to a shared memory. The system is asynchronous; We do not assume any bound on messages delays and on relative process speeds. Processes are subject to crash failures and do not behave maliciously (e.g., no Byzantine failures).

The store contains a set $D = \{x_1, x_2, ...\}$ of data items. Each data item x is a tuple $\langle k, v \rangle$, where k is a key and v a value. We assume that the store exposes an interface with two operations: *read(k)* returns the value of a given k, and *write(k,v)* sets the value of key k to value v. We refer to those transactions that contain only read operations as *read transactions*. Transactions that contain at least one write operation are called *write transactions*.

We assume that clients execute transactions locally and then submit the transaction to the store to be committed. When executing a transaction, the client may read values from its own local cache. Write operations are buffered until commit time.

The isolation property that the system provides is *one-copy serializability*: every concurrent execution of committed transactions is equivalent to a serial execution involving the same transactions [107]. To ensure consistency, the store implements optimistic concurrency control. All *read transactions* are served directly by the store. To commit a *write transaction*, the client submits its buffered writes together with all values that it has read. The store only commits a transaction if all values in the submitted transaction are still current. As a mechanism for implementing this check, the system uses a *compare(k,v)* operation, which asserts that the value of k is v (i.e., the value has not changed since the client's last read. In the event of an abort, the server returns *corrections* with the up-to-date values that caused the compares to fail. This allows the client to immediately re-execute the transaction.

System Overview

Figure 7.2 shows a basic overview of NOCC. Transaction requests pass through a NOCC switch, which either forwards the request on to the store, or aborts the

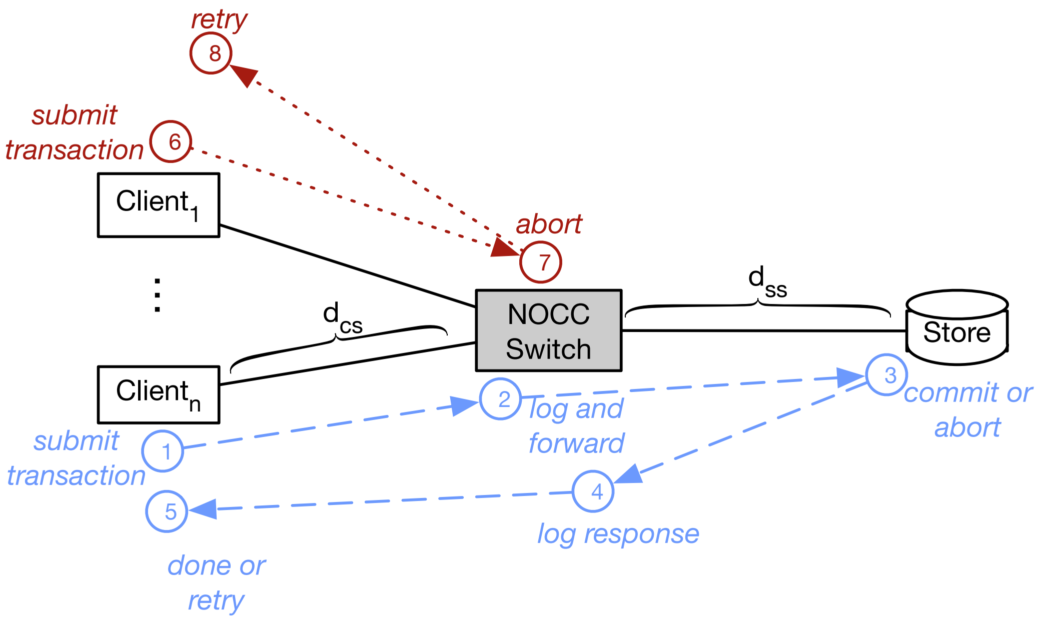


Figure 7.2. Overview of NOCC deployment.

transaction and responds to the client directly.

The blue, dashed-line shows the forwarding case: (1) the client submits the transaction; (2) the switch logs the transaction in its local cache, and forwards it to the store; (3) the store decides to commit or abort the transaction and responds with the decision; (4) the switch logs the result of the execution, and forwards the response to the client. (5) the transaction either completes or the client must re-try.

The red, dotted-line shows the abort case: (7) the switch examines the transaction message. If the switch sees that the transaction is likely to abort based on some previously seen transaction, the switch preemptively aborts the request; (8) Upon receiving the abort message, which contains corrections, the client can re-submit the transaction.

For a transaction that would have aborted at the store, there are two advantages of the *speculative verification offload* approach. First, the store does not waste resources on verifying the transaction, reducing load on the store. Second, the message avoids traveling the distance from the switch to the store, d_{ss}, twice.

Data Store A transaction request message contains three possibly empty lists of operations: *compares*, *reads*, and *writes*. The store first checks the compares for stale values. If any compare fails, the store aborts the transaction. As part of the abort response, the store includes a list of correct values, with the updated values for comparisons that caused the transaction to fail. Otherwise, the store updates the values of its data items with the values from the writes. Then the store responds to the client with all the values that were updated, along with the values that the transaction may read.

Speculative Verification Offload The NOCC switch logs requests and responses from several clients in order to determine if a subsequent transaction is likely to abort. NOCC adopts an aggressive strategy for aborting transactions. It proactively updates its cache with the latest value after the switch has seen a transaction request (step 2 in Figure 7.2). Note that this cache is only used to make decisions about aborting, and does not serve read requests.

We refer to this as a *speculative verification offload* strategy. It is speculative because the switch assumes that any transaction request that it has seen and conforms to its cached values is likely to be committed. As a result, it can make decisions about aborting subsequent transactions sooner. However, this approach may abort transactions that would not have been aborted by the store, which we discuss in Section 7.4.

The logic for speculative verification offload is as follows. The switch has logic for processing both transaction requests and their responses. When the switch receives a transaction from the client, it checks the compares. If any compare operation references a key that is not in the cache, then the switch cannot reason about the validity of the transaction, so it forwards the request to the store. If the compare references a key that is in the cache, then the switch compares the value in the packet with that in the cache. If the values differ, the value in the cache is added to a per-transaction set of corrections. After processing the compares, if there is at least one correction (i.e. there was a comparison that failed), the switch immediately sends an abort response to the client with the set of corrections. When the client receives the abort response, it uses the values in the corrections to recompute and resubmit the transaction immediately, avoiding an extra round trip of requesting the latest value from the store. There are two benefits: the client can retry the aborted transaction right away (lower latency); and the there is less chance that the value will have changed again since receiving the abort (which would be more likely if the client had to re-request the value).

If no comparisons fail, then the switch checks if the request contains write operations. If there are write operations, the switch updates its cache with the new values. Then, it forwards the transaction to the store for processing.

An abort message from the store contains a non-empty list of the corrections. The corrections contain the updated values that caused the transaction to fail. When the switch receives an abort message, it updates its cache with the correct values. If the switch did not update its cache on aborts, it would incorrectly abort subsequent transactions.

We note that although a NOCC switch needs to maintain states in its local cache, the size of the cache does not need to be too large to be effective. It is sufficient to reserve enough space for “hot” data items. The amount of space available to the cache will depend on the target platform for deployment. If the size is restricted, NOCC could use a cache eviction policy to make space available for new items.

Detecting Stale Values In a typical transactional storage system, data items would include a version number that the store would use to determine if a transaction should be aborted due to a stale value. However, with the speculative verification offload strategy, the switch must update its local cache of data items before the store could assign a version number. Therefore, NOCC cannot use version numbers to check for stale values. Instead of comparing version numbers, NOCC compares the actual values of data items. Furthermore, by storing the actual values on the switch, they can be included in switch-generated abort messages, which enables clients to retry transactions immediately with the latest values. This obviates the extra round trip of the client requesting the latest value from the store, reducing transaction retry latency and load on the store.

Expected Deployment We expect that NOCC would be deployed in a Top-of-Rack switch that inspects all traffic in a rack of storage servers. However, if NOCC were deployed in a way that it did not interpose on all traffic to a store (e.g., clients connected to different switches update data at the store), NOCC does not violate correctness. Clients and switches will learn of new values after an abort message from the store. For example, if client$_1$ writes a value v_1 for key k_1, then switch$_1$ will record v_1 in its cache. However, if client$_{n+1}$ had previously written a value v_1' for key k_1, the request from client$_1$ will pass through switch$_1$, but will be aborted by the store. The client and switch$_1$ will learn of the new value v_1' in the abort response from the store. They will both then update their local caches, and the client can re-submit the transaction with the latest value.

Correctness The store ensures one-copy serializability. The serialization order is defined by the arrival order of transactions at the store. A transaction only commits if all the reads it performed during execution are still up to date at the time the transaction is received by the store.

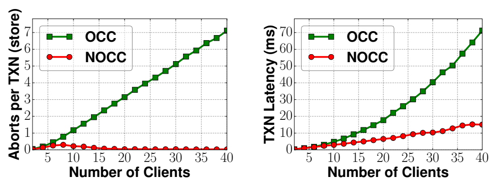
The correctness of the switch logic follows from the fact that (a) the switch does not commit any transaction, although it may abort transactions, and (b) the switch forwards non-aborted transactions to the store without changing their operations. The switch may abort transactions that would not have been aborted by the store; this does not compromise correctness, but has a performance penalty which is outweighed by the benefits of correct aborts.

7.3 Implementation

We have implemented NOCC as a P4 program that runs on a Barefoot Tofino ASIC [15]. We have also implemented a client program and OCC transactional storage system in Python. NOCC uses a custom *transaction* header encapsulated in a UDP packet, followed by a sequence of fixed-width *operation* headers.

The P4 program contains parsers and tables for standard L2 forwarding, as well as processing logic for transaction packets, which is divided into two phases. In the first phase, the program iterates over the operations, checking that the compares are valid (i.e., the value in the packet matches the value in the cache). In the second phase, the program iterates over the operations a second time, either: updating the cache if the transaction is valid; or, updating the invalid values in the packet and returning it to the client as an abort.

The cached values on the switch are stored in registers which allow up to 32 bits to be read or written in a single pipeline stage [11]. This presented us with two challenges: caching large values and accessing them multiple times for the same packet (e.g., for the iterations). To store larger values, we split the value across multiple registers stored in different stages. To iterate over the operations in the packet, we use recirculation; after each iteration, the packet is recirculated through the pipeline, storing the state of the computation (including iteration index) in packet metadata. Although we could recirculate an arbitrary number of times, the number of operations per transaction we support is bounded by how deep the packet can be parsed, which in turn, is bound by the size of the packet header vector [11]. Although recirculation reduces the throughput of the switch, it still provides better performance than a software implementation.



(a) aborts per txn(b) latency vs.
vs. #clients #clients

Figure 7.3. NOCC has low aborts and latency at store.

7.4 Evaluation

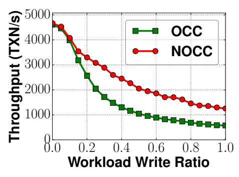
In this section, we describe two sets of experiments that evaluate NOCC. The first set of experiments are microbenchmarks that explore how NOCC impacts the performance for executing transactions on a key-value store under changing operational conditions: number of clients, write ratio, and workload contention (skew).

In the second set of experiments, we explore real-world inspired workloads, by using the TPC-C [106] benchmark with the parameters adjusted to increase contention. Overall, the results demonstrate that NOCC improves throughput and reduces latency for workloads with high contention.

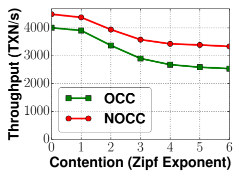
Experimental setup We used two machines, each with 12 cores (dual-socket Intel Xeon E5-2603 CPUs @ 1.6GHz), 16GB of 1600MHz DDR4 memory, and Intel 82599ES 10Gb Ethernet Controllers connected to a Barefoot Tofino switch. All the clients were collocated on one machine, while the store server ran alone on the other machine.

Microbenchmarks

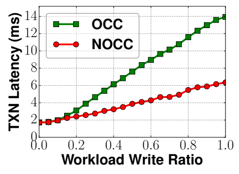
NOCC has higher throughput as the write ratio increases Each client in the benchmarks issues a mix of *read* and *write* transactions on the same key. The write ratio dictates the percent of the total number of transactions that are writes. Figures 7.4a and 7.4c show throughput and latency for 8 parallel clients, as the write ratio changes. These figures clearly demonstrate the effect of write operations, which limit the overall performance of the system. As the write ratio grows, OCC's throughput becomes limited due to the high cost of aborting writes. NOCC's throughput, on the other hand, degrades slowly, since requests are aborted at the switch and clients can optimistically retry transactions sooner. At 0.2 write ratio, NOCC's throughput is already 1.3x that of OCC, reaching 2.2x



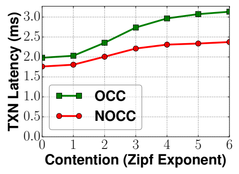
(a) throughput vs. write ratio



(b) throughput vs. contention



(c) latency vs. write ratio



(d) latency vs. contention

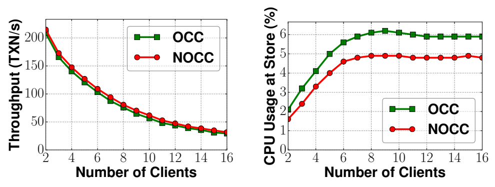
Figure 7.4. NOCC improves throughput and latency as write ratio and contention changes.

that of OCC at 1.0 write ratio.

NOCC does not add overhead for reads When all transactions are reads, a NOCC switch does not perform verification logic. We can see in Figures 7.4a and 7.4c that when the write ratio is 0, NOCC has no overhead compared to OCC.

NOCC scales with the number clients Figure 7.1 reports throughput when the write ratio is fixed at 0.2. NOCC provides a higher transaction rate, past the saturation point of OCC, reaching 4.6x the throughput at 40 clients. Using NOCC, transactions can abort early and optimistically be retried with the latest values, *before the previous conflicting transaction commits*. Offloading verification to the switch reduces the number of aborts the store has to send (Figure 7.3a), freeing the store's resources to commit valid transactions. Because of this, the transaction latency scales linearly with the load, as Figure 7.3b shows.

