OPERATING SYSTEMS ABSTRACTIONS FOR SOFTWARE PACKET PROCESSING IN DATACENTERS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Tudor Simion Marian

January 2011

OPERATING SYSTEMS ABSTRACTIONS FOR SOFTWARE PACKET PROCESSING IN DATACENTERS

Tudor Simion Marian, Ph.D.

Cornell University 2011

Over the past decade, the modern datacenter has reshaped the computing landscape by providing a large scale consolidated platform that efficiently powers online services, financial, military, scientific, and other application domains. The fundamental principle at the core of the datacenter design is to provide a highly available, high performance computing and storage infrastructure while relying solely on low cost, commodity components. Further, in the past few years, entire datacenters have become a commodity themselves, and are increasingly being networked with each other through high speed optical networks for load balancing and fault tolerance. Therefore, the network substrate has become a key component that virtually all operations within and between datacenters rely on. Although the datacenter network substrate is fast and provisioned with large amounts of capacity to spare, networked applications find it increasingly difficult to derive the expected levels of performance. In essence, datacenters consist of inexpensive, fault-prone components running on commodity operating systems and network protocols that are ill-suited for reliable, high-performance applications. This thesis addresses several key challenges pertaining to the communication substrate of the modern commodity datacenter.

First, this thesis provides a study of the properties of commodity end-host servers connected over high bandwidth, uncongested, and long distance lambda networks. We identify scenarios associated with loss, latency variations, and degraded throughput at the attached commodity end-host servers. Interestingly, we show that while the network

core is indeed uncongested and loss in the core is very rare, significant loss is observed at the end-hosts themselves—a scenario that is both common and easily provoked. One common technology used to overcome such poor network performance are packet processors that carry out some sort of performance enhancement protocol.

Second, this thesis shows how packet processors may be used to improve the performance of the datacenter's communication layer. Further, we show that these performance enhancement packet processors can be built in software to run on the commodity servers resident in the datacenter and can sustain high data rates.

And third, this thesis extends the operating system with two novel packet processing abstractions—the *Featherweight Pipes (fwP)* and *NetSlices*. Developers can use the new abstractions to build high-performance packet processing protocols in user-space, without incurring the performance penalty that conventional abstractions engender. Most importantly, unlike the conventional abstractions, fwP and NetSlices allow applications to achieve high data rates by leveraging the parallelism intrinsic of modern hardware, like multi-core processors and multi-queue network interfaces. The key feature of the new abstractions is a design that enables independent work to proceed in parallel while aggressively minimizing the overheads during the contention phases. We demonstrate the performance of packet processors built with these new abstractions to scale linearly with the number of available processor cores.

BIOGRAPHICAL SKETCH

Tudor Marian was born in Câmpia Turzii, Romania, on a snowy Tuesday evening, December 9, 1980. He grew up sharing the bulk of his time between playing tennis (on beautiful red clay) and football (the kind with a round ball), while trying to avoid homework as much as possible. During his early years, he enjoyed tinkering with an old TIM-S computer, the Romanian clone of the Z80 Sinclair Spectrum. He later attended the "Pavel Dan" high school, during which he developed a passion for physics, military jet fighters, and a knack for QBasic. He decided to pursue a career in Computer Science, instead of a career in Physics or one as a military fighter pilot, and promptly joined the Technical University of Cluj-Napoca. Five years later, he graduated with an engineering degree and the will to further pursue his academic endeavors, since he genuinely enjoyed doing research within Professor Ioan Salomie's group, under the supervision of Mihaela Dînşoreanu. In the fall of 2004, he joined the Ph.D. program at Cornell University, where he spent six beautiful years working on systems research under the supervision of Ken Birman and Hakim Weatherspoon.

For my parents, my sister, Katie, and my dearest grandma (măicuţă dragă).

ACKNOWLEDGEMENTS

I wish to express my deepest gratitude to my advisors, Hakim Weatherspoon and Ken Birman. They have taught me the trade of being a systems researcher, one that is firmly anchored within the fundamentals, with an eye towards the art and the engineering of systems building. Hakim strived to mold me into a successful researcher, proving he has incommensurable amounts of patience, and showing extraordinary belief in my abilities. I thoroughly enjoyed the times when I simply walked into his office and had vivid debates about various subjects. Ken guided me in my early years, with enthusiasm and constant invaluable advice. His passion for the field of computer science, his extensive expertise, and his quest for excellence in research have been inspiring. I especially cherish the many stories he recounted of the "good old days" of systems research, and I was always amazed at the accuracy of his predictions on the future of the field. I wish to thank them both for all their guidance and support—without which this thesis would not have been possible.

I am especially grateful to Robbert van Renesse, who was, at times, my substitute advisor. Robbert appeared to me as a paragon of balance that managed to intertwine research with a myriad of other activities in his life. Danny Dolev's visits to Cornell allowed me to learn the fundamentals of the theory of distributed systems from one of the pioneers of the field. I enjoyed the ski trips we took together, the endless talks we had while stuck on ski lifts, and the time we spent together watching NFL football games.

I have had both the luck and the pleasure to be part of a wonderful research group, with remarkable collaborators. Mahesh Balakrishnan has been like a mentor to me, providing much needed support and guidance for which I owe him an enormous debt of gratitude. Likewise, I enjoyed very much my fruitful collaboration with Dan Freedman—he proved to be a valuable source of discussions, and brought to the table a particular, slightly unconventional, and eye opening perspective on things.

I would like to thank my dissertation committee, Hakim Weatherspoon, Ken Birman, Robbert van Renesse, Joe Halpern, and Rafael Pass. Many thanks to the entire administrative staff for all the help they provided me with throughout my stay at Cornell. I would especially like to thank Bill Hogan, for his invaluable aid, and for all the entertaining stories he kindly shared with me.

I thank my office mates, systems lab co-tenants, and colleagues that provided good company over the years, including Dan Sheldon, Sam Arbesman, Thành Nguyen, Chun-Nam Yu, Kevin Walsh, Amar Phanishayee, Hitesh Ballani, Dan Williams, Saikat Guha, Bernard Wong, Alan Shieh, Maya Haridasan, Krzysztof Ostrowski, Lakshmi Ganesh, Tuan Cao, Art Munson, Yee Jiun Song, Ian Kash, Jonathan Kaldor, Muthu Venkitasubramaniam, Anton Morozov, Bistra Dilkina, Daria Sorokina, Benyah Shaparenko, Yogi Sharma, Vikram Krishnaprasad, Alexandru Niculescu-Mizil, Cristian Danescu-Niculescu-Mizil, and Maks Orlovich.

