©Copyright 2018

Antoine Kaufmann

Efficient, Secure, and Flexible High Speed Packet Processing for Data Centers

Antoine Kaufmann

A dissertation
sumbitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2018

Reading Committee:

Thomas Anderson, Chair

Arvind Krishnamurthy

Xi Wang

Program Authorized to Offer Degree:
Computer Science & Engineering

University of Washington

Abstract

Efficient, Secure, and Flexible High Speed Packet Processing for Data Centers

Antoine Kaufmann

Chair of the Supervisory Committee:
Thomas Anderson
Computer Science & Engineering

Data center applications by design rely heavily on network communication. Network bandwidth in data centers is rapidly increasing, but processor performance is only improving at a slower pace. This puts increasing pressure on software packet processing and causes applications to spend more time in the network stack. Existing approaches such as kernel bypass and RDMA reduce software processing overhead but trade off policy compliance or flexibility for performance.

This dissertation demonstrates data center packet processing can be made efficient, scalable, policy compliant, and flexible. I propose a novel architecture for dividing packet processing functionality across the network interface card (NIC), the operating system, and the application. First, with FlexNIC I develop a reconfigurable NIC model that supports scalable NIC-software processing. Second, I demonstrate FlexTCP, a data center TCP network stack for FlexNIC, and show that FlexTCP increases per-core throughput by up to $10.6\times$ compared to Linux and up to $4.1\times$ compared to kernel bypass while still enforcing policy constraints. Finally, I use FlexNIC speed for three data center applications, customizing and integrating NIC processing with application logic, demonstrating a throughput improvement by up to $2.3\times$ for these applications.

Table of Contents

List of Figures v

List of Tables vii

1 Introduction 1

1.1 Reducing Packet Processing Overhead 3
1.2 Flexible Hardware Packet Processing 6
1.3 Goals 7
1.4 Thesis: A Novel Architecture for Packet Processing 8
1.5 Outline 11

2 Background 12

2.1 Data Center Networks 14
2.1.1 Applications 14
2.1.2 Programming Interface 16
2.1.3 Protocols 19
2.1.4 Network Interface Cards 24
2.1.5 Networks 34
2.2 Commodity OS Network Stack 37
2.2.1 Network Stack Architecture 38
2.2.2 TCP Overheads 39
2.2.3 Discussion 42
2.3 Proposal: Kernel Bypass 45
2.3.1 Safe User-Level Access to Network Interface Cards 45
2.3.2 Discussion 47

2.4 Proposal: Protocol Offload . . . 50
2.4.1 TCP Offload Engines . . . 50
2.4.2 Discussion . . . 51
2.5 Proposal: Programmable NICs . . . 53
2.5.1 Network Processor NICs . . . 53
2.5.2 FPGA-based NICs . . . 54
2.5.3 Discussion . . . 54
2.6 Proposal: Remote Direct Memory Access . . . 56
2.6.1 Programming Model . . . 56
2.6.2 Protocols . . . 58
2.6.3 Implementations . . . 60
2.6.4 Comparison to TCP/IP . . . 61
2.6.5 Discussion . . . 63
2.7 Conclusion . . . 64

3 FlexNIC Hardware Model **67**
3.1 Design Goals . . . 67
3.2 Reconfigurable Match Tables in Switches . . . 69
3.3 Applying Reconfigurable Match Tables to NICs . . . 71
3.3.1 Constraints . . . 74
3.4 Building Blocks . . . 75
3.5 Discussion . . . 76

4 TCP Processing **79**
4.1 Goals . . . 79
4.2 Challenges . . . 81
4.3 FlexTCP Network Stack Design . . . 82
4.3.1 FlexNIC Functionality . . . 83
4.3.2 Kernel . . . 87
4.3.3 User-space TCP Stack . . . 89
4.4 Flexible FlexTCP Extension . . . 90
4.5 Implementation . . . 92
4.6 Limitations . . . 94
4.7 Discussion . . . 95

5 Application Integration **97**
5.1 Key-Value Store . . . 97
5.1.1 Motivation . . . 98
5.1.2 Design Goals . . . 100
5.1.3 FlexKVS Components . . . 100
5.1.4 FlexNIC Implementation . . . 103
5.2 Real-time Analytics . . . 104
5.2.1 FlexNIC Implementation . . . 105
5.3 Intrusion Detection . . . 107
5.3.1 FlexNIC Implementation . . . 108
5.4 Discussion . . . 108

6 Evaluation **110**
6.1 FlexTCP . . . 110
6.1.1 Remote Procedure Call (RPC) . . . 112
6.1.2 Packet Loss . . . 116
6.1.3 Key-Value Store . . . 117
6.1.4 Real-time Analytics . . . 119
6.1.5 PCIe NIC Performance . . . 122
6.1.6 Congestion Control . . . 123
6.2 Application Co-Design . . . 126
6.2.1 Methodology . . . 126
6.2.2 Key-Value Store . . . 128
6.2.3 Storm . . . 130
6.2.4 Snort . . . 133
6.3 Discussion . . . 134

7 Related Work **136**
7.1 Software Packet Processing . . . 136
7.1.1 Kernel Stack Improvements . . . 136
7.1.2 Kernel Bypass . . . 138
7.2 Hardware Packet Processing . . . 139
7.2.1 NIC Improvements . . . 139
7.2.2 Programmable Network Hardware . . . 140

7.2.3 Cluster Message Passing 141
7.2.4 GPU Packet Processing 141
7.3 Application Layer Protocols 142
7.3.1 High-performance Applications 142
7.3.2 NIC-Application Co-design 143

8 Conclusion **144**
8.1 Future Work 145
8.1.1 Addressing Limitations 146
8.1.2 Opportunities with FlexNIC 149
8.1.3 Thoughts for the Future 150

Bibliography **153**

A FlexTCP Pseudocode **168**
A.1 NIC Pseudocode 168
A.1.1 Packet Reception 168
A.1.2 Transmission 173
A.1.3 Queue manager events 173
A.2 Rate-based DCTCP control loop 177
A.3 FlexTCP Queue APIs between Components 179
A.4 FlexTCP Low-level Application Interface 181

List of Figures

2.1 Example of a descriptor receive queue. 26
2.2 Example multi-rooted network topology. 35
2.3 Conventional kernel network processing architecture 38
2.4 Kernel bypass processing architecture. 45
2.5 Protocol offload processing architecture. 50

3.1 RMT switch pipeline. 69
3.2 RMT-enhanced NIC DMA architecture. 70

4.1 FlexTCP design overview. 83

5.1 FlexNIC receive fast-path for FlexKVS. 101
5.2 FlexStorm top-n Twitter users topology. 104
5.3 Storm worker design. 106
5.4 Acknowledging incoming FlexStorm tuples in FlexNIC. 107

6.1 FlexTCP single-threaded RPC echo throughput and latency. 113
6.2 FlexTCP single-threaded pipelined RPC throughput. 115
6.3 FlexTCP packet loss throughput penalty. 117
6.4 FlexKVS throughput scalability with FlexTCP. 118

6.5 Throughput for different FlexStorm configurations on FlexTCP. 120
6.6 Single link congestion control sensitivity. 124
6.7 Congestion control flow completion time comparsion. 124
6.8 FlexTCP connection fairness under incast. 125
6.9 FlexKVS throughput scalability with flow-based and key-based steering. . . . 128
6.10 Top-n tweeter throughput on various Storm configurations. 131

List of Tables

2.1 Comparison of previous packet processing architectures. 13

3.1 Meta data format for DMA requests. 73

4.1 Breakdown of lwIP TCP tasks by complexity. 81

4.2 FlexTCP per-flow NIC state 88

6.1 FlexKVS request latency with FlexTCP. 118

6.2 FlexStorm tuple processing time on FlexTCP. 121

6.3 FlexKVS request processing time. 129

6.4 FlexStorm tuple processing time. 132

6.5 Snort throughput and L3 cache behavior. 133

Acknowledgements

I would like to start out by thanking my advisors, Tom Anderson, Simon Peter, and Arvind Krishnamurthy. From the very start of my PhD, Tom has always been happy to help with valuable advice on research, career, and life; I credit him fully with getting me to where I am today. I benefited tremendously from starting out my PhD sitting at a desk right next to Simon, sparking countless discussions and leading to an ongoing fruitful collaboration on ideas, prototypes, and experiments alike. Arvind has always provided interesting technical insights as well as a positive attitude. I would also like to thank Mothy Roscoe, for helping me get started in my research career.

This dissertation and the papers on which it is based have resulted from a collaboration with my advisors as well as Naveen Sharma and Tim Stamler. Both Naveen and Tim have helped with implementing prototypes and experiments and have contributed to many discussions. I would also like to thank Xi Wang, Michael Taylor, and Scott Hauck for serving on my thesis committee.

My time at the University of Washington would not have been such an enjoyable experience without the members of the systems lab: Pedro Fonseca, Helga Gudmundsdottir, Yuchen Jin, Niel Lebeck, Jialin Li, Ellis Michael, Luke Nelson, Naveen Sharma, Haichen Shen, Helgi Sigurbjarnarson, Adriana Szekeres, Qiao Zhang, Irene Zhang, Kaiyuan Zhang, and Danyang Zhuo. In particular, I really enjoyed the many lunches, evenings playing

squash, and other outings with Naveen, Adriana, and Jialin. Xi Wang and Dan Ports have also provided interesting discussions and distraction when needed.

Finally, I thank my family for their support. My parents, Jacqueline and Andreas, have been a constant source of support and encouragement. Most of all, my wife, Justine, has been my best friend and companion through most of this process.

Dedication

to my wife, Justine

Chapter 1

Introduction

Data centers are the platform of choice for increasing numbers of applications and global services. By combining millions of processors, network components, and storage devices into gigantic computers, data centers accommodate even the most resource-intensive applications. At the same time, the consolidation of computational resources into a central pool amortizes operation overheads providing a cost-effective and flexible infrastructure for applications of all sizes. Cloud computing extends these benefits to external customers, by allowing computation to be out-sourced to data centers over the internet. In these multi-tenant data centers, security and isolation mechanisms are needed to protect users.

Modern data centers are composed of individual servers connected by a network and managed as one system. They vary in size from a few hundred to hundreds of thousands of servers. Most applications in data centers run across multiple servers, to take advantage of additional computational capacity, memory, and storage, and for resilience against failures. The different parts of the application use the data center network to communicate. In addition to providing compute resources, data centers also provide additional

services, such as reliable storage, data management, monitoring, and logging. These are also accessed over the network. As a result, many data center applications rely heavily on network communication, both explicitly and implicitly in the services they use. Network communication is increasing as applications continue to scale up to more machines. Small and frequent interactions, such as remote procedure calls (RPCs), are a particularly common communication pattern. This stands in contrast to traditional network usage patterns centering around streaming of large amounts of data.

Data center networks have been growing at a dramatic pace, both to connect increasing numbers of servers and to provide individual servers with more communication bandwidth [122]. Today individual servers have bandwidths of 25 – 40 Gbps, and networks can deliver small messages across a data center within less than $100\,\mu s$ [90]. The exponential trend towards higher bandwidth and lower latencies is continuing for the foreseeable future, with 100 Gbps components available off the shelf and 400 Gbps networks being standardized [50]. This dissertation concerns how to design servers to be able to keep up with high-performance networks on the small message workloads common in today's data centers.

The Ethernet networks most commonly used in data centers split the responsibility for network communication between the network and servers. The network itself implements a basic mechanism to deliver limited size messages from one server to another. It is allowed to drop packets at any point if insufficient capacity is available. Servers cooperatively implement network protocols on top of this basic network for reliable transfers, larger messages, and to manage network resources (congestion control). These protocols use mechanisms such as splitting large messages into multiple smaller messages, re-transmitting lost messages, and estimating available network capacity to limit sending rates.

However, while this split enables cost-effective and highly scalable networks, it also puts pressure on servers to efficiently perform necessary protocol processing. As networks dramatically out-pace gains in processor performance, the need for more efficient server processing continues to grow. A typical communication-heavy data center application can spend the majority of its processor cycles on packet processing. When applications must dedicate much of their computational resources to network processing, that increases cost. As applications continue to scale to more servers and as network bandwidth continues to outstrip processor speeds, applications will be increasingly bottlenecked by packet processing overheads.

1.1 Reducing Packet Processing Overhead

Network packet processing overhead is a longstanding problem. A long line of previous work addresses these overheads by optimizing individual components of the system.

Network Interface Card Capabilities Each server is connected to the network through a network interface card (NIC). The NIC transfers packets between the physical network and software running on the processor. Modern NICs offer a spectrum of capabilities to address certain software overheads. With large segment offload, the NIC splits up chunks of TCP data from software into multiple packet-sized segments to reduce software overhead. NICs transfer packets to and from software through descriptor queues that decouple hardware and software, allowing processing to proceed in parallel, minimizing cache misses, and making efficient use of the I/O interconnect. To enable efficient processing on multiple processor cores, NICs use multiple send and receive queues and offer steering mechanisms to assign incoming packets to receive queues. Most of these NIC capabilities target overheads of large data transfers. They are of limited benefit for the small frequent

interactions typical of most data center communication patterns. And the fixed nature of many NIC features limits their applicability to specific protocol configurations [67].

Operating System Optimization Commodity operating systems such as Linux, Windows, and the BSD family, process packets inside the operating system kernel. Applications send and receive data by issuing system calls to the kernel. Kernel processing enables safe hardware access to the NIC and allows the OS to enforce policies, including protocol invariants and resource allocation. But having this functionality inside the kernel does imply costly context switches between the kernel and applications for every system call and interrupt. Optimizations such as system call batching [126], asynchronous interfaces [44], and uploading application code to the kernel [61] all aim to reduce this overhead. Most of these optimizations work best for large streaming transfers but result in little improvement for small message performance.

Kernel Bypass A more radical approach completely bypasses the operating system (kernel bypass) and allows the application to directly access the NIC [65, 137, 104, 60]. Kernel bypass avoids system call overheads because processing is implemented as a library and invoked with function calls instead of system calls. Applications can further improve performance by tailoring processing to their specific requirements [83]. Recent I/O virtualization features allow operating systems to enforce memory isolation for applications directly accessing the NIC [55]. However, with existing kernel bypass hardware, applications can transmit arbitrary packets. This effectively removes operating system control over what and when applications can transmit. In multi-tenant data centers, this is problematic because the network fundamentally relies on end-host behavior for correct and efficient operation.

Protocol Offload Protocol offload engines take NIC assistance to the logical extreme and fully offload protocol processing to the NIC [24, 21]. This frees up the server processor for executing application logic. Offload also typically improves latency and throughput because a dedicated hardware circuit can execute the mechanical protocol processing steps more efficiently than a general purpose processor [15]. The primary drawback of a hardware protocol implementation in the data center context is the loss of flexibility. After deployment, protocol processing can no longer be modified and new protocols cannot be supported. NICs based on field programmable gate arrays (FPGAs) [108, 143] enable reconfigurable processing after deployment but are considerably more expensive than commodity NICs. [1].

Remote Direct Memory Access An alternative approach is to change the programming model from message passing to remote memory access, that can be completed in hardware without processor involvement on the receiving side [31, 53, 51, 111]. For applications that can be (re-)designed in that model, remote direct memory access (RDMA) can provide efficient network communication. However, a data center scale RDMA implementation must address the same requirements as a TCP implementation with similar mechanisms, including reliable message delivery and congestion control [41, 141]. RDMA combines the new programming model with protocol offload and thereby inherits the same limitations, including the loss of protocol flexibility. The choice of the programming model is orthogonal to how the model is implemented.

[1] In July 2018 on Google Shopping, a commodity Intel XL710 40Gbps NIC was available for $410, while the FPGA-based Mellanox Innova Flex and Solarflare AOE NICs sold for $2050 and $7080 respectively.

1.2 Flexible Hardware Packet Processing

The rapid evolution of network protocols and requirements also leads to operational and deployment challenges for data center networks. Historically, the switches that connect devices in Ethernet networks were proprietary fixed-function devices, that supported a pre-defined set of protocols. OpenFlow [85] and the rise of software-defined networking (SDN) brought flexibility to network control planes, but the actual forwarding functionality and the supported protocols remained fixed. Recent innovation in switch design is leading to a new generation of switches with re-configurable data planes [15, 19, 5]. They support customizable packet formats as well as flexible stateful processing and can be reprogrammed after deployment. At the same time, they operate at aggregate bandwidths of up to 6.5 Tbps.

Reconfigurable match tables (RMT) [15] provide a basis for reconfigurable data planes. In RMT, packets enter the switch through a programmable parser, to identify relevant packet fields. Packets then pass through a fixed-sized pipeline of match-action stages, similar to a systolic array [72]. Each pipeline stage can be programmed to perform table lookups and execute simple actions in response, such as modifying a header field, or manipulating stateful memory. The RMT pipeline architecture places limitations on processing that can be supported, because of the fixed depth and computational capacity of each stage. These limitations are necessary given the throughput requirements. However, these primitives are sufficient for implementing standard Ethernet switch functionality and open up possibilities to implement a wide range of new protocols and behavior.

RMTs are an attractive architecture for reconfigurable hardware packet processing at line-rate. But there are significant differences between packet processing in switches and in servers. Switches only make forwarding and queuing decisions for each packet. Packet processing in servers must also implement complex resource management policies, e.g.

for multi-tenant isolation. NICs must also interact with application and kernel software components running on the server.

1.3 Goals

Thus, any solution to improve the performance and flexibility of server packet processing in the data center must achieve the following goals:

- **Efficiency:** Data center network bandwidth growth continues to outpace processor performance. To manage, data center systems must deliver increasingly efficient packet processing, especially for latency-sensitive small packet communication that is the common case behavior for data center applications and services.
- **Connection Scalability:** Packet processing must also support increasing numbers of connections, as applications and services scale up to larger numbers of servers inside the data center.
- **Performance Predictability:** Another consequence of this scale is that predictable performance is becoming as important as high common case performance for many applications. In large-scale systems, individual user requests can access thousands of backend servers [58, 98] causing one-in-a-thousand request performance to determine common case performance in many cases.
- **Policy Compliance:** At the same time, processing must enforce security and isolation policies across multiple tenants. From a security point of view, applications from different tenants must be prevented from intercepting of and interfering with network communication from other tenants. Interference can take many forms, including spoofing other applications, overloading resources at the end-host or inside

the network, or exploiting implementation vulnerabilities. Processing must be able to enforce policies such as bandwidth limits, memory isolation, network address translation, or the use of specific protocols including congestion control.

- **Protocol Flexibility:** Data centers are a fast moving environment. Network infrastructure is evolving and data center network protocols are an active area of research. New applications are also constantly being rolled out and existing applications are evolving, leading to changes in application protocols, architecture, and performance characteristics. Packet processing must be flexible in adapting to new protocols, network infrastructure, and application requirements.
- **Cost Efficiency:** Finally, any architecture for accelerating packet processing has to be economical. The total cost of ownership factors in hardware cost, available processor capacity for applications, and energy. For hardware cost, I use chip area as a proxy. Costs for hardware extensions, for example, must be justified by reduced processor time for software processing or reduced energy consumption.

1.4 Thesis: A Novel Architecture for Packet Processing

Thesis: *It is possible for data center network packet processing to be made efficient, scalable, predictable, policy compliant, flexible, and cost effective through a novel architecture that splits protocol processing between a programmable NIC and a combination of kernel and application software.*

In this dissertation, I propose and evaluate a novel and unique data center packet processing architecture that achieves these goals. Applications send and receive packets directly through the NIC, bypassing the kernel. The NIC is designed with a special configurable engine to allow it to directly perform most common network and application

protocol processing steps, yet can run at high line rates and is efficient to implement in hardware. The kernel is left to handle less frequent protocol events and out-of-band protocol processing. I show that the architecture is rich enough to implement TCP, the most common protocol used in data centers today. The architecture improves application per-core throughput by up to $10.6\times$ compared to Linux, and up to $4.1\times$ compared to kernel bypass.

This dissertation makes the following contributions:

FlexNIC: Reconfigurable Network Interface Hardware Architecture I propose FlexNIC, a new hardware architecture for reconfigurable NIC processing. FlexNIC allows kernel software to install packet processing, memory transfer, and rate-limit rules into the NIC. This tailors NIC operation to handle common case packet processing traditionally done in software, reducing memory and processing overheads. Operating systems can use FlexNIC to improve packet processing performance for existing and new network protocols. Applications can upload processing rules to FlexNIC via the operating system to reduce application-layer processing overheads.

Unlike fixed function hardware, FlexNIC retains the flexibility to adapt to new network protocols and application requirements. Unlike kernel solutions, FlexNIC enables efficient small message communication. Unlike kernel bypass with commodity NICs, FlexNIC allows the operating system to enforce policy while applications directly access the NIC. Unlike RDMA, FlexNIC separates the programming model from the protocol implementation and can support both message passing and memory access semantics.

FlexTCP: Integrated High-Performance TCP Stack To demonstrate the utility of the FlexNIC architecture, I design and implement FlexTCP, a new flexible protocol stack that implements full TCP semantics. TCP functionality is split between the application li-

brary, operating system kernel, and FlexNIC. Applications send and receive data directly through FlexNIC, where processing rules implement reliable TCP data transfers and assist the kernel in enforcing congestion control. Kernel software handles infrequent operations, such as opening new connections, and digests congestion feedback out-of-band to adjust NIC rate limits. FlexTCP enforces resource isolation and provides tighter performance bounds under load relative to Linux, improving packet handling performance, fairness, and tail-latency by orders of magnitude, while still providing connection scalability into the thousands of active flows, policy compliance, and protocol flexibility.

Accelerating Applications with Customized Network Processing Once kernel protocol processing is moved to FlexNIC, application packet processing can still be a bottleneck and can benefit from being moved onto FlexNIC. With FlexNIC, I can offload application-level protocol processing, steer packets to cores to match application locality, and customize the NIC-software interface to streamline the application-specific request processing. Through three application case studies, I show that this approach reduces application request processing time, improves scalability, and improves cache utilization. When compared to high-performance kernel bypass network stack without FlexNIC, my prototype implementations achieve $2.3\times$ better throughput for a real-time analytics platform modeled after Apache Storm, 60% better throughput for an intrusion detection system, and 60% better latency for a key-value store. All in the context of a system that unlike other kernel bypass solutions, provides policy compliance and flexible protocol deployment.

Neither the outlined goals nor the proposed architecture are limited to data centers. This dissertation focuses on data center networks because the combination of the scale, high bandwidth, and low latency puts enormous pressure on end-host packet processing, beyond that of typical wide-area, enterprise, or mobile networks. Packet processing in these

and other regimes is beyond the scope of this dissertation and it remains future work to evaluate my architecture in this context.

1.5 Outline

The rest of this dissertation is structured as follows:

I begin with an overview of the state-of-the-art in packet processing — covering data center networks, network stacks, NIC hardware, and RDMA — in chapter 2. Next, I present my FlexNIC hardware architecture in chapter 3. Chapter 4 presents FlexTCP, my TCP stack based on co-designed NIC, kernel, and application processing. In chapter 5, I use three case studies to show how packet processing can be customized for specific applications. Chapter 6 evaluates the performance of FlexTCP and the three co-designed applications. I discuss related work in chapter 7. I conclude and discuss future work in chapter 8.

Chapter 2

Background

Existing software and hardware packet processing architectures fail to fully accomplish the set of goals for data center communication outlined above. Different architectures achieve different subsets of goals. In this chapter, I discuss these trade-offs and their architectural origins. This discussion provides the basis for the FlexNIC architecture in the following chapters.

I start by describing the context for data center network communication section 2.1, including application characteristics, protocols, and the software and hardware architecture for data center networks. Next I discuss the Linux network stack architecture in section 2.2 as a representative example for the majority of today's data center systems. With kernel bypass in section 2.3, protocol offload in section 2.4, programmable NICs in section 2.5, and RDMA in section 2.6, I present four existing approaches for reducing overheads in kernel processing systems such as Linux. All four approaches trade off either policy compliance, protocol flexibility, or cost efficiency for better performance, but in section 2.7 I outline how to combine insights from Linux and these approaches to fully achieve the outlined goals.

	Efficient	Scalable	Predictable	Compliant	Flexible	Economical
Linux Kernel	✗ Syscall overhead, complex code	✗ Shared queues, cache pressure, TX scheduling	✗ Multiple queues, complex code, HoL blocking	✓ Complete OS control	(✓) Kernel SW; no app-level changes	(✓) Commodity NICs; CPU overhead
Kernel Bypass	(✓) No syscalls; SW pkt. processing	(✓) No shared queues; cache pressure, TX scheduling	(✓) Isolated; complex code, HoL blocking	✗ No OS enforcement	✓ All SW, can be app speciifc	(✓) Commodity NICs; CPU overhead
Offload	✓ Efficient HW processing	✓ Only limited by NIC RAM	✓ Good, fixed HW timing	(✓) Better; limited HW knobs	✗ Inflexible	✓ Special purpose processing (when applicable)
FPGAs	✓ Better than SW; worse than ASIC	✓ Only limited by NIC RAM	✓ Good, fixed HW timing	✓ Complete OS control	✓ Full flexibility, hard to program	✗ Expensive
RDMA	✓ Efficient HW processing	✓ Only limited by NIC RAM	✗ PFC limits isolation	(✓) Better; limited HW knobs	✗ Inflexible	✓ Special purpose processing (when applicable)
FlexNIC	✓ Efficient HW processing	✓ Only limited by NIC RAM	✓ Good, fixed HW timing	✓ Complete OS control	✓ Reconfigurable	✓ Reconfigurable hardware processing

Table 2.1: Comparison of previous packet processing architectures with respect to my previously defined goals.
Iconography legend: ✓ means the architecture achieves the goal, ✗ means the architecture does not achieve the goal, and **(✓)** means the architecture has pros and cons.

2.1 Data Center Networks

External factors significantly constrain the design space for server packet processing in data centers. Some of these constraints are inherent due to the operating environment, while others are due to design decisions in individual components. Applications have a range of different performance characteristics and requirements. The programming interfaces provided by existing architectures are widely used by many applications. Similarly, protocols used for data center networks are standardized, typically fixed because of support inside the network hardware as well as in external servers. The NIC is responsible for moving data between the physical network and software and includes mechanisms to do so efficiently. Finally, the physical networks that connect individual servers rely on specific server behavior to operate correctly and efficiently.

2.1.1 Applications

Modern data centers run a wide range of applications with different performance characteristics. Some applications such as MapReduce [25] jobs perform large data transfers that are throughput heavy. Other applications primarily send small messages and are sensitive to communication latency, such as distributed lock services [16] or replication protocols [73]. Finally, there are applications such as in memory caching servers [98] that process large volumes of small messages that are sensitive to both latency and throughput.

Communication Model: Remote Procedure Calls

Across this wide range of transfer sizes, a dominant communication pattern for many data center applications is remote procedure calls (RPCs). RPCs consist of a request message from the client to a server, computation of a response on the server, and a response

message from the server back to the client. As such, RPCs are a natural fit for interaction in client server settings but are also an equally good fit for symmetrical settings such as coordination among nodes in a distributed application. To support this widespread use, a wide range of frameworks and libraries provide generic RPC functionality [40, 125, 134]. Other applications implement RPCs manually on top of basic transport protocols such as TCP or UDP.

Regardless of whether RPCs are implemented in a library or directly by the application, most applications require their reliable and ordered delivery. Most RPCs have side effects or other interactions with concurrent calls. As a result lost or re-ordered request or response messages would result in loss, corruption, or general inconsistency of application data. Modern data center networks typically deliver messages reliably and in order. However, occasional packet loss and re-ordering does occur and servers need to implement mitigation when they do occur. Applications can achieve reliable and ordered transmission either by using a reliable transport protocol offered by the OS such as TCP, or by implementing loss recovery and ordering over an unreliable transport such as UDP.

In addition the correctness critical requirements of ordering and reliability, RPCs also present performance challenges for network communication. They are often both latency and throughput sensitive; the client has to wait for the response before it can proceed and the server processes many requests often only with little application logic running, causing communication overheads to dominate. Overheads are further exacerbated by inefficiencies involved with handling small messages in existing operating systems. RPCs involve many small messages because typically the request or response messages are compact, often both.

Deployment Model

The choice of RPCs for communication affects application performance, but it is not the only significant factor. For small messages in particular, the chosen deployment model can introduce additional overheads for sending and receiving data. Applications in data centers are deployed using one of three deployment models: 1) As regular processes on physical hosts, 2) as containers, and 3) as virtual machines. Running applications directly on physical hosts incurs minimal overhead, but the inherently weak isolation limits its applicability to machines not shared by multiple tenants. Containers isolate tenants with operating system mechanisms, including per-packet operations such as scheduling and address translations, resulting in increased overhead. Finally, with virtual machines (VMs), applications run in separate OS instances and a virtual machine monitor processes outgoing packets and passes them to and from the NIC. VMs achieve the highest degree of isolation but also incur higher overheads because of additional protection boundary crossings into the monitor. Because VMs include whole OS instances they also take longer to spin up and down, and consume additional resources. The result is a current trend towards containers as isolation mechanisms continue to mature, to improve performance, resource requirements, and agility while preserving isolation. The rest of this dissertation focuses on the host architecture for processes and containers and leaves VM support as future work.

2.1.2 Programming Interface

The deployment model for applications affects performance and isolation, but regardless of the model applications need a programming interface for network communication. Berkeley sockets are the dominant interface and are used on Unix-based systems including Linux, BSD, and OS X but also other architectures including Windows. This stan-

dardized interface simplifies porting applications between systems. However, individual implementations typically extend sockets with additional calls, often to address fundamental performance problems. The following discussion focuses on the Linux version of the API as an example.

Connection Management Before an application can communicate over the network it needs to create a socket. Connection-less protocols such as UDP require a listening socket. Connection-oriented protocols can either initiate or accept connections. An application can directly initiate connections. To accept incoming connections the application has to open a listening socket. Next the application can accept incoming connections on the listening socket. After a connection or listening socket is no longer required the application can close them. Linux represents listening sockets and connection sockets as file descriptors.

Data Transfers After a connection socket (or listening socket for connection-less protocols) is established the application can issue send and receive calls. Send and receive calls take a socket, a pointer to a buffer, a number of bytes, and potentially additional meta data. The number of bytes passed to these calls specifies an upper bound and both calls can return with less than a full buffer sent or received. Short receives occur if fewer bytes are available in the socket receive buffer. Short sends occur if not enough space is available in the send buffer.

Non-Blocking Operations Both receive and send block if no data is available or the send buffer is full. Initiating a connection and accepting a connection are also blocking operations. When initiating a connection, the call will block until the connection is accepted and fully established. The accept call blocks if there is no pending connection request.

Blocking calls prevent the application thread from doing other work while waiting for completion of the blocking operation. Blocking is problematic for threads that handle multiple connections in parallel. Worse, in most cases the application cannot predict if a call will block or not.

Applications can configure sockets to operate in non-blocking mode. In non-blocking mode any operation that would block immediately returns an error code instead. Calls can be repeated at a later time to check for changes. Initiating a connection is an exception as the kernel will asynchronously finish establishing the connection. The application can check the status of a connection using a separate call. Non-blocking mode simplifies processing multiple connections on the same thread. It also allows language runtimes to avoid blocking all threads when multiplexing multiple threads on top of operating system threads.

Multiplexing Sockets When serving multiple sockets on a single thread, applications need to decide which socket to operate on at what time. Simply cycling through non-blocking sockets is expensive and does not scale. Berkeley sockets include `select`, a multiplexing call to determine which sockets in a set are ready to send and receive data. However, because of overheads with the `select` interface, the major implementations all provide other non-standardized but more efficient multiplexing mechanisms.

Linux provides `epoll`, an event multiplexing call to allow the application to collect and wait for events across multiple sockets. Events include new data on a connection, available transmit buffer space, new connection requests, and closed connections. Applications can add and remove file descriptors, including sockets, to/from these `epoll` sets. Applications can also wait for events on any file descriptors in a set. Each wait call can return more than one event up to a specified limit. The application calls the corresponding API functions to process these events. For example a receive event requires a follow-up

receive call to obtain the data. This abstraction allows the application to efficiently collect multiple events on a set of sockets from the kernel. It also allows the application to block and be re-enabled if an event on *any* connection arrives. For RPCs, the consequence of this interface is that applications typically need at least interface system calls for each RPC, to wait for events, receive the request, and send out the response.

2.1.3 Protocols

The network stack implementation translates between application interface calls and network packets according to network protocol specifications. A stack of protocols provides higher-level abstractions for end-to-end message delivery and by-directional pipes on top of physical hardware. These protocols also provide mechanisms for sharing the network between multiple tenants while ensuring isolation and efficient use of the network. Servers need to process packets according to each of those protocols.

Unreliable Datagram Protocol

Individual end-hosts (or virtual machines) typically run multiple applications and services that communicate on the network. This dictates the need for a multiplexing layer that shares the local IP address between multiple applications and allows incoming and outgoing packets to be identified. The unreliable datagram protocol (UDP) [105] provides exactly that: a multiplexing layer to offer multiple endpoints on a single host. To this end UDP adds additional address information to each packet, in the form of numeric ports. A server application running on a host can listen on a known port number and then receives all packets with the specified destination port. Each packet contains both the source and destination port number so receivers know where to respond to. UDP does not provide services beyond multiplexing. There is no reliable delivery and message size is limited to

the size supported by the underlying network minus the protocol headers. UDP also does not require connections, resulting in minimal processing because no connection needs to be opened and no connection state managed by the end-host. Because UDP does not implement congestion control, typical multi-tenant data center configurations place rate-limits on UDP traffic to preserve some level of isolation.