Skewed workloads benefit from NOCC This experiment characterizes the effect of contention on the performance of NOCC. Clients submit transactions that can access one of 10 keys, and the popularity of each key is dictated by a Zipf distribution. Figure 7.4b shows how increasing the Zipf exponent affects throughput.



(a) #clients vs. throughput (b) #clients vs. CPU usage

Figure 7.5. TPC-C Payment transaction

With the Zipf exponent at 0, all keys are accessed with the same probability, and contention increases for larger exponents. As contention increases, the number of aborts grows, limiting the performance of write transactions for OCC. NOCC, on the other hand, by optimistically aborting and retrying transactions closer to the client, is less affected by the increase in contention.

TPC-C

The TPC-C [106] benchmark models an online transaction processing (OLTP) workload for a fictional wholesale supplier that maintains warehouses for different sales districts.

We note that TPC-C is not a good benchmark for evaluating NOCC, since the benchmark is intentionally designed to avoid queries that result in high-contention. Nevertheless, TPC-C is a widely accepted standard for evaluating transaction processing systems. We therefore focus on the TPC-C payment transactions, which has contentious dependencies.

To further increase contention, we modified the parameters of the benchmark. The TPC-C specification states that the benchmark should be run with a set of specific parameter values: 1 warehouse, 10 districts, 3000 customers and 100,000 items. For our evaluation, we used the following settings: 1 warehouse, 2 districts, 10 customers and 10 items.

As a TPC-C driver, we modified an open source Python implementation from Pavlo et al. [108]. Since the store does not have indexes, we modified the transaction executor to only perform exact selects. To represent the TPC-C database in our store, we map each record to a key-value pair. The client driver keeps a copy of all records it has read or written, essentially mirroring the store with a local cache. This eliminates the unnecessary latency of issuing read requests for records that have not changed. Consistency is guaranteed by validating the transaction before committing, ensuring the read values (possibly from the local cache) are up-to-date.

Figure 7.5a shows the throughput for the Payment transaction, and figure 7.5b shows the CPU usage at the store, for increasing contention. In Figure 7.5b, we can see that NOCC reduces the load on the store (by up to 22% compared to OCC), while maintaining slightly higher throughput.

Discussion After seeing the microbenchmark results, we expected TPC-C to maintain higher throughput with NOCC. However, the performance was limited by incorrect speculative decisions. The switch aborts transactions that would not have been aborted by the store. Nevertheless, NOCC performed better than the OCC baseline, suggesting that NOCC can speed-up transaction processing for high-contention OLTP workloads.

7.5 Conclusion

NOCC moves transaction processing logic into the network, using a custom packet header and programmable switches to identify and abort doomed transactions as early as possible. For write-intensive, high-contention workloads, NOCC reduces load on the store and increases system throughput.

Overall, concurrency control is a key component of storage systems, and NOCC demonstrates how tighter integration with the network can lead to significant improvements in performance.

Chapter 8

Transaction Triaging

This chapter describes Transaction Triaging (TT), a set of techniques that manipulate streams of transaction requests and responses while they travel to and from a database server. Compared to non-triaged transaction streams, the manipulated ones execute faster once they reach the database server. The triaging algorithms do not interfere with the transaction execution nor require adherence to any particular concurrency control method, making them portable across systems.

At the core of our TT switch implementation is a queue data structure. This demonstrates how application data can be buffered on the switch efficiently. Furthermore, some of the triaging techniques use recirculation on the switch to process packets with an arbitrary number of transactions.

We validate the TT techniques using a high-performance in-memory database and a programmable switch. The results show that TT can improve workloads such as TPC-C [106] and YCSB [109], both on an IP-based stack and an RDMA-enabled one. For IP-based protocol stacks, a triaged stream can offset up to 97% of the network overhead. In other words, the server processes more transactions per unit of time, which almost cancels the networking overhead. For RDMA-enabled networks, which are already low-overhead, triaging can bring a peculiar benefit. It becomes faster to execute a triaged transaction stream coming from the network clients than if local clients generated a non-triaged workload.

8.1 Background

Improving transaction processing performance has long been a critical concern for database systems [110]. There are many techniques for simultaneously handling sets of concurrent transactions [107] and for executing them efficiently [111].

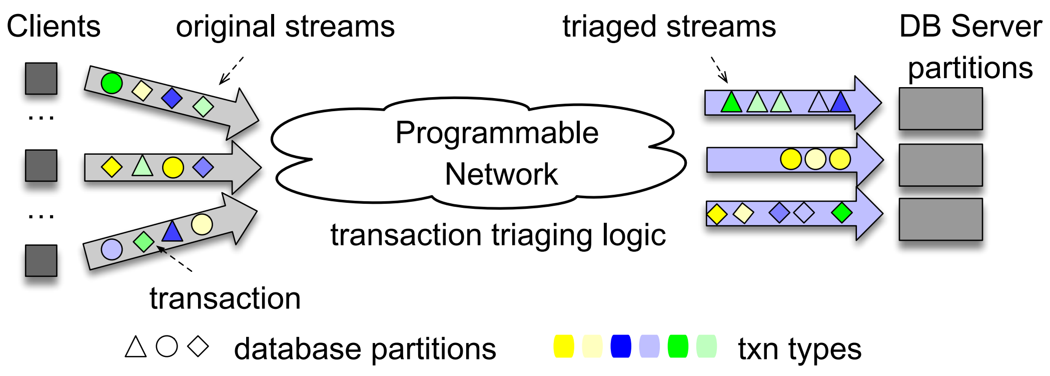


Figure 8.1. In-network triaging fosters faster transaction execution.

These techniques apply to transactions that have already reached the database server. However, a portion of a transaction's lifetime is spent on networking: transferring a client's request into the database server and shipping the results back. For in-memory databases, such networking time can represent a high cost. Our experiments (described in detail in §8.6)show that for some small but typical workloads, the transaction delivery and return alone can take up to 70% of the user-perceived response time. Technologies such as RDMA [6, 112] can significantly reduce this overhead, but the techniques we present here enhance even those networks, as we discuss shortly. Moreover, not all networks have RDMA-capable hardware. Consider, for example, a transaction that starts on a mobile device. It does not enjoy the benefit of RDMA for at least part of the path into a database server. For another instance, suppose a transaction requires replication across datacenters, where the interconnect lacks RDMA support. Recent advances notwithstanding, many still consider the networking overhead to be an unavoidable "tax" that activities such as transaction processing pay for distribution.

The overhead exists because data transfers take time but do not advance any computation. The switches performing the transfer, however, contain significant computing power. They can parse and make routing decisions for a handful of billions of packets per second. These high-end switches are invariably based on specialized chips designed to execute a fixed set of networking protocols.

In this chapter, we use INC to address the network overhead that transaction processing incurs. We propose several techniques that rearrange streams of transaction requests and responses while they are in-flight, as Figure 8.1 depicts. We call such techniques Transaction Triaging (TT). For example, TT can batch (coalesce) many network packets into one, amortizing the per-packet handling

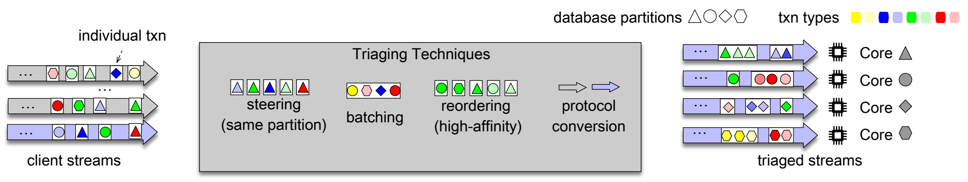


Figure 8.2. Conceptual view of the triaging techniques. Transactions with the same shape target the same partition. Transactions with a similar color have high affinity; executing them in sequence will likely improve performance.

overhead across many transactions. Of course, a single client could batch its transactions, but only the network can perform batching across clients. Moreover, the server can also batch responses to distinct transactions, offloading onto the network the task of breaking them down according to their destination.

Creating TT techniques is challenging in at least two ways. First, it requires finding effective transaction manipulations that influence the server's performance. Batching is just one of the several TT techniques we present. The second challenge is encoding these techniques as algorithms that INC can support. As we mentioned above, INC does not turn the network into a general-purpose computer. Instead, it requires adhering to a particular programming model that only supports *forward-logic*.

8.2 Transaction Triaging

Triaging aims to produce streams of transactions carefully designed to improve server efficiency. We identify four triaging techniques that have the following properties: (a) they inter-operate, (b) potentially impact server performance, and (c) can be carried by programmable networks. Figure 8.2 illustrates the individual techniques we identified: *batching*, *steering*, *re-ordering*, and *protocol conversion*.

The batching technique bundles several single-packet transaction requests into a larger network packet. In OLTP, clients naturally produce single-packet transactions as they need the results of a current transaction to determine what to do next. Batching amortizes the packet-receiving overhead across multiple transaction requests.

The steering technique influences how a multi-core database interacts with the network. The network may elect a single core to receive an incoming packet or use RSS to load balance the related overhead across the cores. In-memory databases often choose to partition the data horizontally and map partitions to cores (e.g., partitioning TPC-C by `warehouseID`) [113, 114, 115]. RSS is bound to deliver transactions to a "wrong" partition if it does not take that mapping into consideration. Our steering technique allows RSS to use partitioning information to guide the NIC's RSS algorithm.

Transaction reordering manipulates the sequence in which the transactions arrive at the server. It attempts to place similar transactions close to one another using an *affinity* metric. For example, transactions with similar access patterns may reuse instructions or data caches [116, 117]. As another example, re-ordering can influence the likelihood of an abort in a database with optimistic concurrency control.

Lastly, protocol conversion translates packets to use a more efficient network stack between the switch and the server. Quite often, a top-of-rack switch has a fast connection with the servers in that rack. We assume this is the case and allow the switch to use techniques such as RDMA in the last hop of the communication, even if the client-to-switch portion of that stream cannot benefit from RDMA. This technique leverages the fact that for a database server the bulk of the RDMA benefits come from how the receiving NIC interacts with the system.

Combining the four techniques produces the following result: our pipeline generates streams that carry multi-transaction packets, each containing transactions for a given database partition/core, while all transactions within a batch share some beneficial degree of affinity. The batch is delivered directly to the core most involved with the execution, and the delivery utilizes low overhead protocols.

8.3 In-Network Algorithms

The Transaction Triaging techniques described in the previous section can be implemented on an `x86` CPU using standard data structures and programming languages. However, our goal is to execute such logic on high-speed switches. We must create algorithms that respect the logical constraints of the computing model and the physical limitations of the current generation of such equipment.

There are at least two design challenges that need to be addressed in that context. First, we must separate our algorithms' logic into a control plane component—which runs offline before deployment—and a data plane component—which is expected to process transaction requests at line-rate. In our case, some algorithms use conversion tables (e.g., given a transaction type what batch should it join) that are calculated offline. Some other portions of our algorithms manipulate packets while they are in-transit in the data plane (e.g., actually place a

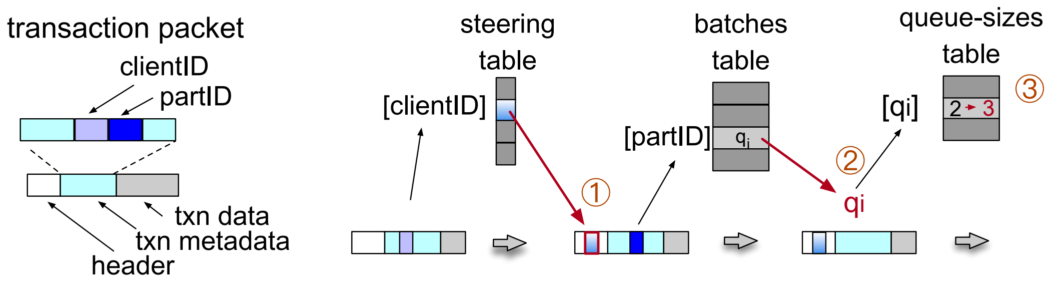


Figure 8.3. Example of side-effects of processing a packet on the switch. (1) a packet is matched with the steering table using the `clientID` field and changing the packet header as a result; (2) the lookup of field `partID` results in associating a queue number with the packet; and (3) the queues' size table is incremented to account for the imminent packet buffering.

transaction on a batch using the table).

The second design challenge is to express the data plane component in the *forward-logic* style the switch imposes. This model enforces a strict ordering on operations; all data reads must occur before all writes. Moreover, each stage can perform a limited number of steps, and all logic must fit into the fixed number of stages in the switch. By design, there are no loops—network devices are designed to process packets in a single pass.[1]

We introduce a simple example of forward-logic next and describe our algorithms in detail afterward. Our goal is to highlight some of the implementation challenges for this particular set of algorithms while providing a broader perspective on how to implement other algorithms or data structures on a PISA device.

Switch Programs. Figure 8.3 illustrates how a sequence of match-action tables (MATs) modify a packet carrying a transaction. In step (1), the program performs a lookup on a `steering` table, which we describe shortly, using the `clientID` field as a key, and updates the packet's header with the contents of the match. The altered packet moves to the next stage (2), where a lookup on table `batches` is done using field `partID`. Note that the side-effect this time is to record the batch (queue) to which the packet belongs in its metadata. Lastly, the packet moves to the next stage (3), where a lookup on table `queue-sizes` is done using the q_i from the previous step. This step increments the value of an entry on that table, the size of queue q_i.

[1] Iteration can be achieved by re-circulation a packet through the pipeline, at the cost of reducing the throughput of the device.