I was fortunate enough to have amazing friends outside of Upson Hall as well. My soccer teammates and friends, without whom the summer, fall, spring, and even winter days (when we played indoor) would have felt dull—thank you for enjoying the "beautiful game" with me, and making my daily routine less monotonous and immensely more joyful.

Lastly, I would like to extend all my gratitude to my family for all their continuous support and encouragement, and to Katie, my lovely wife, who was patient enough with me, and picked up the slack when instead of doing my designated chores, I was working towards this very dissertation. I hope I can make it up to you.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The modern datacenter has taken the center-stage as the dominant computing platform that powers most of today's consumer online services, financial, military, and scientific application domains. Further, datacenters are increasingly being networked with each other through high speed optical networks. Consequently, virtually every operation within and between datacenters relies on the networking substrate. This thesis focuses on the study and the enhancement of the datacenter's network layer.

1.1 The Modern Commodity Datacenter

Over the past decade, the modern datacenter has emerged as the platform of choice for an increasingly large variety of applications. Datacenters are based on the simple concept of leveraging the aggregate resources of a cluster of inexpensive, *commodity*, servers and network equipment to perform tasks at large scales [58, 6]. According to the Merriam-Webster dictionary, a commodity item is a mass-produced unspecialized product supplied without qualitative differentiation across a market. The success of the datacenter and its rapid industry adoption to replace legacy computing platforms like high-end mainframes and supercomputers can be attributed to several factors. First, network bandwidth became an abundant resource. As can be seen in Table 1.1, the bandwidth of commodity networks has been increasing exponentially, at a rate that outpaced the growth of single processor core speed [103, 163] (expressed in metrics such as CPU core frequency or operations executed per core per second). Second, the performance to price ratio of commodity hardware components has made them viable alternatives to high-end, consolidated, mainframe servers and custom network interconnects [58, 42].

Table 1.1: Single processor and network speed evolving over the years.

Year	Single CPU frequency (MHz)	Network data rate (Mbps)
1982	12	10
1995	133	100
2001	1800	1000
2008	3200	10000
2012 (projected)	3500	40000

And third, the shift towards Internet applications, like search and email, whose typical workloads are inherently parallel by virtue of user parallelism which means they can be distributed across a cluster of independent machines with little effort [97, 119, 87].

Financial, military, scientific, and other organizations have turned to datacenters with commodity components to lower operational costs [122, 132]. As a result, more functionality (i.e., applications) and data that has previously resided on personal computers is migrating towards an online service model [44, 4, 1, 2, 3, 16, 19, 11, 34, 13, 35, 15, 5, 161, 162]. Today, users are not only interacting with fundamentally online services, like search and online games, they also perform tasks like document, spreadsheet, image, and video processing while manipulating data that is stored remotely [18, 30]. All user activity is channeled over high-throughput networks to be handled by datacenters, making the respective online services constantly available around-the-clock from anywhere in the world. These online services are specifically engineered to be deployed on commodity computing components.

Notably, commodity computing components have continuously evolved over time, and should be referenced within the time-frame that encompasses them. For example, "commodity" meant something different in the past than what commodity means today in the context of the modern datacenter, and it will surely mean something else in the future. Table 1.2 provides a reference system by summarizing the characteristics of a commodity server that can be found in today's modern datacenter, while Table 1.3 shows

Table 1.2: Typical Specifications of Datacenter-hosted Commodity Server.

Component	Specifications
Processors (CPUs)	2-16 cores (x86), clocked at 2-3GHz
Last Level Cache	2-16MB L2 or L3, 10-40 cycles access time
Cache Coherent Interconnect	Shared bus or point-to-point (e.g. HyperTransport)
Memory (RAM)	4-16 GB, 50ns access time, 20GB/s throughput
Storage (Disks)	2TB, 10ms access time, 200MB/s throughput
Network (Ethernet)	2-8 1GbE interfaces, 1-4 10GbE interfaces

Table 1.3: Typical Specifications of NOW Commodity Workstation.

Component	Specifications
Processors (CPUs)	1-2 cores (RISC, Alpha), clocked at 30-100MHz
Last Level Cache	0-1MB (optional) L2, 3-16 cycles access time
Cache Coherent Interconnect	Shared bus
Memory (RAM)	2-128 MB, 100ns access time, 200MB/s throughput
Storage (Disks)	500 MB, 15ms access time, 2-10MB/s throughput
Network (Ethernet)	Single 10Mb/s Ethernet or 155Mb/s ATM interface

the characteristics of the typical NOW workstation (*cca.* 1994). The network of workstations (NOW) project [58] along with the high performance computing community's Beowulf clusters [6] are early examples of modern datacenters. For example, looking at processing units (CPUs), the Table shows the current trend of multiprocessor commodity systems, and in particular *multi-core* systems, where several independent processors (or cores) share the same silicon chip. This recent architectural shift was caused by the inability to continuously scale in frequency a single processor, due to the energy limitations and cooling requirements of current semiconductor technology [178].

In fact, the datacenters themselves have become a commodity. For example, "plug-and-play" datacenters-in-a-shipping-container [40, 48, 111] can be purchased today. Such a datacenter comes equipped with commodity off-the-shelf hardware and software, requiring only to be plugged with electrical power, network, and potentially a cooling source before being operational [64]. This great ease, coupled with the avail-

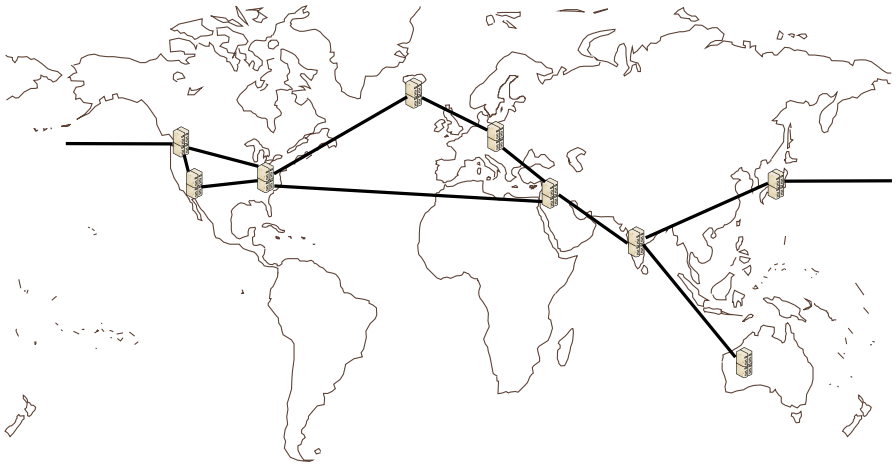


Figure 1.1: A network of geographically spread data centers.

ability of relatively cheap or already emplaced "dark" (unused) optical fiber is ushering in the global network of datacenters [120].