Transmission Control Protocol

While basic multiplexing is a useful primitive, applications communicating over UDP potentially need to implement a wide range of communication related functionality. In particular application message handling is substantially simpler if the underlying protocol abstracts hardware details, such as maximum packet size, guarantees reliable in-order arrival, determines a sending rate appropriate for current network conditions (congestion control), and avoids overwhelming receiver buffers (flow control). The transmission control protocol (TCP) [106] layers on top of IP to provide applications with these features. To applications, TCP exposes the abstraction of reliable byte streams for sending and receiving data over each connection. Applications simply open a connection, append data to the outgoing stream and remove data from the incoming stream, while the protocol implementation (TCP stack, typically in the operating system) generates outgoing network messages as needed and processes incoming messages. The majority of data center applications communicate over TCP, both within the data center, as well out to the internet.

Connections Like UDP, TCP multiplexes multiple connections and endpoints over one IP address using port numbers. Applications ready to accept incoming connections issue a listen command to the TCP stack with the port number. To open an outgoing connection, an application issues a connect command to the TCP stack with the IP address of the remote host as well as the destination TCP port. At the protocol level, connections are ini-

tiated with the three-way handshake: 1) The host initiating the connection sends a packet with the `SYN`-flag set to the remote host, including both port numbers and IP addresses. 2) The receiving host, assuming it is ready to accept connections on this port, sets up local connection state and responds with a packet containing the `SYN` and `ACK` flags to confirm. 3) The initiating host completes the handshake with a packet containing the `ACK` flag. The two `SYN` packets also negotiate protocol extensions called TCP options supported by both peers. After the handshake both sides are ready to send and receive data.

In-order Transmission To implement reliable in-order data transmission over an unreliable network without ordering guarantees, TCP combines multiple mechanisms. During the handshake, TCP negotiates a sequence number for the two data streams in opposite directions. This sequence number identifies the position of the first byte of data to be transmitted in each direction, incremented as the host transmits data. When a sender sends out a packet with TCP stream payload, referred to as a segment in TCP terminology, the packet header contains the sequence number of the first payload byte in the packet. The sequence number in each packet allows the receiver to find the position in the stream for the payload. This allows receivers to re-order out-of-order packets and detect lost packets.

Reliable Transmission To signal successful reception of stream data to the sender, the receiver sends cumulative acknowledgements to the sender. A cumulative acknowledgement specifies the sequence number up to which the receiver has received all data. If a packet arrives where the sequence number indicates that other data has been lost or reordered, the receiver still responds with a cumulative acknowledgement indicating the last sequence number that was received in order (i.e. not the sequence number in this segment). The sender buffers transmitted data until it receives an acknowledgement. The

sender tracks both the last sequence number that it sent out, as well as the last sequence number that has been acknowledged. When the sender transmits a segment with payload, it also arms a timer that will be cancelled when it receives a corresponding acknowledgement. If the network drops the segment (or the acknowledgement), the timer expires and triggers a retransmission of this segment and re-arms the timeout. Because it is impossible for the sender to determine if the packet has been lost, or simply delayed inside the network, timeouts are typically chosen so as to be conservative to avoid unnecessary re-transmissions. As a result, timeouts can significantly delay data transmission. Fast retransmit [127] reduces the reliance on timeouts. In the common case, one packet is lost in the middle of a continuous train of packets for a connection. In this case, the receiver returns a duplicate (cumulative) acknowledgement for each packet following the lost one. With fast retransmit a sender counts duplicate acknowledgements, and issues an immediate retransmission (without waiting for a timeout) as soon as a configurable threshold (often 3) of duplicate acknowledgements has been reached.

Flow and Congestion Control In addition to mechanisms for recovering from packet loss, TCP also includes mechanisms to minimize packet loss. A receiver might not have sufficient buffer space to store an arriving packet. This occurs when a sender sends data faster than the application on the receiver processes it. TCP flow control aims to avoid this situation, by requiring receivers to signal how much receive buffer space they still have available, in each packet (or ACK) sent in the opposite direction. A sender is not allowed to send more data than specified by the receive window. Congestion control, aims to minimize congestion related drops inside the network by adjusting the sending rate to match network capacity. Since the available capacity is not known and changes as other servers in the network start sending or go idle, congestion control algorithms continuously adapt the sending rate based on signals from the network.

In heterogeneous networks such as the internet, packet loss is the only widely supported mechanism for end-hosts to detect network congestion. Packet loss as an implicit congestion signal is attractive because it requires no additional signalling or mechanisms. For end-hosts, however, using packet loss as a signal complicates detecting congestion. Protocols need to rely on mechanisms such as sequence numbers and timeouts to detect loss, and they need to distinguish between the case of a message being lost or simply delayed due to queueing. Packet loss as a signal is inefficient, both within the network and on end-hosts. The message has to be re-transmitted requiring additional network bandwidth and end-host processing time. Latency until the message successfully arrives at the destination is also affected.

Explicit congestion notification (ECN) [110] instead provides an explicit in-band signal for the network to notify end-hosts of congestion. With ECN, switches mark packets with the congestion experienced (CE) flag inside the packet header to signal queue build up. The receiver forwards this flag to the sender when it acknowledges the packet. ECN enables early signalling without introducing extra packet loss when capacity is available. However, once queues are full even ECN-enabled switches have no choice but to drop packets.

TCP congestion control uses signals from the network to continuously estimate current network capacity. At a high level, it increase the sending rate for successful transmissions, and reduces the sending rate on losses, explicit congestion marks, or increases in the round-trip time. Congestion control is still a very active research area for TCP and many different algorithms exist and are implemented in various TCP implementations [90, 17, 26, 34, 81, 91, 94, 139, 141]. Different algorithms use different signals and control laws for adjusting the sending rate. Most traditional algorithms calculate a byte window for how much data can be in flight at a time, and only send out more data as ac-

knowledgements arrive. Data center TCP (DCTCP) [3] has been designed for data centers with ECN support and is widely used in different variants.

2.1.4 Network Interface Cards

For sending and receiving network packets generated by the protocol stack, server software interacts with the NIC. NICs provide software network stacks with access to the physical network. Historically, network interface cards provided simple mechanisms to transfer packets between the processor and the physical network. Modern network cards, however, additionally provide a plethora of mechanisms aimed at making packet processing more efficient.

Partial Protocol Offload

Packet processing often consists of long sequences of steps, many of which are mechanical. Some NICs support performing — or "offloading" — some of these steps by the NIC. Offloading processing from the CPU to the NIC reduces CPU overhead. Common offloads include checksums, segmentation, and encapsulation/decapsulation.

Checksums Checksum offload validates checksums for received packets and inserts checksums for transmitted packets. When validating checksums NICs include flags in descriptors to indicate incorrect checksums. For transmit, checksum offload software needs to include flags in transmit descriptors to enable the right combination of offloads. Ethernet CRC offload has been offered by NICs for a long time. Modern NICs also support offloading IP, UDP, and TCP checksums. Transmit offloads that modify packets require NICs to generate correct checksums after modifications.

Large Segment Segmentation offload delegates the task of splitting up data into individual packets according to the maximum transfer unit (MTU) supported by the network. Software can pass packets larger than the MTU as one packet; the NIC generates smaller packets as needed. This reduces software overhead. Segmentation offload also reduces bus traffic because only one descriptor has to be passed in each direction for the whole packet. TCP large segment offload (TSO) is the most common segmentation offload. For TSO, software passes a large TCP packet including protocol headers to the NIC. The NIC generates MTU sized TCP segments by copying the initial header and adjusting the sequence numbers and checksums.

Receive-side Coalescing For receiving packets, receive-side coalescing (RSC) is the inverse of segmentation offload. The NIC combines multiple smaller packets into a single large packet before passing the packet to software. RSC again reduces software processing overheads and bus traffic. NIC RSC typically only coalesces packets if they arrive back-to-back; otherwise partial packets would need to be held in NIC memory. Assuming no packets for other connections arrive, the NIC stops coalescing after a maximum size is reached or after a specified time after the first packet arrives.

Encapsulation and Decapsulation Protocols such as VLAN, VXLAN, GRE, IP-in-IP tunnels add additional encapsulation headers to packets. Encapsulation and decapsulation offload can add and remove those headers when sending and receiving packets. This again reduces software processing overheads and can also be used to transparently (to software) encapsulate and decapsulate packets for certain send and receive queues. Encapsulation protocols also interact with other NIC features such as packet steering or other offloads. A NIC not supporting an encapsulation protocol can no longer parse inner packet headers. The NIC may see outer headers and use them for steering, or not

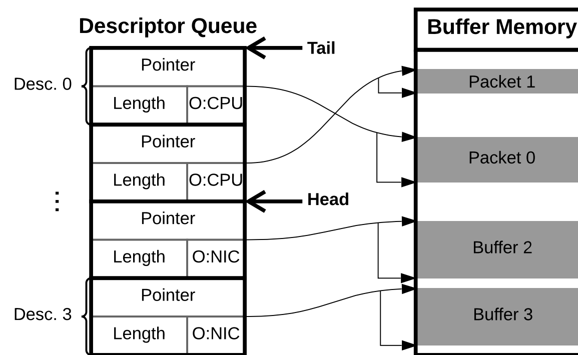


Figure 2.1: Example of a descriptor receive queue. The first two descriptors point to received packets and are marked as CPU-owned. The other two descriptors are marked as NIC-owned and point to unused buffers.

recognize headers at all.

Some NICs support a wide variety of other offloads. Additional examples of offloads are precision time-stamping, IPSec en-/decryption and authentication, or network address translation.

Software Interface

At a high level, most modern NICs have a similar interface consisting of two parts: configuration registers and descriptor queues. Software uses configuration registers to initialize the NIC and control its behavior. Configuration registers reside on the NIC and are accessed by software through memory mapped I/O over the I/O bus. Descriptor queues transfer packets and notifications between the NIC and software in both directions. Descriptor queues reside in host memory and are accessed by the NIC through direct memory access (DMA) over the I/O bus. During initialization software allocates memory for descriptor queues and writes the address and length of each queue to the corresponding configuration register.

Descriptor Queues A descriptor queue is a circular queue consisting of descriptors used for bi-directional communication. One end of the queue is owned by software and the other end by the NIC. Each descriptor queue has a head pointer and a tail pointer. The head pointer points to the first descriptor owned by the NIC, and the tail pointer points to the last descriptor owned by the NIC. The head pointer is owned by the NIC, and the tail pointer is owned by software. Software adds a descriptor by writing it to the next entry after the tail pointer and then incrementing the tail pointer. The NIC removes a descriptor by reading the entry pointed to by the head pointer and then incrementing the head pointer. Software never moves the head pointer and the NIC never moves the tail pointer.

Each descriptor corresponds to one request or notification. The specific format and meaning of a descriptor depends on the context and the direction. Transmit queues carry packet descriptors from software to the NIC, and completion notifications from the NIC to software. Receive queues carry buffer descriptors from software to the NIC, and packet notifications from the NIC to software.

To send a packet, software adds a packet descriptor consisting of the packet length and a pointer to the packet buffer to the transmit queue. The NIC then asynchronously removes packets from the transmit queue. After reading the packet from the location specified in the descriptor, the NIC sends the packet out over the network. After a packet is sent the NIC enqueues a completion notification on the transmit queue. This completion notification lets software know that the buffer can be freed. The notification also passes a descriptor back to software, so that the NIC only owns descriptors for packets that are waiting to be sent.

For receiving packets, software starts by allocating receive buffers and adding corresponding buffer descriptors to the receive queue. When the NIC receives a packet it

removes the next buffer descriptor from the queue, thereby obtaining a pointer to the receive buffer. After writing the packet to this receive buffer, the NIC adds a packet notification to the receive queue. The packet notification specifies the packet buffer and the packet length.

PCIe Performance Constraints A standard architecture is to attach the NIC to the I/O bus — PCIe in today's servers. In which case, the NIC communicates with the CPU over the PCIe bus. PCIe transfers have higher latency and limited bus bandwidth. Devices issuing many small PCIe transfers effectively have less bandwidth available for data because of per-operation transfer overheads. Memory access latency is asymmetric; accesses from the CPU incur lower latencies (especially for cache hits) than accesses from the NIC. These latency and throughput overheads constrain the design space for communication between the NIC and CPU.

Descriptor queue implementations combine multiple PCIe mechanisms to achieve good performance. The NIC maintains the head and tail pointers as configuration registers. After adding a descriptor to the queue, software has to write to the tail pointer register to let the NIC know that there is a new descriptor. This tail pointer write to a configuration register is called a *PCIe doorbell* because it notifies the NIC. For the CPU, doorbell writes are writes to un-cachable memory and thus more expensive than regular memory writes. Doorbell writes however do not necessarily lead to CPU pipeline stalls because the CPU does not have to wait for a result.

Reads from configuration registers are more expensive. Un-cachable reads do cause pipeline stalls because execution cannot continue until the result is available. Reading the head pointer configuration register is thus not an appropriate mechanism for software to determine how many new descriptors are available. Instead descriptors contain an ownership bit indicating if a descriptor is currently owned by the NIC or the CPU.

When adding a descriptor to a descriptor queue, software will mark it as NIC owned. The NIC marks descriptors as software owned once processing is complete. Descriptors between the head and the tail pointer are marked as NIC owned, all other descriptors are marked as software owned. Software reads descriptors until it finds a descriptor that is marked as NIC owned or until it reaches the tail position in case of an empty queue. The NIC may also issue interrupts to asynchronously notify software that an event occurred. An interrupt does not convey any additional information such as what event occurred or how many descriptors were added. Interrupts merely signal that software should check queues for new descriptors. Because of the overhead of taking an interrupt, most high performance NICs make interrupts configurable — limiting them to a configurable rate and allowing kernel software to dynamically disable interrupts temporarily on high packet rates.

Because of high DMA read latency, NICs cache a small number of receive descriptors in internal memory [56]. Receive descriptor caching reduces packet processing latency. Instead of first issuing a DMA read to determine the next receive buffer address, the NIC can immediately issue writes for the packet to the buffer and the descriptor. Descriptor caching also reduces packet buffering required to avoid packet drops while receive descriptor fetches are pending.

Descriptor queues lend themselves well to batching as a performance optimization. Instead of reading descriptors individually the NICs issue DMA reads for multiple descriptors at a time. Combining smaller DMA reads into a larger one amortizes DMA latency. Combining reads also improves PCIe bandwidth utilization by amortizing per-operation bus overheads. Descriptor write-backs can also be combined. For transmit queues, delayed write-backs work well because freeing transmit buffers is not on the critical path. For receive queues, combined descriptor write-back can increase receive latency

because the new packet will be delayed until all packets in the batch are processed. This is a trade-off between latency and PCIe bandwidth utilization. NICs usually offer configuration registers to tune these parameters to the workload characteristics.

Doorbell writes provide another opportunity for batching. When registering multiple receive buffers together only one doorbell write is required for the whole batch. Doorbell batching amortizes the cost for the un-cached memory write over multiple operations. For transmit queues doorbell batching provides a trade-off between latency as well as CPU overhead and PCIe bandwidth. Applications processing packets at high rates often need doorbell batching to achieve maximum throughput.

On modern X86 machines, PCIe DMA is cache coherent. Devices will see any modified data in CPU caches, and the CPU sees data written by devices immediately. Writes to data that is in a cache cause the cached copy to be invalidated. When software reads descriptors that the NIC wrote back, the CPU incurs a cache miss. However, descriptor queues lend themselves well to pre-fetching. Software knows ahead of time which memory location will be accessed so it can issue a manual pre-fetch. And because descriptors are linear in memory the CPU will automatically issue additional pre-fetches if multiple descriptors are read. When receiving packets after reading the descriptor software accesses the corresponding receive buffer. Receive buffers for new packets have also been written to by the NIC and accesses also cause cache misses. These misses can also be avoided with manual pre-fetching. This is especially effective when receiving multiple packets at once. Multiple pre-fetches for packet buffers can be issued in parallel after reading the corresponding descriptors.

Descriptors are often smaller than a cache line. Descriptor writes from the NIC can cause cache misses for adjacent descriptors that software has already pre-fetched. For receive queues these misses disappear if enough packets arrive to keep the NIC ahead

by a small number (number of descriptors per cache-line) of descriptors. Thus this behavior can be self-correcting; as cache misses cause overhead, software processing slows down and the NIC gets ahead. Transmit queues cause cache misses not avoidable by pre-fetching if only few packets are in the queue. For transmit queue these misses are not self-correcting because slowing down software processing causes shorter queues and more misses. Transmit descriptors do not provide information beyond the fact that the packet has been sent and can be freed. Because the NIC transmits packets in order, the only information that software needs is the last packet sent. Intel NICs provide a feature called *TX head index write-back*. With head index write-back the NIC no longer writes transmit descriptors back. Instead the NIC only writes the head index, i.e. the position of the last packet sent, to a specified memory location. Head-index write-back thus avoids those pathological misses because the NIC no longer writes to the transmit queue.

Recent Intel server processors support data direct I/O (DDIO). With DDIO PCIe DMA operations go directly to the shared last level cache. DMA reads are served from cache without requiring invalidation to memory first. DMA writes directly update the last level cache, reducing last level cache misses when the CPU accesses the written data. DDIO reduces CPU access latency as well as memory bus traffic. Even with DDIO, L1 and L2 caches are still invalidated. DMA writes cause L1/L2 cache lines to be marked as invalid. DMA reads will cause cache lines in modified or exclusive states to transition to shared. When re-using transmit buffers, transitions of the cache lines from shared to modified occur. These transitions are similar in overhead to L2 cache misses. They can again be avoided using manual pre-fetching.

Multi-Core Performance

When receiving and sending packets on multiple cores shared descriptor queues incur synchronization and cache coherence overhead. Modern NICs provide multiple transmit and receive descriptor queues to avoid these overheads. Each core can be assigned a dedicated transmit and receive queue. Because each queue is only accessed from one core no synchronization is required. A core sends packets by adding descriptors to its assigned transmit queue. Similarly each core receives packets through its assigned receive queue.

With multiple receive queues the NIC needs to assign incoming packets to queues. This assignment of incoming packets to queues is called *packet steering*. Simply assigning packets round-robin or randomly does not work well for existing software protocol processing. When processing packets from multiple queues in parallel, packets would be no longer totally ordered. Protocols such as TCP expect packets to arrive in order and out-of-order arrivals are more costly to process. Many protocols also require per-connection state in memory for processing. Packets for one connection arriving on different cores cause cache-coherence overheads by transferring connection state between private caches.

Receive-side Scaling The most widely available mechanism for NIC packet steering is receive-side scaling (RSS). To assign an incoming packet to a queue the NIC calculates a hash over packet fields identifying the connection and assigns packets based on this hash. A simple option is to simply use the hash modulo the number of queues to calculate the destination queue. Consistently assigning packets based on a connection identifier hash guarantees that all packets for a single connection arrive in the same queue. This preserves packet ordering within a connection and eliminates cache-coherence overhead when accessing connection state. Assigning equal ranges of hashes to queues can lead to load-imbalance if some connections have higher throughput. To partially address this,

many NICs use an additional configurable redirection table that maps ranges of hashes to queues. These tables are relatively small, commonly 128 or 256 entries. This additional level of indirection allows software to correct load-imbalances.

Flow Steering An application consuming data on a different core than the one packets arrive on will incur additional communication overhead. To minimize this, packets should arrive and be processed on the same core where the application consumes the data and generates responses. While RSS with a redirection table enables load balancing, it is too coarse-grained to assign individual connections to specific cores. High-performance NICs support *flow steering* mechanisms that enable fine-grained assignment of connections to queues. Examples of flow steering mechanisms include Intel's FlowDirector and Solarflare's Accelerated RFS, that support steering 10,000s of connections. Flow steering provides a lookup table that maps flow identifiers to queues. The flow identifiers used as a lookup key are limited to a fixed set of packet fields supported by the NIC, such as source and destination IP address and TCP port numbers.

Flow steering is usually implemented using a hash table to scale to large numbers of flows. Often only one type of flow identifier can be supported at once. For example TCP connections can be steered while listening sockets that need a wild-card match for the source port cannot be steered. Some NICs provide additional steering mechanisms. Intel's 82599 10 GbE NICs provide 128 5-tuple filters, a flow steering mechanism that supports wild cards in individual entries. Other examples include steering based on MAC addresses for virtual machines. All of these steering mechanisms are limited to known protocol fields specified by the NIC vendor. These fields are extracted by a packet parser on the NIC that cannot be configured by software to extract different fields. A minor exception for this is the Intel FlowDirector that can filter based on a field at a configurable offset in the packet. This lack of flexibility limits these mechanisms to only the specific set

of configurations they are intended for, and cannot be used for steering based on higher layer protocols.

2.1.5 Networks

The NICs transfer packets between server software and the physical link that connects the sever to the rest of the network. The network is then responsible for transferring packets between individual servers. At data center scale, achieving this is not an easy task. To manage, networks are arranged in complex topologies and are highly tailored to the protocols used. As a consequence correct and efficient network operation fundamentally depends on data center servers correctly implementing network protocols. If all components operate correctly, data centers networks provide high bandwidths, low latencies, and predictable performance.

Topology

Data center networks today typically use multi-rooted tree topologies [2, 39, 97, 122] (also referred to as Clos networks). Servers (leaves in the tree) are arranged in racks and each rack has a top-of-rack switch (ToR). Each top-of-rack switch connects to multiple aggregation layer switches, and each aggregation layer switch in turn connects to multiple spine (or core) layer switches. This multi-rooted tree topology provides multiple redundant paths between racks to ensure availability in case of failures and to increase bisection bandwidth.

An implication of the multi-rooted tree topology is that at each layer there are multiple equivalent paths up the tree from where all destinations can be reached. When routing a packet to a destination, each switch looks at the destination address and decides whether it can directly deliver the packet on one of its downward facing ports, or whether the

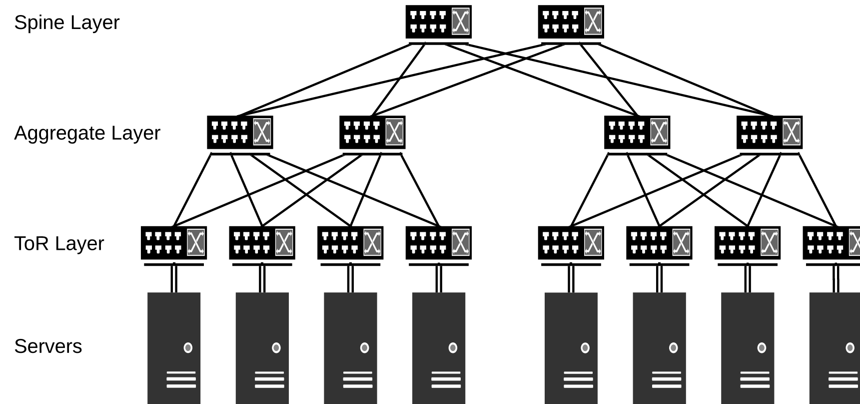


Figure 2.2: Example multi-rooted network topology.

packet needs to be forwarded up the tree. For forwarding up the tree multiple equivalent options are available, so the switch needs to decide which route to use for each packet. The switch should route packets so that the load on all available upward links is balanced. However, a conflicting goal for data center networks is to avoid re-ordering packets. While servers do not expect the network to guarantee ordering and higher layer protocols such as TCP are equipped to handle out-of-order packets, re-ordering will nevertheless increase end-host processing overheads. But end-host protocol processing is only sensitive to the order of multiple packets within each connection, while tolerating re-ordering across connections without performance penalties. Thus, ensuring that packets for a particular connection use a deterministic route avoids overheads while still using multiple paths efficiently, provided there are many connections. Equal cost multi-path routing (ECMP) [47] achieves this by calculating a hash over the connection identifying fields in the packet, and then using this hash to index into the set of available equivalent routes. ECMP thus guarantees that packets for a particular connection use a deterministic route without tracking dynamic per-connection state at switches.

As a way to reduce cost, some multi-rooted tree topologies used in data centers contain

oversubscribed links [122]. There is less aggregate upward bandwidth than the aggregate downward bandwidth. When there is oversubscription, typically it is a small multiple — networks most commonly are provisioned to allow most servers to send simultaneously. Other data centers avoid oversubscription and provide full bisection bandwidth [39]. For oversubscribed topologies, upward facing links can experience congestion and eventually packet loss. But even networks with full bisection bandwidth suffer from congestion due to fan-in, i.e. incoming traffic from multiple ports forwarded to the same output port at a switch. As a result end-host congestion control is critical for overall performance. We discuss options for congestion control later in this chapter.

Switches

Ethernet switches connect servers and other switches and forward packets between them. A switch has multiple ports and each port can be connected to another Ethernet device. When an incoming packets arrives on a port, the switch looks at its destination address to determine which output port to use. Modern data center switches are non-blocking and can move data between any two ports in parallel. However, each port is limited by the maximum bandwidth of the link. Congestion occurs if a faster port is sending data to a slower port or, more commonly in a data center network, if multiple servers are using the same destination port.

Each switch has some buffer memory to queue packets that cannot immediately be forwarded because of congestion. Switches dynamically assign buffer memory to queues for each output port, as needed within configurable limits. As long as queue space is available, incoming packets can be enqueued and will eventually be forwarded. When no more queue space is available, the switches silently drop packets and end-hosts must detect and re-transmit the lost data.

Performance Parameters

Modern data center networks offer aggregate bandwidth measured in petabits per second with 10–40 Gbps network links per server. [122]. Even 100 Gbps Ethernet NICs and switches are available as commodity components off the shelf. An individual switch can forward a small packet in a few 100 nanoseconds when there is no queueing. Even adding in queuing delay and data center scale infrastructure, network round trip times are in the order of magnitude of a few 10 microseconds. Data center switches typically offer around 10 MB of shared buffers [3, 41], enough to queue a few thousand packets. Given the high bandwidths of data center networks and the low switching latencies, end-to-end latency is typically dominated by end-host processing and in some cases queueing delay inside the network. Packet loss rates in well managed networks are typically below 0.1% [122, 142], with loss primarily due to congestion but transient losses due to outages or packet corruption also occur less frequently. While packets within a particular connection usually arrive in the same order they are transmitted, transient reordering can occur in response to failures or configuration updates.

2.2 Commodity OS Network Stack

Commodity operating systems including Linux, Windows, BSD, and OS X process network packets in the operating system kernel. The operating system implements all network protocols and interacts with the NIC to send and receive packets. Applications use system calls to manage connections and transfer data. Figure 2.3 shows this interaction between components graphically. The rest of this section uses Linux as an example implementation.

I start with a discussion of the internal software architecture of the network stack,

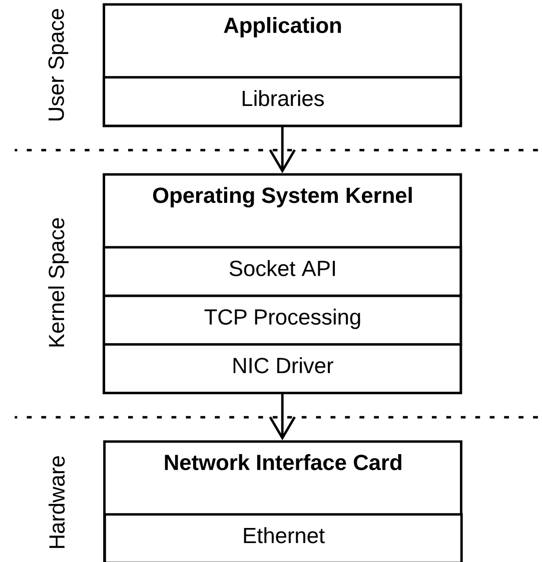


Figure 2.3: Conventional operating systems implement all network processing inside the kernel.

move on to TCP-specific overheads, and conclude with a discussion of OS kernel processing relative to the outlined goals.

2.2.1 Network Stack Architecture

Because Linux is used for everything from data centers to smart phones, its kernel network stack is engineered for a wide range of use-cases and configurability. At the top level there is the user space API that is independent of what protocols are enabled or what NIC hardware is available. The protocol implementations are agnostic to what lower level protocols or hardware they are running on. The NIC drivers are also independent of the higher level protocols used or what additional processing is configured.

The combination of layers and modules with defined interfaces is a known technique for engineering complex systems. Developers can re-combine modules in different ways without modifying them. The TCP implementation can be used for communicating over

an Ethernet network, a serial port with SLIP, or a virtual private network wrapped in UDP. Kernels also insert additional functionality such as firewalls, traffic shaping, or packet capture for debugging between layers.

At the lowest layer network device drivers interact with NICs for sending and receiving packets. When receiving an interrupt for new packets, the driver will asynchronously schedule packet processing and disable additional interrupts for the device until packets are processed. At high packet rates some drivers completely disable interrupts and rely on periodic polling. After receiving packets the driver passes them on to higher protocol layers of processing. For sending packets, the packet scheduler, or queueing discipline in Linux, calls the driver to add packets to the transmit descriptor queue. The network device driver interface is the same for all network drivers. This complex control and data flow through multiple components causes significant overhead.

2.2.2 TCP Overheads

TCP packet processing in particular is complex and notorious for its overhead. Most implementations have at least two state machines per connection. One manages connection setup and tear-down handshakes and the other handles common case operation for established connections, such as flow control, congestion control, packet re-ordering, acknowledgement generation, and loss detection. On multicores, TCP packet processing overheads include mutual exclusion on shared data structures, cache line invalidations, and mismatched cache locality. The resulting overheads impact application performance in the following ways.

Per-Packet Processing Latency Although TCP is a streaming protocol, in data centers and over the internet it is often used for RPCs. Key-value stores [87, 112], distributed

lock managers [16] and file systems [45] are just a few examples of applications that employ RPCs over TCP. In addition to server throughput, the latency from sending a request to receiving the response is a primary performance factor for these applications [93]. Data center networks can deliver packets within a few microseconds between clients and servers. Since application-level processing time is often small, the dominant factor is TCP per-packet handling costs, which can take 10s of microseconds (see Figure 6.1). The discrepancy of several orders of magnitude between processing and network speeds is the main contributor to remote procedure call overheads for these applications.

Connection Scalability TCP is a stateful protocol and requires access to per-connection data. This per-connection state is typically large and complex (e.g. in Linux the `tcp_sock` structure contains more than 100 fields). As the number of connections grows this causes cache pressure and pollutes application cache contents. This state also contains complex data structures that require pointer chasing, e.g. for timers or handling out of order packets. Accessing these data structures incurs significant overhead.

The stateful nature of TCP processing also results in cache coherence and synchronization overhead on multi-core systems. For example, per-socket locks in the Linux kernel serialize access to socket data and thus per-connection processing cannot be parallelized. Instead, techniques like receive side scaling (RSS) are designed to deliver packets directly to the appropriate core responsible for each connection. However, this is not a complete solution. For example, the core used for sending replies may be different from the core used for receiving, causing cache invalidations and synchronization [103] in the common case.

Further, application cache locality does not always match TCP connection locality. For example, a key-value store might receive requests for a hot set of keys on numerous client connections. When different cores modify the same values, there will be added syn-

chronization and cache invalidation overheads. When different cores read the same data, cache lines become duplicated in L1 and L2 caches, reducing overall cache effectiveness.

Queueing and Fairness Most kernel TCP implementations employ multiple shared queues for both incoming and outgoing traffic. For example, Linux employs at least three queues for outgoing TCP traffic [135]. The first is a per-socket queue holding the current transmit window. The second is a shared per-core *queueing discipline* of configurable size used for traffic shaping. The third is a shared per-core NIC driver queue of fixed size. These queues add CPU overhead. Since packets can be held up in any of the queues for unpredictable delays, this can also create situations of unstable performance, when queues fill up and drain in a bursty fashion. This impacts fairness in particular for the increasingly common case of many concurrent connections: shared kernel queues can run out of space, dropping packets. Different packet drop strategies are employed for each queue. For example, when the Linux NIC driver signals that its outgoing queue is full, the queueing discipline re-enqueues overflow packets, incurring extra overhead [135]. To reduce latency, Linux employs an adaptive queueing mechanism called TCP small queues [23] that restricts each connection to at most 2 packets and a pre-defined byte limit for outgoing queues. This mechanism puts a bound on latency, but it incurs the aforementioned CPU overheads when packets need to be re-enqueued. Queuing also increases the complexity of the software TCP implementation. Queues make it more difficult to debug performance problems.

This complexity is an artifact of the modular architecture and the requirement for configurability in Linux. Only the queues for sending and receiving in the driver are fundamentally required. They decouple the NIC from software, and allow the two to operate in parallel.

2.2.3 Discussion

I wrap up this section with a discussion of Linux kernel OS processing in the data center, specifically which of my outlined goals it satisfies.

Efficiency ✗ Applications interacting with the kernel network stack need to use system calls. Each system call results in a transition to kernel mode followed by a transition back to user mode. These transitions have direct and indirect costs [126]. Direct costs include CPU overheads for switching protection mode and kernel overheads for saving and restoring registers. In addition there are indirect costs of reduced cache locality and pipeline efficiency. Kernel system call handlers implement careful parameter and authorization checks that often require data structure accesses. Executing this code pollutes the CPU instruction and data caches. The CPU also cannot speculate instructions across system calls. Even simple system calls cost 100s of cycles. As a result processing for a UDP echo server that receives a small UDP message and sends it back out takes $3 - 6\,\mu\mathrm{s}$ [104].

Relying on a single shared generic kernel stack limits opportunities for optimizations. Maintainers of shared stacks can only accept optimizations that do not degrade performance for any other applications, and standardized interfaces can also cause additional overheads. For example the Linux kernel struggles to support new NIC offloads because they require changes to interfaces and multiple layers [130]. The socket interface across a kernel boundary is also not well suited for high performance applications. Many common operations require multiple system calls. These could be combined into one single special-purpose call, at added complexity.