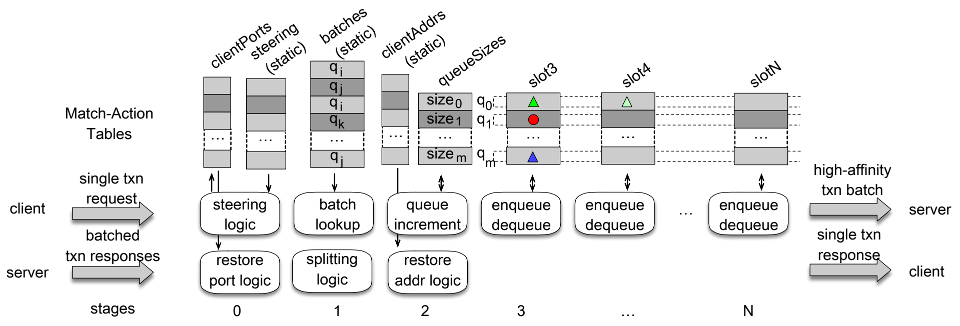


Figure 8.4. Placement of TT-related match-action tables across the programmable switch stages.

Algorithm Notation. We use pseudo-code that captures common forward logic restrictions without using the syntax of any particular programming language. Note that our prototype is implemented in P4 [19], but other languages such as Broadcom's NPL [118], Huawei's POF [119], or Xilinx's PX [18] could have been used. The data plane logic is separated into a sequence of stages, indicated by the **Stage:** keyword, where each stage corresponds to one match-action unit in the pipeline. The logic at each stage is triggered when a request (to the server) or response (from the server) is received, which is indicated by the **upon** … **do** code block. We use the notation **with** *var* **as lookup** *key* **in table** *id* to indicate that we perform a lookup operation on the table named *id* with a specified *key*. The result of a lookup is bound to the variable name *var*, in the lexical scope indicated by the indented text.

We present more detailed algorithms for each of our four TT techniques next. Figure 8.4 depicts how the techniques are integrated in a single switch program.

8.3.1 Steering

When the server's NIC receives a packet, the RSS algorithm computes a hash on five specific packet header fields—the source/destination addresses and ports, and the IP protocol—and uses the hash to select a CPU core to interrupt. RSS simply load balances packets across cores (e.g., with 12 cores, RSS selects core number *hash*%12).

To steer a packet to a specific database thread, we influence the NIC's RSS algorithm by changing the values of some of the five header fields. Note that we cannot change the source and destination address, as that would interfere with routing in the network. Nor can we change the destination port, because the database server is already expecting packets on certain ports. This leaves a single candidate to modify: the UDP source port.

Finding the UDP source port for steering a transaction can be done offline using an exhaustive search. The exhaustive search algorithm finds a random UDP port that, hashed with the other four RSS fields, would induce the NIC to send the packet to the desired partition/core [120]. The search space is limited because the number of UDP ports and the number of connected clients is limited. We assume that the database client addresses are known upfront. This algorithm produces the `steering` table, which maps a client address `srcAddr` and a `partID` to a `srcPort` (for the RSS) and `dstPort` (where the partition/core is). This table can either be used at the client or directly in a network switch.

Algorithm 2: Steering Transactions and Responses

```
   Stage: 0
   Table: steering                         ▷ Stores ports that steer client requests
   Table: clientPorts                      ▷ Stores the original srcPort for clients
1  upon request pkt do
2      with row as lookup (pkt.clientID) in table clientPorts
3          row.port ← pkt.srcPort
4      with row as lookup (pkt.srcAddr, pkt.partID) in table steering
5          pkt.srcPort ← row.srcPort
6          pkt.dstPort ← row.dstPort
7  upon response pkt do
8      with row as lookup (pkt.clientID) in table clientPorts
9          pkt.dstPort ← row.port
```

Algorithm 2 shows how the switch uses the `steering` table to steer a transaction packet to a specific database thread. The logic can fit in a single stage of the switch pipeline. Upon receiving a transaction request packet, the switch first stores the packet's original source port. It does so by looking up the client (identified by `pkt.clientID`) in the `clientPorts` table and updating the row with the port (line 2). The switch loads the ports associated with the partition the transaction wishes to access (lines 4–6). It looks-up the client's address (`pkt.srcAddr`) and the partition ID (`pkt.partID`). It then substitutes the source and destination ports (`srcPort`, `dstPort`) in the packet, effectively causing the packet to go to that destination instead.

Upon receiving a response (line 7), the switch must restore the `pkt.srcPort` from the original request packet. Otherwise, the client will receive a packet for an unknown port (the client is not aware of the translation). The switch loads the previously-stored source port, `pkt.srcPort`, which is now the destination port of the packet, `pkt.dstPort` (line 9).

8.3.2 Batching

Conceptually, the logic for creating batches is straightforward. The main idea is to combine transactions from multiple request packets into a single batched packet. However, to implement this in the network, we must address several non-trivial issues. First, we must manage several queues (batches) in a pipeline architecture. Second, we must determine which transactions to place within the same batch. Third, we must keep track of where transaction requests came from, in order to forward the corresponding individual response when splitting the batched response packets.

Algorithm 3 gives an outline of our forward-logic implementation of batching. At a high level, as requests arrive, the transactions are put in one among many possible queues. When a queue is full, it is drained, and the transactions are combined into a single request. There are three main steps in the algorithm: (i) choosing the queue (batch) for a transaction; (ii) updating the state (size) of the chosen queue; and (iii) either enqueueing a transaction or draining the queue when it is full. When the queue is drained, the transactions in the queue are appended to the current packet.

To guarantee that batches will not stay incomplete indefinitely, the control plane can inject special, per-queue timeout requests, which we discuss in Section 8.5. Such packets trigger a batch-send in case the batch has not increased for a given time. For simplicity, we omit this mechanism in Algorithm 3.

In stage 1, the algorithm picks a queue for the transaction by performing a lookup on table `batches` (line 2). The table contains a mapping from the transaction's partition (`pkt.partID`) to a queue ID (`qid`); this ensures that each batch contains transactions for the same partition and is steered to the appropriate database thread. We discuss in Section 8.4 how the table `batches` can be generated offline (potentially considering more information than simply the transaction's partition).

In stage 2, the algorithm loads the current queue size (`m.qsize`) for the chosen queue (line 5). The queue size is then incremented and stored. If the queue size has reached the batch size, then it wraps around to 0.

The packet passes through the rest of the stages, each of which holds one index entry of all queues. The head of the queue is in stage 3, and the tail moves down the pipeline as transactions are enqueued. Depending on whether the queue is full or not, the stages perform different actions. If the queue is not full, the transaction in the packet is stored when the packet reaches the stage corresponding to the tail of the queue (line 14). If the queue is full, the transaction stored in each stage is loaded into the packet (line 16); once the packet reaches the end of the pipeline, it will contain all the transactions from the queue, and the batch is sent.

After executing the transactions, the database responds with a batched response packet. This packet contains the result for transactions submitted by different clients. The switch has to split this packet into multiple packets, each addressed to the originating client.

Our algorithm relies on making copies of the packet as it iterate over the transactions within it, as shown in Algorithm 4. Programmable switches usually provide very efficient mechanisms to support such packet copy operations. In stage 1, the algorithm checks whether the packet contains multiple transactions (line 2). If there are indeed multiple transactions, the packet is split into two packets: a copy which contains all but the first transaction; and the original packet, with only the first transaction. The original packet continues to the next stage, while the copy is sent back to the beginning of the pipeline.

Stage 2 receives the original packet from stage 1. In this stage, the packet must be addressed to the client that originally sent the transaction. The client's address is looked-up in the `clientAddrs` table (line 8), which contains a mapping of clientIDs to addresses. This mapping can be established during the initialization of each client's connection. For simplicity, we omit such logic and assume that, for the purpose of this explanation, the `clientAddrs` table is static.

8.3.3 Reordering

As described above, Algorithm 3 picks a queue for each transaction based solely on the transaction's target (or main) partition. This constitutes a coarse-grained mechanism to reorder transactions. The ordering can be further manipulated in several ways. For example, we can also classify transactions by their type (e.g., NewOrder or Payment in TPC-C), in addition to their target partition. The table `batches` in Algorithm 3, line 2 would then map a packet's `txnType, partID` (instead of just `partID`) into a `qid`, effectively enforcing a separation policy. If certain transaction types should be delivered together to the server, the `batches` table would map them to the same `qid`. Conversely, transaction types that interfere with one another could be assigned to different queues. The number of queues is limited only by the memory available on the switch, but programmable switches can have memory for even thousands of such queues (as the batch sizes are usually small).

Algorithm 3: Batching Transactions

```
    Stage: 1
    Table: batches                                  ▷ Maps transactions to queue IDs
1   upon request pkt, metadata m do
2       with row as lookup (pkt.partID) in table batches
3           m.qid $\leftarrow$ row.qid
    Stage: 2
    Table: queueSizes                               ▷ Stores the current size of each queue
4   upon request pkt, metadata m do
5       with row as lookup (m.qid) in table queueSizes
6           m.qsize $\leftarrow$ row.qsize + 1
7           if m.qsize = BATCH_SIZE then                          ▷ Queue is full
8               row.qsize $\leftarrow$ 0
9           else
10              row.qsize $\leftarrow$ m.qsize
    Stage: N: 3 ... 3+BATCH_SIZE
    Table: slotN                                    ▷ Stores txn at position N-3 in the queue
11  upon request pkt, metadata m do
12      with row as lookup (m.qid) in table slotN
13          if N−2 = m.qsize then                                 ▷ Tail of queue
14              row.txn $\leftarrow$ get txn from pkt
15          else if m.qsize = BATCH_SIZE then                     ▷ Queue full
16              append row.txn to pkt
```

Having a finer control over the transaction ordering within the same batch is also possible. With a slightly different logic in Stages `N` (lines 11 to 16), Algorithm 3 could perform an insertion sort and place a transaction anywhere within the batch. Lines 13 to 14 would instead swap a transaction in the first position when the incoming transaction's priority is higher than the existing one. If a transaction gets displaced this way, it becomes the current transaction—and the insertion process repeats, starting at the stage the displacement occurred. When draining, the algorithm preserves the order in which the transactions appear on a queue, therefore inducing the same order onto the server, upon the receipt of a new batch.

In summary, the algorithms described here are flexible with respect to the ordering criteria to use. The layout and contents of the table `batches` and the

Algorithm 4: Splitting Batched Transaction Responses

```
   Stage: 1
1  upon response pkt do
2      if pkt contains multiple txns then
3          pkt′ ← copy pkt
4          remove first txn from pkt′
5          truncatate all but first txn from pkt
6          send pkt′ to stage 0
   Stage: 2
   Table: clientAddrs                         ▷ Stores each client's address
7  upon response pkt do
8      with row as lookup (pkt.clientID) in table clientAddrs
9          pkt.dstAddr ← row.addr
```

matches performed by Algorithm 3 can be tailored to specific cases. We discuss some suitable alternatives in more detail in Section 8.4.

8.3.4 Protocol Conversion

This technique has both a strategic and a practical purpose. The strategic one is to take advantage of RDMA as a low-overhead protocol between the switch and the database server. We can do so even if RDMA is not available for part of the client-server interconnect. The practical purpose is related to the current generation of programmable switches. They can buffer a handful of packet fields: the transaction's ID, type, and originating client. However, to buffer the entire transaction payload would require full packet manipulation involving an area of the switch that is not (yet) fully programmable: the traffic manager. There have been attempts to address this issue [121, 122], but we took an alternative approach instead. To overcome what we believe is a temporary limitation, we buffer the transaction directly on the database server via one-side RDMA initiated by the switch.

Figure 8.5 provides an overview of the two channels we use to send transactions to the server. When the switch receives a transaction (1), it assigns the transaction to a queue as described above, as well as a memory address in a ring buffer on the server. It stores the transaction metadata, along with the memory address in the queue in switch memory. It then transforms the transaction packet into an RDMA WRITE request and forwards it to the server (2). The server's NIC

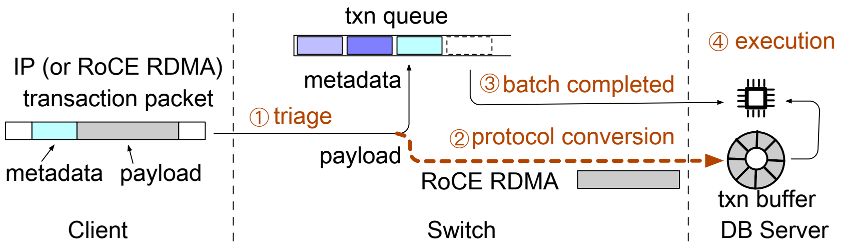


Figure 8.5. Using protocol conversion to RDMA for delivering transaction payloads. (1) the transaction metadata is buffered on the switch; (2) the transaction parameters are forwarded to the transaction buffer area in the server; (3) when the batch completes, the server received it as a single packet; and, (4) the server gets the parameters of the transactions from the buffer area.

receives the WRITE and stores the transaction in the ring buffer at the address specified by the switch. Note that this does not involve the server's CPU.

When a batch is full, the switch pops all the transactions from the queue (3), including the memory address in the server's ring buffer. This batched packet reaches the server through an RDMA SEND operation. The database process receives the batched packet. The database reads all the transaction memory addresses in the packet to load the previously stored transactions from the ring buffer (4).

Our protocol conversion technique has another advantage. It minimizes the changes to the networking subsystem of a database that wishes to integrate Transaction Triaging. The transactions metadata (steered, batched, and reordered) reach the switch via the normal client connection, as we discuss in Section 8.5. The only necessary change is for the system to read the transactions payload from the designated queues.

8.4 Transaction Affinity

The algorithms described in Section 8.3.3 can effectively program the switch to carry out the following mapping:

$$transactionType, partitionID \rightarrow queueID, priority.$$

Based on the contents of the `batches` table and on a transaction's target partition, its type, and its priority, the logic on the switch decides which queue to pick to direct an incoming transaction. Optionally, it also decides the relative position of the transaction in a queue. This scheme is powerful enough to express many different policies.

The simplest strategy is to separate transactions by their partition and type. The necessary number of queues corresponds then to the product of the number of different transaction types by the number of partitions in which the database is divided. For workloads with a limited number of transaction types, this is a very suitable scheme. For instance, running TPC-C (5 transaction types) on a 12-core machine would call for 60 queues on the switch. We show how TPC-C can benefit from this technique in Section 8.6.