Geographically spread datacenters are interconnected with semi-dedicated or private high-speed optical links for scenarios such as load balancing and failover (as depicted in Figure 1.1). For example, data may be mirrored between datacenters to protect against disasters, while client requests may be redirected to the closest datacenter to improve latency, provide localized services (e.g., news aggregators like Google News [17] serve content in English for North America), and to balance the load in general. These optical networks are known as *lambda* networks since they employ multiple wavelengths to provide independent communication channels on a single optical fiber strand. Each independent channel has a capacity of 10 or 40 Gigabits/second (Gbps), while a standard for 100Gbps for the same physical medium is currently being drafted [170]. A single fiber strand may therefore carry tens to hundreds of wavelengths of light, providing large amounts of aggregate bandwidth. In this thesis we consider packet-switched lambda networks that are largely uncongested (are lightly used) with little to no traffic that competes for shared physical resources along paths.

1.2 Challenges

The underlying network substrate has become a first-class citizen of the modern commodity datacenter, with virtually all operations within and between datacenters relying on it. We identify three research questions and related challenges pertaining to the commodity datacenter's communication substrate. First, *what are the properties of the communication traffic between commodity servers over long-distance, uncongested lambda networks?* Although lambda networks have sufficient bandwidth, are dedicated for specific use, and often operate with virtually no congestion [29], end-hosts and applications find it increasingly harder to derive the full performance they might expect [50, 154, 46, 150, 180]. Second, given the abundance of commodity servers within a datacenter, *how can we leverage them to build efficient, high-speed packet processors in software, in order to enhance the performance of the (inter-) datacenter communication substrate?* A packet processor is a forwarding element in a packet-switched network that is capable of performing additional processing on the stream of packets that flows through it. For example, a packet processor may perform a performance enhancement protocol to increase throughput or mask packet loss. And third, *how can we provide primitive functional building blocks and abstractions that enable constructing software packet processor applications capable of operating at high, or even maximum line, data rates?* Current modern operating systems do not provide developers with abstractions for building packet processors at the application level, without additionally incurring a severe performance penalty cost.

This thesis is concerned with addressing the three above mentioned questions that have not been cohesively addressed by prior work. We proceed to discuss, detail, and elaborate each question in turn.

Lambda Networked Commodity Servers: Lambda networks play an increasingly central role in the infrastructure supporting globally distributed, high-performance systems and applications [120]. However, the end-to-end characteristics of commodity servers communicating over high-bandwidth, uncongested, and long-distance lambda networks have not been well understood [163]. Although convenient, relying on conventional transport protocols to utilize the network efficiently and yield nominal performance with commodity servers has proven to be difficult [163, 142, 166, 107, 50]. Simply connecting commodity servers to a lambda network without understanding their properties yields degraded performance which is common and easily provoked. For example, the Transmission Control Protocol (TCP), the de-facto protocol used for reliable communication over the Internet, yields poor throughput on high bandwidth / high delay links [129, 130, 217, 227, 63], making TCP a less than ideal candidate for tasks such as remote site backup and mirroring.

Consequently, datacenter operators have had little choice but to develop custom protocols that best suit the challenges of such an environment. For instance, instead of using conventional TCP, Google disseminates newly computed search indexes to all its datacenters spread across the globe by means of a custom protocol. This protocol uses the User Datagram Protocol (UDP) to send data at high rates, and TCP to reliably determine which parts of the data have arrived safely at the destination and which parts of the data need to be retransmitted (a protocol reminiscent of RBUDP [133], SABUL [125], and Tsunami [166, 211]).

The challenge lies in understanding the characteristics of the communication between commodity servers connected over lambda networks, especially when relying on conventional protocols like TCP and UDP. This knowledge may be subsequently used in designing new network protocols and supporting abstractions with the goal of improving

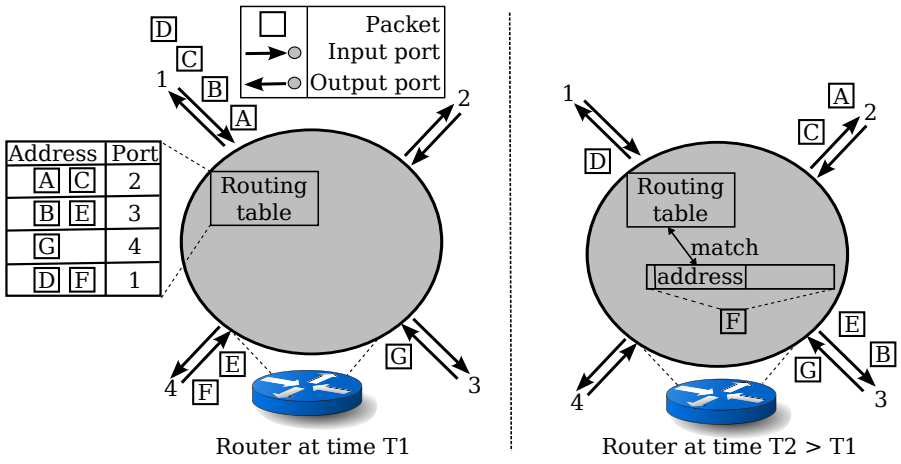


Figure 1.2: Depiction of a router with four bidirectional network interfaces. The router is depicted at two points in time, the initial configuration before any packets are forwarded on the left, and a subsequent configuration after it forwarded five packets and is processing the sixth on the right.

the performance of the underlying communication layer.

Software Packet Processors: As datacenter operators find it increasingly difficult to drive the high-speed lambda networks to their full potential with commodity servers, more and more custom network protocols are being developed. Such protocols are very diverse, are usually used to improve the quality of the underlying communication, and are typically deployed using a network of *packet processors*. A packet processor fetches network traffic from an input network interface controller/card (NIC), performs some amount of computation per packet, and forwards one or more resulting packets on output interfaces. The quintessential example of a packet processor is the Internet Protocol (IP) router. A router forwards packets between input and output interfaces by comparing the address information stored in the packet header with its internal routing table—as shown in Figure 1.2. Additionally, the IP standard requires routers to alter each packet by decrementing the time-to-live (TTL) header field, and to adjust header checksums accordingly so as to reflect the TTL modification. A router may also generate and issue new packets, in addition to the traffic that flows through it. For example, if a router

receives a packet with the TTL value of one, it will drop the original packet and generate a "time-exceeded" Internet Control Message Protocol (ICMP) packet that is returned to the original sender. Moreover, a router may have to fragment a large packet into several smaller ones due to maximum transmission unit (MTU) restrictions on network segments it is attached to.