Connection Scalability ✗ The complex kernel processing architecture also results in problems for connection scalability. Shared queues between stack components can introduce pathological behavior if the number of active connections grows beyond the fixed

queue size. This can results in excessive timeouts and even connection resets. Because of the modular structure, there is no single central point where packets are scheduled for transmission. Instead packets are injected from the top layer when ready, and individual components implement separate back pressure and scheduling mechanisms. The generic nature of the protocol implementation also results in complex and larger connection state in memory. As a result, systems handling large numbers of active connections experience poor cache utilization in the kernel as well as the application.

Performance Predictability ✗ Processing packets for all applications in a shared network stack also reduces performance predictability. Applications and connections share various receive and transmit queues. Packets are de-queued and at least partially processed in the order in which they arrive, regardless which application is currently running. This results in performance crosstalk between applications. Depending on current internal stack state a packet from a latency sensitive application might either be sent immediately, or deferred almost indefinitely if multiple timeouts and back-off occur because of full intermediate queues. Latency-sensitive applications are likely to experience head-of-line blocking if mixed with buffer-filling applications.

Policy Compliance ✓ The network links connecting the server to the network are shared by all applications running on the same server. The kernel implements resource management policies to isolate applications from negatively affecting other applications on the same server. For example a single application should not be allowed to use more than its fair share of network bandwidth if other applications are trying to send and receive data. The kernel enforces resource management policies and arbitrates between applications.

Applications can also disrupt applications running on other machines. For example sending a lot of data to a machine with a slower network link causes other packets to that

machine to be dropped. TCP *congestion control* aims to fairly allocate bandwidth for any bottleneck link in the network, not just at end hosts. The kernel also prevents application from spoofing source address information in packets to impersonate other applications. Some protocols and implementations are vulnerable to maliciously crafted packets [118]. The kernel guarantees that the protocol is used correctly.

Protocol Flexibility (✓) With kernel processing all protocols are implemented in software. Adding support for new protocols or modifying the implementation of a protocol can be achieved by re-compiling the kernel. Kernel processing also implement all processing centrally and all applications running on the machine will automatically use the modified version without recompilation. Administrators can update applications and the network stack independently. Even deploying additional functionality such as firewalls or overlay networks does not require modifying applications.

However, this flexibility only extends to global modifications for all applications. Because of the centralized and shared nature, application specific modifications are generally not possible. This applies to code modifications and even some configuration changes.

Cost Efficiency (✓) Finally, in this architecture the NIC is only responsible for sending and receiving packets. All protocol processing is implemented in software in the OS. As a result the NIC only needs to implement minimal packet processing available in all commodity NICs. While the hardware cost is minimal, the software overheads mentioned above do impact cost efficiency. Many applications spend large fractions of their CPU cycles on kernel network processing. With lower overheads, applications would need fewer CPU cores for the same processing.

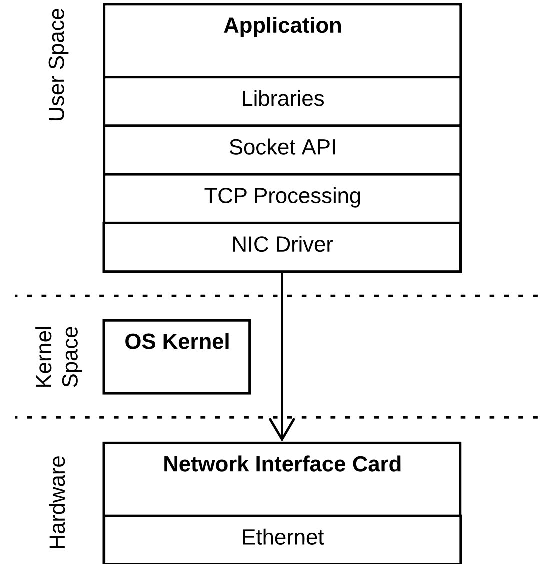


Figure 2.4: Kernel bypass architectures avoid overheads for kernel crossings by moving the protocol implementation into the application and providing applications with direct access to the NIC for sending and receiving packets.

2.3 Proposal: Kernel Bypass

Kernel processing by designed enforces multi-tenant policies, but fundamentally suffers from high overhead. Kernel bypass instead aims to address the performance and flexibility problems of kernel processing by moving all packet processing completely into applications, as shown in Figure 2.4. In the resulting architecture applications send and receive packets by directly interacting with the NIC, avoiding all system calls.

2.3.1 Safe User-Level Access to Network Interface Cards

Allowing multiple applications safe access to a NIC requires a combination of mechanisms. In early work, U-Net [137] proposed a NIC presenting multiple virtual NICs. These virtual NICs are then assigned to individual applications. Each virtual NIC consists of a send and receive descriptor queue as well as a fixed memory range for trans-

mit and receive buffers. A virtual NIC only accepts descriptors pointing into the fixed packet buffer area. This ensures applications cannot circumvent process isolation using descriptor queues. A key limitation is that U-Net is designed for ATM networks instead of Ethernet. ATM provides virtual channels for multiplexing connections. U-Net assigns incoming packets to virtual NICs based on the virtual channel ID in the packet. The VNICs only allow virtual channel IDs assigned to this virtual interface when transmitting packets.

I/O memory management units (IOMMUs) provide another mechanism to confine direct memory access for a device to specific ranges. The primary use-case for IOMMUs is safely passing individual PCIe devices through to virtual machines. IOMMUs are not specific to NICs. In contrast to U-Net, IOMMUs do not require support from the device. IOMMUs are memory management units using the same virtual address translation mechanisms as CPUs. The hypervisor or kernel sets up a separate virtual address mapping for each device. After a mapping is set up, the device uses virtual addresses. The IOMMU translates those addresses to physical addresses and prevents accesses without valid translations. Thus the IOMMU guarantees memory isolation even when a VM or application has direct (virtual) access to a device.

By itself, an IOMMU allows a single application to access a PCIe device while guaranteeing memory isolation. PCI single-root I/O virtualization (SR-IOV) generalizes this to multiple applications: a single PCIe device presents multiple virtual copies as separate devices. SR-IOV refers to these virtual copies as virtual functions (VFs). To the system virtual functions look and act as separate devices. The IOMMU can assign different virtual address mappings to each virtual function. The primary use-case for SR-IOV is sharing a single physical device between multiple virtual machines but it can also be used to shera a physical device among multiple applications. The device and not SR-IOV specifies se-

mantics for virtual functions.

SR-IOV NICs provide separate sets of receive and transmit queues to virtual functions. The kernel or hypervisor assigns separate MAC addresses to each virtual function. The NIC then assigns incoming packets to virtual functions based on the packet's destination MAC address. For outgoing packets the NIC also ensures that each virtual function only uses its assigned source MAC address. SR-IOV NICs allocate link bandwidth among VFs according to configurable policies. Today's NICs support aggregate rate limits and priority-based schemes. The combination of IOMMUs and SR-IOV NICs allows protected access to the NIC by virtual machines or applications.

2.3.2 Discussion

Efficiency (✓) With kernel bypass, applications do not need to use system calls for sending and receiving individual packets. This avoids the direct cost of executing system calls as well as the indirect overheads due to cache pollution and pipeline stalls. Avoiding system call overheads frees up CPU cycles for application processing and reduces request processing latency. However, even with kernel bypass and inspite of the broad range of NIC hardware optimizations, applications still spend a large fraction of their CPU time executing protocol processing (see chapter 6).

Connection Scalability (✓) How kernel bypass scales to larger number of connections depends on the protocol stack implementations. User space protocol stack implementations such as mtcp [60] are designed to run fully partitioned between cores, avoiding shared intermediate queues and cross-core shuffling of packets. As a result they typically scale better than Linux. However, even with kernel bypass protocol processing still executes fully in software. As a result the cache footprint and overheads for accurate transmit

scheduling still limit scalability, albeit beyond Linux.

Performance Predictability (✓) With user space protocol processing performance isolation and predictability tend to improve. No packets for other applications are processed as an application is executing. But processing still involves relatively complex control flow leading to non-predictable performance. On the transmit side, the NIC is responsible for arbitrating packets from different applications. Relatively simple round-robin policies combined with batching for DMA efficiency are typical. When mixing latency and throughput bound applications this can still lead to considerable head-of-line blocking.

Policy Compliance ✗ Policy compliance is the Achilles heel for kernel bypass systems. With kernel bypass the NIC needs to execute all correctness-critical processing. Because applications send and receive packets by communicating with the NIC directly the operating system cannot interpose and inject additional processing. If the NIC does not support some required kernel feature, kernel bypass is not feasible. This critical processing includes demultiplexing packets, egress packet filtering, and resource allocation.

On the send path, the Intel 82599, for example, only supports checking the source MAC address. It cannot enforce that an application only uses a specific source IP or port number. The NIC also schedules packet transmissions between multiple applications, and as such needs to support the required scheduling policy. Today's NICs typically only offer per-virtual function rate limits. Thus a policy limiting transmit bandwidth for specific destinations is not feasible. As a consequence, kernel bypass cannot enforce the same protocol correctness guarantees as traditional operating systems. Software in a trusted kernel can guarantee correct protocol behavior such as respecting TCP congestion control, even with malicious applications. But for kernel bypass only the NIC is in a position to validate outgoing packets.

For cases when the application is trusted, such as operator provided services, the lack of policy enforcement is often not a problem. In these settings, kernel bypass provides a performance boost on existing hardware. But in any multi-tenant setting running non-trusted software, kernel bypass can typically not be used.

Protocol Flexibility ✓ Even with kernel bypass all protocol processing is implemented inside the application. This provides protocol flexibility, as the application can implement arbitrary processing, including application-specific behavior. And each application can use its own customized processing if required, otherwise standard library stacks can be used.

One caveat here is that there are some limits due to what mechanisms the NIC supports. An example is demultiplexing for incoming packets. For each packet the NIC must decide which virtual NIC should receive the packet. The NIC needs to recognize the protocols involved and offer packet steering based on the required fields. These mechanisms are typically limited to standard protocol fields, such as the traditional 5-tuple of protocol, source and destination IP/port number. This lack of flexibility can still cause significant overhead, as discussed in chapter 5.

Cost Efficiency (✓) Similar to in-kernel processing, kernel bypass too can be implemented on top of commodity NICs, and as a result the hardware cost is low. The still significant amount of processor time spent on packet processing rather than application logic, does result in additional cost. However, kernel bypass is significantly more cost efficient than in-kernel processing.

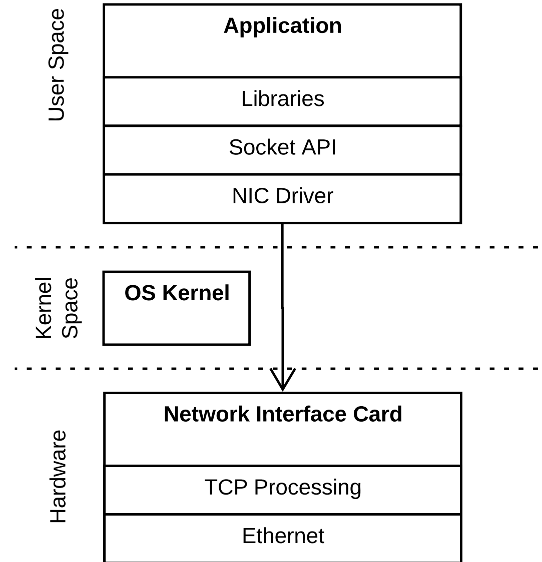


Figure 2.5: Protocol offload implements protocol processing in hardware on the NIC.

2.4 Proposal: Protocol Offload

Both in-kernel and kernel bypass packet processing execute all protocol processing steps in software, reducing the CPU time available for applications. Protocol offload instead completely moves protocol processing into hardware on the NIC. In contrast to the partial NIC offload features that only offload small parts of processing, such as checksums or segmentation, full protocol offload offloads all processing. With full protocol offload, NICs expose a higher-level interface instead of individual packets.

2.4.1 TCP Offload Engines

TCP offload engines (TOEs) implement all or most of the required protocol processing for TCP in hardware. On the receiving path, the NIC passes the TCP payload into a software defined buffer. On transmit, the NIC takes payload from software and generates TCP packets. TOEs implement all aspects of TCP processing, including congestion control and

reliable data transfer.

TOEs are classified into two categories: full TCP offload, and partial (also chimney) TCP offload. Full TCP offload implements all aspects of TCP on the NIC, including connection establishment and tear-down. Partial TCP offload performs connection setup and tear-down in software and hands off connections to the NIC for data reception and transmission. Implementing connection setup and tear-down in software simplifies the NIC and gives software control over what connections can be established. TCP connection setup and tear-down accounts for a significant amount of TCP's complexity. At the same time, for long-lived connections the majority of CPU cycles are spent receiving and transmitting data.

Historically, TOEs were used to reduce the CPU overhead for in kernel processing. This type of system primarily targets large transfers, as the overheads for small transfers are often comparable. Integration in this design is also challenging because it completely bypasses parts of the network stack (all protocol processing), but retains tight integration with the sockets layer.

Modern TOEs instead target kernel bypass. Vendors typically ship application libraries that provide drop-in compatibility with sockets without application modifications. These libraries contain a user-space NIC driver, and implementations of socket calls that translate into low-level commands for the driver. In these deployments the kernel is completely off the critical path for data transfers.

2.4.2 Discussion

TCP offload has failed to gain widespread adoption in data centers to date. While TOEs improve performance, they come at a cost of a complete loss of protocol flexibility as well as integration and management challenges [92].

Efficiency ✓ Protocol offload combined with kernel bypass is highly efficient. On the CPU, only minimal processing is required to initiate and complete data transfers. All protocol processing is implemented in fixed hardware circuits that outperform more flexible execution engines, including CPUs and FPGAs, in terms of throughput, latency, power consumption, and silicon area.

Connection Scalability ✓ With protocol offload, connection scalability is only limited by the available memory for connection state on the NIC. The NIC has a global view of all connections and can schedule connections centrally and efficiently in hardware.

Performance Predictability ✓ Because of the implementation of processing as fixed hardware circuits, the timing is also completely predictable (modulo protocol-level events such as timeouts).

Policy Compliance (✓) Unlike Ethernet kernel bypass, protocol offload is able to enforce protocol invariants and other policies. Because applications provide the NIC with higher level commands and because the NIC understands the protocol, the NIC can mediate application actions. However, any enforcement is limited to the available hardware mechanisms of a particular implementation.

Protocol Flexibility ✗ The main drawback of the hardware implementation of protocol processing is the complete lack of flexibility. The NIC supports a fixed set of protocols, configurations, and policy enforcement mechanisms that cannot be changed after deployment. Any incompatible change, such as moving to a new encapsulation protocol or congestion control algorithm cannot be accommodated.

Cost Efficiency ✓ However, the special purpose nature of the silicon implementation does result in minimal chip area requirements for the supported processing. The resulting hardware cost is lower than other alternatives.

2.5 Proposal: Programmable NICs

Programmable NICs address the inflexibility of fixed protocol offload by replacing the fixed processing logic with a programmable engine. They allow all or part of the packet processing to be specified by software. Network processor NICs and FPGA-based NICs are the two most common types of programmable NICs.

2.5.1 Network Processor NICs

Network processors (NPUs) consist of regular CPU cores and additional network processing specific units. NPUs can be programmed as regular CPUs. They typically offer familiar tool chains such as C compilers; some run full-fledged operating systems such as Linux. Thus, existing protocol processing code and even whole applications can be ported to NPUs. Some NPUs such as Cavium Octeon [18] include up to 48 out-of-order cores running at high clock frequencies. Other NPUs such as Netronome NFP [96] rely on larger numbers of smaller specialized cores running at lower clock frequencies.

NPU-based NICs can run a wide variety of processing, from offloads such as encryption to full applications such as deep packet inspection that perform all processing on the NIC. Compared to fixed function NICs, similar processing on NPUs incurs additional latency. The achievable throughput for an application depends on how the required processing parallelizes. Since NPUs employ CPU cores for processing, overheads for processing on these cores are often similar to the host CPU, limiting the benefit of offload. Compared to kernel bypass, they share many of the same advantages and disadvantages

already discussed. The main differences are that NPUs provide policy enforcement by allowing OS software no the NIC to mediate access but at a significantly higher hardware cost.

2.5.2 FPGA-based NICs

Field programmable gate arrays (FPGAs) have gained traction for accelerating a wide range of computationally intensive tasks. NICs with built-in FPGAs allow processing to be specified as reconfigurable hardware circuits. How FPGAs are integrated into NICs varies by product. NetFPGA [143] NICs implement most of the NIC on the FPGA and only implement physical interfaces to the network and the PCIe bus as fixed hardware. For Solarflare FPGA NICs, the FPGA is integrated as a *bump in the wire* and can modify packets as they are received or sent. With bump in the wire processing the host interface is a fixed-function NIC that dictates the software interface.

FPGA NICs have historically been used primarily as research and prototyping vehicles [143]. At least one cloud operator recently deployed bump in the wire FPGA-based NICs at data center scale in production [108]. Based on publically available information, they are currently using the NICs primarily to offload network management functionality, such as address translation for virtual machines.

2.5.3 Discussion

The following discussion focuses on FPGAs, as NPUs have a lot of commonalities with software solutions.

Efficiency ✓ Relative to a software solution on the host CPU or an NPU, FPGAs provide performance much closer to a hardware implementation. However, the reconfigurability

in FPGAs does come at a cost and as a result circuits typically run at lower frequencies than fixed hardware circuits.

Connection Scalability ✓ As with hardware protocol offload, connection scalability is primarily limited by available NIC memory. In general this will depend on what processing is offloaded to the FPGA, as anything inherently sequential that cannot be pipelined would cause bottlenecks (also applies to fixed hardware offloads).

Performance Predictability ✓ Because processing here is also represented as hardware circuits, performance is predictable. Again this also depends on what processing is offloaded.

Policy Compliance ✓ The OS can leverage the FPGA to retain policy enforcement even if applications bypass the kernel. The flexibility of FPGAs provides the OS with a wide range of mechanisms that can be implemented, from high-level application interfaces, various resource limits, to other validity checks on outgoing packets.

Protocol Flexibility ✓ FPGAs are programmed with hardware circuits, and as such provide a high degree of control over processing. As such they expose the fine-grained parallelism inherent in hardware circuits. They are programmed in hardware description languages (HDLs) such as VHDL or Verilog, that complicate porting of existing functionality. Fully taking advantage of FPGAs requires significant engineering effort and manual tuning.

Cost Efficiency ✗ The flexibility of FPGAs is achieved by mapping circuits onto reconfigurable logic arrays. The chip area required after mapping a circuit onto an FPGA is typically multiple times up to an order of magnitude [15] higher than a fixed hardware

implementation of the same circuit. And to retain the flexibility to add additional processing in the future, FPGA sizes for an application have to be chosen generously. As a result the hardware cost for FPGA NICs is much higher than commodity NICs or even sophisticated offload capable NICs.

2.6 Proposal: Remote Direct Memory Access

All four packet processing architectures discussed so far aim to implement the same network level protocols with different performance characteristics. Remote direct memory access (RDMA) takes a radically different approach by introducing a new programming model, a new protocol, and a hardware implementation of the protocol. The RDMA programming model centers around asynchronous shared memory reads and writes. The protocol then specifies how to translate these requests into messages and how to transport them over the network. Finally an RDMA NIC implements the protocol in hardware, to both issue operations to other hosts and to respond to incoming operations, typically without involving the CPU. With this combination, RDMA provides a case study of a clean slate and cross-layer design.

This section starts by presenting the RDMA programming model, protocols, and hardware implementations. Next follows a comparison of RDMA to TCP/IP systems at the different levels, before wrapping up with a discussion of RDMA in the context of the outlined goals.

2.6.1 Programming Model

The defining characteristic for RDMA is the availability of one-sided operations to directly access remote memory while bypassing the CPU on the remote host. This is possible for `READ` and `WRITE`. In addition, modern RDMA hardware also offers normal message pass-

ing operations, SEND and RECV. The major implementations of RDMA [53, 52, 119] have two additional operations: ATOMIC performs a 64-bit compare-and-swap or fetch-and-add memory operation that is guaranteed to complete atomically with respect to other RDMA operations but not with respect to CPU operations. WRITE with immediate combines a memory write with a notification for the remote host. After performing the memory write, the remote application receives a notification with a 32-bit payload.

Replicating the range of options with message passing, some RDMA implementations offer range of service types. Infiniband [53] and RoCE [52] implementations offer reliable connected (RC), unreliable connected (UC), and unreliable datagram (UD) operation. RC guarantees reliable in-order delivery over established connections, similar to TCP. UC provides best-effort semantics (unreliable, unordered) over established connections. UD offers the same best-effort message semantics as UDP.

Because of the focus on performance, RDMA operates asynchronously allowing applications to issue multiple operations in parallel and to process completion notifications as they arrive. Each connection is associated with a dedicated queue pair on each peer, consisting of a send and receive queue. For each queue pair the application specifies a completion queue that can be shared between multiple queue pairs. The hardware posts notifications about completed operations (if requested), as well as incoming SEND and WRITE with immediate operations, on the completion queue for the application.

Before accepting RDMA READ and WRITE operations, the application has to register at least one memory region with the hardware. The RDMA NIC driver returns a handle (rkey) for each memory region. This handle is a capability (applications can forward to other nodes) to issue one sided operations to the associated memory region. One-sided operations are only valid if they contain a valid handle for the address. This mechanism provides fine-grained control over what memory regions can be accessed remotely by

which peers. Because one-sided operations bypass the kernel only the RDMA hardware can enforce access control for memory.

There are number of APIs used with RDMA. The verbs [53] interface provides direct asynchronous access to RDMA primitives, including queues and management of memory regions. Verbs provide applications with full control but also no abstraction over low-level details. The message passing interface [88], MPI, provides higher-level primitives for message passing, synchronization, and one-sided data transfers. MPI is particularly popular for high performance computing (HPC) applications. rsockets [100] implement the standard sockets API over RDMA. However, the sockets API is inherently two-sided and do not allow the application to leverage one-sided operations.

2.6.2 Protocols

Three standardized protocols provide RDMA semantics: Infiniband [53], RDMA over converged Ethernet (RoCE) [51, 52], and the Internet wide-area RDMA protocol (iWARP) [111]. I limit the discussion to Inifiniband and RoCE as the only protocols with documented data center scale deployments.

Infiniband Infiniband [53, 54] is an interconnect originally designed for communication within a system between processors and I/O devices, and then extented to work over short distances between servers. The specification covers the full stack from the physical layer up to and including the transport layer. In contrast to Ethernet, Infiniband is designed to provide reliable communication with techniques such as link-level flow control and retransmission under hardware control. Mechanisms such as congestion control and recovery from out-of-order packets are not required or can be simplified. As a result, even when using unreliable RDMA operations (no transport layer acknowledgements and re-

transmissions) in a busy cluster petabytes of data can be transferred without losses [63].

Commodity systems primarily use PCIe as the internal interconnect but Infiniband is popular for high-throughput low-latency network communication in clusters using PCIe to Infiniband adapters. High performance computing (HPC) clusters commonly use Infiniband [132] but data centers and enterprise networks are heavily dominated by Ethernet. While smaller Infiniband clusters can achieve single digit micro second latencies, at data center scales latencies of $90\,\mu s$ in the common case and hundreds of microseconds in the tail are expected [41]. Of the major cloud providers currently only Microsoft Azure offers an instance type with Infiniband [66].

RDMA over Converged Ethernet Because Infiniband requires new network infrastructure and more expensive hardware and does only work over limited distance, it is less attractive in the data center setting. RDMA over Converged Ethernet (RoCE) [51, 52] instead layers the RDMA protocol layers over Ethernet protocols. The original RoCE [51] layers the Infiniband network and transport layer protocols directly over Ethernet. This was not a good fit for the L3 IP routing used pervasively in data centers. RoCEv2 [52], also known as routable RoCE, instead layers the Infiniband transport layer protocol over UDP (on IPv4 or IPv6), providing compatibility with IP routing and ECMP and enabling deployment on top of existing network infrastructure. RoCE is usually implemented in hardware for performance, but software implementations using regular Ethernet NICs exist too [79].

To avoid congestion related packet drops, RoCE requires enabling Ethernet priority flow control (PFC) [49] in the network. In case of congestion, a PFC-enabled switch temporarily tells the upstream switch or NIC to pause and then re-enables the sender when sufficient buffer space is available. PFC is challenging to configure correctly, requires additional buffer space on switches, and (depending on the configuration) can lead to

livelock, deadlock, and head-of-line blocking [41]. Backpressure propagates from congestion points back towards the sender. The granularity for flow control is coarse-grained, supporting only up to 8 priority classes per switch port (in practice often less because of limited buffers [41]), each of which can be independently suspended. As a result, backpressure caused by a single connection generally affects multiple connections causing collateral damage. Modern RoCEv2 implementations combine PFC with ECN-based end-host congestion control [141, 90] to back off when congestion is building up before PFC triggers backpressure. Because of inherent link distance limitations in PFC, current implementations cannot be deployed at full data center scale but only within individual network sections [41].

With RoCE, lossless Ethernet is not a strict correctness requirement, as the transport layer already has to detect and retransmit non-congestion losses. However, RoCE performance degrades significantly (some implementations reach a livelock) with as little as 0.4% packet loss [41], because it uses extremely simple loss recovery. Newer RoCE implementations avoid livelock and improve performance under loss with more sophisticated retransmission schemes [41, 86].

2.6.3 Implementations

For both protocols, commodity implementations are available. To achieve true one-sided operation, RDMA must be implemented in hardware on the NIC. More generally a hardware implementation provides RDMA with similar performance benefits as TCP offload for operations involving the CPU, by minimizing software processing. But software implementations do exist and can offer compatibility on existing commodity hardware.

Modern hardware implementations for all three protocols are designed around safe kernel bypass. In these cases applications issue RDMA commands directly through in-

memory queue pairs to the NIC. The driver in this case is split into a kernel driver for privileged tasks such as initializing NIC state and setting up memory mappings in the NIC address translation table for registered memory regions. Because the NIC implements all protocol processing, significant amounts of on-NIC memory are needed for connection state, memory mappings, and caching of queue entries.

2.6.4 Comparison to TCP/IP

Different systems use different aspects of RDMA with different motivation. Some systems benefit from the RDMA programming model, others rely on RDMA because the (hardware) implementation outperforms TCP stacks in commodity operating systems for the specific use-case.

Programming Model The primary distinguishing characteristic of RDMA is the ability to directly access memory on a remote host. This is a fundamentally different programming model compared to the pure data stream abstraction offered by TCP and datagrams offered by UDP. One-sided operations exhibit lower latency and reduced CPU utilization compared to implementing the same operations in software using message passing. Applications that only need basic remote memory reads and writes can leverage RDMA to significantly improve performance with one-sided operations. However, memory reads and writes are only a basic communication primitive. Many applications require more complicated data structures that cannot be accessed with just individual reads and writes, and are often accessed concurrently. In these cases remote locking and multiple round-trips are generally required, resulting in reduced benefits for one-sided operations and potentially even in higher latency and lower throughput compared to an RPC-based implementation over messages [62].

Protocol At a protocol level, both Infiniband and RoCE feature different divisions of responsibility between endhosts and the network. Both protocols assume mechanisms within the network (including hardware flow-control) to avoid packet loss due to congestion. TCP on the other hand, does not assume hardware flow control and employs end-host congestion control to adapt sending rates and recover from congestion losses. In the data center context the TCP and RoCE protocols are converging from opposite directions. While TCP starts from minimal network assumptions with no reliable delivery, no ordering, and no visibility into the network beyond packet drops, RoCE starts assuming reliable delivery and minimal endhost responsibilities. Data center TCP [3] improves congestion control performance and minimizes congestion loss by leveraging explicit network feedback. RoCE on the other hand leverages end-host congestion control [141] to improve performance in large networks, and with resilient RoCE [86] relaxes network requirements and enables operation in lossy networks.

Implementation Even applications that only rely on RDMA messaging primitives leverage RDMA to improve performance compared to using commodity operating systems network stacks [63]. These performance improvements arise from implementation aspects: direct kernel bypass NIC access for applications, offload of processing to NIC, stronger network requirements, and application interface details. Dating back to VIA [31], RDMA has primarily been implemented in hardware to maximize performance, offering kernel bypass as well as full data path offload to the NIC. Applications enqueue RDMA commands for the NIC, and the NIC will generate packets as well as process incoming packets and notify the application. Instead of the blocking socket calls, RDMA verbs offer a fully asynchronous queue-based interface.

2.6.5 Discussion

Efficiency ✓ With its richer programming model, RDMA provides a larger design space for applications. One-sided operations provide low-latency direct access to remote memory while completely bypassing the remote CPU. For applications that can effectively use one-sided operations, the RDMA programming model enables more efficient operation than pure message passing. For all other applications RDMA effectively functions as protocol offload for message passing. As such, RDMA hardware implementations offer at least the efficiency of hardware protocol processing, with additional latency and CPU overhead improvements for applications using one-sided operations.

Connection Scalability ✓ As with hardware protocol offload, RDMA also has to store connection state in hardware, and as such is limited by available NIC memory. RDMA NICs typically also support spilling connection state to host memory, but paging state in and out of NIC memory is expensive and only helps with inactive connections. Protection meta data for one-sided operations takes up additional NIC memory, increasing memory footprint compared to pure message passing use. Besides the memory limit, RDMA does not place other limits on connection scalability.

Performance Predictability ✗ In terms of end-host processing RDMA achieves the same predictable hardware processing as hardware protocol offload. However, the use of hardware flow control in RoCE (as the most popular data center RDMA implementation) can lead to unpredictable performance. Because PFC back-pressure occurs per-link, and can propagate out into the network, it introduces head of line blocking and cross-talk. The requirement for PFC has also made full data center scale deployments of RDMA impossible, because the limitations that PFC places on wiring distance prevent communication across the spine layer [41].

Policy Compliance (✓) While RDMA does use kernel bypass, the combination with a high-level application interface and a hardware protocol processing enables the OS to enforce policies. But the policies that the OS can enforce are completely dependent on the mechanisms offered by the RDMA protocol and the NIC implementation.

Protocol Flexibility ✗ The fixed nature of a NIC that implements one particular protocol in hardware, by definition results in limited to no protocol flexibility. RDMA vendors have been extending both the protocol and the NICs with new features, but these extensions require deployment of new hardware. Protocol extensions in deployments that mix multiple generations of hardware can also only be used among the nodes that support the extension.

Cost Efficiency ✓ As purpose-built hardware implementations, RDMA NICs implement their processing in minimal silicon resulting in low hardware cost. But again the lack of protocol flexibility potentially implies costly upgrades as requirements and infrastructure evolve.

2.7 Conclusion

Data center networks fundamentally rely on server protocol processing for correct and efficient operation. Servers implement protocols, such as TCP, for reliable communication over the unreliable network. Implementing these protocols consumes significant resources on end-hosts. The design space for solutions is constrained because much of it has to be performed by a trusted layer in multi-tenant systems.

Traditionally this trusted layer has been the operating system kernel. This architecture is capable of enforcing multi-tenant policy and does not require special-purpose hard-

ware, but the main drawback is high processor overhead for applications. Bypassing the kernel by allowing applications to directly access the NIC, reduces kernel-crossing overheads but still requires the application to implement all protocol processing. More importantly, bypass sacrifices policy compliance because applications are able to craft arbitrary packets and violate congestion control and is as such a no-go for multi-tenant uses. Protocol offload implements protocol functionality in hardware on the NIC, freeing up processor cycles and allowing the NIC to police the application by enforcing configured protocol operation. However, while the fixed hardware implementation of protocol offload offers high performance at low cost, it lacks protocol flexibility. Programmable FPGA NICs address this by offering flexible protocol offload for a wide range of functionality. However, the flexibility comes at the cost of vastly increased chip area and a HDL programming model. Finally, RDMA provides applications with an alternate programming model centered around shared memory operations instead of message passing. For applications that can benefit from one-sided memory accesses, the RDMA programming model can significantly improve efficiency. However, current commercially available RDMA NICs struggle with protocol limitations for data centers and complete lack of protocol flexibility.

From a performance and efficiency point of view, kernel bypass is necessary but not sufficient. Protocol offload further boosts efficiency, and is also a necessary requirement for policy enforcement with kernel bypass. But what packet processing architecture can provide the necessary protocol flexibility in an economical way?

Existing NICs are on two opposite ends of the spectrum of programmability. Commodity NICs (including TCP offload and RDMA) implement fixed processing in hardware and offer no programmability. NPUs and FPGAs are at the other end of the spectrum and allow (almost) arbitrary processing to be implemented. In this dissertation, I argue that the sweet spot lies in the middle of this spectrum. Limited programmability in

the NIC can be augmented with software processing in operating system and applications to efficiently implement even complex protocols.

Chapter 3

FlexNIC Hardware Model

I now present FlexNIC, an architecture for efficient, scalable, predictable, policy compliant, flexible, and cost effective NIC packet processing. In the previous chapter, I concluded that while the combination of protocol offload and kernel bypass improves performance without compromising policy compliance, existing architectures are either inflexible and economical or completely flexible but not cost effective. FlexNIC instead explores the middle ground with limited programmability in a cost effective architecture. The OS and applications alike can leverage FlexNIC to achieve high and predictable performance with large numbers of connections while providing security for multi-tenancy. FlexNIC serves as the basis for both FlexTCP in chapter 4 and integrated application processing in chapter 5.