In real-world scenarios, however, we expect much more elaborate transaction sets and clustering techniques. One way to triage transactions is through micro-benchmarking them offline and identifying opportunities. For instance, STREX [116] and ADDICT [117] are techniques based on static analysis of query plans. They find transaction types that share instruction patterns and can thus benefit from instruction cache reuse if executed together (on the same core, within a short amount of time).

Another technique to obtain a mapping would be exhaustive evaluation. A starting point is the combination of the number of transaction types per size of batch. (The size of a batch is an implementation detail that depends on the flavor of the queue used on the switch and structural properties of the switch.) This number can be aggressively pruned by reducing the number of different transaction types in a batch. We envision a calibration tool that performs such tests automatically.

8.5 Integration to Existing Systems

Integrating TT into a database server requires a few peripheral changes. We list them below.

Transaction buffer. The server continues to read transaction request packets as before, although the latter will only contain the transactions' metadata. To receive the transactions' contents, the server must create a transaction buffer, as depicted above in Figure 8.5. The switch is responsible for delivering the portions of the transactions it does not buffer into that area. The server informs the switch of the location of this area at initialization time. When the server receives a metadata batch to execute, it fetches the additional transaction information from the buffer.

Reliability. The switch performs triaging over transactions carried by UDP/IP or RDMA UD. These protocols are easier to manipulate in part because they lack reliability. To compensate, we assume that a client would re-send a transaction request if it does not receive a response within a certain time. Moreover, the client would periodically inform the server of the most recent response it has received.

The server maintains a transaction response cache. If it received a transaction request whose response is in the cache, it re-sends those results rather than re-executing the transaction. The cache is cleared using the clients' acknowledgement messages or after an established timeout. This scheme assumes that both clients and transactions can be uniquely identified.

Packet Format. We assume that network packets have a standard transaction metadata header that allows the switch to manipulate arbitrary transactions uniformly. The header carries information about the client and the transaction. Of particular interest here is a transaction's type, e.g., NewOrder or Payment in TPC-C, and the primary partition which the transaction targets, e.g., a hash of the warehouseID. The client and the server may negotiate some parameters at connection time, such as the database partitioning criteria. We assume that a transaction is an instance of a *stored procedure* initiated by an OLTP application [123]. Moreover, the application can fill all the transaction's input parameters at the same time, when issuing the transaction's execution request. The database client's library is responsible for filling the transaction metadata fields.

8.6 Evaluation

To evaluate Transaction Triaging, we carried out four sets of experiments. The first set establishes a baseline comparison by quantifying the overhead attributed to the network (Section 8.6.1). The second set evaluates the impact of each optimization in isolation under various system parameters (Sections 8.6.2-8.6.4). The third set evaluates the optimizations using different network transport layers (Section 8.6.5). Finally, we compare the performance under different workloads (Section 8.6.6).

Experimental Setup and Environment. As a representative in-memory transactional database, we used Silo [124]. Silo is open-source and capable of executing hundreds of thousands to millions of transactions per second. For this reason, Silo is often used as a benchmark for new concurrency control algorithms [125, 126, 127]. We extended Silo with a network component for each of the protocol stacks we use, as the open-source version does not support networking. We refer to this extended version of Silo as `NetSilo`.

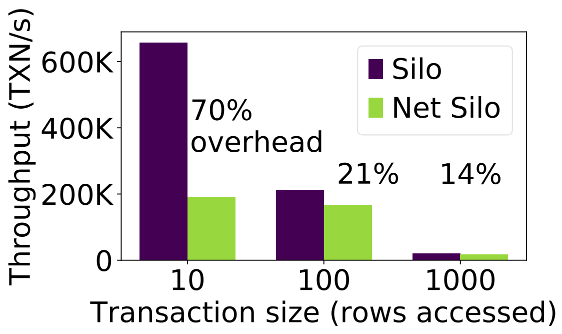


Figure 8.6. Overhead of network communication on in-memory database.

We ran all experiments on a pair of servers, one acting as the database server, the other as multiple clients. Each server had dual-socket Intel Xeon E5-2603v3 CPUs @ 1.6GHz with a total of 12 cores and 16GB of 1600MHz DDR4 memory. Each server had both an Intel 82599ES 10 Gb/s (DPDK compatible) and a Mellanox ConnectX-5 100 Gb/s (RDMA) Ethernet controller. The servers were connected to a 32-port 100 Gb/s programmable switch based on the Tofino ASIC [15].

8.6.1 Networking Overhead

Transmitting and handling transaction requests accounts for a portion of the perceived transaction response time. As we mentioned, this overhead can be substantial. To quantify this impact, we pre-generated an entire set of transactions and placed them in memory on our Silo server (`LocalSilo`). We then compared the time to execute this local workload versus sending the same workload through remote clients. For this experiment, we used a UDP/IP stack.

Figure 8.6 shows the throughput of the locally- and remotely-generated workload executions. We used a workload of a single transaction that accesses either 10, 100, or 1000 rows from a database table. With networking, executing relatively small transactions adds about 70% overhead compared to executing the transaction locally. As the transaction size increases, the overhead decreases, which suggests that the absolute per-transaction overhead is constant.

8.6.2 Steering Experiments

To evaluate how steering contributes to performance, we run `NetSilo` in three modes: with RSS disabled, with standard RSS, and with our semantic RSS. In each mode a different core (or set thereof) receives an interrupt from the NIC

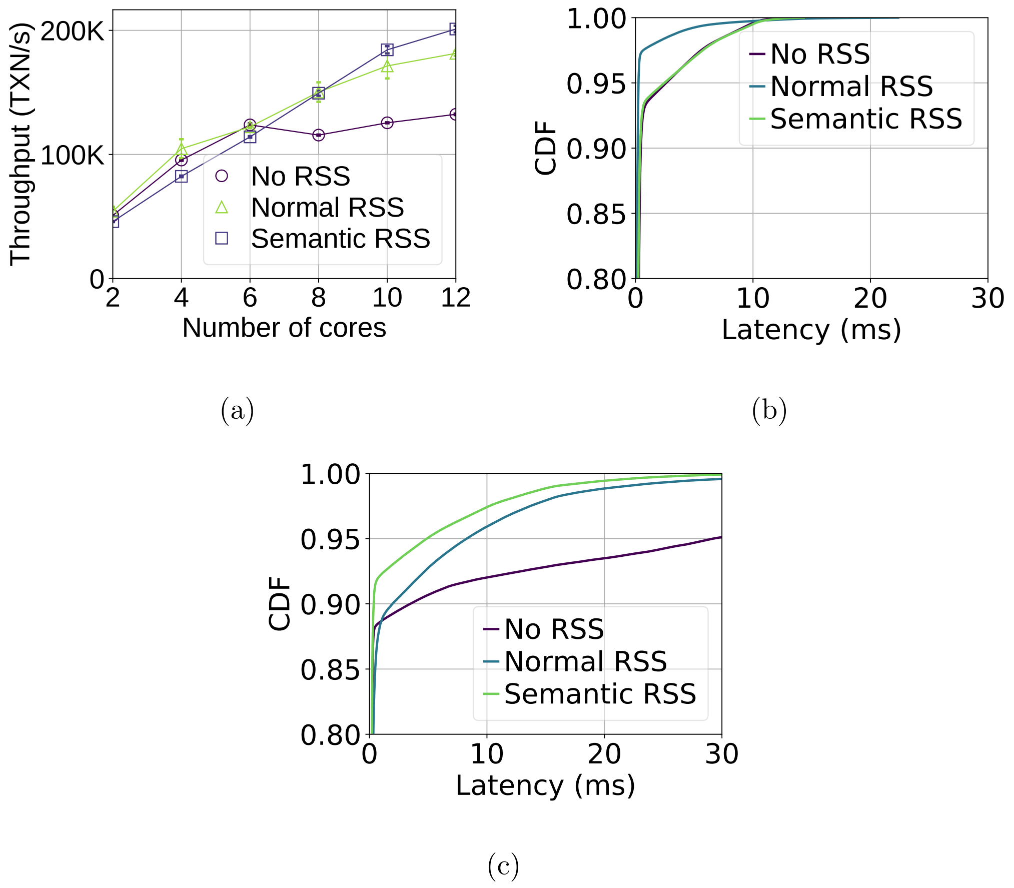


Figure 8.7. Steering throughput (a) and latency CDF for 6 (b) and 12 (c) cores.

to indicate that a new packet arrived. Without RSS, a single core is interrupted. With RSS, the cores are selected via hashing some fields of the packet's IP headers. Lastly, semantic RSS delivers each packet to the primary database partition the transaction refers to.

Figure 8.7a shows the effect of varying the number of cores on transaction throughput. In this experiment, RSS brings an improvement when compared to single-core interrupts. Semantic RSS delivers some further improvement for higher numbers of cores. Moreover, semantic RSS also reduces tail latency. Figures 8.7b and 8.7c show that latency significantly improves using Semantic RSS as we move from 6 to 12 cores.

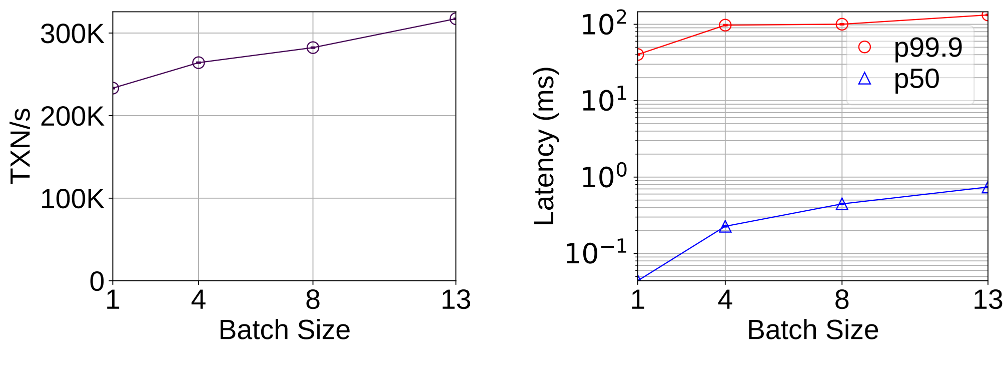


Figure 8.8. Throughput and latency due to batching.

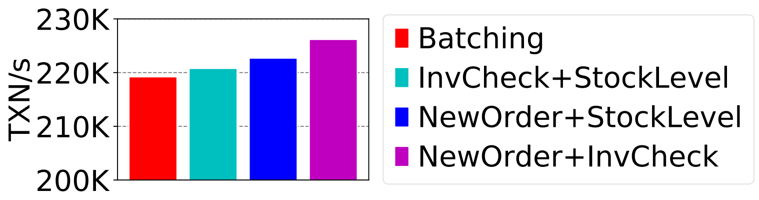


Figure 8.9. Transaction pairs with different affinities.

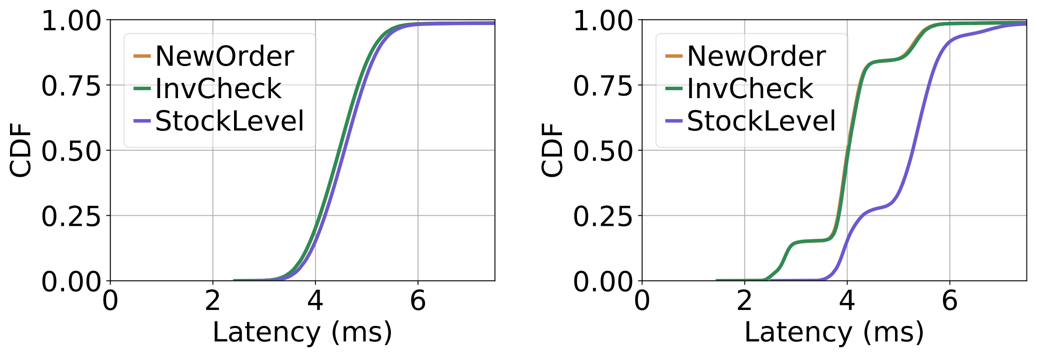
8.6.3 Batching Experiments

To test the idea that the batch size affects throughput, we configured the switch to send increasingly larger batches. Due to switch resources, our implementation stores the metadata of up to 12 transactions and unloads them at the 13th packet. Figure 8.8a shows the throughput for increasing batch sizes with the TPC-C benchmark. The most significant speedup is between no batching (i.e., batch size 1) and a batch size of 4. There are diminishing benefits for larger batch sizes. As the batch size increases, the overhead of packet processing becomes smaller, relative to transaction execution.

Creating batches on the switch requires queuing transactions as they arrive. As expected, batching increases the average latency, as Figure 8.8b shows. We assume in the experiments that batches do not time out. In practice, the switch control plane would send regular packets into the switch to unload batches that reach a given latency threshold.

8.6.4 Reordering Experiments

To evaluate how transaction reordering affects performance, we try separating transactions according to their *affinity*. The affinity criterion we use is transac-



(a) (b)

Figure 8.10. Batching and affinity pairing latency. NewOrder and InvCheck overlap on both charts.

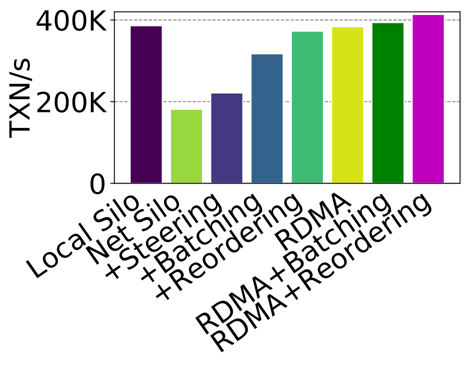
tions that perform similar work. To apply such an effect in a controlled manner, we created a new TPC-C transaction called InvCheck, which is similar to the read portion of NewOrder. (It reads the inventory but does not change it.)