Packet processors are often more general than the IP router. They can implement various performance enhancement functions either at the end-hosts or at some point along a network path [227, 142, 224, 107, 146, 200, 225]—in the latter case they act as proxies or middleboxes. Traditionally, packet processors trade off performance, quantified in terms of data and packet processing rates, for flexibility, extensibility, and programmability. *Hardware packet processors* provide higher performance, yet they are hard, if not impossible in practice, to extend with new functionality beyond basic forwarding, network address translation (NAT), and simple counting statistics [31, 36, 27, 12, 9, 32]. By contrast, *software packet processors* yield tremendous flexibility, at the cost of performance [43, 103, 158]. Importantly, software packet processors may be deployed on any of the commodity servers a datacenter has in abundance, and can be quickly prototyped by developers that take advantage of a familiar, rich environment, using powerful development, debugging, and testing tools. Further, software packet processors are highly extensible, since unlike hardware counterparts, modifying both the data and control plane functionality requires simple software upgrades. For example, a typical router is conceptually divided into two operational planes: **i)** the data plane that performs packet forwarding in hardware, and **ii)** the control plane that is the software which keeps up to date the routing table by exchanging information with other routers. Only the latter can be easily upgraded.

However, building software packet processors that run on the commodity servers

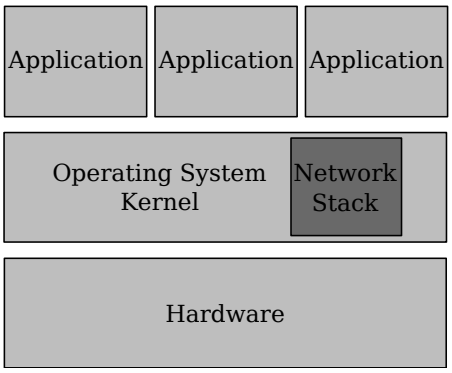


Figure 1.3: Operating system kernel multiplexing hardware resources and performing requests received from several user-space applications.

resident in a datacenter is challenging due to the high data / packet rates involved. Datacenter networks are fast and provisioned with vast amounts of available capacity to spare, therefore packet processors must be capable of handling large volumes of aggregate data from potentially many machines. The challenge is architecting efficient software packet processors that are scalable, transparent, deployable in today's datacenters, and can comprehensively address the problem of degraded performance on large bandwidth-delay links.

Packet Processing Abstractions: Modern *operating systems* are a vital component of the software stack which runs on top of the datacenter's commodity servers. Unfortunately, operating systems do not provide developers with abstractions for building packet processing applications capable of achieving sufficiently high data rates. As a result, high-speed software packet processors have been traditionally built at a low level, close to the bare hardware resources and within the operating system kernel, so as to incur as little overhead as possible from the software stack [147, 103]. An operating system is the low-level software layer that multiplexes access to the hardware resources (like disks, NICs, processors, memory) and safely exports the resources to user applications (i.e., to "user-space") by means of higher-level abstractions (like file descriptors, sock-

ets, processes, address spaces). User-space applications do not interface directly with the hardware, instead their access is mediated via the operating system—as depicted in Figure 1.3. Operating systems must therefore provide applications with efficient access to physical resources with little to no interference, should performance be of paramount importance. Further, operating systems must allow applications to take advantage of newly emerging hardware.

Efficient operating systems support is required for developers to build general purpose software packet processing applications that run in user-space and achieve performance levels similar to the low-level software which has direct access to the hardware resources (like the operating system's kernel). Unfortunately, applications that reside outside the kernel, or in user-space, incur a severe performance penalty since modern operating systems fail to provide efficient access to the entire network traffic in bulk. An operating system's conventional software network stack in general, and the *raw socket* in particular, exert excessive amounts of pressure on the memory subsystem and are inefficient in utilizing the available physical resources. The raw socket is the principal mechanism for relaying the entire network traffic to user-space applications. Consequently, software packet processing applications relying on conventional abstractions like the raw socket are unable to scale with, and take advantage of, modern multi-core processors to drive multi-gigabit network adapters at line rates (e.g., 10Gbps for 10GbE lambda networks).

Since general purpose software packet processing applications reside outside the kernel, namely in user-space, the challenging aspect is designing and building a set of abstractions that enable developers to take advantage of the emerging hardware, like multi-core processors, to deliver upon the required levels of performance. Ideally, such abstractions would also be familiar, un-encumbering, and universally usable by develop-

ers. Furthermore, such abstractions should minimize performance overheads of packet processing at a high level in the software stack.

1.3 Contributions

We present three contributions towards studying and enhancing the commodity datacenter's communication substrate. First, we investigate the network properties of commodity end-hosts connected across large geographical spans by high throughput optical links. Next, we demonstrate how packet processing network protocols can be built in software and run on commodity servers to enhance datacenter communication patterns. Finally, we provide new operating systems abstractions that enable developers to build such software packet processors that can perform complex performance enhancement protocols in user-space, without compromising performance. Importantly, unlike the existing conventional abstractions for packet processing in user-space, the new operating systems abstractions we introduce are able to take advantage of modern hardware and enable applications to scale linearly with the number of processor cores.

Lambda Networked Commodity End-hosts High-bandwidth, semi-private optical lambda networks carry growing volumes of data on behalf of large data centers, both in cloud computing environments and for scientific, financial, defense, and other enterprises. In this thesis we undertake a careful examination of the end-to-end characteristics of an uncongested lambda network running at high speeds over long distances, identifying scenarios associated with loss, latency variations, and degraded throughput at attached end-hosts. We use identical commodity source and destination server platforms, hence expect the destination to receive more or less what we send. We observe otherwise: degraded performance is common and easily provoked. In particular, the re-

ceiver loses packets even when the sender employs relatively low data rates. Data rates of future optical network components are projected to vastly outpace clock speeds of commodity end-host processors [163] (also see Table 1.1 and Figure 3.1), hence more and more applications will confront the same issue we encounter. Our work thus reveals challenges faced by those hoping to achieve dependable performance in such uncongested optical networked settings, and has two implications. First, there is a growing need for packet processors and packet processing support to improve networking performance between commodity end-hosts. Second, commodity servers communicating through conventional operating system abstractions are shown to perform poorly, therefore more efficient abstractions are required to support packet processing applications. Packet processing applications typically handle significantly more traffic per second than end-host applications, like the ones employed in this study.