3.1 Design Goals

My outlined goals for data center packet processing guide the development of FlexNIC and each goal has a implications for the NIC design:

- **Efficiency:** In addition to minimizing processor overhead for packet processing, FlexNIC must be able to support processing at the line rates of tomorrow's data center network link speeds (at least 100 Gb/s).
- **Connection Scalability:** FlexNIC should not restrict scalability to large numbers of connections. However, all stateful protocol offloads are limited by available NIC memory. I explore memory requirements for stateful offload in chapter 4.
- **Performance Predictability:** Hardware packet processing with FlexNIC should preserve the same predictable hardware timing as fixed protocol offloads.
- **Policy Compliance:** FlexNIC needs to support the protection and isolation guarantees provided by the OS, while allowing applications to install their own offloading primitives in a fast and flexible manner.
- **Protocol Flexibility:** FlexNIC must be flexible enough to serve the offload requirements of data center protocols and applications that change at software development timescales. The architecture has to support existing protocols and applications to enable incremental deployment and interaction with hosts outside of the data center.
- **Cost Efficiency:** The required additional hardware has to be economical, such that it fits the pricing model of commodity NICs.

I start with the reconfigurable match table (RMT) model recently proposed for flexible switching chips [15] to the NIC DMA interface. The RMT model processes packets through a systolic sequence of match and action (M+A) stages.

In this dissertation I propose the FlexNIC hardware model, and evaluate it using an emulation methodology [42].

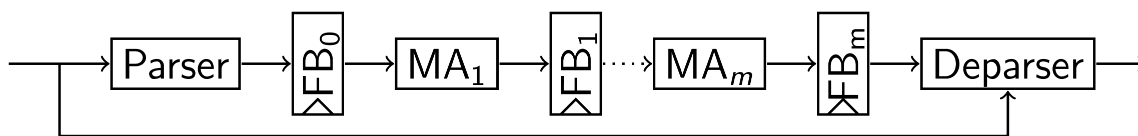


Figure 3.1: RMT switch pipeline.

A hardware implementation of FlexNIC is outside of the scope for this dissertation. However, I can estimate its hardware cost. A typical commodity switch uses merchant silicon to support sixteen 40 Gbps links at a cost of about $10K per switch in volume, including the switching capacity, deep packet buffers, protocol handling, and a match and action (M+A) table for route control. In fact, it is generally believed that converting to the more flexible RMT model will *reduce* costs in the next generation of switches by reducing the need for specialized protocol processing [15]. Extensions to RMT switches for advanced stateful processing and programmable scheduling have also been shown to incur area overheads of less than 2% each compared to a baseline RMT switch [123, 124]. As a result, I believe that adding line-rate FlexNIC support is both feasible and less expensive than adding full network processor or FPGA support.

3.2 Reconfigurable Match Tables in Switches

I now briefly describe the RMT model [15] used in switches before moving on to discuss how to adapt it to support flexible packet processing in a NIC. RMT switches can be programmed with a set of rules that match on various parts of the packet, and then apply data-driven modifications to it, all operating at line rate for the switched packets. This is implemented using two packet processing pipelines that are connected by a set of queues allowing for packets to be replicated and then modified separately.

Such an RMT pipeline is shown in Figure 3.1. A packet enters the pipeline through

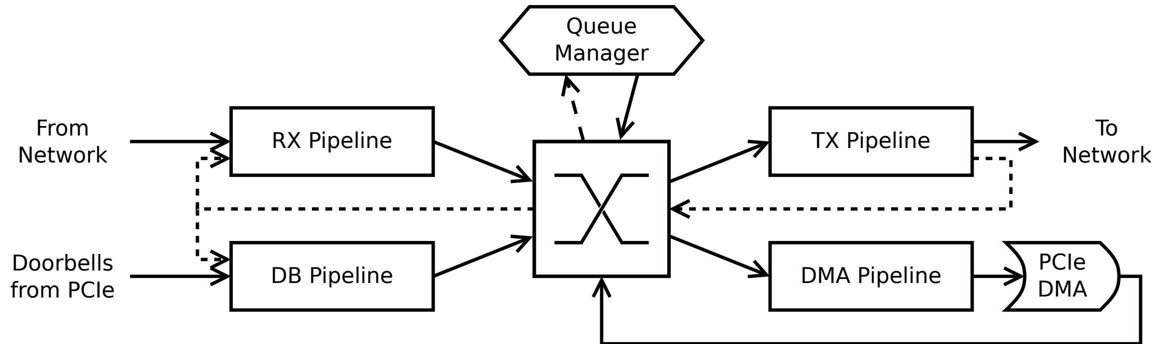


Figure 3.2: RMT-enhanced NIC DMA architecture.

the fully programmable parser, which identifies all relevant packet fields as described by the software-defined parse graph. It extracts the specified fields into a field buffer (FB_0) to be used in later processing stages. The relevant fields pass through the pipeline of M+A stages ($MA_1..MA_m$) and further field buffers ($FB_1..FB_m$). In a typical design, $m = 32$. An M+A stage matches on field buffer contents using a match table (of implementation-defined size), looking up a corresponding action, which is then applied as it moves on to the next field buffer. Independent actions can be executed in parallel within one M+A stage. The deparser combines the modified fields with the original packet data received from the parser to get the final packet. To be able to operate at high line rates, multiple parser instances can be used. In addition to exact matches, RMT tables can also be configured to perform prefix, range, and wildcard matches. Finally, a limited amount of switch-internal SRAM can maintain state across packets and may be used while processing.

3.3 Applying Reconfigurable Match Tables to NICs

The model has four identical RMT packet processing pipelines connected by an interconnect (as shown in Figure 3.2). The receive and PCI doorbell pipelines process incoming packets from the network and notifications from the host CPU, respectively. The transmit and DMA pipelines apply rules to packets received from other pipelines and pass them on to the network or via a DMA engine to/from host memory. In particular, an arriving packet can trigger both a DMA of its contents into application memory and an acknowledgment back to the sender, under RMT control. Finally, the queue manager is responsible for packet scheduling.

Pipelines As with a switch RMT model, each processing pipeline consists of a programmable parser, multiple match-and-action (M+A) stages that execute a fixed number of operations for each packet, and a programmable deparser. Packet fields can be added, stripped, or modified. Operations can also read and write stateful memory local to each pipeline for keeping state across packets. Packets have associated metadata that operations can manipulate. The interconnect reads metadata to determine the destination for a packet. The DMA interface also uses metadata to determine how many bytes to exchange with a particular host memory range.

Interconnect The interconnect routes packets between individual pipelines, and between pipelines and the queue manager. It also buffers packets and arbitrates conflicts if more than one packet arrives at the same time, e.g. a PCIe doorbell and a packet from the network. For each pipeline the interconnect manages a queue of packets waiting to be processed. The interconnect provides back-pressure to components injecting new packets, i.e. the PCIe bus, the network, and the queue manager, to ensure no packets are dropped

after being admitted to the system. To this end the interconnect manages reservations for packets in queues.

Packet metadata fields control routing of packets between components as well as reservations. One metadata field specifies the packet destination, and a second field specifies reservations as a bitmap. The reservation bitmap contains a bit for each pipeline and specifies a super set of pipelines that the packet might be directed to in its remaining lifetime. After initial admission into the system, bits in the reservation bitmap can only be cleared but never be set. The reservation bitmap is used to hold reservations in queues for individual pipelines that a packet can still reach. For each source injecting new packets into the system, i.e. the network, the PCIe bus, and the queue manager, an initial bitmap is specified by a configuration parameter. The interconnect only admits new packets if queue slots in each possible destination pipeline are available. Different sources of packets react differently to back-pressure. The network will simply drop packets, the PCIe bus propagates back-pressure to the CPU, and the queue manager delays injecting the next packet.

DMA Engine To maximize flexibility in my approach, I enhance commodity NIC DMA capabilities by integrating an RMT pipeline with the DMA engine. Interaction with the host is fully controlled by the DMA pipeline. In contrast to traditional NICs that use descriptor queues in host memory, FlexNIC can interact with host memory in many different ways. In addition to descriptor queues FlexNIC can implement other data structures such as linear packet buffers or even RDMA semantics where an incoming packets reads or writes to memory before generating a response packet.

The DMA pipeline issues requests to the DMA controller for transferring data between host memory and packet buffer. Requests to the DMA engine are passed from the DMA pipeline in packet metadata. The relevant metadata fields are an address in host

Field	Description
`offset`	Byte offset in the packet
`length`	Number of bytes to transfer
`direction`	From/To memory
`memBase`	Start address in memory

Table 3.1: Meta data format for DMA requests.

memory, the offset in the packet, the size of the transfer, and the transfer direction, as shown in Table 3.1. Each packet can issue a small number of DMA requests in parallel (I assume 2-4). By combining table-lookups, stateful memory, and arithmetic instructions, the DMA pipeline can generate DMA requests for a wide range of data structures. This design is easily extended to include support for CPU cache steering [48, 102] and atomic operations [101] if supported by the PCIe chipset.

Queue Manager To provide fairness and to enforce resource allocation, FlexNIC needs to arbitrate access to the DMA and transmit pipelines among arriving packets, doorbells, and packets from previously queued outbound connections. FlexNIC is also inherently event driven and generates a single output event for each input event (packet or PCIe doorbell). Looping back to another pipeline is desirable for more complex operations, but ties up packet buffers potentially causing uncontrolled packet drops and backpressure to the PCIe bus. I propose a *queue manager* (QM), as shown in Figure 3.2, to provide packet scheduling and to allow incoming events to be decoupled from resulting outgoing events.

The QM provides a number of rate-limited token buckets. Each packet exiting a pipeline arrives at the QM, with metadata specifying a target bucket and number of tokens to fill (e.g., packet size). The QM stores only the total number of tokens in each bucket, a per-bucket rate limit, and a per-bucket maximum drainage quantum (such as the maximum transfer unit). The QM drains buckets under rate limits and injects *token*

packets containing the originating bucket ID and number of tokens drained (up to the quantum). As long as the interconnect is available, buckets without rate limit are drained immediately in a round-robin fashion. Buckets with rate limits are drained at their rate, oldest-first. The QM alternates between draining buckets with and without rate limits.

The QM decouples events that fill buckets from the events that drain them, allowing drained events to be scheduled fairly. For example, FlexTCP assigns a rate-limited QM bucket to each TCP flow for fair transmit scheduling and congestion control. Given that the QM keeps only bucket levels, its hardware footprint is modest. Prior work exploring programmable packet scheduling [124] in switches has shown feasibility for 1,000s of queues with much higher aggregate throughput than is currently required for NICs.

An example use-case of the queue manager are transmit queues that support doorbell batching. A single doorbell informs the NIC that K entries were added to the transmit queue. Based on this the doorbell pipeline instructs the QM to add K tokens to the token bucket corresponding to this queue. The QM now asynchronously starts injecting K token packets through the interconnect to the DMA pipeline. In the DMA pipeline a table lookup translates the token bucket identifier in the packet to a queue id, provides the memory address of the queue, and a stateful memory location holding the current queue position. Based on this information, the DMA pipeline can issue a DMA request to read the packet from memory and then direct it to the transmit pipeline.

3.3.1 Constraints

To make packet processing at high line-rates feasible, the RMT model is explicitly not freely programmable and several restrictions are imposed upon the user. For example, processing primitives are limited. Multiplication, division and floating point operations are typically not feasible. Hashing primitives, however, are available; this exposes the

hardware that NICs use today for flow steering. Control flow mechanisms, such as loops and pointers, are also unavailable, and entries inside M+A tables cannot be updated on the data path. This precludes complex computations from being used on the data path. The amount of stateful NIC memory is also constrained.

3.4 Building Blocks

During the use case exploration of FlexNIC, a number of building blocks have crystallized, which I believe are broadly applicable. These building blocks provide easy, configurable access to a particular functionality that FlexNIC is useful for. I present them in this section and will refer back to them in later sections.

Multiplexing Multiplexing has proven valuable to accelerate the performance of applications. On the receive path, the NIC has to be able to identify incoming packets based on arbitrary header fields, drop unneeded headers, and place packets into software-defined queues. On the send path, the NIC has to read packets from various application-defined packet queues, prepend the correct headers, and send them along a fixed number of connections. This building block is implemented using only ingress/egress M+A rules.

Flow and Congestion Control Today's high-speed NICs either assume the application is trusted to implement congestion control, or, as in RDMA, enforce a specific model in hardware. Many protocols can be directly encoded in an RMT model with simple packet handling and minimal per-flow state. For example, for flow control, a standard pattern I use is to configure FlexNIC to automatically generate receiver-side acks using ingress M+A rules, in tandem with delivering the payload to a receive queue for application processing. If the application falls behind, the receive queue will fill, the ack is not generated,

and the sender will stall.

For congestion control, enforcement needs to be in the kernel while packet processing is at user level. Many data centers configure their switches to mark an explicit congestion notification (ECN) bit in each packet to indicate imminent congestion. I configure FlexNIC to pull the ECN bits out before the packet stream reaches the application; I forward these to the host operating system on the sender to allow it to adjust its rate limits without needing to trust the application.

Hashing Hashing is essential to the scalable operation of NICs today, and hashes can be just as useful when handling packets inside application software. However, they often need to be re-computed there, adding overhead. This overhead can be easily eliminated by relaying the hardware-computed hash to software. In FlexNIC, I allow flexible hashing on arbitrary packet fields and relay the hash in a software-defined packet header via ingress or DMA rules.

Filtering In addition to multiplexing, filtering can eliminate software overheads that would otherwise be required to handle error cases, even if very few illegal packets arrive in practice. In FlexNIC, I can insert ingress M+A rules that drop unwanted packets or divert them to a separate descriptor queue for software processing.

3.5 Discussion

With FlexNIC I propose one possible hardware model for reconfigurable NICs that can enable more efficient software packet processing. For the purposes of this dissertation the goal for FlexNIC is to argue that a restricted programming model, can enable significant end-to-end performance improvements in software processing. A hardware implementa-

tion of FlexNIC is not necessary to validate the abstraction, and thus out of scope for this dissertation.

The model does cover external architectural constraints arising from how NICs interact with the rest of the server and the external network. The model also defines high-level constraints for what types of operations can and cannot be implemented in FlexNIC. Specific low-level hardware limits, such as how many operations of a specific type a pipeline stage supports, are implementation specific and out of scope. This dissertation also makes no claim that FlexNIC is the only or even the ideal NIC hardware model for this purpose. Instead, I demonstrate that a limited reconfigurable NIC model is useful, and provide a starting point for future design exploration.

Hardware Architecture Design Space There is a large design space to explore for future reconfigurable packet processing systems, with design choices along many axes. For the RMT pipeline the two high-level parameters are the number of stages and the number (and type) of parallel operations within each stage. Existing high-throughput switch designs rely on multiple pipeline instances, raising the throughput but effectively partitioning traffic and state into separate units. While RMT fully partitions memory for lookup tables and state among states, other architectures [19] employ central memory pools, offering flexible allocation to stages but requiring a significantly more complex design. The main drawback of a pipeline for processing is the abrupt performance cliff for processing that cannot be performed in one pass through the pipeline. dRMT [22] instead relies on multiple processors that each execute a stream of match-action operations for a packet, leading to throughput linear in the number of operations required. Memory capacity in RMT is also limited because of the need for fast SRAM. I expect that RMT can be extended with DRAM to provide more memory, for example by adding a mechanism for fetches data from DRAM in an early pipeline stage to be available in time for processing

in a later stage.

Programming Interface Similarly, I omit a precise definition of a programming interface. I implement a version of the FlexNIC emulator that implements processing as shown in Figure 3.2. But in this emulation environment the four pipelines that execute specific protocol processing are implemented as general C functions. I manually ensure that the pipeline configurations do not use any operations not supported by the RMT model. At a high level, NIC processing could be specified in a domain specific language for packet processing, such as P4 [14], but I leave this for future work. The more complex stateful processing in NICs likely requires extensions to P4, for example through abstractions such as packet transactions [123].

Chapter 4

TCP Processing

This chapter describes the design and implementation of the FlexTCP stack. Given FlexNIC, my solution for addressing limitations in existing NICs, the next question is how accelerate TCP, the most commonly used protocols in data centers. TCP is notorious for its complex and resource intensive processing. However, FlexNIC provides a restricted NIC model only that only supports offloading simple protocol processing steps. The rest of this chapter describes this challenge in more detail before discussing how the FlexTCP architecture resolves this mismatch and presenting the prototype implementation.

4.1 Goals

FlexTCP has the following specific design goals:

- **Efficiency:** My primary goal is to reduce the CPU overhead of latency-sensitive TCP small packet processing, well beyond that of kernel bypass. To do so, I offload common-case CPU packet processing to FlexNIC and improve application-level cache locality and multi-core scalability.

- **Connection Scalability:** FlexTCP should scale to the tens of thousands of flows typical of data center scenarios and provide fairness and isolation to each one of them.

- **Performance Predictability:** FlexTCP should also provide consistent performance, isolating each application's performance from the behavior of other applications, and minimizing the impact for handling expensive TCP corner cases.

- **Policy Compliance:** At the same time, FlexTCP needs to enforce complex system policy, affected by various environmental conditions, including congestion control protocol, network configuration, firewalls, and resource isolation policy. It has to do so without cooperation from untrusted user-level software, too much NIC complexity, or the overhead of repeated kernel invocations.

- **Protocol Flexibility:** FlexTCP needs to be compatible with existing TCP peers, applications, and networks. But flow and congestion control, as well as user-layer extensions to TCP, are still an active area of research. I do not intend to stifle this innovation by fixing a single TCP implementation and programming interface in hardware. Instead, my goal is to retain maximum flexibility in the choice of flow and congestion control protocols, programming interfaces, as well as session and presentation layer extensions, such as multiplexing enhancements or RPC object steering.

- **Cost Efficiency:** FlexTCP needs to be economical and reduce the total cost of ownership, by making efficient use of resources and with a cost effective hardware model.

Task	LOC	%	Complexity
State machine	948	26	$O(C)$
Out of order	507	14	$O(N)$
Timers	408	11	$O(C)$
Segmentation	310	9	$O(1)$
Debug, API	291	8	$O(C)$
Common TX	226	6	$O(1)$
Common RX	209	6	$O(1)$
TCP Options	207	6	$O(1)$
Flow control	149	4	$O(1)$
Memory mgmt	117	3	$O(M)$
Port mgmt	111	3	$O(C)$
Congestion control	66	2	$O(1)$, mul, div, mod
Checksum	70	2	$O(1)$
Total	3619	100	

Table 4.1: Top lwIP tasks by LOC and primary computational complexity (C = connections, N = out-of-order packets, M = memory fragmentation). Bold items are feasible to implement with FlexNIC.

4.2 Challenges

As motivation for my implementation, I study the TCP implementation in lwIP [82] version 2.0.2 as an example of a minimal, yet fully featured TCP network stack implementation, and classify the code into the different tasks necessary to support TCP and the data and computational primitives required to carry out these tasks. I use the number of lines of code (LOC) of each task's implementation as a proxy for its complexity.

Table 4.1 shows the result. The majority of the code is dedicated to managing the connection state machine, including connection handshake and maintaining listening and recently closed connections, handling out-of-order packets, and timer-related functionality, including retransmission of lost packets, sending keep-alives, delayed acknowledgments, and timing out of connections. These tasks are data structure intensive. They frequently walk linked lists of connections and TCP segments, requiring a non-constant number of

operations per processed packet. A smaller, but related non-constant task is the allocation of open port space. All of these tasks make frequent use of a library of heap memory management functions to (de-)allocate packet and connection state that is also non-constant. Together, these tasks constitute 57% of the lwIP TCP implementation. Because they have non-constant complexity, they are infeasible to offload to FlexNIC.

Common-case receive and transmit code has constant complexity with each processed packet. The same holds for segmentation, TCP options processing, flow control, and checksum calculations. None of these tasks requires any additional data structures. They operate solely on packet headers or submitted payload. Thus, they are potentially offloadable, subject to computation and per-flow memory constraints. Together, they constitute 33% of the implementation, or 1,170 lines of code. I explain in the next section how to map these tasks on the available hardware primitives.

Only 2% of code is dedicated to congestion management. However, congestion management requires multiplication, division, and modulo operations for round-trip time estimation that are not available in the FlexNIC model. And while approximations [121] could enable implementation of individual congestion control algorithms, congestion management is an area of TCP innovation that is easier to evolve when implemented using a familiar programming model [95]. Hence, I retain it on the CPU.

The remaining 8% of code are dedicated to debugging and API-related functionality, such as calling callbacks. I do not offload them to the NIC.

4.3 FlexTCP Network Stack Design

FlexTCP has three components: FlexNIC program, trusted kernel stack, and untrusted per-application user-space stack. All components are connected via a series of optimized queues in host memory. Queues between user-space and the kernel reside in shared mem-

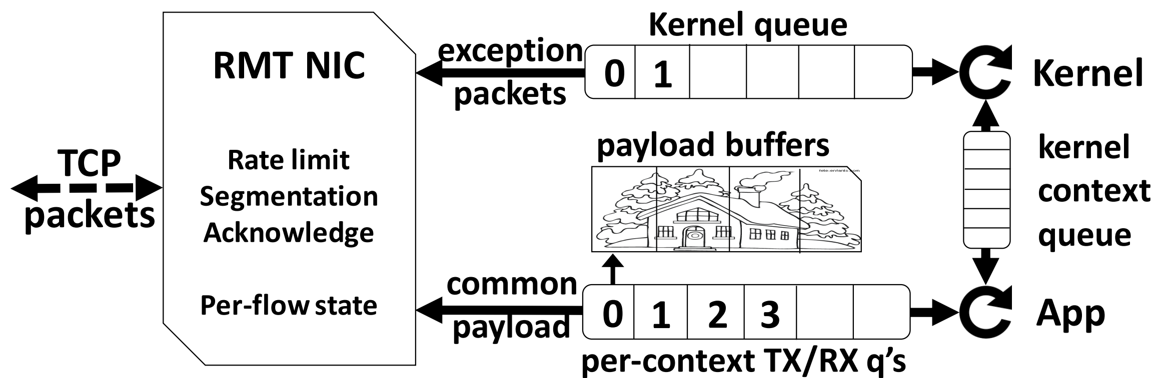


Figure 4.1: FlexTCP design overview. The exception queue is in kernel memory; everything else is in user memory.

ory and are optimized for cache-efficient message passing [7]. Queues between NIC and software use PCI doorbells, identical to those in commodity NICs [56].

Figure 4.1 shows an overview of how FlexTCP components interact. The NIC is responsible for handling common case packet exchanges. To do so, it deposits the payload of incoming packets directly in user-space per-flow *receive payload buffers*, notifying the user-space TCP stack of data arrival via a *receive context queue*. Outgoing payload is written by the user-space TCP stack into per-flow *transmit payload buffers*, notifying the NIC via a *transmit context queue*. The NIC fetches and encapsulates payload according to per-connection *rate limits* that are dynamically configured by the kernel. User-level TCP stacks export the standard POSIX API to applications. Applications do not need to be modified. For connection setup and teardown user-level stacks interact with the kernel using *kernel context queues*.

4.3.1 FlexNIC Functionality

FlexNIC handles common-case exchange of packets on established connections. It also must detect and respond to exceptions, such as out-of-order arrivals and unknown con-

nections, and enforce congestion policy. It processes protocol headers, sends TCP acknowledgments, and performs segmentation.

Common-Case Receive FlexTCP assumes that packets are commonly delivered in order by the network. This is true for data center networks today due to connection-stable multipath routing [39, 122]. With in order packets, the NIC can discard all network headers and directly insert the payload into a user-level, per-flow, circular payload receive buffer.

Circular payload buffers are more efficient than the DMA descriptor queues used by commodity NICs. The NIC can directly write received payload to host memory, notifying an appropriate context queue by identifying the connection and number of bytes received. No prior DMA descriptor reads are necessary. I also do not experience the internal memory fragmentation of fixed-size DMA buffers, allowing for streaming receives without scatter-gather IO. Finally, linear buffers prefetch well in CPU caches and require no more NIC state than DMA descriptors.

Per-flow payload buffers simplify flow control and improve isolation, making a FlexNIC implementation feasible. With shared buffers determining an accurate flow control window requires iteration over all connections sharing the buffer, imposing non-constant per-packet overhead.

When a payload buffer is full, the NIC simply drops the packet. When a context queue is full, the NIC will inform the user-level stack upon future packet arrivals if the queue is available. User-level stacks are free to define and configure contexts. I describe them in more detail in subsection 4.3.3.

NIC-Generated ACKs After depositing the payload of an in-order packet, the NIC automatically generates an acknowledgement packet and transmits it to the sender to update its TCP window. Handling TCP acknowledgements on the NIC is important for safety.

If user-space was given control over acknowledgements, as in many kernel bypass solutions, it can use it to defeat TCP congestion control [118]. The acknowledgements also provide correct ECN feedback, and accurate TCP timestamps for RTT estimation.

Common-Case Send User-level stacks send data on a flow by appending it to a flow's circular transmit buffer. Per-flow send buffers are required to alleviate head-of-line blocking under rate and flow control. To inform the NIC, the stack issues a TX command on a context queue and sets a doorbell. The NIC fills a flow-specific QM bucket with the new amount of data to send. Asynchronously, the QM drains these buckets, depending on the configured rate-limit and the receiver's TCP window to enforce congestion and flow control. When data can be sent, the NIC fetches the appropriate amount from the transmit buffer, produces TCP segments, prepends packet headers for the connection, and transmits.

ACK Processing Any payload that has been sent remains in the transmit buffer until acknowledged by the receiver. The NIC parses incoming acknowledgements, updates per-flow sequence and window state, frees transmit payload buffer space, and informs user-space of reliably delivered packets by issuing a notification with the number of transmitted bytes for the corresponding flow. This requires constant time. With DMA descriptor queues the NIC would have to scan the descriptors to determine the (parts of) buffers freed. The NIC also uses TCP timestamps to provide the kernel with an accurate RTT estimate for congestion control and timeouts.

NIC State To enforce policy, the NIC requires the per-flow state shown in Table 4.2. The opaque field is specified by and relayed to user-space to help it identify the corresponding connection. Similarly, the doorbell register helps the NIC identify what per-core receive

and transmit queue to use. RX/TX buffer state is used for management of per-flow buffers in user-space. The kernel can read and write NIC state as device memory. In all, FlexTCP requires 102 bytes of per-flow state. Current commodity NICs supply about 8 MB of SRAM at reasonable cost (section 4.5). This allows us to keep the state of more than 80,000 active flows in fast memory. Packets for any flows that do not fit are directed to the kernel queue for traditional kernel processing. Integrating ideas to reduce NIC state (e.g., SENIC [109]) is future work.

Exceptions The NIC detects out-of-order arrivals by matching arrivals against expected sequence numbers in the per-flow `seq` register. It drops them and generates an acknowledgement specifying the next expected sequence number. When processing incoming acknowledgements the NIC counts duplicates and triggers fast recovery, without kernel intervention, after three duplicate acknowledgements by resetting the sender state as if those segments had not been sent yet. The NIC also increments a per-flow retransmit counter to inform the kernel to reduce the flow's rate limit.

As an optimization, the NIC tracks one interval of out-of-order data in the receive buffer (starting at `ooo_start` and of length `ooo_len`). The NIC accepts out-of-order segments of the same interval if they fit in the receive buffer. In that case, the NIC writes the payload to the corresponding position in the receive buffer. When an in-order segment fills the gap between existing stream and interval, the NIC notifies the user-level stack as if one big segment arrived, and resets its out-of-order state.

Other exceptions, such as unidentified connections, corrupted packets, and packets with unhandled flags, are filtered and sent to the kernel for processing.

4.3.2 Kernel

The kernel implements all policy decisions and management mechanisms that have non-constant per packet overhead or are too expensive or stateful to process on the NIC. This includes congestion control policy, connection management, user-space TCP stack registry, handling timeouts and other exceptional situations.

Congestion Control FlexTCP enforces congestion control by configuring the NIC to limit each connection to a specific rate. The kernel updates flow rates periodically out-of-band, based on congestion feedback collected by the NIC. Based on received ACKs, the NIC maintains for each flow: bytes acknowledged with and without ECN marks, number of fast retransmits, and current RTT estimate. The kernel runs a control loop iteration for each flow every control interval (configurable, by default every 2 RTTs). It retrieves congestion feedback from the NIC, then runs a congestion control algorithm to calculate a new flow rate, and finally updates the flow rate on the NIC.

This provides a generic framework to implement different congestion control algorithms. I implement DCTCP [3] and TIMELY [90] (adapted for TCP by adding slow-start). I adapt DCTCP to operate on rates instead of windows by applying the control law (rate-decrease proportional to fraction of ECN marked bytes) to flow rates. During slow start I double the rate every control interval until there is an indication of congestion, and during additive increase I add a configurable step size (10 mbps by default) to the current rate. To prevent rates from growing arbitrarily in absence of congestion, I ensure at the beginning of the control loop that the rate is no more than 20% higher than the flow's send rate. FlexTCP's rate-based DCTCP implementation is compatible with Linux peers.

Stack Management To associate new user-space TCP stacks with FlexTCP, the kernel has to be informed via a special system call. If the request is granted, the kernel creates an

Register	Bits	Description
opaque	64	Application-defined flow identifier
doorbell	16	Associated TX doorbell
bucket	24	Associated QM bucket ID
rx|tx_start	128	RX/TX buffer start
rx|tx_size	64	RX/TX buffer size
rx|tx_head|tail	128	RX/TX buffer head/tail position
tx_sent	32	Sent bytes from tx_head
seq	32	Local TCP sequence number
ack	32	Peer TCP sequence number
window	16	Remote TCP receive window
dupack_cnt	4	Duplicate ACK count
local_port	16	Local port number
peer_ip|port|mac	96	Peer 3-tuple (for segmentation)
ooo_start|len	64	Out-of-order interval
cnt_ackb|ecnb	64	ACK'd and ECN marked bytes
cnt_frexmits	8	Fast re-transmits triggered count
rtt_est	32	RTT estimate

Table 4.2: Required per-flow NIC state (102 bytes total).

initial pair of context queues that the user-space stack uses to create connection buffers, etc.

Connection Management Connection management is complex. It includes port allocation, negotiation of TCP options, maintaining ARP tables, and IP routing. I thus handle it in the kernel. User-level TCP stacks issue a `new_flow` command on the kernel context queue to locally request new connections. If granted, the kernel establishes the connection by executing the TCP handshake in software and, if successful, installs the established flow's state in the NIC and allocates a rate-limited QM bucket. Remote requests are detected by the NIC and forwarded to the kernel, which then completes the handshake in software.

Servers can listen on a port by issuing a `listen` command to the kernel. The kernel informs user-space of incoming connections on registered ports by posting a notification in the kernel context queue. If user-space decides to accept the connection, it may issue

the `accept` command to the kernel (via the kernel context queue), upon which the kernel establishes the flow in software. To tear down a connection, user-space issues `close`, upon which the kernel executes the appropriate handshake in software and removes the flow state from the NIC. Similarly, for remote teardowns, the kernel informs user-space via a `close` command.

Retransmission Timeouts I handle retransmission timeouts in the kernel. When collecting congestion statistics for a flow from the NIC, the kernel also checks for unacknowledged data.If a flow has unacknowledged data with a constant sequence number for multiple control intervals (2 by default) the kernel instructs the NIC to start retransmitting by adding a command to the kernel context queue. In response to this command the NIC will reset the flow and start transmitting exactly as described above for fast retransmits.

4.3.3 User-space TCP Stack

The user-space TCP stack presents the programming interface to the application. The default interface is POSIX sockets so applications can remain unmodified, but per-application modifications and extensions are possible, as the interface is at user-level [104, 83, 8]. The TCP stack is responsible for managing connections and contexts, as well as sending and receiving payload. To fulfill the performance goals, common-case overhead of the TCP stack is minimal.

Context Management User-space stacks are responsible for defining and allocating contexts. Contexts are useful in various ways, but typically stacks allocate one context per application thread for scalability, as it allows cores to poll only a private context queue, rather than a number of shared payload buffers. This is especially important with many connections. Contexts do not need to match connections and performance can be gained

when not doing so (section 4.4). Stacks allocate contexts via management commands to the kernel.

4.4 Flexible FlexTCP Extension

FlexTCP is flexible. This allows us to build higher-level extensions that can improve application efficiency and scalability. I have developed one such extension, which I present in this section.

Server applications can improve performance by steering similar application-level objects to the same set of cores in order to benefit from increased cache locality and utilization [77]. However, TCP connections are difficult to process across multiple cores because of the complex per-connection state involved [60], prohibiting this approach. If I can perform common-case TCP connection processing on the NIC, I can steer incoming objects to identical worker cores without incurring the management overhead.

Overview I provide an object steering extension to address this problem. I define an overlay stream of variable-size objects on top of the TCP byte stream. Objects are application-identified sequences of bytes, such as RPC requests and responses. I continue to support TCP's guarantees with relaxed ordering of objects delivered to different cores. Applications are responsible to specify steering semantics such that object dependencies are not violated. This is in-line with the primary use-case of object steering, which is to facilitate parallel processing of multiple incoming remote procedure calls. For example, to support pipelined random access memory (PRAM) consistency, writes would continue to be steered by connection, while reads can be steered to any available CPU.

For connections using object steering, the TCP byte streams contain sequences of objects. Each object consists of a header, a key, and payload. The header frames the object,

specifying its total length, as well as the length of the key. Upon receiving an object, the NIC uses the key to determine the target core for the object. I currently provide two configurable steering mechanisms for each connection: hash-based and direct. Hash-based steering calculates a hash over the key and then uses it to choose the target core, akin to receive side scaling [57]. Direct steering uses the key directly as the target core identifier.