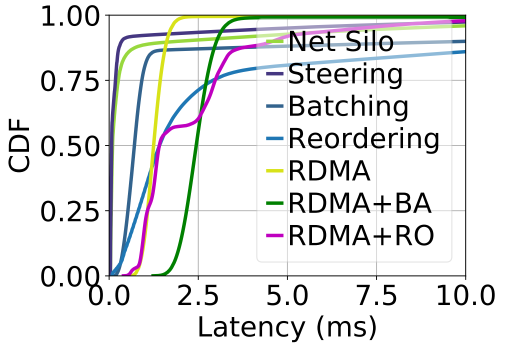
Figure 8.9 shows the result of running the NewOrder, InvCheck, and StockLevel transactions. The switch was programmed to batch the transactions randomly or reorder them using various combinations. When NewOrder is paired with InvCheck, the throughput is the highest. This suggests that other pairings have lower affinity.

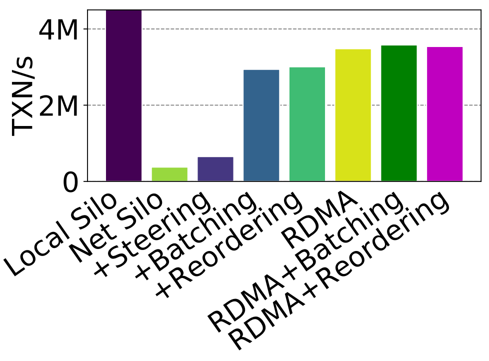
Figures 8.10a and 8.10b show the per-transaction latencies for the baseline and high-affinity reordering, respectively. In the baseline, as the experiment does not queue the transactions, all the transaction types have a similar latency. With the high-affinity reordering, however, the two transactions that are grouped, NewOrder and InvCheck, have the same average latency, which is lower than that of StockLevel.

8.6.5 Comparing UDP/IP and RDMA stacks

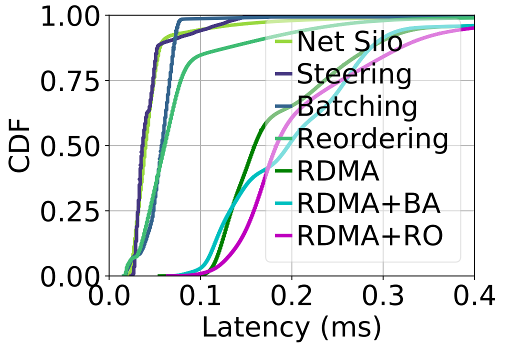
Figure 8.11a compares the throughput obtained by applying our triaging techniques on a TPC-C workload. We use `LocalSilo` as a baseline, i.e., the pre-generated workload loaded into the server's memory. `LocalSilo` runs at 386 Ktps. We then add one triaging technique at a time, starting from the original UDP/IP stack, which runs at 182 Ktps. This represents a networking overhead of 53%. By the time we are running all the techniques, the throughput increases to 373 Ktps, a 2.05× improvement over `NetSilo`. This amounts to 97% of the original local throughput. Hence, our triaging techniques almost entirely compensate







(c)



(d)

Figure 8.11. TPC-C throughput (a) and latency (b) at 80% of maximum throughput. YSCB throughput (c) and latency (d) at 80% of maximum throughput.

for the network overhead on a UDP/IP stack.

We implemented a different networking module for `NetSilo` that uses two-sided RDMA SEND over Unreliable Datagrams (UD). This setting best approximates the implementation decisions we took for the UDP/IP stack. We test the RDMA stack, which reaches 383 Ktps, as Figure 8.11a shows. The numbers reflect how efficient that stack already is; the networking overhead amounts to less than 1%. Silo becomes the bottleneck for the performance as opposed to the network. We add triaging techniques to the RDMA stack, one by one as before. The final throughput is 414 Ktps, a 1.08× improvement over the already efficient RDMA stack. To put things in perspective, our techniques allow the execution of additional 31 Ktps over this stack. As we will show shortly, batching and reordering allow transactions to be more efficiently executed on the server.

Figure 8.11b shows the latency CDF curves for each of the scenarios above.

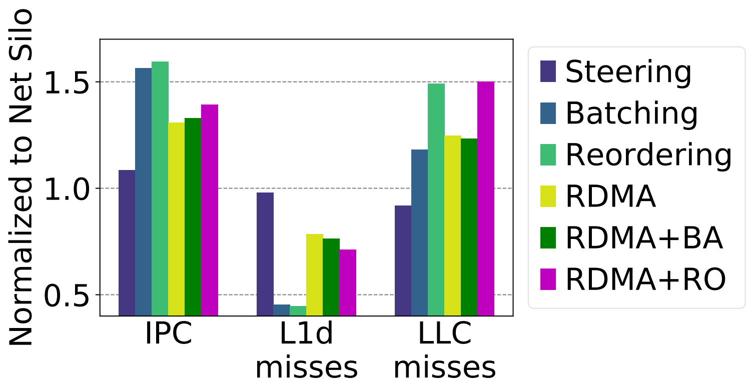


Figure 8.12. TPC-C CPU micro-architecture analysis.

Steering improves on the `NetSilo` baseline because it reduces the overhead of delivering the transactions. All the other techniques hurt latency to some extent. This is expected, as we buffer transactions on the switch.

Figure 8.12 shows micro-architecture measurements for the scenarios above. Each technique improves the instructions per clock (IPC) value of the previous technique and lowers the rate of L1 cache misses. We see, however, an increase in LLC misses. This is not uncommon; batching/reordering of transactions makes it so that more data is touched per unit of time, hence yields more last layer cache misses [128].

8.6.6 Comparing TPC-C and YCSB

Figure 8.11c compares the throughput obtained by combining the triaging techniques on a YCSB workload. As YSCB transactions are very light, almost all the response time is due to networking overhead. In our experiments, the throughput of `LocalSilo` is 17.5 Mtps, whereas the UDP/IP `NetSilo` delivers 377 Ktps, a 98% networking overhead. Although the combined techniques over UDP/IP do not match the throughput of `LocalSilo`, batching and reordering have the most substantial speedup: they deliver 3 Mtps, an increase in throughput of 7.95× compared to `NetSilo`. The more networking overhead, the more opportunities to recover it when applying TT techniques.

In contrast, the baseline RDMA delivers 3.48 Mtps and sees only marginal improvement from our techniques, at 3.58 Mtps when batching on that network stack. This is quite far from `LocalSilo`'s 17.5 Mtps. We attribute this discrepancy to our testing environment. Generating very high transaction throughput requires a notoriously high CPU cost [129, 130], something beyond the capacity of the machine on which we run our clients.

Figure 8.11d shows the latency CDF for the various techniques. As before, the impact on the UDP/IP stack is small. Here again, our techniques add more latency with the RDMA stack. This reflects how very small transactions can be more sensitive to latency. Note, however, that the x-axis in Figure 8.11d is measured in tens of milliseconds. For a mere 0.1 ms of additional latency, our techniques execute approximately 100 Ktps more transactions than before.

8.6.7 Evaluation Summary

Our experiments demonstrate that networking overhead has a significant impact on transaction processing performance. However, applying Transaction Triaging to in-flight transactions can compensate for the overhead by fostering better execution times: steering avoids the latency penalties that occur when delivering transactions to random cores; batching helps amortize the cost of receiving and sending packets across a larger number of transactions; reordering improves data and cache instructions reuse during transactions execution; and, lastly, protocol conversion takes advantage of high-speed, low-overhead networking where it is available. The effectiveness of these techniques improves with more cores, larger batches, and higher affinity. The results hold across diverse workloads.

8.7 Conclusion

In this chapter, we introduced Transaction Triaging, a set of techniques that execute in the center of the network, in order shape the stream of transactions before its delivery to a database server. We showed that performing Transaction Triaging can reverse the network overhead by providing server performance improvements. We also showed that our algorithms only use transaction metadata, and thus interact with existing database systems with very few changes. With more information about the transaction workload (i.e. transaction affinity), the techniques are even more effective.

As programmable networks become an off-the-shelf technology, we see our techniques as one more step towards revisiting the traditional separation of concerns between networking and database systems, allowing a new generation of systems to emerge.

Chapter 9

Related Work

In this chapter we provide an overview of related work. First, we discuss work that leverages the programmable data plane in general (§ 9.1). Then, we compare our INC classification to other recent classifications (§ 9.2). The rest of the sections discuss related work that is specific to each of the INC services we implemented.

9.1 Dataplane Programming

Several recent projects have made heavy use of state in the forwarding plane to offload or accelerate systems services. Marple [29] uses stream processing techniques to process telemetry data on switches. Jose et al. [131] describe a congestion control mechanism that leverages switch statistics.

Several recent projects have explored moving application logic into programmable network devices. Dang et al. [33] proposed the idea of moving consensus logic in to network devices. Paxos Made Switch-y [132] describes an implementation of Paxos in P4. István et al. [133] implement Zookeeper's atomic broadcast on an FPGA. Speculative Paxos [134] and NoPaxos [31] use programmable hardware to increase the likelihood of in-order delivery, and leverage that assumption to optimize consensus à la Fast Paxos [135]. NetCache [32] implements a key-value store. NetChain [34] provides a network implementation of a coordination service. HovercRaft [36] offloads the packet processing of the consensus leader to a programmable switch. R2P2 [24] uses a switch to accelerate the routing of RPC messages.

9.2 INC Classifications

Recently, there have been various classifications of INC applications and hardware. Benson [22] presents a taxonomy of functionality that can be deployed with INC, which includes: caching, network function virtualization (NFV), consensus, machine learning and stream processing. Ports and Nelson [23] include the following primitives in their classification: sequencing, replicated storage, caching, deep neural network (DNN) training, DNN inference, database reductions, database hash joins, virtual networking and telemetry. Furthermore, for comparing the primitives, they use objective metrics: operations per packet, state per packet, and packet gain. McCauley et al. [136] describe classes of applications suitable for INC (load balancing, telemetry, scheduling and congestion control) and unsuitable (aggregation and coordination). Yuta et al. [21] compare the suitability of INC hardware (smart NICs, SoCs, FPGAs and switch ASICs) for three applications: a key value store, a consensus protocol and a domain name system (DNS) server.

Our hardware classification in Section 3.1 does not distinguish the type of INC NIC (i.e. it groups SoCs together with smart NICs). The categories of building blocks we present in Section 3.2 provide a higher level of abstraction to applications than the application categories in the related work mentioned above.

9.3 Publish/Subscribe

Packet subscriptions are related to pub/sub messaging, network languages, and information-centric networking.

Publish/subscribe messaging system. Many application-level middleware messaging services provide pub/sub communication, such as Kafka, ActiveMQ, and Siena [60]. Eugster et al. provide a comprehensive survey [47].

Network programming languages. Several languages support the control-plane configuration of switches, including Frenetic [137], Pyretic [138], Merlin [139], and NetKAT[140]. In contrast to this work, packet subscriptions provide stateful filtering rules that realize a form of in-network processing, and therefore amount to data-plane programs. Marple [29] evaluates telemetry queries in-network. BDDs have long been used as a compressed representation of relations. In networking, BDDs have been used to verify network properties [141], check network configurations [142], and optimize compilation of OpenFlow rules [53]. Our compiler also uses BDD as an efficient internal representation, but differs from this prior work in that it generates a switch pipeline configuration.

Information-centric networking. With information-centric networking (ICN), packets are routed using symbolic names rather than network addresses. Some ICN architectures support the rich pub/sub semantics of packet subscriptions [143], but the mainstream architectures (CCN and NDN) are based on a "pull" model and on a stateless prefix matching that is significantly less expressive than the content-based and stateful filtering of packet subscriptions. In any case, prior work in ICN achieves a maximum throughput that is well below the line-rate throughput of packet subscriptions [144, 145, 146, 147].

9.4 Stream Processing

P4 Benchmarks. Whippersnapper [148] is a P4 benchmark. NOCC (chapter 7) describes an implementation of a benchmark as a case study for stateful data-plane programming.

Implementations of Linear Road. Linear Road was first described in a language-agnostic logical specification [66]. It has since been implemented for various streaming systems. Notably, a version written in CQL [69] ran on the Stanford STREAM data stream management system [62]. It was used to benchmark the Aurora [63] and Borealis [149] streaming engines. Jain et al. describe an implementation written in SPL [150] running on IBM's Infosphere Streams [64].

9.5 String Search

DFAs. Prior work has explored compacting DFAs [151]. Sherwood et al. [152] proposed splitting a DFA into DFAs that match a subset of the input bits. The NetKat [53] compiler matches packets using a Binary Decision Diagram, which has a structure similar to a DFA.

Hardware solutions. There are several techniques for using TCAMs, including compacting transition symbols and states [153], using LPM to share state [154] and variable striding [155]. DFC [99] uses cache-friendly data structures for pattern matching on general purpose CPUs. Others have implemented pattern matching on GPUs [97, 156, 157]. HAWK [158] implements an FPGA pipeline using a bitsplit technique proposed by Sherwood et al.. HARE [159] adds RegExp support to HAWK by adding a character class translation stage to the pipeline, as well as counters for RegExp quantifiers. Sapio et al. [37] also perform aggregation tasks in programmable switches. To the best of our Knowledge, no prior work on programmable switches performs string searches.

9.6 Optimistic Concurrency Control

NOCC is superficially similar to Eris [35], in the sense that they both use programmable switches to accelerate transaction processing. However, they have very different execution models. The Eris model is based on prior work on independent transactions [114, 160], in which transactions are ordered first, and then executed. In contrast, with the NOCC model, clients pre-execute transactions locally, and then submit the result for validation (i.e., ordering).

Proxies and caches. The idea of using a proxy to extend distributed services is a well-established idea [161] that has been widely adopted [162, 163, 164, 165]. Proxies are often used to scale services by caching copies of data closer to clients, such as with content distribution networks (CDNs) [166, 167, 168]. CDNs typically are used for static content, although there are examples of proxies used for dynamic content [169]. Prior work has also explored the possibility of leveraging the network to route requests dynamically to proxies to service requests [168]. Notably, SwitchKV [31] uses OpenFlow-enabled switches to dynamically route read requests to proxy caches. NetCache [32] provides a P4-based implementation of a key-value store to cache hot-data items for highly skewed read workloads. NOCC differs from this work in that it is not a cache, per se. It keeps copies of transaction requests, but it does not service client read requests. Rather, it uses copies of previous requests to make informed decisions about when to abort transactions early, with the goal of reducing latency for write-heavy workloads.