Software Packet Processors Connecting datacenters over vast geographical distances is challenging, especially when relying on traditional communication protocols [163]. One common technology used to tackle this problem are packet processing perimeter middleboxes that perform some form of performance enhancement network protocol. We show that such packet processors may be built in software and may be deployed on the commodity servers resident within the datacenter to thoroughly improve the communication between datacenters. Further, we show that these software packet processors performing performance enhancement protocols can sustain high network data rates, thus providing a readily available commodity alternative to otherwise proprietary, dedicated, hardware equipment.

Packet Processing Abstractions Packet processing (user-space) applications have traditionally incurred a severe performance penalty due to the conventional software

network stack that overloads the memory subsystem. We introduce two new packet processing operating systems abstractions to address the issue—Featherweight Pipes (fwP) and NetSlices. FwP allows developers to run user-space packet processing applications at high data rates on commodity, shared-bus, multi-core platforms. FwP provides a fast, multi-core aware, path for packets between network interfaces and user-space, reducing pressure and contention on the memory subsystem, in addition to removing other sources of overheads, like system calls, blocking, and scheduling. In our experiments, a security appliance, an overlay router, an IPsec gateway, and a protocol accelerator, all using fwP, are shown to run on many cores at gigabit speeds.

The NetSlice operating system abstraction takes fwP one step further, targeting multi-core non uniform memory access (NUMA) architectures and modern multi-queue network adapters. Unlike the conventional raw socket, NetSlice is designed to tightly couple the hardware and software packet processing resources, and to provide the application with control over these resources. To reduce overall contention, NetSlice performs coarse-grained spatial partitioning (i.e., exclusive, non-overlapping, assignment) of CPU cores, memory, and NIC resources, instead of time-sharing them. Moreover, NetSlice provides a streamlined communication channel between NICs and user-space. We show that complex user-space packet processors, like a protocol accelerator and an IPsec gateway, built with NetSlice and running on commodity components, can scale linearly with the number of cores and operate at nominal 10Gbps network line speeds.

1.4 Organization

The rest of the thesis is organized as follows. In Chapter 2, we present an argument in support of software packet processing abstractions, describe our experimental methodol-

ogy, and demonstrate how high-speed packet processing network protocols can be built to improve the communication between datacenters. In particular, we show how various complex performance enhancement proxies can be carefully built within the operating system software layer of commodity servers to process packets at high data rates. Moreover, these packet processors serve as baseline comparison for user-space equivalent implementations. In Chapter 3, we present a study of the properties of uncongested lambda networks interconnecting 10GbE commodity end-hosts over large distances. The study provides key insight into some of the performance roadblocks standing in the way of achieving communication at full line-rate between different datacenters. Chapter 4 details the overheads faced by operating system abstractions for building packet processors in user-space, and introduces Featherweight Pipes (fwP)—a new operating system abstraction for packet processing that overcomes these overheads. Chapter 5 introduces NetSlice—a new operating systems abstraction that builds upon fwP to enable developers to build packet processors in user-space, while also taking advantage of modern hardware, like NUMA multi-core processors and multi-queue network adapters. Importantly, packet processors built with NetSlice are able to scale linearly with the number of CPU cores. We present related work in Chapter 6, and finally discuss potential future research directions and conclude in Chapter 7.

CHAPTER 2

THE SCOPE OF THE PROBLEM AND METHODOLOGY

Extensible and programmable router support is becoming more important within today's experimental networks [20, 26, 14, 177]. Indeed, general purpose packet processors enable the rapid prototyping, testing, and validation of novel protocols. For example, OpenFlow [165] evolved quickly into a mature specification, and was able to do so by leveraging highly extensible NetFPGA [157] forwarding elements. Moreover, the OpenFlow specification is currently being incorporated into silicon fabric by enterprise grade router manufacturers. At the same time, extensible router support seamlessly enables the deployment of functionality that is currently implemented by network providers through special purpose network middleboxes, such as protocol accelerators and performance enhancement proxies [31, 36, 27, 12, 9, 32, 63].

The prospect of software packet processors is attractive since they are highly extensible and programmable. For example, modifying both data and control plane functionality of a software router requires simple software upgrades. Moreover, employing commodity servers means that packet processors may rapidly take advantage of the new semiconductor technology advancements. Further, software packet processors benefit from the large-scale manufacturing of commodity components.

Traditionally, the tradeoff between specialized hardware packet processors and software packet processors running on general purpose commodity hardware has been, and remains still, one of high performance versus extensibility and ease of programmability. The currency for packet processors is performance. More recently, several significant efforts strived to render networking hardware more extensible [47, 23, 165, 157, 10]. Conversely, software routers have successfully harnessed the raw horsepower of modern hardware to achieve considerably high data rates [103, 159]. However, for the sake

of performance, such software routers were devised to run within the operating system's kernel, at a low level immediately on top of the hardware.

Writing a packet processor on domain specific, albeit extensible, hardware is difficult since the developer needs to be aware of low level issues, intricacies, and limitations. We argue that building packet processors in the kernel, even when taking advantage of elegant frameworks such as Click [147], is equally difficult. In particular, the developer does not simply learn a new "programming paradigm." She needs to be aware of the idiosyncrasies of the memory allocator (e.g., small virtual address spaces, the limit on physically contiguous memory chunks, the inability to swap out pages), understand various execution contexts and their preemptive precedence (e.g., interrupt context, bottom half, task / user context), understand synchronization primitives (like various spinlock variants, mutexes, and semaphores) and how they are intimately intertwined with the execution contexts (e.g., when an execution context is not allowed to block), deal with the lack of standard development tools like debuggers, and handle the lack of fault isolation. A bug in a conventional monolithic kernel brings the system into an inconsistent state and is typically lethal—leading at best to a crash, or worse, may corrupt data on persistent storage or cause permanent hardware component failure.

Although user-space packet processing applications could potentially ease the developer's burden, the premium on performance has rendered such an option largely invalid for all but modest data rates. Packet processors running in user-space on modern operating systems (OSes) are rarely able to saturate modern networks [103, 168, 93, 43], given that 10 Gigabit Ethernet (GbE) Network Interface Controllers (NICs) are currently a commodity. Yet the opportunity to achieve both performance and programmability rests in taking advantage of the parallelism intrinsic in modern hardware (like multi-core processors and multi-queue NICs). However, to scale linearly with the number of

cores, contention must be kept to a minimum. Conventional wisdom, and Amdahl's law [53, 184, 136], states that when adding processors to a system, the benefit grows at most linearly, while the costs (cache coherency, memory / bus contention, serialization, etc.) grow quadratically. Unfortunately, current operating systems fail to provide developers with user-space abstractions for building high-speed packet processors that take advantage of the modern emerging hardware.