Implementation To implement object steering, I extend the NIC flow state with a flag to mark the flow as an object steering connection, a flag controlling which steering mechanism to use, and two 32 bit registers `rx/tx_objrem` storing the number of bytes left in the current object across segments used for receive and transmit operations. Kernel processing is modified to set up this state and to extend the slow path as described below. The protocol requires that objects always start on TCP segment boundaries and that the header and key fields are fully contained in that segment, but an object can span multiple segments.

Sending Objects The NIC requires `send` commands on the per-context transmit queue to identify object boundaries. Upon send, the NIC uses the `tx_objrem` register to determine if a new object starts with this segment, or whether there is an object that has been partially sent. For new objects, the NIC inspects the object header after fetching a segment to determine the length of the object. The NIC stores the object's length in `tx_objrem` and then truncates the segment if it contains more than this object. If there is an object that has been partially sent, the NIC fetches the minimum of a full segment and `tx_objrem`. Finally, for both cases, the NIC decrements `tx_objrem` based on the number of bytes sent.

Receiving Objects Upon receiving a segment, the NIC uses the `rx_objrem` register to determine if this segment is the start of a new object, or if the segment is a continuation of the current partial object. For new objects, the NIC stores the total length of the object in `rx_objrem`, uses the key to determine the destination application core, and updates the `doorbell` register accordingly. For both cases, the NIC will then subtract the current number of payload bytes from `rx_objrem`. A higher number of payload bytes than `rx_objrem` is a protocol error and I direct the packet on the slow path for the kernel process. The notification on the per-core receive queue is extended to include the position in the receive buffer where the object starts. These notifications can cover partial objects, but the NIC ensures that all notifications for a particular object are received in order by the same core.

Notifications for objects from different connections might be interleaved. Thus, when processing object notifications, the TCP stack needs to track the start position of a partial object for each connection. Due to asynchronous processing, different cores can be processing notifications for the same connection, thus this state needs to be local to the core. The NIC and TCP stack will guarantee that objects are received in their entirety even if split among TCP segments.

4.5 Implementation

I have implemented FlexTCP from scratch, both as a software NIC extension to a commodity Intel NIC [56] using DPDK [27] in 2,931 lines of C code, as well as an RMT program for the Netronome Agilio-CX flow processor [96], implemented in the P4 programming language [14]. For both implementations, the host-side components are the same: a trusted kernel component (3,744 lines of C) and a user-level library providing the POSIX sockets API that is dynamically linked to unmodified application binaries (3,452 lines of

C).

Kernel I currently implement all kernel-related functionality in a separate user-level process. This has no performance impact but simplifies development. To bootstrap the kernel context queues, FlexTCP requires applications to first connect to the kernel via a named UNIX domain socket. Applications use the socket to set up a shared memory region for the context queues. The kernel process also uses the socket for automatic cleanup, to detect when application processes exit by receiving a hangup signal via the corresponding socket.

Software Emulation I developed a software emulator, emulating the NIC model and implemented FlexTCP on top. The software emulator uses a configurable number of dedicated host cores, which can be adjusted based on the NIC line rate, replicating each RMT pipeline, and a shared memory interface to system software that mimics the hardware interface. Since the emulator replicates each RMT pipeline over multiple cores, each emulator core exposes a queue pair to the kernel and to each application context to avoid synchronization. The NIC's RSS mechanism ensures that packets within flows are assigned to the same pipeline and not reordered. A hardware implementation would expose only one queue pair to kernel and application contexts, and thus be slightly faster than the emulation.

RMT Implementation One of my collaborators implemented FlexTCP in the P4 programming language and compiled it to the fully programmable Netronome Agilio-CX 40G NIC [96]. Netronome's P4 compiler currently does not support parsing variable-length headers (this is not a limitation of the RMT approach). Thus, I padded packet headers to reasonable sizes where necessary and implemented parsing variable-sized TCP

options in C. The RMT implementation has 320 lines of P4 and 270 lines of C code.

SoftTCP Implementation I adapted the emulation software to build a host-only version of FlexTCP called SoftTCP, removing artifacts of the NIC hardware model not needed in this setting, such as the emulation of PCIe doorbells and RMT pipelines. As with the emulator, SoftTCP runs in a separate privileged user-level process. Unlike kernel bypass or systems that rely on batching to reduce kernel-user switches, SoftTCP runs on its own CPU cores (configured to match the application workload), using queues to communicate with the user-space TCP stack. I find that the optimizations needed to streamline TCP execution on a NIC—separating out the common case, reducing state, and avoiding non-constant time operations—are also effective at reducing purely software packet processing overhead.

4.6 Limitations

Fixed Connection Buffer Sizes FlexTCP requires connection send and receive buffers to be fixed upon connection creation. I do not currently implement any buffer resizing depending on load. For workloads with large numbers of inactive connections, buffer resizing (via additional management commands) is desirable.

Wire-Format for Object Steering Object steering in FlexTCP places restrictions on how messages are split into individual TCP segments. I believe that avoiding this limitation requires either an extension to the NIC hardware model or additional serialized CPU processing. I leave further exploration of these trade-offs as future works.

No IP Fragments The current design does not support fragmented IP packets. I believe this is sufficient, as IP fragmentation does not normally occur in the data center. The

principal difficulty in handling IP fragment is that not all fragments include TCP headers. Handling fragments typically involves buffering fragments until the message is complete, which is beyond the capabilities for FlexNIC offload.

4.7 Discussion

I have presented FlexTCP, a TCP stack based on a unique split of processing between FlexNIC, the operating system kernel, and the application. My approach improves performance without sacrificing flexibility or multi-tenant policy enforcement. Despite the well-known complexity of TCP processing, FlexTCP can take advantage of the limited abstractions provided by FlexNIC.

In FlexTCP, I make a number of design decisions, not all of which are fundamental to the architecture, but some are based on experiences from previous work on data center packet processing. One example is my use of rate-based congestion control instead of traditional window-based congestion control. Window-based congestion control can result in bursty transmit behavior [17] where burst of packets are followed by periods of no transmissions, while rate-based packet pacing spreads out packets evenly. In this setting FlexNIC serves to enable previously prohibitively expensive packet pacing at line-rate.

FlexTCP implements a simplified mechanism for recovering from packet loss and reordering compared to most software TCP implementations. In subsection 6.1.2, I explore the impact of this simplification and find that for data center loss-rates the cost seems acceptable. But there are other options that likely perform better for higher loss rates. One option is to add support for selective acknowledgements, allowing the receiver to only retransmit lost bytes. On the sender side, selective acknowledgements (SACKs) can be processed in the NIC, similar to how duplicate ACKs are currently handled. Another option is to move this processing into software and issue re-transmits from the kernel,

simplifying NIC processing. On the receiver side, SACKs can also be generated in the NIC. But tracking multiple ranges of out-of-order data, to generate SACKs and to recognize once gaps are filled in, requires additional NIC state. These and other opportunities for optimizations can be addressed by follow-up work and could extend the applicability of my approach to a wider range of operating conditions.

In a multi-tenant data center setting running virtual machines, FlexTCP would also need to be integrated with the hypervisor. While I expect that this is feasible, I leave a concrete design and implementation of hypervisor integration as future work.

Chapter 5

Application Integration

In this chapter I present three case studies of how applications can leverage my flexible architecture to reduce application processing overhead. After accelerating kernel protocol processing many applications still incur significant overhead for application-level processing. Part of this overhead is due to the required protocol processing at the application level. In addition, many applications incur memory and synchronization overheads due to the interaction between the application and the (accelerated) network stack. For three typical data center applications, I discuss this processing overhead and demonstrate how my architecture, FlexNIC in particular, can help reduce it.

5.1 Key-Value Store

I now describe the design of a key-value store, FlexKVS, that is compatible with Memcached [87], but whose performance is optimized using the functionality provided by FlexNIC. To achieve performance close to the hardware limit, I needed to streamline the store's internal design, as I hit several scalability bottlenecks with Memcached. The authors of MICA [77] had similar problems, but unlike MICA, I assume no changes to the

protocol or client software. I discuss the design principles in optimizing FlexKVS before outlining its individual components.

5.1.1 Motivation

To motivate offload of application processing to FlexNIC, I start by describing common performance bottlenecks in memcached, and describe how flexible offload can mitigate them.

Memory-Efficient Scaling To scale request throughput, today's NICs offer receive-side scaling (RSS), an offload feature that distributes incoming packets to descriptor queues based on the client connection. Individual CPU cores are then assigned to each queue to scale performance with the number of cores. Additional mechanisms, such as Intel's FlowDirector [56], allow OSes to directly steer and migrate individual connections to specific queues. Linux does so based on the last local send operation of the connection, assuming the same core will send on the connection again.

Both approaches suffer from a number of performance drawbacks:

1. Hot items are likely to be accessed by multiple clients, reducing cache effectiveness by replicating these items in multiple CPU caches.

2. When a hot item is modified, it causes synchronization overhead and global cache invalidations.

3. Item access is not correlated with client connections, so connection-based steering is not going to help.

FlexNIC allows us to tailor these approaches to Memcached. Instead of assigning clients to server cores, I can *partition* the key space [77] and use a separate key space

request queue per core. I can install rules that steer client requests to appropriate queues, based on a hash of the requested key in the packet. The hash can be computed by FlexNIC using the existing RSS hashing functionality. This approach maximizes cache utilization and minimizes cache coherence traffic. For skewed or hot items, I can use the NIC to balance the client load in a manner that suits both the application and the hardware (e.g., by dynamically re-partitioning or by routing hot requests to two cores that share the same cache and hence benefit from low latency sharing).

Streamlined Request Processing Even if most client requests arrive well-formed at the server and are of a common type—say, GET requests—network stacks and Memcached have to inspect each packet to determine where the client payload starts, to parse the client command, and to extract the request ID. This incurs extra memory and processing overhead as the NIC has to transfer the headers to the host just so that software can check and then discard them. Measurements using the Arrakis OS [104] showed that, assuming kernel bypass, network stack and application-level packet processing take half of the total server processing time for Memcached.

With FlexNIC, I can check and discard Memcached headers directly on the NIC before any transfer takes place and eliminate the server processing latency. To do so, I install a rule that identifies GET requests and transfers only the client ID and requested key to a dedicated fast-path request queue for GET requests. If the packet is not well-formed, the NIC can detect this and instead transfer it in the traditional way to a slow-path queue for software processing. Further, to support various client hardware architectures, Memcached has to convert certain packet fields from network to host byte order. I can instruct the NIC to carry out these simple transformations for us, before transferring the packets into host memory.

5.1.2 Design Goals

Minimize Cache Coherence Traffic FlexKVS achieves memory-efficient scaling by partitioning the handling of the key-space across multiple cores and using FlexNIC to steer incoming requests to the appropriate queue serving a given core. Key-based steering improves cache locality, minimizes synchronization, and improves cache utilization, by handling individual keys on designated cores without sharing in the common case. To support dynamic changes to the key assignment for load balancing, FlexKVS's data structures are locked; in between re-balancing (the common case), each lock will be cached exclusive to the core.

Specialized Critical Path I further optimize FlexKVS by offloading request processing work to FlexNIC and specializing the various components on the critical path. FlexNIC checks headers, extracts the request payload, and performs network to host byte order tranformations. With these offloads, the FlexKVS main loop for receiving a request from the network, processing it and then sending a response consists of fewer than 3,000 x86 instructions including error handling. FlexKVS also makes full use of the zero-copy capabilities of Extaris. Only one copy is performed when storing items upon a SET request.

Figure 5.1 depicts the overall design, including a set of simplified FlexNIC rules to steer GET requests for a range of keys to a specific core.

5.1.3 FlexKVS Components

I now discuss the major components of FlexKVS: the hash table and the item allocator.

Hash Table FlexKVS uses a block chain hash table [77]. To avoid false sharing, each table entry has the size of a full cache line, which leaves room for a spin-lock, multiple

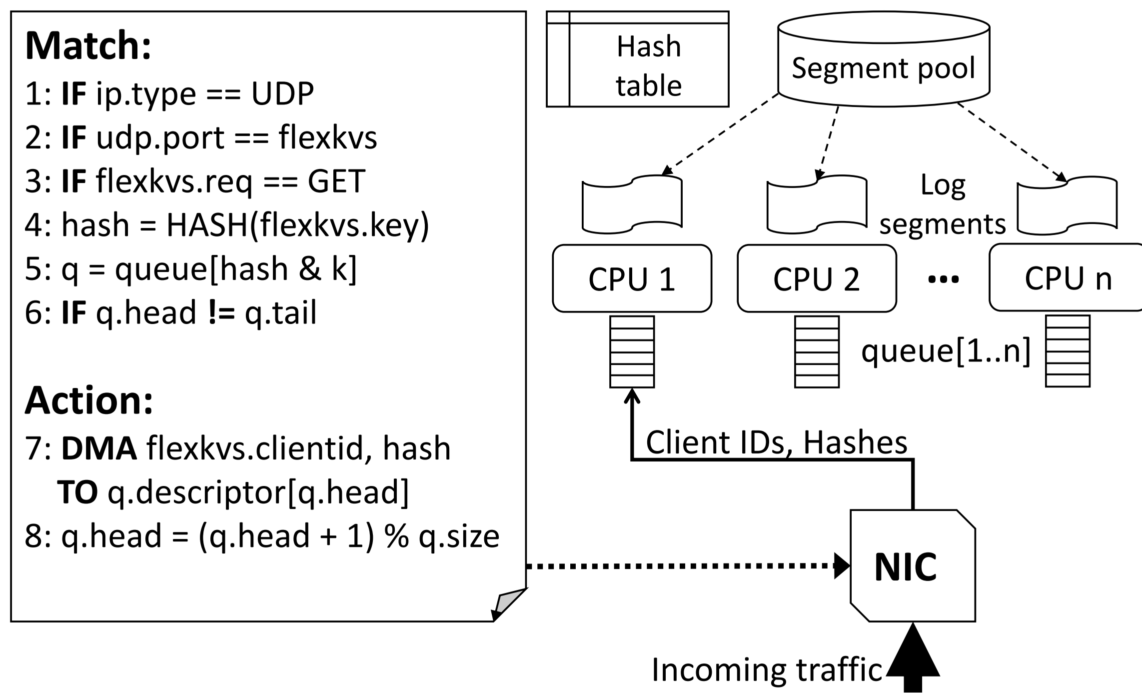


Figure 5.1: FlexNIC receive fast-path for FlexKVS: A rule matches GET requests for a particular key range and writes only the key hash together with a client identifier to a host descriptor queue. The queue tail is updated by FlexKVS via a register write.

item pointers (five on x86-64), and the corresponding hashes. Including the hashes on the table entries avoids dereferencing pointers and touching cache lines for non-matching items. If more items hash to an entry than there are available pointers the additional items are chained in a new entry via the last pointer. I use a power of two for the number of buckets in the table. This allows us to use the lowest k bits of the hash to choose a bucket, which is easy to implement in FlexNIC. k is chosen based on the demultiplexing queue table size loaded into the NIC (up to 128 entries in the prototype).

Item Allocation The item allocator in FlexKVS uses a log [116] for allocation as opposed to a slab allocator used in Memcached. This provides constant-time allocation and minimal cache access, improving overall request processing time. To minimize synchronization, the log is divided into fixed-size segments. Each core has exactly one active segment that is used for satisfying allocation requests. A centralized segment pool is used to manage inactive segments, from which new segments are allocated when the active segment is full. Synchronization for pool access is required, but is infrequent enough to not cause noticeable overhead.

Item Deletion To handle item deletions, each log segment includes a counter of the number of bytes that have been freed in the segment. When an item's reference count drops to zero, the item becomes inactive and the corresponding segment header is looked up and the counter incremented. A background thread periodically scans segment headers for candidate segments to compact.When compacting, active items in the candidate segment are re-inserted into a new segment and inactive items deleted. After compaction, the segment is added to the free segment pool.

5.1.4 FlexNIC Implementation

The FlexNIC implementation consists of key-based steering and a custom DMA interface. I describe both.

Key-Based Steering To implement key-based steering, I utilize the hashing and demultiplexing building blocks on the key field in the FlexKVS request packet, as shown in Figure 5.1. When enqueuing the packet to the appropriate receive queue based on the hash of the key, the NIC writes the hash into a special packet header for software to read. This hash value is used by FlexKVS for the hash table lookup.

Custom DMA Interface FlexNIC can also perform the item log append on SET requests, thereby enabling full zero-copy operation for both directions, and removal of packet parsing for SET requests in FlexKVS. To do so, FlexKVS registers a small number (four in the prototype) of log segments per core via a special message enqueued on a descriptor queue. These log segments are then filled by the NIC as it processes SET requests. When a segment fills up, FlexNIC enqueues a message to notify FlexKVS to register more segments.

FlexNIC still enqueues a message for each incoming request to the corresponding core, so remaining software processing can be done. For GET requests, this entails a hash table lookup. For SET requests, the hash table needs to be updated to point to the newly appended item.

Adapting FlexKVS to the custom DMA interface required adding 200 lines for interfacing with FlexNIC, adding 50 lines to the item allocator for managing NIC log segments, and modifications to request processing reducing the original 500 lines to 150.

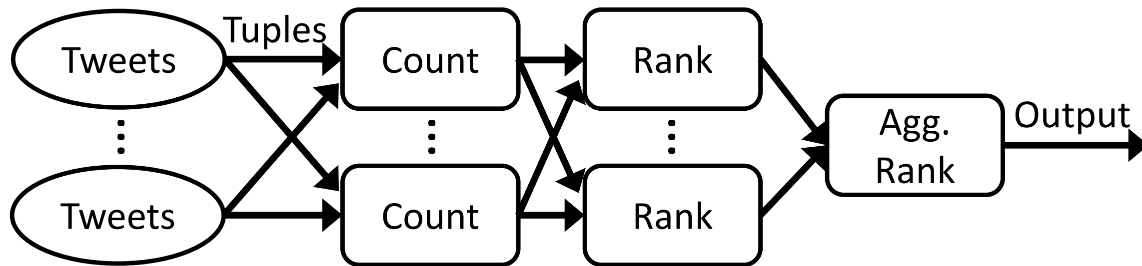


Figure 5.2: FlexStorm top-n Twitter users topology.

5.2 Real-time Analytics

Real-time analytics platforms are useful tools to gain instantaneous, dynamic insight into vast datasets that change frequently. To be considered "real-time", the system must be able to produce answers within a short timespan (typically within a minute) and process millions of dataset changes per second. To do so, analytics platforms utilize data stream processing techniques: A set of *worker nodes* run continuously on a cluster of machines; data *tuples* containing updates stream through them according to a dataflow processing graph, known as a *topology*. Tuples are emitted and consumed worker-to-worker in the topology. Each worker can process and aggregate incoming tuples before emitting new tuples. Workers that emit tuples derived from an original data source are known as *spouts*.

In the example shown in Figure 5.2, consider processing a live feed of tweets to determine the current set of top-n tweeting users. First, tweets are injected as tuples into a set of counting workers to extract and then count the user name field within each tuple. The rest of the tuple is discarded. Counters are implemented with a sliding window. Periodically (every minute in this case), counters emit a tuple for each active user name with its count. Ranking workers sort incoming tuples by count. They emit the top-n counted users to a single aggregating ranker, producing the final output rank to the user.

As shown in Figure 5.2, the system scales by replicating the counting and ranking

workers and spreading incoming tuples over the replicas. This allows workers to process the data set in parallel. Tuples are flow controlled when sent among workers to minimize loss. Many implementations utilize the TCP protocol for this purpose.

I have implemented a real-time analytics platform FlexStorm, following the design of Apache Storm [133]. Storm and its successor Heron [71] are deployed at large-scale at Twitter. For high performance, I implement Storm's "at most once" tuple processing mode. In this mode, tuples are allowed to be dropped under overload, eliminating the need to track tuples through the topology. For efficiency, Storm and Heron make use of multicore machines and deploy multiple workers per machine. I replicate this behavior.

FlexStorm uses DCCP [69] for flow control. DCCP supports various congestion control mechanisms, but, unlike TCP, is a packet-oriented protocol. This simplifies implementation in FlexNIC. I use TCP's flow-control mechanism within DCCP, similar to the proposal in [36], but using TCP's cumulative acknowledgements instead of acknowledgement vectors and no congestion control of acknowledgements.

As topologies are often densely interconnected, both systems reduce the number of required network connections from per-worker to per-machine connections. On each machine, a demultiplexer thread is introduced that receives all incoming tuples and forwards them to the correct executor for processing. Similarly, outgoing tuples are first relayed to a multiplexer thread that batches tuples before sending them onto their destination connections for better performance. Figure 5.3 shows this setup, which I replicate in FlexStorm.

5.2.1 FlexNIC Implementation

As we will see later, software demultiplexing has high overhead and quickly becomes a bottleneck. We can use FlexNIC to mitigate this overhead by demultiplexing tuples in the NIC. Demultiplexing works in the same way as for FlexKVS, but does not require hashing.

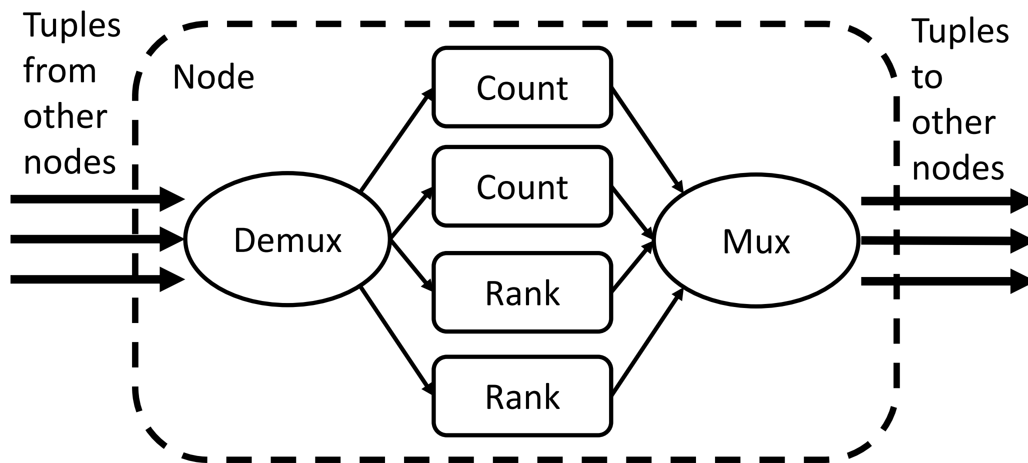


Figure 5.3: Storm worker design. 2 counters and 2 rankers run concurrently in this node. (De-)mux threads route incoming/outgoing tuples among network connections and workers.

We strip incoming packets of their headers and deliver contained tuples to the appropriate worker's tuple queue via a lookup table that assigns destination worker identifiers to queues. However, our task is complicated by the fact that we have to enforce flow control.

To implement flow-control at the receiver in FlexNIC, we acknowledge every incoming tuple immediately and explicitly, by crafting an appropriate acknowledgement using the incoming tuple as a template. Figure 5.4 shows the required M+A pseudocode. To craft the acknowledgement, we swap source and destination port numbers, set the packet type appropriately, copy the incoming sequence number into the acknowledgement field, and compute the DCCP checksum. Finally, we send the reply IP packet, which does all the appropriate modifications to form an IP response, such as swapping Ethernet and IP source and destination addresses and computing the IP checksum.

To make use of FlexNIC we need to adapt FlexStorm to read from our custom queue format, which we optimize to minimize PCIe round-trips by marking whether a queue position is taken with a special field in each tuple's header. To do so, we replace FlexStorm's

Match:	**Action:**
IF ip.type == DCCP	SWAP(dccp.srcport, dccp.dstport)
IF dccp.dstport == FlexStorm	dccp.type = DCCP_ACK
	dccp.ack = dccp.seq
	dccp.checksum = CHECKSUM(dccp)
	IP_REPLY

Figure 5.4: Acknowledging incoming FlexStorm tuples in FlexNIC.

per-worker tuple queue implementation with one that supports this format, requiring a change of 100 lines of code to replace 4 functions and their data structures.

5.3 Intrusion Detection

In this section I discuss how I leverage flexible packet steering to improve throughput for the Snort intrusion detection system [115]. Snort detects malicious activity, such as buffer overflow attacks, malware, and injection attacks, by scanning for suspicious patterns in individual packet flows.

Snort only uses a single thread for processing packets, but is commonly scaled up to multiple cores by running multiple Snort processes that receive packets from separate NIC hardware queues via RSS [42]. However, since many Snort patterns match only on subsets of the port space (for example, only on source or only on destination ports), I end up using these patterns on many cores, as RSS spreads connections by a 4-tuple of IP addresses and port numbers. Snort's working data structures generally grow to 10s of megabytes for production rule sets, leading to high cache pressure. Furthermore, the Toeplitz hash commonly used for RSS is not symmetric for the source and destination fields, meaning the two directions of a single flow can end up in different queues, which can be problematic for some stateful analyses where incoming and outgoing traffic is examined.

5.3.1 FlexNIC Implementation

My approach to improve Snort's performance is similar to FlexKVS. I improve Snort's cache utilization by steering packets to cores based on expected pattern access, so as to avoid replicating the same state across caches. Internally, Snort groups rules into port groups and each group is then compiled to a deterministic finite automaton that implements pattern matching for the rules in the group. When processing a packet, Snort first determines the relevant port groups and then executes all associated automatons.

I instrument Snort to record each distinct set of port groups matched by each packet (which I call *flow groups*) and aggregate the number of packets that match this set and the total time spent processing these packets. In my experience, this approach results in 30-100 flow groups. I use these aggregates to generate a partition of flow groups to Snort processes, balancing the load of different flow groups using a simple greedy allocation algorithm that starts assigning the heaviest flow groups first.

This partitioning can then be used in FlexNIC to steer packets to individual Snort instances by creating a mapping table that maps packets to cores, similar to the approach in FlexStorm. I also remedy the issue with the Toeplitz hash by instructing FlexNIC to order 4-tuple fields arithmetically by increasing value before calculating the hash, which eliminates the asymmetry.

5.4 Discussion

In this chapter I have demonstrated how FlexNIC can streamline application processing and remove scalability bottlenecks in three case studies. I have explored different levels of integration with the different case studies, varying from deep integration with application data structures in the key-value store, to only modifying NIC packet steering in the

intrusion detection system.

I have focused the discussion on performance benefits enabled by integrated processing. But practical deployment of application-specific NIC processing will require addressing additional aspects that I leave for future work. First, when running application processing on the NIC, memory isolation for DMA needs to be enforced by the operating system. I imagine reserving the last pipeline stage(s) in the FlexNIC DMA pipeline for the OS to validate DMA accesses. The kernel can use a match-action table to track address regions that individual applications are allowed access to and catch accesses that violate this policy. Next, applications should only be able to influence the processing of packets that are destined to/sent from them. As an initial step manual review of FlexNIC programs can verify this property. But eventually tool chain support for combining configurations from multiple applications and checking correctness properties would be required. I expect that techniques developed in network verification efforts can be modified for this purpose. Finally, a production deployment would also require support for dynamically starting and terminating applications. This requires operating system support for modifying, compiling, and verifying FlexNIC configurations at run-time.

Chapter 6

Evaluation

This chapter evaluates my approach for integrated processing in FlexTCP and the three application case studies. Using a combination of software emulation and creative re-use of existing hardware features I evaluate performance in micro benchmarks as well as full applications.

6.1 FlexTCP

For FlexTCP, I seek to answer the following questions:

- By how much does FlexTCP improve CPU efficiency, latency, and connection scalability for remote procedure call operation compared to state-of-the-art software solutions? How much is due to streamlining (evaluated via SoftTCP) and how much is due to FlexNIC offload (evaluated via emulation)?
- Do these improvements result in better end-to-end throughput and latency for data center applications? How do these workloads scale with the number of CPU cores?

- Can object steering provide the same performance improvements for TCP-based applications as it does for unreliable, connectionless protocols (see section 6.2)?
- Does the simplified fast-path TCP operation negatively affect performance under packet loss or congestion?
- Is the labor split among host and FlexNIC impacting FlexTCP performance when FlexNIC sits across the PCIe bus? Is the amount of required per-flow state reasonable for a FlexNIC?

To answer these questions we first evaluate RPC performance on a number of systems using microbenchmarks. We then evaluate two data center application workloads: a typical, read-heavy, key-value store application and a real-time analytics framework. Finally, we validate our results using our P4 implementation on the Netronome Agilio-CX 40G NIC and an ns-3 simulation.

Testbed Cluster Our evaluation cluster contains a 24-core Intel Xeon Platinum 8160 (Skylake) system at 2.1 GHz with 196 GB RAM, 33 MB aggregate cache, and an Intel XL710 40Gb Ethernet adapter. We use this system as *the server*. There are also six 6-core Intel Xeon E5-2430 (Sandy Bridge) systems at 2.2 GHz with 18MB aggregate cache, which we use as clients. These systems have Intel X520 (82599-based) dual-port 10Gb Ethernet adapters with both ports connected to the switch. We run Ubuntu Linux 16.04 (kernel version 4.4) with DCTCP congestion control on all machines. We use an Arista 7050S-64 Ethernet switch, set up for DCTCP-style ECN marking at a threshold of 65 packets. The switch has 10G ports (connected to the clients) and 40G ports (connected to the server).

Baseline We compare FlexTCP/SoftTCP performance to both the Linux in-kernel TCP stack (using `epoll`) and to the mTCP user-level TCP stack [60], an efficient, zero-copy,

and scalable network stack. However, mTCP does **not** provide the same safety guarantees as FlexTCP/SoftTCP or Linux. mTCP has no trusted entity that can reliably enforce policy.

Peer Compatibility Our benchmarks do not mix peer systems, but we confirm that FlexTCP and SoftTCP interoperate with existing TCP peers by comparing the aggregate throughput of 100 flows between two hosts among all combinations of Linux and FlexTCP/SoftTCP senders and receivers. Line rate was achieved in all cases.

6.1.1 Remote Procedure Call (RPC)

RPC is a demanding, but necessary mechanism for many server applications. RPCs are both latency and throughput sensitive. Scaling reliable RPCs to many connections has been a long-standing challenge due to the high overhead of software TCP packet processing [98, 113, 117]. To demonstrate the per-core efficiency benefits of FlexTCP, we evaluate a simple single-threaded event-based RPC echo server. SoftTCP requires at least two cores, an application core and a core for the network stack, so we divide SoftTCP throughput results by two to achieve a fair comparison.

Connection Scalability

For each benchmark run, we establish an increasing number of client connections to the server and measure RPC throughput and latency over 1 minute. To do so, we use multi-threaded clients running on as many client machines as necessary to offer the required load. Each client thread leaves a single 64-byte RPC per connection in flight and waits for a response in a closed loop. Clients measure latency by embedding a send timestamp in the RPC that is evaluated when the echo response is received.

Figure 6.1 shows throughput as we vary the number of client connections. On a sin-

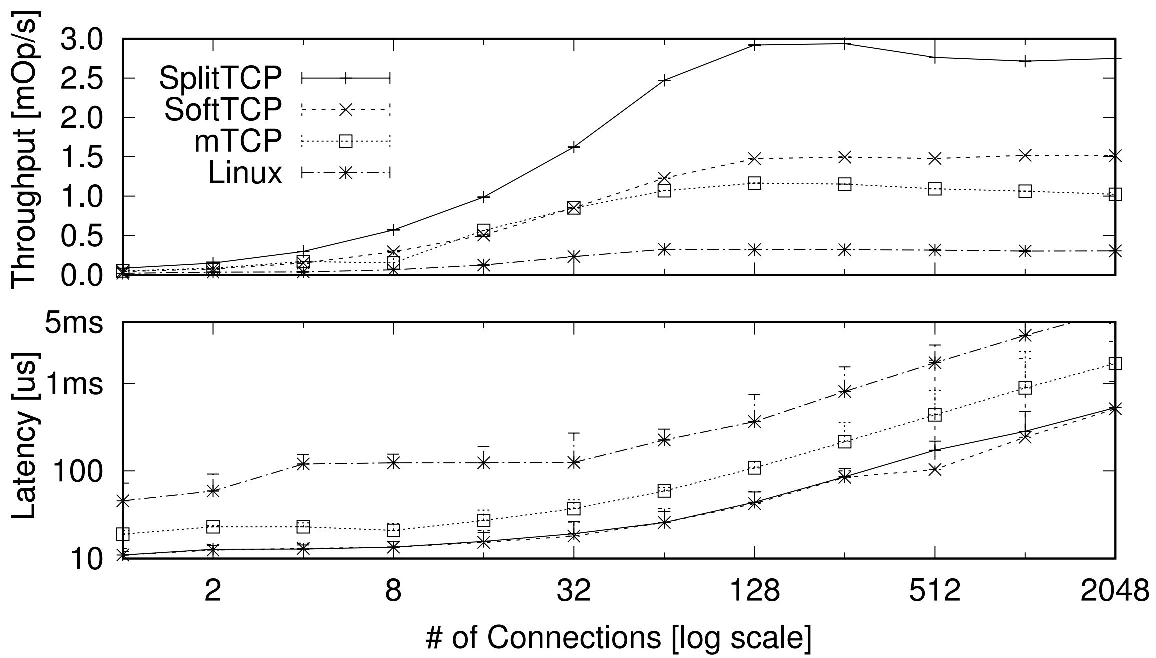


Figure 6.1: RPC echo throughput and latency (median and 99th percentile) for a single-threaded server.

gle connection SoftTCP shows throughput of 2× Linux, but 15% lower than mTCP. The improvement versus Linux is because SoftTCP streamlines processing and thus gains efficiency, while mTCP does not provide protection. All stacks except Linux then scale until saturation of the application thread. SoftTCP's division of labor into common and uncommon TCP processing experiences less contention on uncommon data structures, resulting in improved throughput. At this point, SoftTCP shows throughput of 5× Linux and 1.4× mTCP. By offloading to the (emulated) FlexNIC, FlexTCP achieves even better throughput of 1.7× mTCP, 4× Linux, and 2× SoftTCP on a single connection. At saturation, FlexTCP has a throughput of 2.5× mTCP, 9× Linux, and 1.8× SoftTCP.