9.7 Transaction Triaging

We divide the related work into two broad categories. First, we consider work that used similar underlying transaction management techniques to those we presented, although without relying on networking support. Then, we compare work that also provides low-overhead networking.

Transaction Management Techniques. Partitioning databases is a common way to optimize transaction management and achieve scalability in multi-core systems [170, 171, 172, 114, 124]. These systems schedule transactions immediately upon arrival and can benefit from having the network deliver transactions to a core respecting the database partitioning.

Steering techniques that go beyond RSS have been proposed in the context of OS support for low-latency transactions [173, 174, 175, 176]. One of the works even leverages a programmable NIC [177]. These techniques try to re-assigning cores to applications in case some cores find themselves with more work than others. Our techniques, in contrast, do not require any changes to the OS.

Transaction batching is a widespread technique that databases use in different execution stages: during transaction execution [113], logging of transactions (group commit) [178, 179], and at replication time [180]. We perform batching in-network, which eliminates the cost of doing so on the database server.

Transaction reordering is also a known technique. Many systems seek to minimize concurrency conflict in such a way [181, 113, 115]. The schedulers in those systems try to select a next transactions to execute that would cause the minimal interference to ongoing transactions. These techniques are complementary to ours. Reordering has also appeared in the context of maximizing resource sharing during execution [116, 117]. We have shown similar improvements, although by resorting to in-network mechanisms.

RDMA and fast networking. Several database algorithms take advantage RDMA-enabled networks [182, 183, 184, 185, 186, 187, 188, 189]. These works reflect the more recent change in systems in which networking becomes more powerful at the same time that CPU's performance plateaus [190]. Our work broadly falls into this category but it is unique in that it leverages both network programmability and fast networking.

Chapter 10

Conclusion

This thesis explores how INC can be leveraged to provide services that improve application performance. By exposing some application-level information, INC services can either perform computation or make optimizations that increase application throughput and reduce latency. Overall, this thesis made the following contributions:

Service Characterization. We used a principled approach to characterize services that can be provided by INC. We began by identifying the properties of INC devices that give them a comparative advantage to software implementations (see § 3.1). The most important property is the ratio between I/O and processing expressiveness; INC is particularly well suited for applications that are I/O bound, but require some moderately expressive processing. Conversely, INC is not suitable for applications that require very expressive processing, a lot of state, and do not benefit from fast I/O.

Building Blocks. This thesis described application agnostic services that can be provided by INC (see § 3.2). By making services reusable, they can be used by a wide variety of applications. Furthermore, we showed that these services are built on top of some common data structures: pub/sub and string search both use an automata (DFA); stream processing uses maps, counters and sets; OCC uses maps; and TT uses queues. We believe that these data structures can be adapted to implement other types of services.

Separation of Concerns. While some of the services can be deployed transparently to applications, others require the active cooperation of applications. For example, PPS can be deployed transparently as bump-in-the-wire (see § 6.2.1), searching arbitrary packets that pass through the switch; on the other hand, Pub-/sub, stream processing, OCC and TT require the application to include specific information in the packet. We believe this integration with the application is justified by the performance improvements.

INC Techniques. Although some INC services have processing that seems simple, implementing it on an INC device is not so straight-forward (see § 2.4.3). We presented some common data structures that can be used for diverse services (§ 3.3.1). Furthermore, we described the techniques for implementing the services on switches (§ 3.3.2). This includes simple techniques for handling state on the switch, as well as more complicated techniques for implementing unbounded computation like iteration.

Evaluation on Hardware. We built five different services that run on programmable switches. The artifacts in themselves validate the feasibility of implementing the data structures and techniques we described. For the evaluation, we used real-world workloads and integrated with existing systems. The results show promising results compared to software baselines, as well as compared to other hardware devices. Our evaluation showed that INC services provide performance improvements to applications while reducing CPU utilization on end hosts, with potential savings in energy and cost.

10.1 Results

To evaluate the hypothesis of this thesis, we built several systems to answer the research questions presented in Section 1.2. Below we provide a summary answer to each question, referencing previous sections with more detail for specific systems.

What types of applications can benefit from INC? There are three main criteria for an application to benefit from INC. First, it must be possible to implement an application in the network, given the expressiveness and computational constraints of the devices. Second, there must be a comparative advantage of executing application logic in the network. The most important indicator of application suitability is the I/O to processing expressiveness ratio: I/O bound applications with moderate complexity benefit the most from INC. The string search and stream processing applications especially exhibit these characteristics (see § 6.7 and § 5.6). Finally, applications that perform less I/O can also benefit from INC, because of its centrally located processing. This is the case for pub/sub, NOCC and Transaction Triaging (see § 4.6, § 7.5 and § 8.7).

What is the right level of granularity for INC building blocks? This question is about where to place the interface between the application and the building block. Some building blocks, like messaging and string search, provide a fairly high-level interface that can be used by a wide variety of applications (see § 4.6 and § 6.2.1). Others, like stream processing and transaction triaging, are closer to the application and thus execute more application logic (see § 5.6 and § 8.7). There is a trade-off between granularity and performance: granular building blocks can be used by more applications, but at the expense of performance.

What and how much application-level information should be exposed to the network? Some applications can use INC transparently. For example, PPS can be deployed as a bump-in-the-wire appliance (see § 6.2.1). Pub/sub is more flexible because it lets the application decide what information should be used for forwarding (see § 4.4). On the other hand, some applications must be tightly integrated: in stream processing and Transaction Triaging, the switch must be fully aware of application data structures (see § 5.4 and § 8.7). Ideally, the building block should use the minimum amount of application-level information required to provide a speed-up to the application. Stream processing, for example, can provide a large speed-up to the application, but this comes at the expense of reusability—this building block cannot be easily ported to other applications (see § 5.6).

What are the potential performance benefits? To answer this question, we compared INC to other software or hardware implementations for the same applications. We found that applications that have to process large quantities of data have the largest speed-ups compared to software implementations. This is the case for pub/sub (see § 4.5), stream processing (see § 5.5) and string search (§ 6.6). For the latter, INC even provided performance (and cost benefits) compared to other hardware solutions. Even applications that are not I/O bound, like NOCC (§ 7.4) and transaction triaging (§ 8.6) benefited, benefited from INC; the central location of the switch helped to manage the workload, reducing the load on the server.

What must the application trade-off, in terms of features or expressiveness? Although INC can speed-up applications, we found that it can come at the expense of application functionality. In some cases, it is not possible to fully implement complex operations on the switch (e.g., some queries from the Linear Road benchmark (see § 5.3.2). Some application workloads may require more memory than is available on the switch. This limitation can be traded-off for throughput or accuracy (e.g., PPS can store hashes instead of entire patterns, § 6.3.2). On the other hand, building blocks that do not look deep into the application data (e.g., pub/sub and transaction triaging), do not compromise the expressiveness of the application, but only provide performance improvements.

10.2 Future Work

This thesis focuses on five INC systems that we built to explore the benefits that INC can bring to applications. Each system provides different insight into INC systems, including the implementation challenges and techniques that can be used. Although we believe these systems are representative of a broad range of INC capabilities, we could have made different design decisions or drilled deeper into each application. For each system, below we describe alternative approaches or what we would have done if we had more time.

Publish/Subscribe An important feature of messaging systems is reliability: when a client sends a message, it would like guarantees of reliable delivery. Our Packet Subscriptions system does not ensure reliable delivery. Adding presents two main challenges. First, unlike one-to-one communication systems, pub/sub is one-to-many (multicast), and the sender is not aware of the recipients. It is difficult for receivers to detect packet loss in such settings. Second, reliable delivery can reduce throughput and add latency. To tackle this, we would likely want to use a NACK based scheme, which only adds latency in the unlikely event of packet loss.

Stream Processing The Linear Road benchmark is intended to be representative of stream processing workloads. The benchmark, however, does not exhaustively test all aspects of a stream processing system. Ideally, we would have used different types of analytics workloads (i.e. other than Linear Road) and more varied queries. This could provide more insights into operators that INC does or does not provide.

String Search There are many string search algorithms, and we chose the Aho-Corasick because it was the best fit for PISA. Implementing other algorithms may not yield positive results. However, there are some extensions to Aho-Corasick that could increase the system throughput or reduce the memory footprint. For example, before executing the DFA, the switch could translate the symbols in the input stream into a more compact representation. We are not sure this would be feasible in the current generation of switches we are using, but it may be possible with future generations with more memory.

Currently, for multiple DFAs, we randomly partition the patterns multiple times and pick the partitioning that yields DFAs with the fewest states. This is an approximation of the optimal partitioning, which we believe would require performing an exhaustive search. We leave finding the optimal partitioning for future work.

Efficient data levitation (see Section 6.4) is critical for reducing the load on storage machines. More work would be needed with bypass techniques like RDMA to efficiently move data from the servers to the switch.

We evaluated PPS functioning as a *dedicated appliance*. PPS can also be deployed as *bump-in-the-wire*, to provide, for example, a network intrusion detection system (NIDS). However, this scenario requires the original packet to pass through unmodified. This would require more work on holding the original packet while PPS scans a copy.

Optimistic Concurrency Control Our NOCC implementation is designed for an environment where clients execute transactions speculatively, and send the result to the server for validation. More work would be necessary to integrate with other systems. We began an initial feasibility study for integration with Galera Cluster [191], a distributed version of MySQL. We also envision combining NOCC with complimentary techniques for read-heavy workloads, e.g., using a cache to service read requests [32, 31].

Database Transaction Triaging This work presents a technique that reorders transactions based on their affinity. We evaluated reordering with one affinity metric: transaction type. However, there are more types of affinity. Some other candidate affinity types could be transaction working set and transaction size. Preliminary experiments showed that the reordering based on transaction size reduces latency for small transactions. More work is needed to validate reordering using these alternative affinity metrics.

10.3 Final Remarks

The advent of programmable hardware is not just a revolution for network-centric applications, but for applications in general. The most suitable applications for INC are those which can be expressed in the programmable hardware, are I/O bound, and benefit from being centrally located in the network. By exposing the right level of application-specific information to the network, applications can offload computation to reusable INC building blocks. Overall, this thesis revisits the separation of concerns between the application and the network, showing that co-design is not only possible, but also beneficial.

Bibliography

[1] Theo Jepsen, Masoud Moshref, Antonio Carzaniga, Nate Foster, and Robert Soulé. Life in the fast lane: A line-rate linear road. In *ACM SIGCOMM Symposium on SDN Research (SOSR)*, pages 10:1–10:7, March 2018.

[2] Theo Jepsen, Leandro Pacheco de Sousa, Masoud Moshref, Fernando Pedone, and Robert Soulé. Infinite resources for optimistic concurrency control. In *Proceedings of the 2018 Morning Workshop on In-Network Computing*, pages 26–32, New York, NY, USA, August 2018.

[3] Theo Jepsen, Masoud Moshref, Antonio Carzaniga, Nate Foster, and Robert Soulé. Packet subscriptions for programmable asics. In *Workshop on Hot Topics in Networks*, pages 176–183, New York, NY, USA, November 2018.

[4] Theo Jepsen, Daniel Alvarez, Nate Foster, Changhoon Kim, Jeongkeun Lee, Masoud Moshref, and Robert Soulé. Fast string searching on pisa. In *ACM SIGCOMM Symposium on SDN Research (SOSR)*, pages 21–28, New York, NY, USA, April 2019.

[5] DPDK. `http://dpdk.org/`.

[6] Infiniband architecture specification. `https://www.infinibandta.org/ibta-specifications-download/`.

[7] Jeffrey C. Mogul. Tcp offload is a dumb idea whose time has come. In *9th Workshop on Hot Topics in Operating Systems*, page 5, USA, 2003. USENIX Association.

[8] Srihari Makineni, Ravi Iyer, Partha Sarangam, Donald Newell, Li Zhao, Ramesh Illikkal, and Jaideep Moses. Receive side coalescing for accelerating tcp/ip processing. In *Proceedings of the 13th International Conference*

on High Performance Computing, HiPC'06, pages 289–300, Berlin, Heidelberg, 2006. Springer-Verlag.

[9] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: taking control of the enterprise. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 1–12, August 2007.

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

[11] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 99–110, August 2013.

[12] Xilinx Alveo. `https://www.xilinx.com/products/boards-and-kits/alveo.html`, 2020.

[13] Agilio CX SmartNICs - Netronome. `https://www.netronome.com/products/agilio-cx/`, 2020.

[14] Noa Zilberman, Yury Audzevich, G Adam Covington, and Andrew W Moore. Netfpga sume: Toward 100 gbps as research commodity. *IEEE micro*, 34(5):32–41, 2014.

[15] Barefoot Tofino. `https://www.barefootnetworks.com/products/brief-tofino/`, 2020.

[16] Broadcom Trident 3. `https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56870-series`, 2020.

[17] Nik Sultana, Salvator Galea, David Greaves, Marcin Wojcik, Jonny Shipton, Richard Clegg, Luo Mai, Pietro Bressana, Robert Soulé, Richard Mortier, Paolo Costa, Peter Pietzuch, Jon Crowcroft, Andrew W Moore, and Noa Zilberman. Emu: Rapid prototyping of networking services. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 459–471, Santa Clara, CA, 2017. USENIX Association.

[18] G. Brebner and Weirong Jiang. High-Speed Packet Processing using Reconfigurable Computing. *IEEE/ACM International Symposium on Microarchitecture*, 34:8–18, January 2014.

[19] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM Computer Communication Review (CCR)*, 44(3):87–95, July 2014.