The rest of the chapter is structured as follows. Section 2.1 describes the experimental testbeds used throughout this thesis for both measurements and system evaluation. Section 2.2 details the commodity hardware configurations employed by the testbeds, while Section 2.3 reports on the metrics we used to perform quantitative assessments. Finally, Section 2.4 presents three examples of complex software packet processors that can improve the performance of datacenter communication.

2.1 Testbeds

Throughout this thesis, we employ two testbed configurations on which we perform our measurements and system evaluation. The first testbed, called the Cornell NLR Rings, leverages a high-end production lambda network that is currently in use by the scientific community. The second consists of the Emulab network emulation testbed—a public resource freely available to most researchers worldwide.

2.1.1 Cornell NLR Rings

To emulate a pair of datacenters connected over large geographical distances, we created a "personal" lambda network by tapping into the National Lambda Rail [26] network

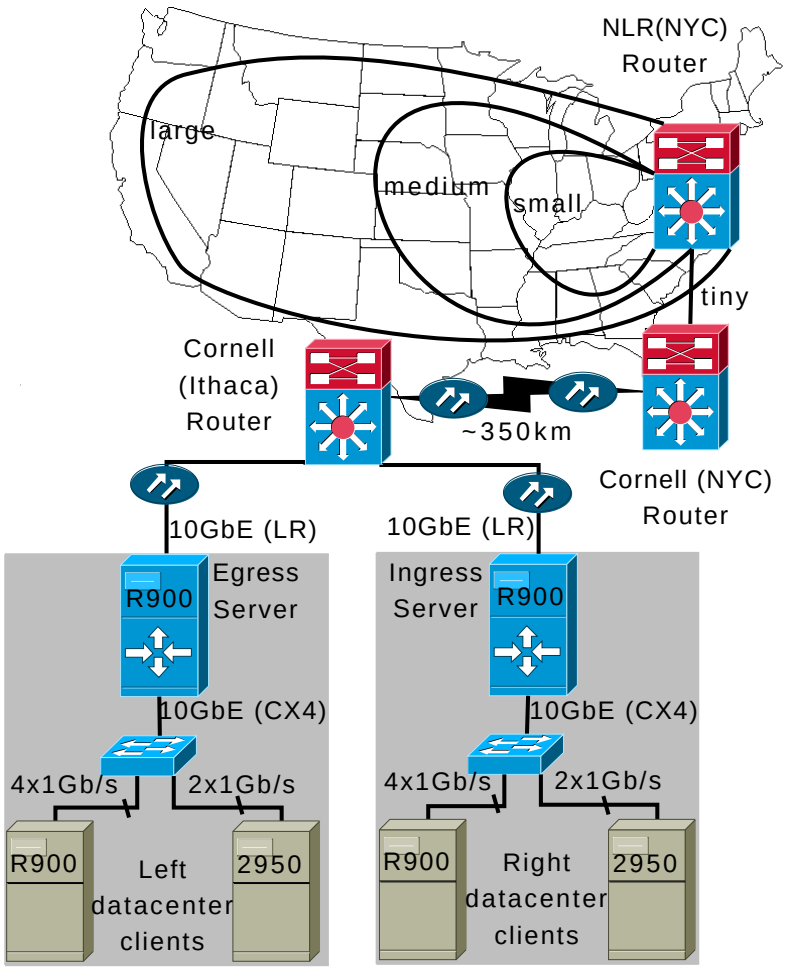


Figure 2.1: The testbed topology.

resources. In particular, we created our own network measurement testbed centered at Cornell University and extending across the United States; this is the Cornell National LambdaRail (NLR) Rings testbed.

Depicted in Figure 2.1, our Cornell NLR Rings testbed takes advantage of the existing National LambdaRail [26] backbone infrastructure. Two commodity servers are connected to the backbone router in Ithaca, New York, and function as *Ingress* and *Egress* servers; these are four-way 2.4 GHz Xeon E7330 quad-core Dell PowerEdge R900 servers with 32GB RAM, each equipped with an Intel 10GbE LR PCIe x8 adapter

(EXPX9501AFXLR) and an Intel 10GbE CX4 PCIe x8 adapter (EXPX9502CX4). They run a preemptive 64-bit Linux 2.6.24 kernel, with the Intel `ixgbe` driver version 1.3.47. The generic segmentation offload (GSO) was disabled since it is incompatible with the Linux kernel packet forwarding subsystem. We connected a pair of client machines to each commodity router through a pair of HP ProCurve 2900-24G switches. The client machines are Dell PowerEdge R900 and PowerEdge 2950 respectively, with Broadcom NetXtreme II BCM5708 Gigabit Ethernet cards (R900 has four such 1Gbps NICs while the 2950 has two). They run the same Linux 2.6.24 kernel as the routers, with stock bnx2 network drivers for the Broadcom Gigabit network interfaces.

Through a combination of IEEE 802.1Q virtual Local Area Network (VLAN) tagging and source-, policy-, and destination-based routing, we have established four static 10GbE full duplex routes that begin and end at Cornell, but transit various physical lengths: a *tiny* ring to New York City and back, a *small* ring via Chicago, Atlanta, Washington D.C., and New York City, a *medium* ring via Chicago, Denver, Houston, Atlanta, Washington D.C., and New York City, and a *large* ring across Chicago, Denver, Seattle, Los Angeles, Houston, Atlanta, Washington D.C., and New York City (Figure 2.1). The one-way latency (one trip around the ring) as reported by the `ping` utility is 8.0 ms for the tiny path, 37.3 ms for the small path, 68.9 ms for the medium path, and 97.5 ms for the large path. All optical point-to-point backbone links use 10GbE with Dense Wavelength Division Multiplexing (DWDM) [210], except for a single OC-192 Synchronous Optical Networking (SONET) [37] link between Chicago and Atlanta. The Cornell NLR Rings employ all NLR routers and 10 of the 13 NLR layer three links.

The NLR routers are Cisco CRS-1 [7] devices, while the Cornell (Ithaca and NYC) backbone routers are Cisco Catalyst 6500 series [8] hardware. These routers all have sufficient backplane capacity to operate at their full rate of 10Gbps irrespective of the

traffic pattern; the loads generated in our experiments thus far have provided no evidence to the contrary. The Cisco Catalyst 6500s are equipped with WSX6704-10GE Ethernet modules with centralized forwarding cards. The Quality of Service (QoS) feature on these routers was disabled, hence in the event of an over-run, all traffic is equally likely to be discarded. In particular, packets are served in the order in which they are received. If the buffer is full, all subsequent packets are dropped, a discipline sometimes referred to as first-in-first-out (FIFO) queueing with drop-tail [90]. Enabling QoS requires wholesale reconfiguration of the production NLR network by NLR engineers, and was not feasible.