Figure 6.1 also shows RPC round-trip time (RTT) as we increase the number of client connections. We can see that FlexTCP and SoftTCP both achieve 42% and 76% better median latency with a single connection than mTCP and Linux, respectively. As we increase the number of concurrent connections, latency increases for all configurations, but more slowly for FlexTCP and SoftTCP. At 1024 concurrent connections, they achieve 68% and 92% better median latency than mTCP and Linux, respectively. Again, this is expected due to FlexTCP's streamlining. Linux does not perform well in the tail, while mTCP performs roughly equivalent to the median until 128 connections, after which queues start to build. The division of labor in FlexTCP and SoftTCP provides lower common-case tail latency and queues only start to build after 1024 connections.

Pipelined RPC

In cases without dependencies, RPCs can be pipelined on a single connection. These transfers can still be limited by TCP stack overheads, depending on RPC size. We compare pipelined RPC throughput for different sizes by running a single-threaded event-based server processing RPCs on 100 connections, partitioned equally over 4 client machines

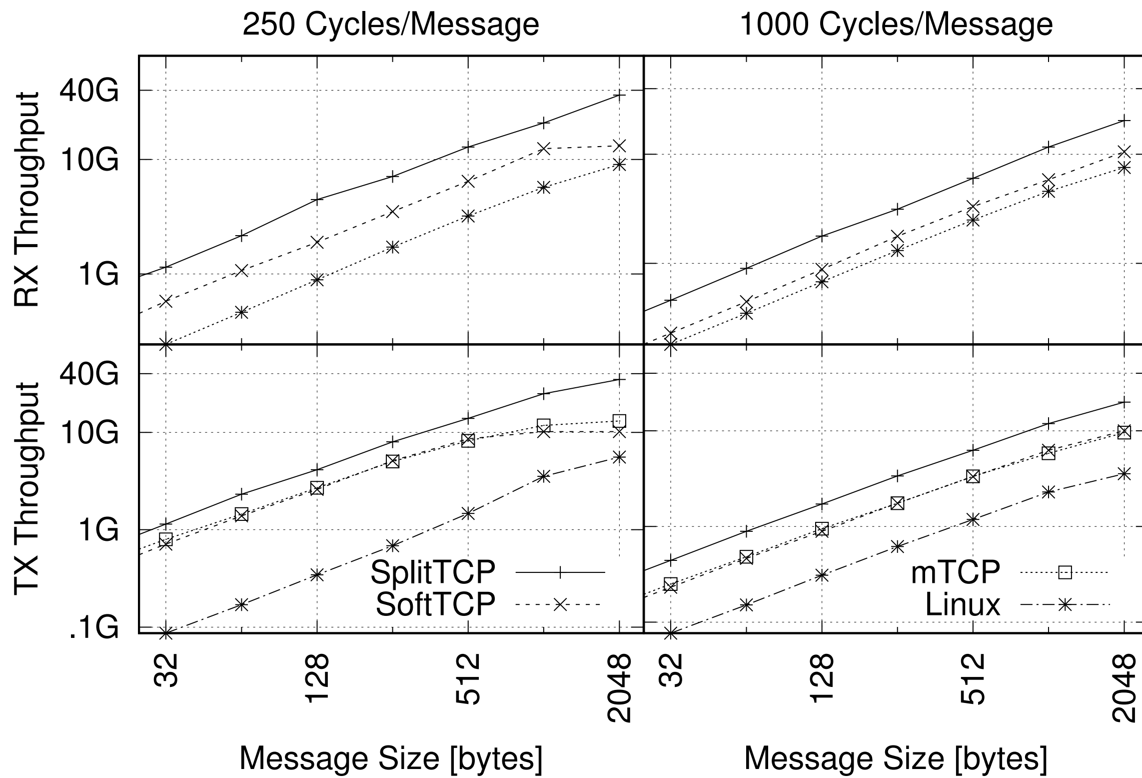


Figure 6.2: Pipelined RPC throughput, varying per-RPC delay and size, for a single-threaded server.

using 4 threads each. After each RPC the server waits for an artificial delay of 250 or 1000 cycles to simulate application processing. To break out improvements in receive and transmission overhead, we run separate benchmarks, one where the server only receives RPCs and one where it only sends.

Figure 6.2 shows the results. When receiving small ($\leq$ 64B) RPCs, FlexTCP provides up to 5.5$\times$ better throughput than Linux. FlexTCP's improvement reduces to 4$\times$ as RPCs become larger. FlexTCP reaches 40G line-rate with 2KB RPCs for 250 cycles of processing while Linux barely reaches 10G. For 1000 cycles of processing, no stack achieves line-rate and FlexTCP provides a steady throughput improvement around 2.7$\times$ regardless of RPC size. SoftTCP's efficiency is between Linux and FlexTCP, which is expected given that it

requires an additional host core to run the stack. mTCP locks up in this experiment.

When sending small and moderate (≤ 256B) RPCs at 250 cycles processing time, FlexTCP provides up to 13.7× Linux and 1.6× mTCP efficiency. For large (2KB) RPCs, FlexTCP's advantage declines to 6.3× Linux, but improves to 2.7× mTCP. mTCP reaches scalability limitations beyond 512B RPCs, while Linux catches up as memory copying costs start to dominate. FlexTCP again achieves 40G line-rate at 2KB RPC size, while Linux and mTCP do not reach beyond 10G. This shows that simplifications in common-case send processing, such as removing intermediate send queueing, can make a big difference.

This difference again diminishes as application-level processing grows to 1000 cycles. In this case, FlexTCP provides a steady improvement of up to 5.9× Linux, regardless of RPC size. Compared to mTCP, FlexTCP provides up to 2× improvement. SoftTCP performs comparably to mTCP in both transmit cases, but does provide protection.

We conclude that FlexTCP indeed provides better RPC latency and throughput when compared to both state-of-the-art in-kernel and kernel bypass TCP stack solutions. Further, SoftTCP provides throughput on par with and better latency than kernel bypass stacks while retaining traditional OS safety guarantees. Thus we improve performance and efficiency of all networked data center applications relying on RPCs over TCP.

6.1.2 Packet Loss

Even in a data center environment, minimal (≤1%) packet loss can occur due to congestion and transmission errors. FlexTCP uses a simplified recovery mechanism and we are interested how packet loss affects FlexTCP throughput in comparison to Linux. We quantify this effect in an experiment measuring throughput of 100 flows over a single link between two machines under different rates of induced packet loss between 0.1% and 5%. We compare FlexTCP with receiver out-of-order processing and without it (simple

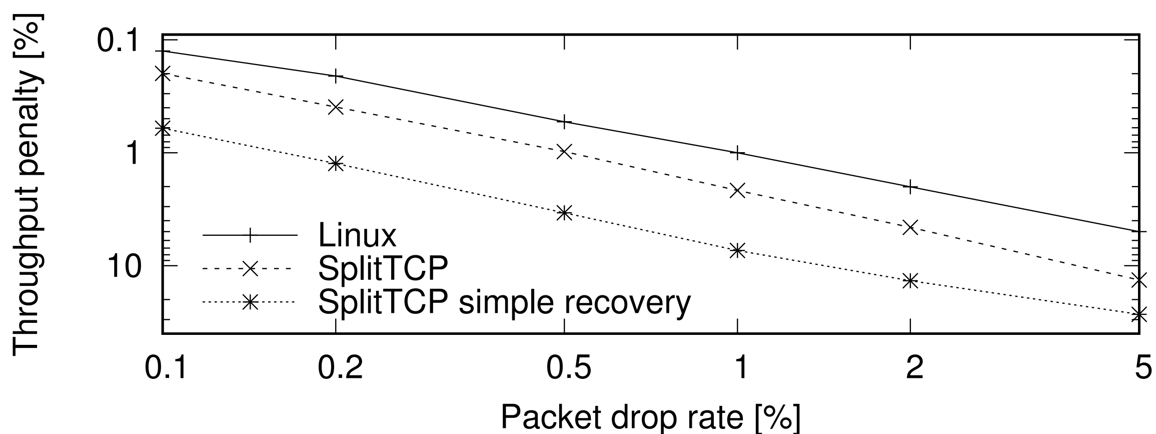


Figure 6.3: Throughput penalty with varying packet loss rate.

go-back-N).

Figure 6.3 shows the penalty relative to the throughput achieved without loss. We can see that FlexTCP throughput is minimally affected (up to 1.5%) for loss rates up to 1%. For a loss rate of 5%, FlexTCP incurs a throughput penalty of 13%. Overall, FlexTCP's penalty is about 2× that of Linux. Linux keeps all received out-of-order segments and also issues selective acknowledgements, allowing it to recover more quickly. FlexTCP only keeps one continuous interval of out-of-order data, requiring the sender to resend more in some cases. Without receiver out-of-order processing, the penalty increases by a factor of 3. We conclude that limited out-of-order processing has a benefit, but full out-of-order processing has minimal impact for the loss rates common in data centers.

6.1.3 Key-Value Store

Key-value stores strongly rely on RPCs. Due to the high TCP processing overhead, some cloud operators use UDP for reads and use TCP only for writing. In this section, we demonstrate that FlexTCP is fast enough to be used for both reading and writing, simplifying application design. To do so, we evaluate an optimized key-value store modeled

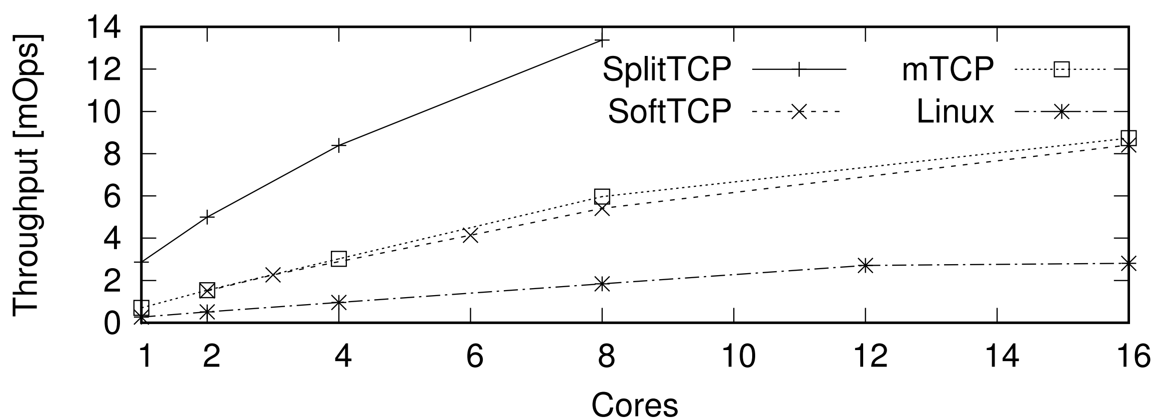


Figure 6.4: Key-value store throughput scalability. Error bars show min/max over 5 runs.

Latency [μs]	**Median**	**90th**	**99th**	**99.9th**
Linux	124	153	200	275
mTCP	28	33	40	71
FlexTCP / SoftTCP	16	18	21	37

Table 6.1: Key-value store request latency in microseconds.

after memcached [87]. We send it requests at a constant rate using a tool similar to the popular memslap benchmark. The workload consists of 100,000 key-value pairs of 32 byte keys and 64 byte values, with a skewed access distribution (zipf, s = 0.9). The workload contains 90% GET requests and 10% SET requests. Throughput was measured over 2 minutes after 1 minute of warm-up.

Throughput Scalability To conduct throughput benchmarks we run 4 clients, each using 4 cores and each core establishing 4 concurrent connections, all directed to the server. We run the benchmark, varying the number of server application cores available. Figure 6.4 shows the result, counting all host cores in use (except NIC emulation cores). We can see that FlexTCP outperforms Linux and mTCP in total throughput by up to 10.6× and 4.1×, respectively. We run out of emulation cores after 8 application cores and thus

do not report FlexTCP scalability beyond this number. SoftTCP and mTCP perform comparably at 3× Linux, but SoftTCP provides protection.

Latency We also conduct single-core latency experiments under 15% bandwidth utilization, so that queues do not build excessively. Table 6.1 show the result. We can see that FlexTCP outperforms both Linux and mTCP by a median 7.8× and 1.8×, respectively. FlexTCP attains even better tail latency versus Linux, but an equivalent difference in tail latency versus mTCP when compared to the median.

We conclude that FlexTCP can greatly improve the performance of RPC-based client-server applications, such as key-value stores. It exceeds state-of-the-art network stacks in both latency and throughput by a comfortable margin, both in median and the tail. As such, FlexTCP can simplify the design of RPC-based applications by allowing them to rely on TCP instead of application-level solutions. SoftTCP again provides comparable throughput and better latency than kernel bypass.

6.1.4 Real-time Analytics

Next, we evaluate the performance of a TCP version of FlexStorm from section 5.2. Figure 6.5 and Table 6.2 show average achievable throughput and latency at peak load on this workload. Throughput is measured over a runtime of 20 seconds, shown raw and per core over the entire deployment. Per-tuple latency is broken down into time spent in processing, and in input and output queues, as measured at user-level, within FlexStorm. We deploy FlexStorm on 3 machines of our client cluster. We evenly distribute workers over the machines to balance the load.

Linux Performance Overhead introduced by the Linux kernel network stack limits FlexStorm performance. Even though per-tuple processing time is short, tuples spend several

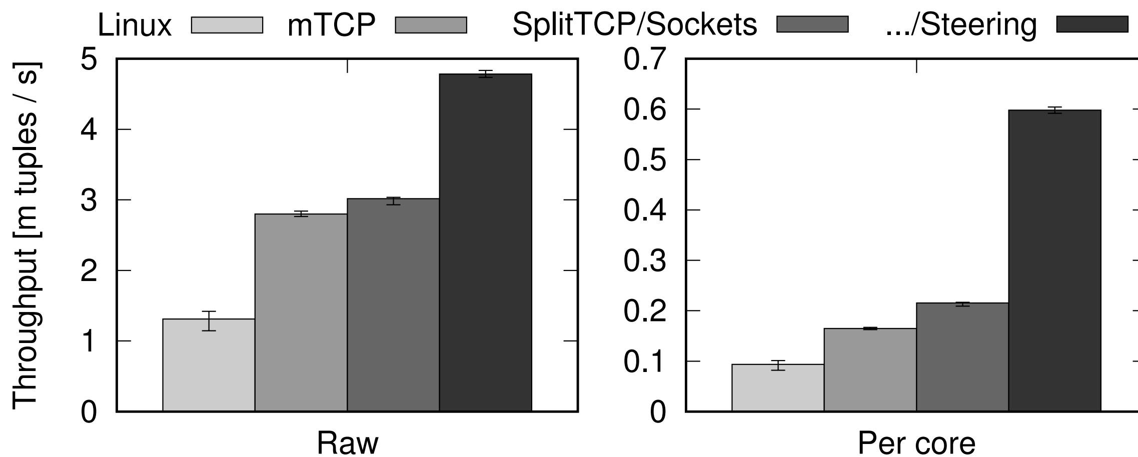


Figure 6.5: Average throughput on various FlexStorm configurations. Error bars show min/max over 20 runs.

milliseconds in queues after reception and before emission. Queueing before emission is due to batching in the multiplexing thread, which batches up to 10 milliseconds of tuples before emission. Input queueing is minimal in FlexStorm as it is past the bottleneck of the Linux kernel and thus packets are queued at a lower rate than they are removed.

mTCP Performance Running all FlexStorm nodes on mTCP yields a 2.1× raw throughput improvement versus Linux, while utilizing an additional core per node to execute the mTCP user-level network stack. The per-core throughput improvement is thus lower, 1.8×. We could not run mTCP threads on application cores, as mTCP relies on the NIC's symmetric RSS hash to distribute packets to isolated per-thread stacks for scalability. This does not work for asymmetric applications, like FlexStorm, where the sets of receiving and sending threads are disjoint. The bottleneck is now the FlexStorm multiplexer thread. Input queuing delay has increased dramatically, while output queueing delay decreased only slightly. This is primarily because mTCP collects packets into large batches to minimize context switches among threads. Overall, tuple processing latency has decreased

	Input	Processing	Output	**Total**
Linux	6.96 μs	0.37 μs	20 ms	**20 ms**
mTCP	4 ms	0.33 μs	14 ms	**18 ms**
FlexTCP	7.47 μs	0.36 μs	8 ms	**8 ms**
Steering	–	0.30 μs	–	**0.30 μs**

Table 6.2: Average FlexStorm tuple processing time.

only 10% versus Linux due to the much higher amount of batching in mTCP.

FlexTCP Performance Running all FlexStorm nodes on FlexTCP yields an 8% raw throughput improvement versus mTCP, but eliminates the dedicated core for the network stack. The per-core throughput improvement is thus 26%. The improvement is only small as the bottleneck remains the multiplexer thread. Overall, tuple processing latency has decreased 56% versus mTCP. This is because FlexTCP does not require any batching to achieve its performance.

Object Steering Enhancing FlexStorm on FlexTCP with object steering support yields a 1.6$\times$ raw throughput improvement versus the non-steering version. Object steering has eliminated the multiplexer and demultiplexer threads on each node. Due to the elimination of their queues, tuple processing latency has decreased by 4 orders of magnitude versus not steering, while per-core throughput has improved by 2.8$\times$. The busiest workers in the system now operate at 90% CPU utilization. Throughput is 13% lower than the DCCP-based version of FlexStorm. The difference is due to TCP's longer protocol header and congestion control mechanism (DCCP only supports flow control). To support object steering, we had to modify the processing loop of FlexStorm's worker threads to use FlexTCP's object API and eliminate the (de-)multiplexing threads. This entailed changing roughly 20 lines of code (LOC) and removing hundreds of LOC.

Moving application-level packet (de-)multiplexing functionality into the NIC eliminates the need for multiplexing or network stack threads, which can grow large under high line-rates. It also yields performance benefits, while reducing the amount of time that tuples are held in queues. This provides the opportunity for tighter real-time processing guarantees under higher workloads using the same equipment.

6.1.5 PCIe NIC Performance

To validate that my emulation-based results are not negatively impacted by offloading to a NIC attached to the PCIe bus, a collaborator evaluated the throughput attained by our FlexTCP implementation running on an Agilio-CX 40G NIC.

Our testbed in this case is a simple setup of two Xeon E5-2680 v3 servers at 2.5 GHz, each equipped with an Agilio-CX NIC and connected back-to-back (without a switch). We run the FlexTCP stack on each NIC to conduct a simple TCP throughput benchmark between the servers to measure maximum attainable throughput under FlexTCP execution and compare to the basic NIC without FlexTCP. We send packets from one server to the other for 15 seconds and measure average packet throughput at the sink. We repeat the experiment, varying the packet size.

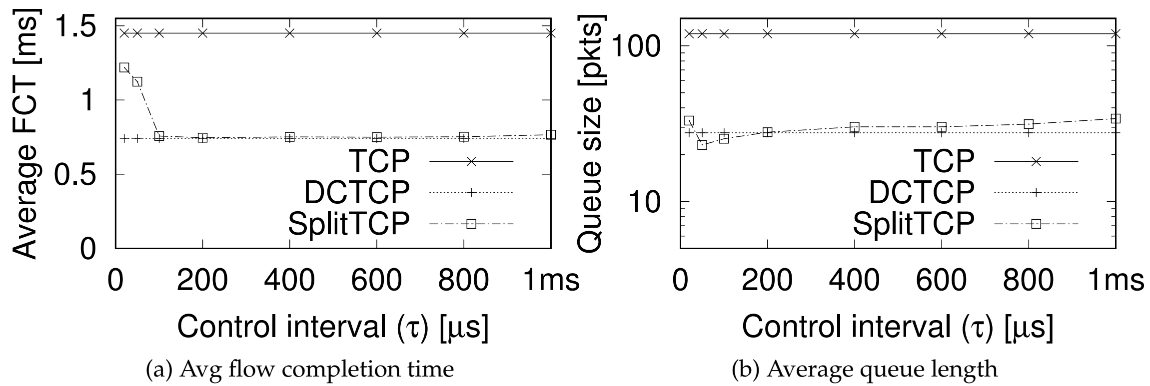
Attainable throughput of our FlexTCP prototype is similar to the emulation-derived results presented in earlier sections and attains line-rate starting at 512 byte packet size. FlexTCP throughput penalty versus the basic NIC is minimal, but increases slightly with smaller packets, until it approaches 8.8% with 256 byte sized packets. The smallest packet size in our Agilio-CX prototype is 174 bytes due to packet header padding. We also validated whether 102 bytes of per-flow state would indeed support a large number of active flows. Using DRAM and an SRAM cache, our NIC is able to support more than 5 million flows—orders of magnitude more than required.

We conclude that our emulation adequately represents the performance of a real NIC prototype. We attribute the moderate slow down of our FlexTCP prototype versus the basic NIC to the early stage of our implementation. The overall slow-down at 256 byte sized packets is due to the architecture of the NIC. We expect it to vanish with an RMT hardware implementation.

6.1.6 Congestion Control

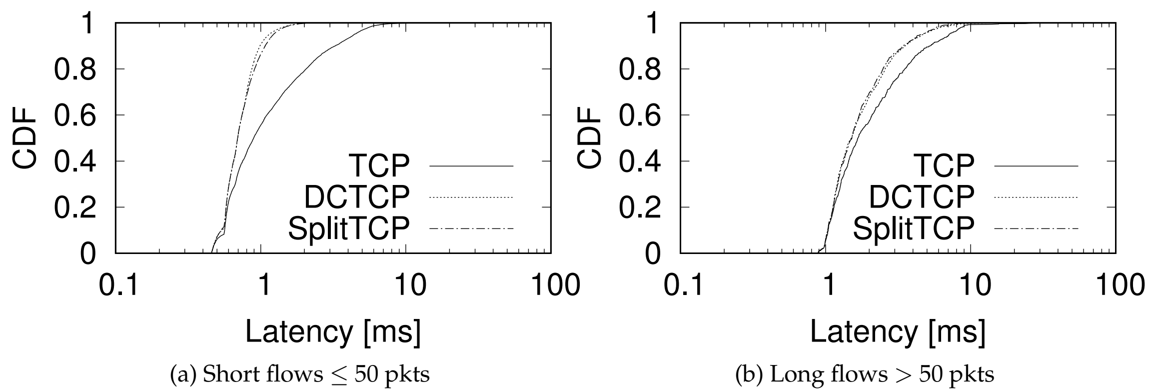
We implemented DCTCP congestion control in FlexTCP with the key difference that transmission is rate based, with rates updated periodically for all flows by the kernel at a fixed pre-defined control interval τ. We investigate the impact of τ on congestion behavior via ns-3 simulations, comparing to vanilla DCTCP. First, we simulate a single 10Gbps link with an RTT of 100µs at 75% utilization with Pareto-distributed flow sizes and varying τ. Next, we simulate a large cluster of 2560 servers and a total of 112 switches that are configured in a 3-level FatTree topology with an oversubscription ratio of 1:4. All servers follow and on-off traffic pattern, sending flows to a random server in the data center at a rate such that the core link utilization is approximately 30%. Finally, we investigate congestion fairness experimentally with $\tau = 2 \times \mathrm{RTT}$ (as measured for each flow) under incast.

Single Link Figure 6.6 shows average flow completion time (FCT) and average queue size with varying τ for the single 10Gbps link. The average FCT for FlexTCP is very similar to that of DCTCP when τ is greater than the RTT. However, if τ is set too low, frequent fluctuations in congestion window cause slow convergence and long completion times. The average queue length is very similar to that of DCTCP and grows, but slowly, as τ increases beyond the RTT, due to delayed congestion window updates.



(a) Avg flow completion time (b) Average queue length

Figure 6.6: Simulation of a single 10Gbps link.



(a) Short flows ≤ 50 pkts (b) Long flows > 50 pkts

Figure 6.7: Flow completion times for large cluster simulation.

Large Cluster Figure 6.7 shows the average flow completion times for short and long flow sizes in the large cluster simulation with the control interval τ set to 100µs. The performance of FlexTCP is similar to that of DCTCP in both cases. 100µs is a reasonable amount of time for the kernel to update congestion windows for thousands of flows. Even with larger values of τ, queue size is only minimally affected and FCTs stay approximately identical. We thus conclude that our out-of-band approach works to provide DCTCP-compatible congestion behavior.

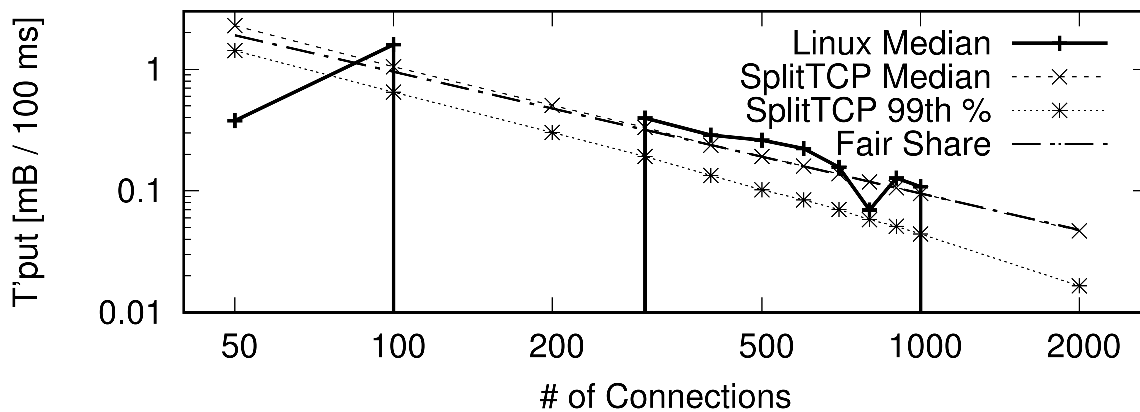


Figure 6.8: Distribution of connection rates under incast.

Tail-Latency Under Incast To evaluate performance under congestion, we measure tail latency under incast with 4 machines sending to a single receiver (operating at line rate) with different numbers of connections. We record the number of bytes received on each connection every 100ms on the receiver over the period of a minute, discarding a warmup 20 seconds. Figure 6.8 shows the median (and 99th percentile) throughput over the measured intervals and connections on Linux (using DCTCP) and FlexTCP. For FlexTCP, the tail falls within $1.6\times$ and $2.8\times$ of the median, while the median is close to each connection's fair share. Linux median (and tail—not shown) behavior fluctuates widely, showing significant starvation of flows in some cases.

Linux fairness is hurt in three interacting ways: (1) Linux window based congestion control creates bursts when windows abruptly widen and contract under congestion. (2) Window-based congestion control limits the control granularity for low-rtt links. (3) The Linux TCP stack architecture requires many shared queues that can overflow when flows are bursty, resulting in dropped packets without regard to fairness. Rate-based packet scheduling and per-flow queueing in FlexTCP smoothes bursts and eliminates unfair packet drops at end hosts.

6.2 Application Co-Design

Next I present my measurement and evaluation results for my three application co-design case studies.

6.2.1 Methodology

To quantify the benefits of integrated processing, I leverage a combination of hardware and software techniques that I implement within the test cluster. Whenever possible, I re-use existing hardware functionality to realize FlexNIC functionality. When this is impossible, I emulate the missing functionality in software on a number of dedicated processor cores. This limits the performance of the emulation to slower link speeds than would be possible with a hardware implementation of FlexNIC and thus favors the baseline in my comparison.

Hardware Features Re-used We make use of the X520's receive-side scaling (RSS) capabilities to carry out fast, customizable steering and hashing in hardware. RSS in commodity NICs operates on a small number of fixed packet header fields, such as IP and TCP/UDP source and destination addresses/ports, and are thus not customizable. We attain limited customization capability by configuring RSS to operate on IPv6 addresses, which yields the largest contiguous packet area to operate upon—32 bytes—and then moving the relevant data into these fields on the sender. This is sufficient for our experiments, as MAC addresses and port numbers are enough for routing within our simple network. RSS computes a 32-bit hash on the 32 byte field, which it also writes to the receive descriptor for software to read. It then uses the hash as an index into a 128-entry redirection table that determines the destination queue for an incoming packet.

Software Implementation We implement other needed functionality in the software FlexNIC emulator. For these experiments our emulator implements flexible demultiplexing, congestion control, and a customizable DMA interface to the host. To do this, the emulator uses dedicated host cores (2 send and 2 receive cores were sufficient to handle 10Gb/s) and a shared memory interface to system software that mimics the hardware interface. For performance, our emulator makes use of batching, pipelining, and lock-free queueing. Batching and pipelining work as described in [42]. Lock-free queueing is used to allow scalable access to the emulated host descriptor queue from multiple emulator threads by atomically reserving queue positions via a compare and swap operation. We try to ensure that our emulation adequately approximates DMA interface overheads and NIC M+A parallelism, but our approach may be optimistic in modelling PCI round trips, as these cannot be emulated easily using CPU cache coherence interactions.

Baseline In order to provide an adequate comparison, we also run all software on top of the high-performance Extaris user-level network stack [104] and make this our baseline. Extaris runs minimal code to handle packets and is a zero-copy and scalable network stack. This eliminates the overheads inherent in a kernel-level network stack and allows us to focus on the improvements that are due to integrated processing.

Testbed Cluster Our evaluation cluster contains six machines consisting of 6-core Intel Xeon E5-2430 (Sandy Bridge) systems at 2.2 GHz with 18 MB total cache space. Unless otherwise mentioned, hyperthreading is enabled, yielding 12 hyperthreads per machine. All systems have an Intel X520 (82599-based) dual-port 10 Gb Ethernet adapter, and we have both ports connected to the same 10 Gb Dell PowerConnect 8024F Ethernet switch. The machines run Ubuntu Linux 14.04.

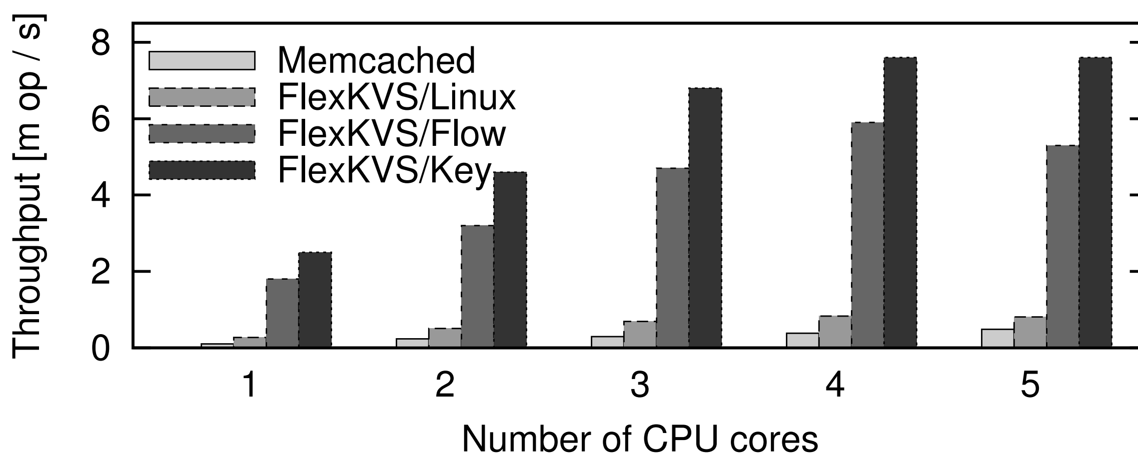


Figure 6.9: FlexKVS throughput scalability with flow-based and key-based steering. Results for Memcached and FlexKVS on Linux are also provided.

6.2.2 Key-Value Store

We evaluate different levels of integration between FlexKVS and the NIC. The baseline runs on the Extaris network stack using UDP and RSS, similar to the Memcached configuration in Arrakis [104]. A more integrated version uses FlexNIC to perform the hash calculation and steer requests based on their key, and the third version adds the FlexNIC custom DMA interface. FlexKVS performance is reduced by hyperthreading and so we disable it for these experiments, leaving 6 cores per machine.

Key-based Steering We compare FlexKVS throughput with flow-based steering against key-based steering using FlexNIC. We use three machines for this experiment, one server for running FlexKVS and two client machines to generate load. One NIC port of each client machine is used, and the server is connected with both ports using link aggregation, yielding a 20 Gb/s link. The workload consists of 100,000 key-value pairs of 32 byte keys and 64 byte values, with a skewed access distribution (zipf, $s = 0.9$). The workload contains 90% GET requests and 10% SET requests. Throughput was measured over 2

	Steering		
	Flow	Key	DMA
Median	1110	690	440
90th Percentile	1400	1070	680

Table 6.3: FlexKVS request processing time rounded to 10 cycles.

minutes after 1 minute of warm-up.

Figure 6.9 shows the average attained throughput for key- and flow-based steering over an increasing number of server cores. In the case of one core, the performance improvement in key-based steering is due to the computation of the hash function on the NIC. As the number of cores increases, lock contention and cache coherence traffic cause increasing overhead for flow-based steering. Key-based steering avoids these scalability bottlenecks and offers 30-45% better throughput. Note that the throughput for 4+ cores with key-based steering is limited by PCIe bus bandwidth, as the workload is issuing a large number of small bus transactions. We thus stop the experiment after 5 cores. Modifying the NIC driver to use batching increases throughput to 13 million operations per second.