[20] Muhammad Shahbaz, Sean Choi, Ben Pfaff, Changhoon Kim, Nick Feamster, Nick McKeown, and Jennifer Rexford. Pisces: A programmable, protocol-independent software switch. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 525–538, 2016.

[21] Yuta Tokusashi, Huynh Tu Dang, Fernando Pedone, Robert Soulé, and Noa Zilberman. The case for in-network computing on demand. In *EuroSys*, EuroSys '19, New York, NY, USA, March 2019. Association for Computing Machinery.

[22] Theophilus A. Benson. In-network compute: Considered armed and dangerous. In *17th Workshop on Hot Topics in Operating Systems*, HotOS '19, pages 216–224, New York, NY, USA, May 2019. Association for Computing Machinery.

[23] Dan R. K. Ports and Jacob Nelson. When should the network be the computer? In *17th Workshop on Hot Topics in Operating Systems*, HotOS '19, pages 209–215, New York, NY, USA, May 2019. Association for Computing Machinery.

[24] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2p2: Making rpcs first-class datacenter citizens. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 863–880, Renton, WA, July 2019. USENIX Association.

[25] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, and Lawrence J Wobker. In-band network telemetry via programmable dataplanes. `https://nkatta.github.io/papers/int-demo.pdf`, 2015.

[26] Mojgan Ghasemi, Theophilus Benson, and Jennifer Rexford. Dapper: Data plane performance diagnosis of tcp. In *ACM SIGCOMM Symposium on*

SDN Research (SOSR), SOSR '17, pages 61–74, New York, NY, USA, 2017. Association for Computing Machinery.

[27] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Distributed network monitoring and debugging with switchpointer. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 453–456, Renton, WA, April 2018. USENIX Association.

[28] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, SIGCOMM '17, pages 15–28, New York, NY, USA, 2017. Association for Computing Machinery.

[29] 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 *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 85–98, August 2017.

[30] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven streaming network telemetry. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 357–371, August 2018.

[31] Xiaozhou Li, Raghav Sethi, Michael Kaminsky, David G. Andersen, and Michael J. Freedman. Be fast, cheap and in control with switchkv. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 31–44, March 2016.

[32] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *ACM Symposium on Operating Systems Principles (SOSP)*, October 2017.

[33] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. NetPaxos: Consensus at Network Speed. In *ACM SIGCOMM Symposium on SDN Research (SOSR)*, pages 59–73, June 2015.

[34] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. NetChain: Scale-Free Sub-RTT Coordination. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, April 2018.

[35] Jialin Li, Ellis Michael, and Dan R. K. Ports. Eris: Coordination-free consistent transactions using in-network concurrency control. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 104–120. ACM, October 2017.

[36] Marios Kogias and Edouard Bugnion. Hovercraft: Achieving scalability and fault-tolerance for microsecond-scale datacenter services. In *EuroSys*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[37] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. In-network computation is a dumb idea whose time has come. In *Workshop on Hot Topics in Networks*, pages 150–156, November 2017.

[38] A. Lerner, R. Hussein, and P. Cudré-Mauroux. The case for network accelerated query processing. In *Conference on Innovative Data Systems Research*, 2019.

[39] D. Goncalves, S. Signorello, F. M. V. Ramos, and M. MÃľdard. Random linear network coding on programmable switches. In *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 1–6, September 2019.

[40] David Clark. The design philosophy of the DARPA Internet Protocols. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 106–114, August 1988.

[41] Petr Lapukhov. Internet-scale virtual networking using identifier-locator addressing. `https://www.nanog.org/sites/default/files/20161018_Lapukhov_Internet-Scale_Virtual_Networking_v1.pdf`, 2016.

[42] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software

network load balancer. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 523–535, March 2016.

[43] Open-sourcing Katran, a scalable network load balancer. `https://code.fb.com/open-source/open-sourcing-katran-a-scalable-network-load-balancer/`, 2018.

[44] Tibco rendezvous. `https://www.tibco.com/products/tibco-rendezvous`, 2019.

[45] Jay Kreps, Neha Narkhede, and Jun Rao. Kafka: A distributed messaging system for log processing. In *6th International Workshop on Networking Meets Databases (NetDB)*, June 2011.

[46] IBM MQ. `https://www-03.ibm.com/software/products/en/ibm-mq`, 2019.

[47] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, 35(2):114–131, June 2003.

[48] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 13–24, August 2012.

[49] Rahul Potharaju and Navendu Jain. Demystifying the dark side of the middle: A field study of middlebox failures in datacenters. In *ACM SIGCOMM Internet Measurement Conference (IMC)*, pages 9–22, October 2013.

[50] Nasdaq TotalView-ITCH 5.0 - Nasdaq Trader. `https://www.nasdaqtrader.com/content/technicalsupport/specifications/dataproducts/NQTVITCHspecification.pdf`, 2019.

[51] Micah Adler, Zihui Ge, James F. Kurose, Don Towsley, and Stephen Zabele. Channelization problem in large scale data dissemination. In *Proc. 9th Int. Conf. on Network Protocols*, pages 100–109, November 2001.

[52] How to Build an Exchange. `https://blog.janestreet.com/how-to-build-an-exchange/`, 2017.

[53] Steffen Smolka, Spiridon Eliopoulos, Nate Foster, and Arjun Guha. A fast compiler for netkat. In *International Conference on Functional Programming (ICFP)*, pages 328–341, September 2015.

[54] Xin Jin, Jennifer Gossels, Jennifer Rexford, and David Walker. Covisor: A compositional hypervisor for software-defined networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 87–101, Oakland, CA, May 2015.

[55] S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers (TC)*, 27(6):509–516, June 1978.

[56] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers (TC)*, 35(8):677–691, August 1986.

[57] William Chan, Richard Anderson, Paul Beame, and David Notkin. Combining constraint solving and symbolic model checking for a class of systems with non-linear constraints. In *International Conference on Computer Aided Verification (CAV)*, pages 316–327, June 1997.

[58] Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Călin Caşcaval, Nick McKeown, and Nate Foster. P4v: Practical verification for programmable data planes. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 490–503, August 2018.

[59] Siena Synthetic Benchmark Generator. `http://www.inf.usi.ch/carzaniga/cbn/forwarding/`, 2019.

[60] Antonio Carzaniga and Alexander L. Wolf. Forwarding in a content-based network. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 163–174, August 2003.

[61] Apache. Apache activemq. http://activemq.apache.org/, 2019.

[62] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Principles of Database Systems (PODS)*, pages 1–16, June 2002.

[63] Daniel J. Abadi, Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan

Zdonik. Aurora: A new model and architecture for data stream management. In *The VLDB Journal*, volume 12, pages 120–139, August 2003.

[64] Navendu Jain, Lisa Amini, Henrique Andrade, Richard King, Yoonho Park, Philippe Selo, and Chitra Venkatramani. Design, implementation, and evaluation of the linear road bnchmark on the stream processing core. In *ACM SIGMOD*, pages 431–442, June 2006.

[65] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[66] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. Linear road: a stream data management benchmark. In *30th International Conference on Very Large Data Bases*, pages 480–491. VLDB, August 2004.

[67] Linear Road. `http://www.cs.brandeis.edu/~linearroad/`.

[68] Naveen Kr Sharma, Antoine Kaufmann, Thomas E Anderson, Arvind Krishnamurthy, Jacob Nelson, and Simon Peter. Evaluating the power of flexible packet processing for network resource allocation. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 67–82, March 2017.

[69] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL continuous query language: semantic foundations and query execution. In *The VLDB Journal*, volume 15, pages 121–142, June 2006.

[70] Buğra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu, and Myungcheol Doo. SPADE: The System S declarative stream processing engine. In *ACM SIGMOD*, pages 1123–1134, June 2008.

[71] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: A not-so-foreign language for data processing. In *ACM SIGMOD*, pages 1099–1110, June 2008.

[72] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–14, December 2008.

[73] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. StreamIt: A language for streaming applications. In *11th International Conference on Compiler Construction*, pages 179–196, April 2002.

[74] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming*, pages 277–298, 2005.

[75] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970.

[76] Kyu-Young Whang, Brad T Vander-Zanden, and Howard M Taylor. A linear-time probabilistic counting algorithm for database applications. *ACM Transactions on Database Systems*, 15(2):208–229, 1990.

[77] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In *AofA: Analysis of Algorithms*, pages 137–156, June 2007.

[78] Naveen Kr Sharma, Antoine Kaufmann, Thomas E Anderson, Arvind Krishnamurthy, Jacob Nelson, and Simon Peter. Evaluating the power of flexible packet processing for network resource allocation. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 67–82, March 2017.

[79] In-Network DDoS Detection. `https://barefootnetworks.com/use-cases/in-nw-DDoS-detection`, 2017.

[80] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J Franklin, Benjamin Recht, and Ion Stoica. Drizzle: Fast and adaptable stream processing at scale. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 374–389. ACM, October 2017.

[81] Darin Stewart. Big content: The unstructured side of big data. `https://blogs.gartner.com/darin-stewart/2013/05/01/big-content-the-unstructured-side-of-big-data/`, May 2013.

[82] Shoumik Palkar, Firas Abuzaid, Peter Bailis, and Matei Zaharia. Filter before you parse: Faster analytics on raw data with sparser. *The VLDB Journal*, 11(11):1576–1589, July 2018.

[83] Sangjin Han, Norbert Egi, Aurojit Panda, Sylvia Ratnasamy, Guangyu Shi, and Scott Shenker. Network support for resource disaggregation in next-generation datacenters. In *Workshop on Hot Topics in Networks*, pages 10:1–10:7, 2013.

[84] Noa Zilberman, Andrew W. Moore, and Jon A. Crowcroft. From photons to big-data applications: terminating terabits. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 374(2062), 2016.

[85] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. Understanding pcie performance for end host networking. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 327–341, August 2018.

[86] PCI Sig. Pci express base specifications revision 1.0 a. *PCI SIG*, 2003.

[87] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Communications of the ACM (CACM)*, 20(10):762–772, October 1977.

[88] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2):249–260, March 1987.

[89] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM (CACM)*, 18(6):333–340, June 1975.

[90] Daehyeok Kim, Yibo Zhu Zhu, Changhoon Kim, Jeongkeun Lee, and Srinivasan Seshan Seshan. Generic external memory for switch data planes. In *Workshop on Hot Topics in Networks*, November 2018.

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

[92] Do Le Quoc, Ruichuan Chen, Pramod Bhatotia, Christof Fetzer, Volker Hilt, and Thorsten Strufe. Streamapprox: Approximate computing for stream analytics. In *17th ACM/IFIP/USENIX International Conference on Middleware*, December 2017.

[93] A. Agarwal, M. Slee, and M. Kwiatkowski. Thrift: Scalable cross-language services implementation. `https://thrift.apache.org/static/files/thrift-20070401.pdf`, April 2007.

[94] Snort. http://snort.org/, 2018.

[95] The podesta emails. https://wikileaks.org/podesta-emails/, 2016.

[96] Introduction to tweet json. https://developer.twitter.com/en/docs/tweets/data-dictionary/overview/intro-to-tweet-json.html, 2016.

[97] Cheng-Liang Hsieh, Lucas Vespa, and Ning Weng. A high-throughput dpi engine on gpu via algorithm/implementation co-optimization. *Journal of Parallel and Distributed Computing*, 88:46–56, 2016.

[98] Helios product brief. http://titan-ic.com/assets/img/news/Final-RXPA-APR-18.pdf, 2018.

[99] Byungkwon Choi, Jongwook Chae, Muhammad Jamshed, KyoungSoo Park, and Dongsu Han. DFC: Accelerating string pattern matching for network applications. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 551–565, March 2016.

[100] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

[101] Rakesh Agrawal, Michael J. Carey, and Miron Livny. Concurrency control performance modeling: Alternatives and implications. *ACM Trans. Database Syst.*, 12(4):609–654, November 1987.

[102] Michael J. Carey and Michael Stonebraker. The performance of concurrency control algorithms for database management systems. In *10th International Conference on Very Large Data Bases*, pages 107–118, San Francisco, CA, USA, 1984. Morgan Kaufmann Publishers Inc.

[103] Rakesh Agrawal and David J. Dewitt. Integrated concurrency control and recovery mechanisms: Design and performance evaluation. *ACM Trans. Database Syst.*, 10(4):529–564, December 1985.

[104] Y. C. Tay, Nathan Goodman, and R. Suri. Locking performance in centralized databases. *ACM Trans. Database Syst.*, 10(4):415–462, December 1985.

[105] Peter Franaszek and John T. Robinson. Limitations of concurrency in transaction processing. *ACM Trans. Database Syst.*, 10(1):1–28, March 1985.

[106] Transaction Processing Performance Council. TPC-C Benchmark Revision 5.11.0. `http://www.tpc.org/tpcc/`, 2010.

[107] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., 1987.

[108] Andy Pavlo. Python TPC-C. `https://github.com/apavlo/py-tpcc`, 2020.

[109] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, pages 143–154, New York, NY, USA, June 2010. ACM.

[110] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1992.

[111] Mohammad Sadoghi, Spyros Blanas, and H. V. Jagadish. *Transaction Processing on Modern Hardware*. Morgan & Claypool Publishers, 2019.

[112] Infiniband architecture specification–annex a16: Roce. `https://www.infinibandta.org/ibta-specifications-download/`.

[113] Ippokratis Pandis, Ryan Johnson, Nikos Hardavellas, and Anastasia Ailamaki. Data-oriented transaction execution. *Proc. VLDB Endow.*, 3(1-2):928–939, September 2010.

[114] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (it's time for a complete rewrite). In *33th International Conference on Very Large Data Bases*, pages 1150–1160, 2007.

[115] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 1–12, New York, NY, USA, 2012. Association for Computing Machinery.

[116] Islam Atta, Pinar Tözün, Xin Tong, Anastasia Ailamaki, and Andreas Moshovos. Strex: Boosting instruction cache reuse in oltp workloads

through stratified transaction execution. In *40th International Symposium on Computer Architecture*, pages 273–284, New York, NY, USA, 2013. Association for Computing Machinery.