Note that the Cornell NLR testbed also has a loopback configuration, by connecting the egress and the ingress servers directly back-to-back with a 10 meter optical patch cable. The loopback configuration is typically used for baseline measurements, and for packet processor prototype development and testing.

2.1.2 Emulab

The Utah [109] and Deter [101] Emulab [220] testbeds have been an invaluable resource for prototyping, debugging, and ultimately evaluating the system software artifacts implemented in support of this thesis. Emulab is a network testbed for emulated experiments. Each emulated experiment allows one to specify an arbitrary network topology, and provides a controllable, predictable, and repeatable environment, which includes exclusive access to leased commodity servers. All the resources of an emulated experiment are instantiated over hardware within a datacenter, and are isolated from all other concurrently emulated experiments. For example, the network topology paths are created by segregating virtual LANs (VLANs) within switches, and relying on additional com-

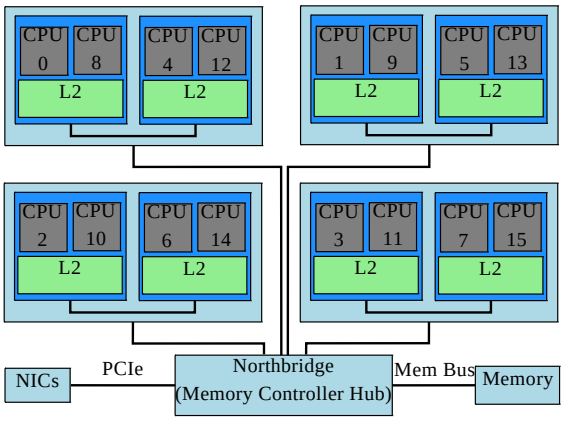
modity servers (also employed exclusively) along paths to perform traffic shaping (e.g., to emulate packet loss, and link delays corresponding to various connection lengths).

Although each Emulab testbed provides various flavors of commodity servers and underlying local area network substrates, we relied exclusively on the `pc3000` class of servers, which are connected to 1Gbps Ethernet. With respect to the available Emulab hardware components, the `pc3000` servers have the best performance while also being equipped with at least two network interface controllers. Each `pc3000` server is a Dell PowerEdge 2850 with one or two hyperthreaded 3.0 GHz 64-bit Xeon processors, 2GB DDR2 RAM, and up to six Intel PRO/1000 MT network adapters. Furthermore, the intermediate nodes performing traffic shaping in our experiments were also `pc3000` servers. Since Emulab allows roughly any operating system to run on the servers, we used various flavors of the Linux 2.6 kernel.

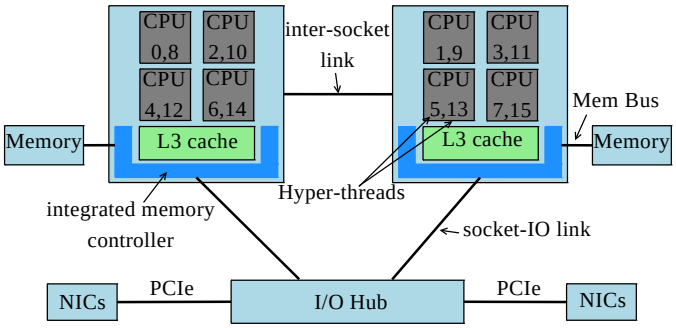
2.2 Commodity Servers Hardware Configuration

The measurements and system evaluations in this thesis use several commodity server hardware configurations. Although the capabilities of the commodity servers differ slightly from one to the other, all servers fall into one of two categories, depending upon the memory architecture. The first class of commodity servers, the most common until recently, consists of shared bus memory architectures. By contrast, the second class of commodity servers consists of non uniform memory access (NUMA) architectures. For a detailed exposition of the architecture of a modern commodity server's memory architecture (i.e., shared bus and NUMA architectures) refer to Appendix A.

The shared bus memory architecture commodity servers comprise of the following:



(a) Shared bus architecture



(b) NUMA architecture

Figure 2.2: Shared bus (a) and NUMA (b) quad core (Intel Xeon) architectures.

- All Emulab machines, namely the Dell PowerEdge 2850 with a single or dual Intel Xeon CPU and 2GB of DDR2 RAM.
- The Cornell NLR Rings machines both in regular and in loopback configuration, used in Chapters 4 and 5 respectively. More precisely, the Dell PowerEdge R900 machines are four socket 2.40GHz quad core Xeon E7330 (Penryn) with 6MB of L2 cache and 32GB of RAM—the E7330 is effectively a pair of two dual core CPUs packaged on the same chip, each with 3MB of L2 cache. The Dell PowerEdge 2950 are dual socket 2.66GHz quad core Intel Xeon X5355 (Core) with 8MB of L2 cache and 16GB of RAM—like the E7330, the X5355 also bundles a pair of two dual core CPUs on the same chip, each with 4MB of L2 cache. Figure 2.2 (a) depicts a diagram of the shared bus Xeon E7330 and X5355 archi-

tectures respectively.

In Chapter 5 we employ more recent hardware, namely a pair of NUMA commodity servers, acting as ingress and egress routers in the Cornell NLR Rings loopback configuration. Each server is a Dell PowerEdge R710 machine, equipped with dual socket quad core 2.93GHz Xeon X5570 (Nehalem) processors with 8MB of shared L3 cache and 12GB of RAM, 6GB connected to each of the two CPU sockets through the QuickPath Interconnect (QPI). Unlike the previous generations of shared bus Xeon processors (e.g., Penryn and Core), the Nehalem CPUs do not package pairs of dual cores on the same chip, instead all four cores were designed to be part of the same silicon die sharing the L3 cache. Further, the Nehalem CPUs support hardware threads, or hyperthreads, hence the operating system manages a total of 16 processors per R710 machine. Figure 2.2 (b) depicts a diagram of the NUMA Xeon X5570 architecture.

All servers are connected exclusively to commodity 1 Gigabit and 10 Gigabit Ethernet networks through a variety of commodity network interface controllers. For example, all Emulab machines are equipped with Intel PRO/1000 MT Gigabit Ethernet adapters, while the Cornell NLR Rings servers are equipped with Broadcom NetXtreme II BCM5708 Gigabit Ethernet adapters. Furthermore, two Power Edge R900 and the two R710 commodity servers part of the Cornell NLR Rings are equipped with Intel or Myricom 10 Gigabit Ethernet LR and CX4 adapters, depending on the experiments. The 10GbE adapters are also commodity components, and provide only basic DMA packet transmit and receive operations; although both the Intel and Myricom 10GbE NICs have the ability to multiplex traffic onto multiple receive and transmit hardware queues, they do not have memory-mapped virtual interface support.