Specialized DMA Interface The second experiment measures the server-side request processing latency. Three different configurations are compared: flow-based steering, key-based steering, and key-based steering with the specialized DMA interface described above. We measure time spent from the point an incoming packet is seen by the network stack to the point the corresponding response is inserted into the NIC descriptor queue.

Table 6.3 shows the median and 90th percentile of the number of cycles per request measured over 100 million requests. Key-based steering reduces the number of CPU cycles by 38% over the baseline, matching the throughput results presented above. The

custom DMA interface reduces this number by another 36%, leading to a cumulative reduction of 60%. These performance benefits can be attributed to three factors: 1) with FlexNIC limited protocol processing needs to be performed on packets, 2) receive buffer management is not required, and 3) log-appends for SET requests are executed by FlexNIC.

We conclude that flexible hashing and demultiplexing combined with the FlexNIC DMA engine improve both latency and throughput considerably for key-value stores. FlexNIC can efficiently carry out packet processing, buffer management, and log data structure management without additional work on server CPUs.

6.2.3 Storm

We evaluate the performance of Storm and various FlexStorm configurations on the top-n user topology. Our input workload is a stream of 476 million Twitter tweets collected between June–Dec 2009 [75]. Figure 6.10 and Table 6.4 show average achievable throughput and latency at peak load on this workload. Throughput is measured in tuples processed per second over a runtime of 20 seconds. Latency is also measured per tuple and is broken down into time spent in processing, and in input and output queues, as measured at user-level, within FlexStorm. For comparison, data center network packet delay is typically on the order of tens to hundreds of microseconds.

We first compare the performance of FlexStorm to that of Apache Storm running on Linux. We tune Apache Storm for high performance: we use “at most once” processing, disable all logging and debugging, and configure the optimum amount of worker threads (equal to the number of hyperthreads minus two multiplexing threads). We deploy both systems in an identical fashion on 3 machines of our evaluation cluster. In this configuration Apache Storm decides to run 4 replicas each of spouts, counters and intermediate

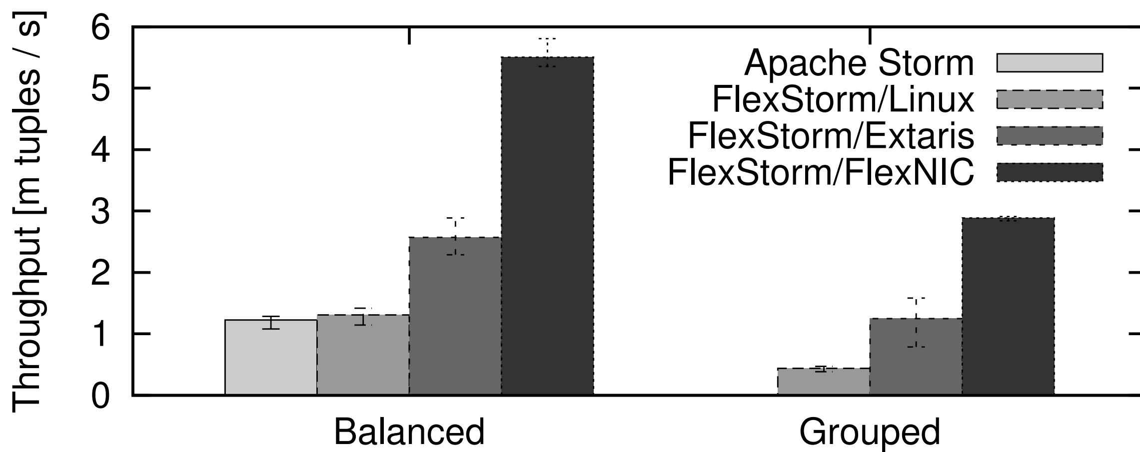


Figure 6.10: Average top-n tweeter throughput on various Storm configurations. Error bars show min/max over 20 runs.

rankers. Storm distributes replicas evenly over the existing hyperthreads and we follow this distribution in FlexStorm (Balanced configuration). By spreading the workload evenly over all machines, the amount of tuples that need to be processed at each machine is reduced, relieving the demultiplexing thread somewhat. To show the maximum attainable benefit with FlexNIC, we also run FlexStorm configurations where all counting workers are executing on the same machine (Grouped configuration). The counting workers have to sustain the highest amount of tuples and thus exert the highest load on the system.

Without FlexNIC, the performance of Storm and FlexStorm are roughly equivalent. There is a slight improvement in FlexStorm due to its simplicity. Both systems are limited by Linux kernel network stack performance. Even though per-tuple processing time in FlexStorm is short, tuples spend several milliseconds in queues after reception and before emission. Queueing before emission is due to batching in the multiplexing thread, which is configured to batch up to 10 milliseconds of tuples before emission in FlexStorm (Apache Storm uses up to 500 milliseconds). Input queueing is minimal in FlexStorm as

	Input	Processing	Output	**Total**
Linux	6.68 μs	0.6 μs	12 ms	**12 ms**
Extaris	4 ms	0.8 μs	6 ms	**10 ms**
FlexNIC	–	0.8 μs	6 ms	**6 ms**

Table 6.4: Average FlexStorm tuple processing time.

it is past the bottleneck of the Linux kernel and thus packets are queued at a lower rate than they are removed from the queue. FlexStorm performance is degraded to 34% when grouping all counting workers, as all tuples now go through a single bottleneck kernel network stack, as opposed to three.

Running all FlexStorm nodes on the Extaris network stack yields a 2$\times$ (Balanced) throughput improvement. Input queuing delay has increased as tuples are queued at a higher rate. The increase is offset by a decrease in output queueing delay, as packets can be sent faster due to the high-performance network stack. Overall, tuple processing latency has decreased 16% versus Linux. The grouped configuration attains a speedup of 2.84$\times$ versus the equivalent Linux configuration. The bottleneck in both cases is the demultiplexer thread.

Running all FlexStorm nodes on FlexNIC yields a 2.14$\times$ (Balanced) performance improvement versus the Extaris version. Using FlexNIC has eliminated the input queue and latency has decreased by 40% versus Extaris. The grouped configuration attains a speedup of 2.31$\times$. We are now limited by the line-rate of the Ethernet network card.

I conclude that moving application-level packet demultiplexing functionality into the NIC yields performance benefits, while reducing the amount of time that tuples are held in queues. This provides the opportunity for tighter real-time processing guarantees under higher workloads using the same equipment. The additional requirement for receive-side flow control does not pose an implementation problem to FlexNIC. Finally, demulti-

	Hashing			FlexNIC		
	Min	Avg	Max	Min	Avg	Max
Kpps	103.0	103.7	104.7	166.4	167.3	167.9
Mbps	435.9	439.8	444.6	710.4	715.6	718.6
Accesses	546.0	553.3	559.3	237.0	241.4	251.0
Misses	26.7	26.7	26.7	19.0	19.2	19.2

Table 6.5: Snort throughput and L3 cache behavior over 10 runs.

plexing in the NIC is more efficient. It eliminates the need for additional demultiplexing threads, which can grow large under high line-rates.

I can use these results to predict what would happen at higher line-rates. The performance difference of roughly 2× between FlexNIC and a fast software implementation shows that we would require at least 2 fully utilized hyperthreads to perform demultiplexing for FlexStorm at a line-rate of 10 Gb/s. As line-rate increases, this number increases proportionally, taking away threads that could otherwise be used to perform more analytics.

6.2.4 Snort

I evaluate the FlexNIC-based packet steering by comparing it to basic hash-based steering. For my experiment, I use a 24 GB pcap trace of the ICTF 2010 competition [131] that is replayed at 1 Gbps. To obtain the flow group partition I use a prefix of 1/50th of the trace, and then use the rest to evaluate performance. For this experiment, I run 4 Snort instances on 4 cores on one socket of a two-socket 12-core Intel Xeon L5640 (Westmere) system at 2.2 GHz with 14 MB total cache space. The other socket is used to replay the trace via the FlexNIC emulator implementing either configuration.

Table 6.5 shows the throughput and L3 cache behavior for hashing and our improved FlexNIC steering. Throughput, both in terms of packets and bytes processed, increases

by roughly 60%, meaning more of the packets on the network are received and processed without being dropped. This improvement is due to the improved cache locality: the number of cache accesses to the L3 cache per packet is reduced by 56% because they hit in the L2 or L1 cache, and the number of misses in the L3 cache per packet is also reduced by roughly 28%. These performance improvements were achieved without modifying Snort.

I conclude that flexible demultiplexing can improve application performance and cache utilization even without requiring application modification. Application memory access patterns can be analyzed separately from applications and appropriate rules inserted into FlexNIC by the system administrator.

6.3 Discussion

My results show significant performance improvements for TCP processing as well as integrated application processing in the use-cases I have evaluated. For FlexTCP I have also demonstrated compatibility with other TCP peers, improved fairness, and validated performance under packet loss and network congestion.

There are some limitations to my evaluation though, most of them because I rely on software emulation for FlexNIC. There are cases where the emulation limits the available throughput. As such, my evaluation presents a lower bound for performance benefits that a hardware FlexNIC implementation would enable. But potentially there are also performance aspects that the emulation underestimates compared to a full hardware implementation, such as the cost of doorbell writes from the CPU. There are two options for enabling a more realistic end-to-end evaluation. First, instead of using software emulation for FlexNIC a full system simulator, such as gem5 [10], can be used to more accurately model overheads and to remove the dependency on the emulator performance. Second, FlexNIC processing for evaluation could be implemented on a NIC based on a network

processor or FPGA to obtain a full end-to-end performance evaluation (subject to any performance overhead associated with the particular NIC).

My TCP evaluation could be extended to evaluate additional aspects of FlexTCP. I currently do not benchmark the slow-path, e.g. by measuring cost for establishing or terminating connections, or flow-completion time for short-lived connections. It is worth noting though that FlexNIC state updates from the kernel are likely one aspect where the emulator is not faithfully representing overheads. With a hardware emulation these updates would likely require multiple un-cached memory writes and potentially reads through PCIe from the CPU, and thus be more expensive than in my emulator with shared memory. An accurate evaluation here would likely require a full system simulator or a hardware implementation. I expect that FlexTCP can scale to larger numbers of connections than my experiments currently evaluate. In my attempts to evaluate these, I ran into protocol limitations with TCP congestion control that degraded performance for 10s of thousands of connections sharing a 10 Gbps link. This is not an FlexTCP limitation, but a consequence of the higher throughput and lower latency achieved in this setup. Linux and mtcp behave much worse in this regime and end up terminating connections because of timeouts.

Chapter 7

Related Work

Related work falls into three categories: 1) modified or new architectures for software protocol processing, 2) hardware architectures for high-performance and flexible processing, and 3) acceleration of specific applications or application protocols.

7.1 Software Packet Processing

On the software side, previous work has investigated the performance of both in-kernel and kernel bypass systems in parallel.

7.1.1 Kernel Stack Improvements

Numerous changes to the system call interface have been made or proposed to reduce overheads. The `select()` call does not scale to large numbers of connections because the kernel and application both need to traverse a data structure linear in the number of connections. To address this, Linux provides `epoll` [80] while BSD-based operating systems implement `kqueue` [74]. Because I/O related system calls can block unexpectedly

preventing the application from performing work until the call returns, asynchronous interfaces [129, 32] have been added so applications can do other work until they receive a completion notification. System call batching [126] amortizes context switch overheads across multiple system calls. Megapipe [44] introduces a new API centered around asynchronous communication channels between the application and the kernel to reduce context switch overheads, avoid file descriptor management, and enable a scalable kernel network stack implementation. IX [8] is a kernel network stack providing a streamlined API for efficient run-to-completion processing with minimal changes to Linux. I apply these ideas in FlexTCP, for both internal and application-facing interfaces. FlexTCP implements a large subset of the sockets interface offered by modern Linux systems to offer backwards compatibility, including `epoll`. The low-level FlexTCP application interface as well as all internal interfaces between components are all asynchronous queues that support batching, similar to Megapipe. While applications can leverage the low level FlexTCP interface for efficient run-to-completion processing similar to IX, FlexTCP also efficiently supports more general communication patterns.

Other work improves performance through internal modifications to the kernel network stack. Scheduling anomalies [29] occur in network stacks that prioritize incoming network packets over application code handling requests. This can lead to live-lock under load. Affinity-accept [103] and Fastsocket [78] improve multicore scalability of the Linux TCP stack by processing each connection on a single core. However, both mechanisms are limited to using the L4 steering mechanisms offered by current NICs and do not support steering based on TCP payload. Internally, FlexNIC processes packets and other events using run-to-completion, and flexible de-multiplexing provides software with full control over scheduling of processing. I build on the lessons learned from affinity-accept and Fastsocket by leveraging the reconfigurability in FlexNIC for higher layer steering to fully

partition processing for a wider range of applications.

Previous research has also investigated different splits of responsibility for processing between the application and the OS kernel. Netmap [114] provides an interface to efficiently send and receive raw network packets without copies while still guaranteeing memory safety and not requiring hardware support. Netmap leaves all protocol processing to applications and does not support multi-tenant isolation as applications are able to craft arbitrary outgoing packets. Instead of moving processing into the application, AFPA [61] proposes the opposite approach and moves server components directly into the kernel to avoid context switch overheads. AFPA requires that applications are trusted to run in kernel mode with full access to the system. I instead investigate how to remove the OS stack from the critical path, while leaving the OS in full control of policy enforcement.

7.1.2 Kernel Bypass

Another line of research completely avoids kernel stack overheads by bypassing the kernel and providing applications with direct access to NICs. U-Net [137] and Arrakis [104] leverage hardware I/O virtualization to enable safe direct access to network devices by multiple applications. They leverage NIC filters to assign incoming packets to applications based on ATM circuits or IP addresses, and validate these parameters on outgoing packets. Neither system is capable of enforcing policies such congestion control. mTCP [60] is a user-level TCP stack optimized to handle short-lived connections on multiple cores, by statically partitioning the TCP stack to application cores. mTCP does not support higher-layer steering, or applications that require handling of connections on specific cores. Chronos [65] aims to minimize application latency by delivering packets directly to the correct application thread based on request data. However, Chronos does

not propose a specific hardware mechanism for de-multiplexing and emulates hardware de-multiplexing in software. In this dissertation, I start with kernel bypass and aim to understand how to provide hardware NIC support so it can also provide the isolation and security properties of in-kernel processing. I adopt the lessons learned from mTCP and Chronos and partition processing to cores, but I propose a flexible hardware architecture to support a wide range of steering mechanisms.

Kernel bypass offers the opportunity to specialize software processing to application requirements, as each application implements its own processing. Sandstorm [83] demonstrates that a TCP stack specialized for 1and tightly integrated into individual applications significantly reduces processing overheads. Sandstorm removes intermediate queuing, allows applications to prepare and cache data in driver send buffers, and uses synchronous run to completion processing. With FlexNIC application acceleration, I leverage similar techniques for UDP workloads. The FlexTCP low-level interface also lends itself to tight application integration, but I leave it as future work to study the attainable benefits for applications.

7.2 Hardware Packet Processing

Previous work has investigated changes to existing NIC architectures, proposed novel hardware architectures, and evaluated the suitability of existing processor architectures for network processing.

7.2.1 NIC Improvements

Research has studied how to how to achieve memory safety and multiplexing for safe kernel bypass [107, 137, 30]. In particular, these projects have proposed NIC mechanisms for validating DMA addresses in the NIC, demultiplexing incoming packets based on

standardized header fields, and validating outgoing packets to prevent spoofing. FlexNIC builds on these mechanisms and provides similar mechanisms that are, however, not tied to fixed protocols. SENIC [109] shows how to scale NIC hardware rate limiting to large numbers of connections with a combination of a hardware scheduler and OS software packet classification. I adapt SENIC's hardware scheduler component into FlexNIC's queue manager and leverage the RMT pipeline to flexibly schedule packets fully in hardware. Other work demonstrates that careful NIC-software interface design can reduce latency and improve bus utilization by minimizing the number of PCIe transitions [35, 11] for sending and receiving small packets. I can implement these and other DMA optimizations in FlexNIC by leveraging flexible DMA to develop application specific communication strategies.

TCP Offload Engines [24, 21] and remote direct memory access (RDMA) [111] go a step further, entirely removing protocol processing from the CPU. Scale-out NUMA [99] extends the RDMA approach by integrating a remote memory access controller that automatically translates memory accesses into remote memory operations. Portals [6] is similar to RDMA, but adds a set of offloadable memory and packet send operations triggered upon matching packet arrival. All of these approaches implement a fixed protocol in hardware. With FlexNIC, I instead show that it is not necessary to implement a fixed protocol. I can support a range of protocols with message passing, shared memory, or hybrid semantics all on one flexible hardware model.

7.2.2 Programmable Network Hardware

In the wake of the software-defined networking [85] trend, a rich set of customizable switch data planes have been proposed. For example, the P4 programming language proposal [14] allows users rich switching control based on arbitrary packet fields, in-

dependent of underlying switch hardware. There is a natural trade-off between programmability, performance, and cost. Reconfigurable match tables (RMT) [15] aim to provide an abstraction for flexible and protocol-independent packet processing that can be implemented for growing data center link speeds at an acceptable cost. Data center switches based on RMTs are commercially available and support aggregate link rates of up to 6.5 Tbps [19, 5]. In this dissertation, I leverage RMT pipelines as a building block for a complete NIC architecture that tightly integrates RMT with novel mechanisms for flexible DMA and packet scheduling.

7.2.3 Cluster Message Passing

High performance computing applications are critically dependent on efficient communication. The SHRIMP multicomputer virtual NIC [13] offers applications direct access to hardware message passing through a memory mapped interface. The Stanford FLASH multiprocessor [46] supports both message passing and cache-coherent distributed shared memory with minimal hardware and software overhead. FLASH integrates MAGIC, a custom fully programmable NIC consisting of three CPU cores arranged as a pipeline, between the CPU and memory, allowing software to snoop and interpose memory accesses translating them to messages and vice-versa. Cluster interconnects are optimized for maximum performance with cost as a secondary factor. They typically assume reliable messaging support in the network. I target the more general case of lossy and potentially congested Ethernet data center networks.

7.2.4 GPU Packet Processing

In addition to optimizing software and hardware, previous work has examined using general purpose graphics processing units (GPGPUs) to accelerate packet processing. GPUs

are attractive accelerators because of their high degree of parallelism and large memory bandwidth. Initial work targeted offloading network packet routing to GPUs [43, 128, 140], SSLShader accelerates SSL processing [59]. GASPP enables more complex stateful packet processing on GPUs [136]. Rhythm accelerates PHP web applications using GPUs [1]. GPUnet [68] removes the need for packets to be sent and received through the CPU by allowing the NIC to directly transfer packets to and from the These GPU performance improvements are often due to better memory latency hiding, and similar performance can be achieved on CPUs with careful optimization [64]. My approach does not leverage GPUs for packet processing. CPUs now have instructions for encryption so the parallelism offered by GPUs is less necessary. Instead I rely on an RMT-based NIC and the CPU, thereby leaving the GPU available for more computationally intensive tasks. While I do not evaluate GPU integration, I expect future work could leverage FlexNIC to efficiently communicate directly with GPUs for applications, such as machine learning or encoding, that process data on GPUs.

7.3 Application Layer Protocols

The work presented so far focuses on improving performance for general packet handling and standard network protocols such as the Ethernet, IP, and TCP stack. There is also a line of work that aims to improve application protocol processing performance.

7.3.1 High-performance Applications

Several researchers have studied how to improve the performance of specific applications. A common target are key-value stores. Three examples are HERD [62], Pilaf [89], and MICA [77], which use NIC hardware features to improve performance. HERD and Pilaf use RDMA operations to offload protocol processing to hardware and to exert fine-

grained control over how requests are processed in software. MICA uses regular Ethernet NICs with kernel bypass and tightly integrates all protocol processing into the application, while leveraging NIC packet steering for scalability. All three systems require require client modifications to be able to make use of these features. FaRM [28] generalizes this approach to other distributed systems. FlexNIC can implement the same optimizations without binding to a specific hardware implementation of a protocol (such as RDMA). But I also show that the flexibility is useful — e.g. to allow server-side optimizations that are transparent to clients.

7.3.2 NIC-Application Co-design

Some research has examined offloading entire applications to programmable NICs, such as key-value storage [20, 12] and map-reduce functionality [120]. KV-Direct [76] presents a key-value store implementation on an FPGA-based NIC and also implements almost all key-value store functionality on the NIC. KV-Direct does implement small parts of processing on the CPU, including memory reclamation, and also leverages the CPU to more efficiently use PCIe bandwidth. In contrast, I only offload critical parts of packet processing to a reconfigurable NIC without requiring expensive network processors or FPGAs on the NIC. FlexNIC instead allows for incremental application integration. As I have shown, FlexTCP supports unmodified applications and FlexNIC can accelerate existing applications with varying degrees of modification.

Chapter 8

Conclusion

In this dissertation, I have proposed and evaluated a novel architecture for high performance data center packet processing. This architecture demonstrates that network communication can be implemented efficiently, scalably, and predictably without compromising multi-tenant policy enforcement, protocol flexibility, or cost efficiency.

This dissertation makes the following contributions:

Reconfigurable NIC Hardware Model With FlexNIC (chapter 3), I designed a flexible hardware NIC model for integrated hardware-software packet processing. This NIC model reduces processor overheads for communication, by allowing protocol processing steps, for standard network protocols and application protocols alike, to be offloaded to hardware. The system is flexible enough to support standard protocols and application protocols alike, scalable to thousands of active connections, and efficient because it can keep up with line rate with minimal processor load. At the same time, the operating system can leverage FlexNIC to enforce policies, essential for kernel bypass operation.

High Performance Data Center TCP Stack To demonstrate the power of FlexNIC, I developed FlexTCP (chapter 4), an efficient high performance TCP implementation for multi-tenant data centers. FlexTCP splits up TCP protocol processing functionality between FlexNIC, the application, and the operating system kernel. Applications directly interact with the NIC to send and receive data, while the operating system manages connections and implements congestion control. FlexTCP improves per-core application throughput by up to $10.7\times$ and latency by up to $7.8\times$ compared to Linux in my experiments.

Integrated Application Packet Processing With three case-studies (chapter 5), I demonstrated that FlexNIC is also useful for accelerating application request processing. I show how to improve scalability of a key-value store with application-level packet steering, resulting in up to 45% higher throughput. FlexNIC can streamline request processing by directing DMA to and from application data structures. The result is an additional 60% reduction in the key-value store request processing time. I show how to accelerate a real-time analytics system by $2.3\times$ by offloading (de)multiplexing and flow control to FlexNIC. Finally, I use FlexNIC to speed up an otherwise unmodified intrusion detection system by 60% by adjusting packet steering based on application locality requirements.

8.1 Future Work

I now discuss directions in which my work could be extended in the future. At a high level, these fall into three categories: 1) addressing limitations that are beyond the scope of this dissertation, 2) exploring new opportunities enabled by FlexNIC, and 3) looking forward to a path for adoption and future challenges.

8.1.1 Addressing Limitations

While I demonstrate that FlexNIC satisfies my outlined goals for data center communication, a practical deployment requires additional work in multiple directions, including a hardware design, programming language and OS integration, and remaining protocol limitations.

Hardware FlexNIC Implementation This dissertation proposed and evaluated abstractions for a reconfigurable NIC from a software point of view. My evaluation results have shown these abstractions reduce software overheads and can improve throughput by more than $10\times$. For cloud customers, this improvement in performance and efficiency has the potential to significantly reduce operating costs and to enable new applications, where communication overheads would otherwise be prohibitive. Achieving these performance improvements in practice an efficient and cost-effective implementation of a reconfigurable NIC.

I have not shown that an efficient hardware implementation exists. There is reason to expect it does, based on existing RMT switch designs, and their extension for stateful processing and programmable scheduling, but there are fundamental differences from a switch. Quantitative evaluation of chip area, power consumption, and hardware timing will require a full hardware design.

Programming Language Support I advocate splitting processing across the NIC, the OS kernel, and the application. This poses implementation challenges; changes involving more than one component require time-consuming modifications in multiple places and using multiple interfaces. An additional complication is that FlexNIC exposes a programming model completely different from the host processor. Programming language support may simplify development and improve reliability by enabling program analysis

across components. For the case of split processing, the compiler would be responsible to turn a single implementation into code for the NIC, the OS kernel, and the application, based on programmer annotations and compiler analysis.

OS Support For Application-Specific Processing Introducing application-specific processing from untrusted applications into a shared network stack poses a set of challenges I have not addressed. Previous work [84, 9, 33, 138] has explored options for safely executing code provided by untrusted applications in the kernel, with approaches such as type-safe languages and simplified instruction sets. These approaches guarantee memory safety and ensure termination of handlers provided by applications. As discussed in this dissertation, multi-tenant packet processing requires additional guarantees. Applications also dynamically start up and terminate, requiring the network stack to adapt at run-time. Adapting NIC processing dynamically, without disturbing processing for remaining applications, presents another challenge.

Data Center Scale TCP Data center management, especially for cloud computing, has grown increasingly finer, from hours to deploy on individual machines, to minutes on virtual machines, through seconds with containers, now down to milliseconds with serverless computing [70, 4, 37]. This transition has also increased application density in the data center, in turn, increasing sharing and network load. In this emerging world, individual physical machines will need to handle 10s to 100s of thousands of active network connections. FlexNIC makes it possible to do scalable packet processing and to adapt to changing conditions, but it does not specify what those protocols should do

Data center congestion control, for example, has to operate across a wider spectrum: a single link could be shared and fully utilized by only a handful of connections or the same link could be used by 100s of thousands of connections with bandwidth-delay product

measured in a few hundreds of packets.. Existing congestion control protocols are not designed for and do not behave well over this range of operating conditions. I have observed that classical TCP triggers instability [38] under these conditions in my experimental setup. As new application requirements and advances in data center infrastructure improve performance further, this and other problems with existing network protocols will become more pronounced. A promising avenue is to observe that existing network resource algorithms are constrained by an assumption of computation in the data path handling code; once I relax that assumption with FlexNIC, a broader range of options becomes available. Understanding and addressing these challenges is essential to support continuing data center evolution and with it the whole cloud ecosystem.

High TCP Connection Churn Network communication in data centers exhibits a range of workload patterns. FlexTCP targets long running TCP connections, the dominant workload for optimized data center applications. But applications with other workloads do exist and also require efficient communication. In some cases applications can be modified to use persistent connections to improve efficiency. For many other applications, such as client-facing servers, the workload and protocols are dictated externally. Short-lived TCP connections with high churn are particularly hard to optimize, but are a common web server workload.

Efficiently supporting short lived connections in FlexTCP requires architectural changes. For connections that only exchange a few data segments before teardown, the slow path does most of the work and the overhead for connection hand-off to the fast path cannot be amortized. As connection establishment and teardown becomes the common case, it should be handled by the fast-path. For a software version of the fast-path this appears feasible. But FlexNIC, as proposed, does not support this split of responsibilities, because it would require adding entries to match-action tables on the data path. Adding this sup-

port to FlexNIC is not trivial, because the hash-table insert requires multiple dependent memory accesses. This is further complicated by hash collisions that would have to be resolved, resulting in a variable time operation problematic for a pipeline. One approach to this is only handing only inserts without collisions on the fast path. Only a careful design and analysis can determine if such an architecture could satisfy the outlined goals.

8.1.2 Opportunities with FlexNIC

The protocol flexibility and predictable performance open up new possibilities for data center communication.

Flexible Remote Direct Memory Access The One-sided communication promised by RDMA is a compelling advantage over message passing systems for many applications. However, network requirements incompatible with modern data centers combined with inflexibility limit the applicability of existing commercial implementations. In addition, while memory accesses augmented with simple atomic operations are sufficient for some applications, many data center applications feature complex data structures and concurrency that are not compatible with this programming model.

FlexNIC has the potential to provide a flexible programming model with application-specific one-sided operations on top of a resource management model compatible with modern data centers. While I do not directly demonstrate it in this dissertation, FlexNIC can support one-sided RDMA operations but can also go a step further. For example, FlexNIC could be programmed with application-level operations such as an append to a shared queue that is fully executed in hardware but would require multiple operations with RDMA. Operations could also be designed to be mostly one-sided but transparently to the sender handle corner cases in software, such as a hash table insertion with colli-

sion handling in software. This programming model could then be layered on top of the resource enforcement techniques developed for FlexTCP. The resulting system would provide a larger set of data center applications with one-sided communication in a resource management model compatible with today's data centers.

Network Layer Service Level Agreements Modern cloud offerings are governed by service level agreements that define objectives for service availability and performance, along with compensation in case the service does not meet the targets. For data center networks, operators today are not willing to guarantee objectives beyond availability, because of the unpredictable performance of end hosts and the network alike. Even services fully controlled by operator, such as storage, typically only specify availability objectives and in some cases worst case response times in multiple orders of magnitude above the average.

The predictable end-host processing offered by FlexNIC and FlexTCP provides an opportunity for operators improve end-to-end predictability and offer service level agreements for networks and services with specific and accurate performance objectives. FlexNIC is only part of the answer for predictable network performance. Congestion control protocols, in-network mechanisms, operating system schedulers, and applications all influence predictability as well as overall performance. Improving predictability in the data center will require solutions that target multiple layers of the stack.

8.1.3 Thoughts for the Future

Looking out into the future, the viability of the proposed architecture depends on first gaining adoption and then keeping track with future network speeds.

A Path to Adoption: Streamlined Software TCP Stack In the course of evaluating FlexTCP, I discovered that streamlined TCP processing also improves CPU efficiency and

overall performance for a pure software implementation. Even accounting for the dedicated CPU cores running the FlexNIC emulation, the resulting software system (SoftTCP) improves both CPU efficiency and overall performance relative to Linux. For most experiments SoftTCP provided performance on par with mtcp (the kernel bypass baseline stack), but without compromising multi-tenancy.

In contrast to FlexTCP, SoftTCP does not require new hardware and can be rolled out incrementally. A software implementation based on the same design principles thereby provides a potential first step on the path towards adoption of FlexTCP. Most of the control plane and the application libraries can be prototyped and evaluated for SoftTCP and then later adapted for hardware offload to FlexNIC or another re-configurable NIC.

The current SoftTCP implementation provides a first step towards validating the core approach of a streamlined software implementation. However, extending it into a full system for practical deployments requires significant additions. For one, SoftTCP currently uses a static number of cores for its fast-path. A complete system has to dynamically adapt resource allocation based on workload requirements. SoftTCP also requires application cores to poll queues for packets, wasting processor cycles during periods of inactivity. Instead, a complete implementation should support blocking operation for applications during less communication intensive periods. These and other required extensions require novel contributions and have the potential to improve efficiency for data center communication.

Future Limits of FlexNIC As network bandwidth continues to grow at a dramatic pace, two critical questions are what the fundamental bandwidth limit of FlexNIC is and what compromises are necessary for supporting higher bandwidths. While precise throughput limits will depend on the specific hardware design and silicon implementation, prior work on reconfigurable stateful processing [123] and flexible scheduling [124] has demon-

strated throughput of a billion packets per second ($\approx$ 500 Gbps with 64-byte packets). Thus I expect that a FlexNIC implementation could support line rate with small packets for 400 Gbps Ethernet.

Scaling up bandwidth significantly beyond 400 Gbps will require compromises and architectural changes. Commercial RMT switches today achieve aggregate line rates of up to 6 Tbps, by partitioning ports to multiple parallel pipelines. This comes at the cost of partitioning state with no sharing between groups of ports, a poor fit for the inherently stateful processing proposed in this dissertation. Depending on processing requirements, the impact could be mitigated with a flexible mechanism for partitioning incoming packets to pipelines, but a partitioned architecture fundamentally loses global visibility in the data path.

RMT pipelines only incur per-packet overheads that are independent of the packet size; larger packets achieve proportionally higher bandwidths, up to 12 Tbps for 1 billion 1500 B Ethernet packets per second. The effective achievable bandwidth depends on the workload. An open question is what packet rates and bandwidths individual future data center servers and applications will need.

Bibliography

[1] S. R. Agrawal, V. Pistol, J. Pang, J. Tran, D. Tarjan, and A. R. Lebeck. Rhythm: Harnessing data parallel hardware for server workloads. In *19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2014.

[2] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *2008 ACM SIGCOMM Conference on Data Communication*, SIGCOMM, 2008.

[3] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). In *2010 ACM SIGCOMM Conference on Data Communication*, SIGCOMM, 2010.

[4] Amazon Web Services. AWS Lambda – serverless compute. `https://aws.amazon.com/lambda/`.

[5] Barefoot Networks. Barefoot Tofino. `https://barefootnetworks.com/products/product-brief-tofino/`.

[6] B. W. Barrett, R. Brightwell, S. Hemmert, K. Pedretti, K. Wheeler, K. Underwood, R. Riesen, A. B. Maccabee, and T. Hudson. The Portals 4.0.1 network programming interface. `http://www.cs.sandia.gov/Portals/portals401.pdf`, Apr. 2013.

[7] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: A new OS architecture for scalable multicore systems. In *22nd ACM Symposium on Operating Systems Principles*, SOSP, 2009.

[8] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2014.

[9] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the SPIN operating system. In *15th ACM Symposium on Operating Systems Principles*, SOSP, 1995.

[10] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The Gem5 simulator. *SIGARCH Computer Architecture News*, 39(2):1–7, Aug. 2011.

[11] N. L. Binkert, A. G. Saidi, and S. K. Reinhardt. Integrated network interfaces for high-bandwidth TCP/IP. In *12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2006.