[117] Pinar Tözün, Islam Atta, Anastasia Ailamaki, and Andreas Moshovos. Addict: Advanced instruction chasing for transactions. *Proc. VLDB Endow.*, 7(14):1893–1904, October 2014.

[118] Network programming language. `https://nplang.org/`.

[119] Haoyu Song. Protocol-oblivious Forwarding: Unleash the Power of SDN Through a Future-proof Forwarding Plane. In *Workshop on Hot Topics in Software Defined Networking*, pages 127–132, August 2013.

[120] Hugo Krawczyk. Lfsr-based hashing and authentication. In *Proceedings of the 14th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '94, pages 129–139, London, UK, UK, 1994. Springer-Verlag.

[121] Naveen Kr. Sharma, Chenxingyu Zhao, Ming Liu, Pravein G Kannan, Changhoon Kim, Arvind Krishnamurthy, and Anirudh Sivaraman. Programmable calendar queues for high-speed packet scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 685–699, Santa Clara, CA, February 2020. USENIX Association.

[122] 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 *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, August 2016.

[123] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-store: A high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.*, 1(2):1496–1499, August 2008.

[124] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 18–32, New York, NY, USA, November 2013. ACM.

[125] Jose M. Faleiro and Daniel J. Abadi. Rethinking serializable multiversion concurrency control. *Proc. VLDB Endow.*, 8(11):1190–1201, July 2015.

[126] Neha Narula, Cody Cutler, Eddie Kohler, and Robert Morris. Phase reconciliation for contended in-memory transactions. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 511–524, Broomfield, CO, October 2014. USENIX Association.

[127] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. Tictoc: Time traveling optimistic concurrency control. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1629–1642, New York, NY, USA, 2016. Association for Computing Machinery.

[128] Utku Sirin, Pinar Tözün, Danica Porobic, and Anastasia Ailamaki. Microarchitectural analysis of in-memory oltp. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 387–402, New York, NY, USA, 2016. Association for Computing Machinery.

[129] Aniraj Kesavan, Robert Ricci, and Ryan Stutsman. To copy or not to copy: Making in-memory databases fast on modern nics. In *Data Management on New Hardware*, ADMS '16, pages 79–94. Springer, 2016.

[130] Yanfang Le, Brent Stephens, Arjun Singhvi, Aditya Akella, and Michael M. Swift. Rogue: Rdma over generic unconverged ethernet. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, pages 225–236, New York, NY, USA, 2018. Association for Computing Machinery.

[131] Lavanya Jose, Lisa Yan, Mohammad Alizadeh, George Varghese, Nick McKeown, and Sachin Katti. High speed networks need proactive congestion control. In *Workshop on Hot Topics in Networks*, November 2015.

[132] Huynh Tu Dang, Marco Canini, Fernando Pedone, and Robert Soulé. Paxos Made Switch-y. *SIGCOMM Computer Communication Review (CCR)*, 44:87–95, April 2016.

[133] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolić. Consensus in a Box: Inexpensive Coordination in Hardware. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 103–115, March 2016.

[134] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 43–57, March 2015.

[135] Leslie Lamport. Fast Paxos. *Distributed Computing*, 19:79–103, October 2006.

[136] James McCauley, Aurojit Panda, Arvind Krishnamurthy, and Scott Shenker. Thoughts on load distribution and the role of programmable switches. *SIGCOMM Computer Communication Review (CCR)*, 49(1):18–23, 2019.

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

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

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

[140] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. Netkat: Semantic foundations for networks. In *Symposium on Principles of Programming Languages (POPL)*, pages 113–126, January 2014.

[141] Hongkun Yang and Simon S Lam. Real-time verification of network properties using atomic predicates. In *IEEE International Conference on Network Protocols (ICNP)*, pages 1–11, October 2013.

[142] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A general approach to network configuration verification. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 155–168, August 2017.

[143] Michele Papalini, Antonio Carzaniga, Koorosh Khazaei, and Alexander L. Wolf. Scalable routing for tag-based information-centric networking. In *Proceedings of the 1st International Conference on Information-centric Networking (HotICN)*, pages 17–26, September 2014.

[144] Diego Perino, Matteo Varvello, Leonardo Linguaglossa, Rafael Laufer, and Roger Boislaigue. Caesar: A content router for high-speed forwarding on content names. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 137–148, October 2014.

[145] Yi Wang, Boyang Xu, Dongzhe Tai, Jianyuan Lu, Ting Zhang, Huichen Dai, Beichuan Zhang, and Bin Liu. Fast name lookup for named data networking. In *IEEE International Symposium of Quality of Service (IWQoS)*, pages 198–207, May 2014.

[146] Haowei Yuan and Patrick Crowley. Reliably scalable name prefix lookup. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 111–121, May 2015.

[147] Michele Papalini, Koorosh Khazaei, Antonio Carzaniga, and Daniele Rogora. High throughput forwarding for ICN with descriptors and locators. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 43–54, 2016.

[148] Huynh Tu Dang, Han Wang, Theo Jepsen, Gordon Brebner, Changhoon Kim, Jennifer Rexford, Robert Soulé, and Hakim Weatherspoon. Whippersnapper: A p4 language benchmark suite. In *ACM SIGCOMM Symposium on SDN Research (SOSR)*, pages 95–101, April 2017.

[149] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Uğur Çetintemel, Mitch Cherniack, Jeong Hyon Hwang, Wolfgang Lindner, Anurag S. Maskey, Alexander Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The design of the borealis stream processing engine. In *Conference on Innovative Data Systems Research*, pages 277–289, January 2005.

[150] Martin Hirzel, Henrique Andrade, Buğra Gedik, Gabriela Jacques-Silva, Rohit Khandekar, Vibhore Kumar, Mark Mendell, Howard Nasgaard, Scott Schneider, Robert Soulé, and Kun-Lung Wu. IBM streams processing language: Analyzing big data in motion. *IBM Journal of Research and Development (IBMRD)*, 57(3/4):7:1–7:11, May/July 2013.

[151] J. Patel, A. X. Liu, and E. Torng. Bypassing space explosion in high-speed regular expression matching. *IEEE/ACM Transactions on Networking*, 22(6):1701–1714, December 2014.

[152] Lin Tan and T. Sherwood. A high throughput string matching architecture for intrusion detection and prevention. In *32th International Symposium on Computer Architecture*, pages 112–122, June 2005.

[153] Kun Huang, Linxuan Ding, Gaogang Xie, Dafang Zhang, Alex X. Liu, and Kavé Salamatian. Scalable tcam-based regular expression matching with compressed finite automata. *Architectures for Networking and Communications Systems*, pages 83–93, 2013.

[154] Anat Bremler-Barr, David Hay, and Yaron Koral. Compactdfa: Generic state machine compression for scalable pattern matching. In *29th IEEE Conference on Computer Communications*, page 659–667, March 2010.

[155] Chad R. Meiners, Jignesh Patel, Eric Norige, Eric Torng, and Alex X. Liu. Fast regular expression matching using small tcams for network intrusion detection and prevention systems. In *USENIX Security Symposium*, August 2010.

[156] Giorgos Vasiliadis, Michalis Polychronakis, Spiros Antonatos, Evangelos P. Markatos, and Sotiris Ioannidis. Regular expression matching on graphics hardware for intrusion detection. In *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection*, pages 265–283, 2009.

[157] Muhammad Asim Jamshed, Jihyung Lee, Sangwoo Moon, Insu Yun, Deokjin Kim, Sungryoul Lee, Yung Yi, and KyoungSoo Park. Kargus: a highly-scalable software-based intrusion detection system. In *19th CCS*, pages 317–328, October 2012.

[158] P. Tandon, F. M. Sleiman, M. J. Cafarella, and T. F. Wenisch. Hawk: Hardware support for unstructured log processing. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 469–480, May 2016.

[159] Vaibhav Gogte, Aasheesh Kolli, Michael J. Cafarella, Loris D'Antoni, and Thomas F. Wenisch. Hare: Hardware accelerator for regular expressions. In *49th IEEE/ACM International Symposium on Microarchitecture*, pages 44:1–44:12, 2016.

[160] James Cowling and Barbara Liskov. Granola: Low-overhead distributed transaction coordination. In *USENIX Annual Technical Conference*, 2012.

[161] Marc Shapiro. Structure and Encapsulation in Distributed Systems: the Proxy Principle. In *6th IEEE International Conference on Distributed Computing Systems*, pages 198–204, May 1986.

[162] Eric A. Brewer, Randy H. Katz, Elan Amir, Hari Balakrishnan, Yatin Chawathe, Armando Fox, Steven D. Gribble, Todd Hodes, Giao Nguyen, Venkata N. Padmanabhan, Mark Stemm, Srinivasan Seshan, and Tom Henderson. A Network Architecture for Heterogeneous Mobile Computing. Technical report, University of California at Berkeley, 1998.

[163] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless Network File Systems. *ACM Transactions on Computer Systems (TOCS)*, 14(1):41–79, February 1996.

[164] Anthony D. Joseph, Alan F. deLespinasse, Joshua A. Tauber, David K. Gifford, and M. Frans Kaashoek. Rover: A Toolkit for Mobile Information Access. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 156–171, December 1995.

[165] Björn Knutsson, Honghui Lu, Jeffrey Mogul, and Bryan Hopkins. Architecture and Performance of Server-Directed Transcoding. *ACM Transactions on Internet Technology*, 3(4):392–424, November 2003.

[166] Mark Nottingham and Xiang Liu. Edge Architecture Specification, 2001. `http://www.esi.org/architecture_spec_1-0.html`.

[167] Michael J. Freedman, Eric Freudenthal, and David Mazières. Democratizing content publication with coral. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 18–18, March 2004.

[168] Limin Wang, Vivek Pai, and Larry Peterson. The Effectiveness of Request Redirection on CDN Robustness. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 345–360, December 2002.

[169] Robert Grimm, Guy Lichtman, Nikolaos Michalakis, Amos Elliston, Adam Kravetz, Jonathan Miller, and Sajid Raza. Na Kika: Secure Service Execution and Composition in an Open Edge-Side Computing Network. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 169–182, San Jose, California, May 2006.

[170] A. Kemper and T. Neumann. Hyper: A hybrid oltp olap main memory database system based on virtual memory snapshots. In *2011 IEEE 27th International Conference on Data Engineering*, pages 195–206, April 2011.

[171] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. Ermia: Fast memory-optimized database system for heterogeneous workloads. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1675–1687, New York, NY, USA, 2016. Association for Computing Machinery.

[172] Yi Lu, Xiangyao Yu, and Samuel Madden. Star: Scaling transactions through asymmetric replication. *Proc. VLDB Endow.*, 12(11):1316–1329, July 2019.

[173] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 345–360, Boston, MA, February 2019. USENIX Association.

[174] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 361–378, Boston, MA, February 2019. USENIX Association.

[175] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 325–341, New York, NY, USA, 2017. Association for Computing Machinery.

[176] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. Arachne: Core-aware thread management. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 145–160, Carlsbad, CA, October 2018. USENIX Association.

[177] Alexander Rucker, Muhammad Shahbaz, Tushar Swamy, and Kunle Olukotun. Elastic rss: Co-scheduling packets and cores using programmable nics. In *Proceedings of the 3rd Asia-Pacific Workshop on Networking 2019*, APNet ’19, pages 71–77, New York, NY, USA, 2019. Association for Computing Machinery.

[178] Dieter Gawlick and David Kinkade. Varieties of concurrency control in ims/vs fast path. *IEEE Database Eng. Bull.*, 8(2):3–10, 1985.

[179] Pat Helland, Harald Sammer, Jim Lyon, Richard Carr, Phil Garrett, and Andreas Reuter. Group commit timers and high volume transaction systems. In Dieter Gawlick, Mark Haynie, and Andreas Reuter, editors, *High Performance Transaction Systems*, pages 301–329, Berlin, Heidelberg, 1989. Springer Berlin Heidelberg.

[180] Frank M. Pittelli and Hector Garcia-Molina. Reliable scheduling in a tmr database system. *ACM Trans. Comput. Syst.*, 7(1):25–60, January 1989.

[181] Bailu Ding, Lucja Kot, and Johannes Gehrke. Improving optimistic concurrency control through transaction batching and operation reordering. *Proc. VLDB Endow.*, 12(2):169–182, October 2018.

[182] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. The end of slow networks: It's time for a redesign. *Proc. VLDB Endow.*, 9(7):528–539, March 2016.

[183] Feilong Liu, Lingyan Yin, and Spyros Blanas. Design and evaluation of an rdma-aware data shuffling operator for parallel database systems. *ACM Trans. Database Syst.*, 44(4), December 2019.

[184] Wolf Rödiger, Tobias Mühlbauer, Alfons Kemper, and Thomas Neumann. High-speed query processing over high-speed networks. *Proc. VLDB Endow.*, 9(4):228–239, December 2015.

[185] Abdallah Salama, Carsten Binnig, Tim Kraska, Ansgar Scherp, and Tobias Ziegler. Rethinking distributed query execution on high-speed networks. *IEEE Data Engineering Bulletin*, 40(1):27–37, 2017.

[186] Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. Query fresh: Log shipping on steroids. *Proc. VLDB Endow.*, 11(4):406–419, December 2017.

[187] Dong Young Yoon, Mosharaf Chowdhury, and Barzan Mozafari. Distributed lock management with rdma: Decentralization without starvation. In *ACM SIGMOD*, pages 1571–1586, New York, NY, USA, 2018. Association for Computing Machinery.

[188] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. The end of a myth: Distributed transactions can scale. *Proc. VLDB Endow.*, 10(6):685–696, February 2017.

[189] Erfan Zamanian, Xiangyao Yu, Michael Stonebraker, and Tim Kraska. Rethinking database high availability with rdma networks. *Proc. VLDB Endow.*, 12(11):1637–1650, July 2019.

[190] John L. Hennessy and David A. Patterson. A new golden age for computer architecture. *Commun. ACM*, 62(2):48–60, January 2019.

[191] Galera cluster. `https://galeracluster.com/`, 2020.