[12] M. Blott, K. Karras, L. Liu, K. A. Vissers, J. Bär, and Z. István. Achieving 10Gbps line-rate key-value stores with FPGAs. In *5th USENIX Workshop on Hot Topics in Cloud Computing*, HotCloud, 2013.

[13] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg. Virtual memory mapped network interface for the SHRIMP multicomputer. In *21st Annual International Symposium on Computer Architecture*, ISCA, 1994.

[14] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Computer Communication Review*, 44(3):87–95, July 2014.

[15] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *2013 ACM SIGCOMM Conference on Data Communication*, SIGCOMM, 2013.

[16] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *7th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2006.

[17] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson. BBR: Congestion-based congestion control. *ACM Queue*, 14(5):50:20–50:53, Oct. 2016.

[18] Cavium Corporation. OCTEON II CN68XX multi-core MIPS64 processors. `http://www.cavium.com/pdfFiles/CN68XX_PB_Rev1.pdf`.

[19] Cavium Corporation. XPliant Ethernet Switch Product Family. `http://www.cavium.com/XPliant-Ethernet-Switch-Product-Family.html`.

[20] S. R. Chalamalasetti, K. Lim, M. Wright, A. AuYoung, P. Ranganathan, and M. Margala. An FPGA Memcached appliance. In *21st ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA, 2013.

[21] Chelsio Communications. TCP offload at 40Gbps. `http://www.chelsio.com/wp-content/uploads/2013/09/TOE-Technical-Brief.pdf`, 2013.

[22] S. Chole, A. Fingerhut, S. Ma, A. Sivaraman, S. Vargaftik, A. Berger, G. Mendelson, M. Alizadeh, S.-T. Chuang, I. Keslassy, A. Orda, and T. Edsall. dRMT: Disaggregated programmable switching. In *2017 ACM SIGCOMM Conference on Data Communication*, SIGCOMM, 2017.

[23] J. Corbet. TCP small queues. `https://lwn.net/Articles/507065/`, July 2012.

[24] A. Currid. TCP offload to the rescue. *ACM Queue*, 2(3), June 2004.

[25] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *6th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2004.

[26] M. Dong, Q. Li, D. Zarchy, P. B. Godfrey, and M. Schapira. PCC: Re-architecting congestion control for consistent high performance. In *12th USENIX Symposium on Networked Systems Design and Implementation*, NSDI, 2015.

[27] DPDK Project. Intel data plane development kit. `http://www.dpdk.org/`.

[28] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation*, NSDI, 2014.

[29] P. Druschel and G. Banga. Lazy receiver processing (LRP): A network subsystem architecture for server systems. In *2nd USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 1996.

[30] P. Druschel, L. Peterson, and B. Davie. Experiences with a high-speed network adaptor: A software perspective. In *1994 ACM SIGCOMM Conference on Data Communication*, SIGCOMM, 1994.

[31] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. M. Merritt, E. Gronke, and C. Dodd. The virtual interface architecture. *IEEE Micro*, 18(2):66–76, Mar. 1998.

[32] K. Elmeleegy, A. Chanda, A. L. Cox, and W. Zwaenepoel. Lazy asynchronous `i/o` for event-driven servers. In *2004 USENIX Annual Technical Conference*, ATC, 2004.

[33] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *15th ACM Symposium on Operating Systems Principles*, SOSP, 1995.

[34] T. Flach, N. Dukkipati, A. Terzis, B. Raghavan, N. Cardwell, Y. Cheng, A. Jain, S. Hao, E. Katz-Bassett, and R. Govindan. Reducing web latency: The virtue of gentle aggression. In *2013 ACM SIGCOMM Conference on Data Communication*, SIGCOMM, 2013.

[35] M. Flajslik and M. Rosenblum. Network interface design for low latency request-response protocols. In *2013 USENIX Annual Technical Conference*, ATC, 2013.

[36] S. Floyd and E. Kohler. Profile for datagram congestion control protocol (DCCP) congestion control ID 2: TCP-like congestion control, Mar. 2006. RFC 4341.

[37] Google Cloud Platform. Cloud Functions – serverless environment to build and connect cloud services. `https://cloud.google.com/functions/`.

[38] S. Gorinsky and H. Vin. Additive increase appears inferior. Technical Report TR2000-18, Department of Computer Sciences, University of Texas at Austin, May 2000.

[39] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: a scalable and flexible data center network. In *2009 ACM SIGCOMM Conference on Data Communication*, SIGCOMM, 2009.

[40] gRPC Authors. grpc. `https://grpc.io/`.

[41] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn. RDMA over commodity Ethernet at scale. In *2016 ACM SIGCOMM Conference on Data Communication*, SIGCOMM, 2016.

[42] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy. SoftNIC: A software NIC to augment hardware. Technical Report UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015.

[43] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: A GPU-accelerated software router. In *2010 ACM SIGCOMM Conference on Data Communication*, SIGCOMM, 2010.

[44] S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy. MegaPipe: A new programming interface for scalable network I/O. In *10th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2012.

[45] T. Haynes and D. Noveck. Network file system (NFS) version 4 protocol, Mar. 2015. RFC 7530.

[46] J. Heinlein, K. Gharachorloo, S. Dresser, and A. Gupta. Integration of message passing and shared memory in the Stanford FLASH multiprocessor. In *6th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 1994.

[47] C. Hopps. Analysis of an equal-cost multi-path algorithm, Nov. 2000. RFC 2992.

[48] R. Huggahalli, R. Iyer, and S. Tetrick. Direct cache access for high bandwidth network I/O. In *32nd Annual International Symposium on Computer Architecture*, ISCA, 2005.

[49] IEEE 802.1. 802.1Qbb priority-based flow control. `https://1.ieee802.org/dcb/802-1qbb/`, June. 2011.

[50] IEEE 802.3bs Task Force. P802.3bs-2017 - IEEE standard for Ethernet amendment 10: Media access control parameters, physical layers and management parameters for 200 Gb/s and 400 Gb/s operation, Jan. 2017.

[51] Infiniband Trade Association. Annex a 16: RoCE. `https://cw.infinibandta.org/document/dl/7148`, Apr. 2010. Release 1.2.1.

[52] Infiniband Trade Association. Annex a 17: RoCEv2. `https://cw.infinibandta.org/document/dl/7781`, Sept. 2014. Release 1.2.1.

[53] Infiniband Trade Association. Infiniband architecture specification volume 1. `https://cw.infinibandta.org/document/dl/7859`, Mar. 2015. Release 1.3.

[54] Infiniband Trade Association. Infiniband architecture specification volume 2. `https://cw.infinibandta.org/document/dl/8125`, Nov. 2016. Release 1.3.1.

[55] Intel Corporation. PCI-SIG SR-IOV primer: An introduction to SR-IOV technology. *Intel application note*, Jan. 2011. Revision 2.5.

[56] Intel Corporation. Intel 82599 10 GbE controller datasheet. `https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/82599-10-gbe-controller-datasheet.pdf`, Jan. 2014. Revision 2.9.

[57] Intel Corporation. Receive side scaling on Intel network adapters. `http://www.intel.com/content/www/us/en/support/network-and-i-o/ethernet-products/000006703.html`, June 2016.

[58] V. Jalaparti, P. Bodik, S. Kandula, I. Menache, M. Rybalkin, and C. Yan. Speeding up distributed request-response workflows. In *2013 ACM SIGCOMM Conference on Data Communication*, SIGCOMM, 2013.

[59] K. Jang, S. Han, S. Han, S. Moon, and K. Park. SSLShader: Cheap SSL acceleration with commodity processors. In *8th USENIX Symposium on Networked Systems Design and Implementation*, NSDI, 2011.

[60] E. Y. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: A highly scalable user-level TCP stack for multicore systems. In *11th USENIX Symposium on Networked Systems Design and Implementation*, NSDI, 2014.

[61] P. Joubert, R. B. King, R. Neves, M. Russinovich, and J. M. Tracey. High-performance memory-based web servers: Kernel and user-space performance. In *2001 USENIX Annual Technical Conference*, ATC, 2001.

[62] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA efficiently for key-value services. In *2014 ACM SIGCOMM Conference on Data Communication*, SIGCOMM, 2014.

[63] A. Kalia, M. Kaminsky, and D. G. Andersen. FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2016.

[64] A. Kalia, D. Zhou, M. Kaminsky, and D. G. Andersen. Raising the bar for using GPUs in software packet processing. In *12th USENIX Symposium on Networked Systems Design and Implementation*, NSDI, 2015.

[65] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat. Reducing datacenter application latency with endhost NIC support. Technical Report CS2012-0977, CSE Department, University of California, San Diego, Apr. 2012.

[66] T. Karmarkar. Availability of H-series VMs in Microsoft Azure. `https://azure.microsoft.com/en-us/blog/availability-of-h-series-vms-in-microsoft-azure/`.

[67] A. Kaufmann, S. Peter, N. K. Sharma, T. Anderson, and A. Krishnamurthy. High performance packet processing with FlexNIC. In *21st International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2016.

[68] S. Kim, S. Huh, X. Zhang, Y. Hu, A. Wated, E. Witchel, and M. Silberstein. GPUnet: Networking abstractions for GPU programs. In *11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2014.

[69] E. Kohler, M. Handley, and S. Floyd. Datagram congestion control protocol (DCCP), Mar. 2006. RFC 4340.

[70] R. Koller and D. Williams. Will serverless end the dominance of Linux in the cloud? In *16th Workshop on Hot Topics in Operating Systems*, HOTOS, 2017.

[71] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter Heron: Stream processing at scale. In *2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD, 2015.

[72] H. Kung and C. E. Leiserson. Systolic arrays (for VLSI). In *Sparse Matrix Proceedings 1978*, pages 256–282, 1979.

[73] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[74] J. Lemon. Kqueue - a generic and scalable event notification facility. In *2001 USENIX Annual Technical Conference*, ATC, 2001.

[75] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[76] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang. KV-Direct: High-performance in-memory key-value store with programmable NIC. In *26th ACM Symposium on Operating Systems Principles*, SOSP, 2017.

[77] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation*, NSDI, 2014.

[78] X. Lin, Y. Chen, X. Li, J. Mao, J. He, W. Xu, and Y. Shi. Scalable kernel TCP design and implementation for short-lived connections. In *21st International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2016.

[79] Linux Kernel Authors. Configure Soft-RoCE (RXE). `https://github.com/linux-rdma/rdma-core/blob/master/Documentation/rxe.md`.

[80] Linux man-pages project. epoll - I/O event notification facility. `http://man7.org/linux/man-pages/man7/epoll.7.html`.

[81] S. Liu, T. Başar, and R. Srikant. TCP-Illinois: A loss and delay-based congestion control algorithm for high-speed networks. In *1st International Conference on Performance Evaluation Methodolgies and Tools*, VALUETOOLS, 2006.

[82] lwIP Authors. lwIP – a lightweight TCP/IP stack. https://savannah.nongnu.org/projects/lwip/.

[83] I. Marinos, R. N. Watson, and M. Handley. Network stack specialization for performance. In *2014 ACM SIGCOMM Conference on Data Communication*, SIGCOMM, 2014.

[84] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference*, USENIX, 1993.

[85] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Computer Communication Review*, 38(2):69–74, Mar. 2008.

[86] Mellanox Technologies. Mellanox simplifies RDMA deployments with enhanced RoCE software. http://www.mellanox.com/page/press_release_item?id=1760, July 2016.

[87] Memcached Authors. memcached – distributed memory object caching system. http://memcached.org/.

[88] Message Passing Interface Forum. MPI: A message-passing interface standard. https://www.mpi-forum.org/docs/, June 2015. Version 3.1.

[89] C. Mitchell, Y. Geng, and J. Li. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *2013 USENIX Annual Technical Conference*, ATC, 2013.

[90] R. Mittal, V. T. Lam, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, and D. Zats. TIMELY: RTT-based congestion control for the datacenter. In *2015 ACM SIGCOMM Conference on Data Communication*, SIGCOMM, 2015.

[91] R. Mittal, J. Sherry, S. Ratnasamy, and S. Shenker. Recursively cautious congestion control. In *11th USENIX Symposium on Networked Systems Design and Implementation*, NSDI, 2014.

[92] J. C. Mogul. TCP offload is a dumb idea whose time has come. In *9th Workshop on Hot Topics in Operating Systems*, HOTOS, 2003.

[93] B. Montazeri, Y. Li, M. Alizadeh, , and J. Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *2018 ACM SIGCOMM Conference on Data Communication*, SIGCOMM, 2018.

[94] K. Nagaraj, D. Bharadia, H. Mao, S. Chinchali, M. Alizadeh, and S. Katti. NUMFabric: Fast and flexible bandwidth allocation in datacenters. In *2016 ACM SIGCOMM Conference on Data Communication*, SIGCOMM, 2016.

[95] A. Narayan, F. Cangialosi, P. Goyal, S. Narayana, M. Alizadeh, and H. Balakrishnan. The case for moving congestion control out of the datapath. In *16th ACM Workshop on Hot Topics in Networks*, HOTNETS, 2017.

[96] Netronome. NFP-6xxx flow processor. `https://netronome.com/product/nfp-6xxx/`.

[97] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. PortLand: A scalable fault-tolerant layer 2 data center network fabric. In *2009 ACM SIGCOMM Conference on Data Communication*, SIGCOMM, 2009.

[98] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation*, NSDI, 2013.

[99] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot. Scale-out NUMA. In *19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2014.

[100] OpenIB.org. rsocket protocol and design guide. `https://github.com/linux-rdma/rdma-core/blob/master/librdmacm/docs/rsocket`.

[101] PCI-SIG. Atomic operations. `https://www.pcisig.com/specifications/pciexpress/specifications/ECN_Atomic_Ops_080417.pdf`, Jan. 2008. PCI-SIG Engineering Change Notice.

[102] PCI-SIG. TLP processing hints. `https://www.pcisig.com/specifications/pciexpress/specifications/ECN_TPH_11Sept08.pdf`, Sept. 2008. PCI-SIG Engineering Change Notice.

[103] A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris. Improving network connection locality on multicore systems. In *7th ACM European Conference on Computer Systems*, EuroSys, 2012.

[104] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. *ACM Transactions on Computer Systems*, 33(4):11:1–11:30, Nov. 2015.

[105] J. Postel. User datagram protocol, Aug. 1980. RFC 768.

[106] J. Postel. Transmission control protocol, Sept. 1981. RFC 793.

[107] I. Pratt and K. Fraser. Arsenic: A user-accessible Gigabit Ethernet interface. In *20th IEEE International Conference on Computer Communications*, INFOCOM, 2001.

[108] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *41st Annual International Symposium on Computer Architecture*, ISCA, 2014.

[109] S. Radhakrishnan, Y. Geng, V. Jeyakumar, A. Kabbani, G. Porter, and A. Vahdat. SENIC: Scalable NIC for end-host rate limiting. In *11th USENIX Symposium on Networked Systems Design and Implementation*, NSDI, 2014.

[110] K. Ramakrishnan, S. Floyd, and D. Black. The addition of explicit congestion notification (ECN) to IP, Sept. 2001. RFC 3168.

[111] RDMA Consortium. Architectural specifications for RDMA over TCP/IP. `http://www.rdmaconsortium.org/`.

[112] Redis Authors. Redis. `http://redis.io/`.

[113] R. Reed. Scaling to millions of simultaneous connections. `http://www.erlang-factory.com/upload/presentations/558/efsf2012-whatsapp-scaling.pdf`, Mar. 2012.

[114] L. Rizzo. netmap: A novel framework for fast packet I/O. In *2012 USENIX Annual Technical Conference*, ATC, 2012.

[115] M. Roesch. Snort - lightweight intrusion detection for networks. In *13th USENIX Conference on System Administration*, LISA, 1999.

[116] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, Feb. 1992.

[117] M. Rotaru. Scaling to 12 million concurrent connections: How MigratoryData did it. `https://mrotaru.wordpress.com/2013/10/10/scaling-to-12-million-concurrent-connections-how-migratorydata-did-it/`, Oct. 2013.

[118] S. Savage, N. Cardwell, D. Wetherall, and T. Anderson. TCP congestion control with a misbehaving receiver. *SIGCOMM Computer Communication Review*, 29(5):71–78, Oct. 1999.

[119] H. Shah, F.Marti, W. Noureddine, A. Eiriksson, and R. Sharp. Remote direct memory access (RDMA) protocol extensions, June 2014. RFC 7306.

[120] Y. Shan, B. Wang, J. Yan, Y. Wang, N. Xu, and H. Yang. FPMR: MapReduce framework on FPGA. In *18th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA, 2010.

[121] N. K. Sharma, A. Kaufmann, T. Anderson, A. Krishnamurthy, J. Nelson, and S. Peter. Evaluating the power of flexible packet processing for network resource allocation. In *14th USENIX Symposium on Networked Systems Design and Implementation*, NSDI, 2017.

[122] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat. Jupiter Rising: A decade of Clos topologies and centralized control in Google's datacenter network. In *2015 ACM SIGCOMM Conference on Data Communication*, SIGCOMM, 2015.

[123] 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 *2016 ACM SIGCOMM Conference on Data Communication*, SIGCOMM, 2016.

[124] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, S.-T. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, S. Katti, and N. McKeown. Programmable packet scheduling at line rate. In *2016 ACM SIGCOMM Conference on Data Communication*, SIGCOMM, 2016.

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

[126] L. Soares and M. Stumm. FlexSC: Flexible system call scheduling with exception-less system calls. In *9th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2010.

[127] W. Stevens. TCP slow start, congestion avoidance, fast retransmit, and fast recovery algorithms, Jan. 1997. RFC 2001.

[128] W. Sun and R. Ricci. Fast and flexible: Parallel packet processing with GPUs and Click. In *9th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS, 2013.

[129] The IEEE and The Open Group. aio.h - asynchronous input and output. `http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/aio.h.html`.

[130] The Linux Foundation. toe. `https://wiki.linuxfoundation.org/networking/toe`.

[131] The UCSB iCTF. The 2010 iCTF data. `https://ictf.cs.ucsb.edu/archive/2010/dumps/`.

[132] Top500 Supercomputer Sites. List statistics. `https://www.top500.org/statistics/list/`.

[133] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy. Storm@Twitter. In *2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD, 2014.

[134] Twitter, Inc. Finagle: A protocol-agnostic RPC system. `https://blog.twitter.com/engineering/en_us/a/2011/finagle-a-protocol-agnostic-rpc-system.html`.

[135] A. Vandecappelle. Linux kernel networking control flow. `https://wiki.linuxfoundation.org/networking/kernel_flow`.

[136] G. Vasiliadis, L. Koromilas, M. Polychronakis, and S. Ioannidis. GASPP: A gpu-accelerated stateful packet processing framework. In *2014 USENIX Annual Technical Conference*, ATC, 2014.

[137] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: a user-level network interface for parallel and distributed computing. In *15th ACM Symposium on Operating Systems Principles*, SOSP, 1995.

[138] X. Wang, D. Lazar, N. Zeldovich, A. Chlipala, and Z. Tatlock. Jitk: A trustworthy in-kernel interpreter infrastructure. In *11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2014.

[139] K. Winstein and H. Balakrishnan. TCP ex machina: Computer-generated congestion control. In *2013 ACM SIGCOMM Conference on Data Communication*, SIGCOMM, 2013.

[140] J. Zhao, X. Zhang, X. Wang, Y. Deng, and X. Fu. Exploiting graphics processors for high-performance IP lookup in software routers. In *30th IEEE International Conference on Computer Communications*, INFOCOM, 2011.

[141] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang. Congestion control for large-scale RDMA deployments. In *2015 ACM SIGCOMM Conference on Data Communication*, SIGCOMM, 2015.

[142] D. Zhuo, M. Ghobadi, R. Mahajan, K.-T. Förster, A. Krishnamurthy, and T. Anderson. Understanding and mitigating packet corruption in data center networks. In *2017 ACM SIGCOMM Conference on Data Communication*, SIGCOMM, 2017.

[143] N. Zilberman, Y. Audzevich, G. Covington, and A. Moore. NetFPGA SUME: Toward 100 Gbps as research commodity. *IEEE Micro*, 34(5):32–41, Sept. 2014.

Appendix A

FlexTCP Pseudocode

A.1 NIC Pseudocode

Here we provide commented pseudocode for the FlexNIC-based implementation of FlexTCP. The NIC implementation is subdivided into common-case packet IO and handling of QM events.

A.1.1 Packet Reception

This code handles packet reception. This involves flow identification, receive payload buffer management, acknowledgement processing and generation, filtering of exception packets to the kernel, buffering of up to one out-of-order segment, flow control, and computation of RTT estimates for congestion control that are later relayed to the kernel.

```
Packet_received(packet):
  If !lookup_flow(packet.5_tuple, flow):
    To_kernel(packet)
  Else:
    Fast_path(packet, flow)
```

```
To_kernel(packet):
  event = {Type: Packet, Len: packet.len}
  event_len = packet.len + sizeof(event)

  If buffer_available(k_ctx.rx) < event_len:
    /* drop packet */
    return

  /* write event to kernel rx queue */
  addr = k_ctx.rx.base + k_ctx.rx.head
  Dma_write(addr + sizeof(event), packet)
  Dma_write(addr, krx_event)

  k_ctx.rx_head += event_len

Fast_path(packet, flow):
  If packet.tcp.flags & (SYN | RST):
    return To_kernel(packet)

  If packet.tcp.payload > 0:
    trigger_ack = 1

  old_tx = tx_available(flow)

  /* Process a valid ack */
  If packet.tcp.flags & ACK and
      ack_valid(flow, packet.tcp.ackno, &tx_bump):
```

```
    /* free space in tx buffer */
    If tx_bump <= flow.tx.in_flight:
      flow.tx.in_flight -= tx_bump
    Else:
      /* this happens if we're in a retransmission
       * and the receiver filled in a gap */
      flow.tx.seq += tx_bump - flow.tx.in_flight
      flow.tx.head += tx_bump - flow.tx.in_flight
      flow.tx.in_flight = 0

    /* count dup-acks */
    If tx_bump != 0:
      flow.dupack_cnt = 0
    Else If ++flow.dupack_cnt >= 3:
      /* fast retransmit */
      Flow_retransmit(flow)
      flow.stats.fast_rexmit++

    /* update congestion control stats */
    flow.stats.ack_bytes += tx_bump
    If packet.tcp.flags & ECE:
      flow.stats.ecn_bytes += tx_bump

  /* Handle out of order packets */
  packet_trim(packet)
  If !seq_valid(flow, packet, &in_buffer):
    /* if the packet is out of order and fits in buffer,
     * see if we can add it to the out of order interval */
    If in_buffer:
```

```
    If flow.rx.ooo_len == 0:
      flow.rx.ooo_start = packet.tcp.seqno
      flow.rx.ooo_len = packet.tcp.payload
      write_payload = 1
    Else If packet.tcp.seqno = flow.rx.(ooo_start + ooo_len):
      flow.rx.ooo_len += packet.tcp.payload
      write_payload = 1
  goto Exit;

/* Update RTT estimate (EWMA) */
If packet.tcp.flags & ACK and
    packet.tcp.tsopt_ecr != 0:
  rtt = current_time - packet.tcp.tsopt_ecr
  flow.rtt_est = ((flow.rtt_est << 8 - flow.rtt_est) +
      rtt) >> 8

/* update flow control window if it changed */
flow_control_update(flow, packet.tcp.win)

/* If we have payload, update state */
rx_bump = packet.tcp.payload
If rx_bump > 0:
  fs.rx.head += rx_bump
  fs.rx.seqno += rx_bump
  write_payload = 1

  /* trim ooo interval if necessary, and see if we
   * caught up */
  trim_ooo(flow)
```

```
    If flow.rx.ooo_len > 0 and
        flow.rx.ooo_start == fs.rx.seqno:
      rx_bump += flow.rx.ooo_len
      fs.rx.head += flow.rx.ooo_len
      fs.rx.seqno += flow.rx.ooo_len
      flow.rx.ooo_len = 0

Exit:
 /* Write payload to position in buffer based on seq */
 If write_payload:
   pos = rx_position(flow, packet.tcp.seqno)
   Dma_write(flow.rx.base + pos, packet.payload)

 /* If we received data or freed transmit buffer, notify
  * application */
 If rx_bump or tx_bump:
   ctx = flow.ctx
   event = {Type: Update, Flow: flow.opaque, Tx: flow.tx.head,
            Rx: flow.rx.head}
   Dma_write(ctx.rx.base + ctx.rx.head, event)
   ctx.rx.head++

 /* Update queue manager if data available for TX changes */
 Queue_manager_set_rate(flow.id, flow.tx.rate)
 If tx_available(flow) != old_tx:
   Queue_manager_add(flow.id, tx_available(flow) - old_tx)

 /* Send ack if necessary */
 If trigger_ack:
```

```
    Flow_tx_ACK(flow, packet.ip.flags & ECN)

Flow_retransmit(flow):
  flow.tx.seq -= flow.tx.in_flight
  flow.tx.head -= flow.tx.in_flight
  flow.tx.in_flight = 0
  flow.dupack_cnt = 0
```

A.1.2 Transmission

This code simply reacts to PCIe doorbells initiated by the user-level TCP stack and, if valid, notifies the queue manager of new data available to send.

```
Doorbell_received(id, data):
  ctx = contexts[id]
  ctx.rx.tail = data.rx_tail

  /* If new entries in Tx queue: notify queue manager */
  If data.tx_tail > ctx.tx.tail:
    Queue_manager_add(id, ctx.tx.tail - data.tx_tail)
    ctx.tx.tail = data.tx_tail
```

A.1.3 Queue manager events

This code handles injected queue manager token packets. This involves identifying the corresponding kernel or user-level context, sending packets from a kernel or user-level transmit payload buffer under rate limit, triggering retransmission initiated by the kernel, and transmit payload buffer management.

```
Queue_manager_event(id):
```

```
  If id < max_contexts:
    If id == 0:
      Kernel_tx()
    Else:
      App_tx(contexts[id])
  Else:
    Flow_tx(flows[id - max_contexts])

Kernel_tx():
  /* read tx queue entry */
  addr = k_ctx.tx.base + k_ctx.tx.head
  k_cmd = Dma_read(addr, sizeof(k_cmd))
  k_ctx.tx.head++

  If k_cmd.type == tx_packet:
    /* send a packet */
    If is_tcp(k_cmd.packet):
      add_timestamp(k_cmd.packet)
    Net_tx(k_cmd.packet)
  Else If k_cmd.type == retransmit:
    /* trigger re-transmits on a flow */
    flow = flows[k_cmd.flow_id]
    old_tx = tx_available(flow)
    Flow_retransmit(flow)
    Queue_manager_add(flow.id, tx_available(flow) - old_tx)
    Queue_manager_set_rate(flow.id, flow.tx.rate)


App_tx(ctx):
```

```
  /* read tx queue entry */
  addr = ctx.tx.base + ctx.tx.head
  cmd = Dma_read(addr, sizeof(cmd))
  ctx.tx.head++

  flow = flows[cmd.flow_id]

  /* calculate available data to be sent capped
   * by flow control window, and free rx buffer */
  old_tx = tx_available(flow)
  old_rx = rx_space(flow)

  flow.tx.tail = cmd.tx_tail
  flow.rx.tail = cmd.rx_tail

  /* Update queue manager if we added data to be sent */
  If tx_available(flow) - old_tx > 0:
    Queue_manager_add(flow.id, tx_available(flow) - old_tx)
    Queue_manager_set_rate(flow.id, flow.tx.rate)

  /* If rx buffer was completely full before, send out
   * flow control window update */
  If old_rx == 0 && rx_space(flow) > 0:
    Flow_tx_ACK(flow, 0)

Flow_tx(flow):
  /* fetch payload */
  seg_len = min(tx_available(flow), MSS)
  packet = Dma_read(flow.tx.base + flow.tx.head, seg_len)
```

```
  TCP_header_prepare(packet, flow)

  flow.tx.seq += seg_len
  flow.tx.head += seg_len
  flow.tx.in_flight += seg_len

  Net_tx(packet)

Flow_tx_ACK(flow, ecn):
  TCP_header_prepare(packet, flow)
  If ecn:
    packet.tcp.flags |= ECE
  Net_tx(packet)
```

A.2 Rate-based DCTCP control loop

This kernel code implements the DCTCP control loop using rate limits and flow information ralyed by the NIC. This consists of computing additive-increase, multiplicative-decrease congestion avoidance and TCP slow-start.

```
/* Configuration options defaults:
 *   dctcp_init_rate = 10Mbps
 *   dctcp_additive_increase = 10Mbps
 *   config.dctcp_min_rate = 10Mbps
 *   dctcp_ecn_weight = 1/16 */

/* Set up state and initialize state */
DCTCP_init(k_flow):
  k_flow.rate = config.dctcp_init_rate
  k_flow.ecn_frac = k_flow.act_rate = 0
  k_flow.slow_start = 1

/* Periodic rate update */
DCTCP_update(k_flow, stats):
  /* calculate actually used rate */
  act_rate = stats.ack_bytes / (time - k_flow.last_time)
  k_flow.act_rate = (7 * k_flow.act_rate + act_rate) / 8
  act_rate = max(k_flow.act_rate, act_rate)

  /* clamp rate to no more than 1.2 times the actual rate */
  k_flow.rate = min(k_flow.rate, 1.2 * act_rate)

  /* slow start */
```

```
If k_flow.slow_start:
  If stats.fast_rexmit == 0 and
      stats.ecn_bytes == 0 and
      stats.timeouts == 0:
    k_flow.rate *= 2
  else:
    k_flow.slow_start = 1

/* congestion avoidance */
If !k_flow.slow_start:
  If stats.fast_rexmit != 0 or
      stats.timeouts != 0:
    /* if there are drops, cut rate by half */
    k_flow.rate /= 2
  Else:
    /* update ECN fraction */
    ecn_frac = stats.ecn_bytes / stats.ack_bytes
    k_flow.ecn_frac = ecn_frac * k_flow.ecn_frac +
      config.dctcp_ecn_weight * (1 - k_flow.ecn_frac)

    If stats.ecn_bytes > 0:
      /* reduction according to ecn fraction */
      k_flow.rate *= 1 - k_flow.ecn_frac / 2
    Else:
      /* additive increase */
      k_flow.rate += config.dctcp_additive_increase

k_flow.rate = min(k_flow.rate, config.dctcp_min_rate)
```

A.3 FlexTCP Queue APIs between Components

These are the queue APIs for the queues between all components in our design (NIC, kernel, application) as shown in Figure 4.1. 4-tuples contain source and destination port numbers and IP addresses. 2-tuples contain destination IP and port number.

Kernel Context Queue (Application to Kernel)

`id, rx/tx_buf = new_flow(2-tuple, opaque)`: Open new flow to `2-tuple` identified by `opaque`. Returns internal identifier `id`, the receive and transmit buffers `rx/tx_buf`.

`listen(port, opaque)`: Listen on `port`, associate `opaque`.

`accept(port, opaque)`: Accept connection on `port` and associate `opaque`.

`close(id)`: Close connection identified by `id`.

Kernel TX Queue

`retransmit(id)`: Start re-transmitting un-acknowledged data for flow `id`.

`transmit_eth(len, data...)`: Transmit raw Ethernet packet `data` of `len`.

Kernel RX Queue

`receive_eth(len, data...)`: Receive raw Ethernet packet `data` of `len`.

Kernel Context Queue (Kernel to Application)

`new_flow(id, rx/tx_buf, opaque_listen, opaque_accept)`:
New connection identified by `id` and `opaque_accept`, with receive and transmit buffers `rx/tx_buf`, accepted on listener identified by `opaque_listen`.

`data(opaque, n)`: Same as RX queue, without `ntx`.

Per-Context RX/TX queue

TX: `send(id, n, nrx)`: Send `n` bytes on connection identified by `id`. `nrx` bytes have been processed.

RX: `data(opaque, n, ntx)`: `n` bytes arrived on the connection identified by `opaque`. `ntx` bytes have been transmitted.

A.4 FlexTCP Low-level Application Interface

This is the low-level FlexTCP interface presented to user-level TCP stacks by the NIC and the kernel.

```
Context Operations:
  /* Create new context */
  create(context *ctx)
  /* Poll for events on context (see events below) */
  poll(context *ctx, int num, event *events)

Listener Operations:
  /* Open listener on specified port */
  open(context *ctx, listener *l, uint port, uint backlog)
  /* Accept connection on listener */
  accept(context *ctx, listener *l, connection *c)
Listener Events:
  /* Listener open done, status indicates success */
  open_done(listener *l, int status)
  /* New connection arrived on listener (not accepted yet) */
  new_conn(listener *l, uint remote_port, ip_t remote_ip)
  /* Connection accepted, status indicates success */
  accept_done(listener *l, connection *c, int status)

Connection Operations:
  /* Open connection to specified destination */
  open(context *ctx, connection *c, ip_t ip, uint port)
  /* Mark oldest available `bytes` of receive buffer as processed */
  rx_free(context *ctx, connection *c, uint bytes)
```

```
  /* Allocate `bytes` of send buffer.
   * Because of circular nature of buffer the allocation can be split */
  tx_alloc(context *ctx, connection *c, uint bytes,
           void **buf_1, uint *len_1, void **buf_2)
  /* Mark `bytes` in send buffer as ready to be sent */
  tx_send(context *ctx, connection *c, uint bytes)
  /* Move connection to new context */
  move(context *ctx, connection *c, context *new_context)
Connection Events:
  /* Connection open done, status indicates success */
  open_done(connection *c, int status)
  /* Received bytes at specified location in receive buffer */
  received(connection *c, void *buf, uint len)
  /* Space freed up in formerly full send buffer */
  sendbuf_avail(connection *c)
  /* Connection successfully moved to new context */
  move_done(connection *c, int status
```