2.3 Metrics

The performance of packet processors and networked end-hosts is quantified and compared in terms of network throughput. The throughput is defined as the average rate of successful data delivery over a communication channel, and is usually measured in bits per second (bps), and sometimes in data packets per second (pps). When quantifying throughput (and other metrics as well), we perform multiple independent measurements, typically in an end-to-end [192] fashion, and we report on the average value accompanied by error bars that denote the standard error of the mean. The standard error of the mean is the standard deviation of the sampling distribution of the mean, and is estimated by $\frac{\sigma}{\sqrt{N}}$ where σ is the standard deviation of the sample and N is the sample size. It can also be viewed as the standard deviation of the error in the sample mean relative to the true mean. Standard error of the mean quantifies variability while accounting for the sample size, unlike standard deviation that only quantifies variability, therefore standard error of the mean indicates the confidence in the measurement of the mean.

Another metric used in this thesis is the end-to-end packet loss fraction (or percentage) that occurs over a communication channel. The measurements in Chapter 3 report on the packet loss percentage of traffic across the Cornell NLR Rings lambda network.

To measure throughput and packet loss, we use the standard Iperf [204] and Netperf [28] measurement applications. Both network tools create TCP and UDP data streams, and allow the user to set various parameters, like UDP sender data rate, or UDP datagram size. The tools report the throughput of the payload, without protocol header encapsulation, like TCP/IP or UDP/IP.

Chapter 3 also reports on the measurement of packet inter-arrival time, a metric denoting the time interval in seconds between consecutively received packets. By contrast,

Chapter 4 reports on the measurement of packet delay as the packet percolates through the software stack of a commodity server, from the time it was picked up by the network interface off the transmission medium, until it was delivered to the application.

In Chapter 4 we also report on the count of processor micro-architecture events during the execution of certain benchmarks and tests. The events are statistically sampled and recorded into the processor's hardware performance counters, counters that are read and recorded both prior and after the tests. To compare different test and benchmark programs we report on the ratios of two related events. In particular, we measure the ratio of memory bus transactions per number of CPU cycles, the load ratio (number of words loaded in the L1 cache per number of CPU cycles), the store ratio, the number of pipeline flushes per number of instructions retired, and the number of read for ownership (RFO) cache consistency messages per memory bus transactions.

2.4 Software Packet Processor Examples

Packet processors at the perimeter of datacenters may be used to improve the performance of the communication over high bandwidth, high latency, uncongested lambda network links. By contrast, conventional protocols that were built for high bandwidth, high latency links (or high bandwidth-delay product links) like the datagram congestion control protocol (DCCP) [146], stream control transmission protocol [200], the explicit control protocol (XCP) [142], and the variable-structure congestion control protocol (VCP) [225] are a poor fit for such an environment, since congestion is not the main issue affecting the performance on semi-private lambda networks [163, 142].

In this section we show how packet processors can be carefully designed within the operating system's kernel layer of commodity servers. We show that these packet

processors are capable of sustaining high data rates while improving the performance of the communication channels over lambda networks. Furthermore, the packet processors are evaluated to provide a baseline for the expected performance equivalent user-space solutions may achieve, when deployed on identical commodity hardware.

For a proof of concept, we have carefully designed, built, and measured three comprehensive and fully functional high-speed packet processors—a TCP protocol accelerator (*TCPsplit*), a protocol independent redundancy elimination proxy (*IPdedup*), and a proactive forward error correction proxy (*IPfec*). All are designed for performance, and are carefully built at a low level close to the bare hardware—namely within the operating system layer as loadable kernel modules—to minimize software stack interference.

2.4.1 TCPsplit

We begin by presenting a packet processor that implements a conventional performance enhancement proxy scheme previously proposed in the past to mitigate high bandwidth-delay product link-related degradations [72, 207].

Networked applications require reliable transport protocols. TCP is currently the de-facto communication protocol on virtually all networks. This holds true for datacenter networks—the vast majority of datacenter traffic consists of TCP flows; TCP has been observed to make up 99.91% of the traffic within datacenters [50]. TCP provides a reliable, connection oriented, bidirectional stream abstraction. Data is delivered in the order it was sent, received data is positively acknowledged, and loss is inferred either by means of timeouts or duplicate acknowledgments. Once loss occurs, TCP assumes it was due to congestion, and enters the congestion control operational mode.

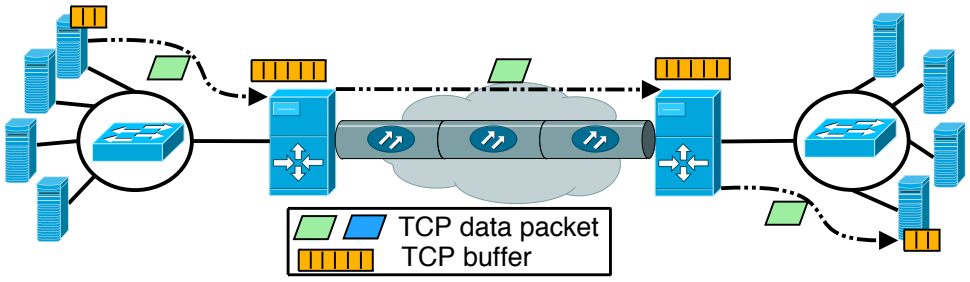


Figure 2.3: TCPsplit diagram depicting traffic pattern.

Importantly, TCP performance depends not upon the transfer rate itself, but rather upon the product between the transfer rate and the round-trip delay time (RTT). This metric, referred to as the "bandwidth-delay product" (BDP) measures the amount of in-flight unacknowledged data that would saturate the capacity of the link. This amount of in-flight data is controlled by the minimum size of the TCP window (i.e., the amount of buffer space) at the sender and at the receiver. However, the TCP window size is an operating system (OS) specific configuration parameter, and requires elevated privileges in order to configure it. Since the window size should be altered to match the BDP between any possible pair of end-hosts, this is hardly a desirable option within a datacenter, where standardization is the key to low and predictable maintenance costs.

The conventional solution is to use a pair of proxies that transparently "split" the TCP connection into three separate TCP flows, while increasing the window size to match the large BDP on the middle (i.e., the large delay) segment, as depicted in Figure 2.3. Note that this solution breaks the end-to-end [192] properties of TCP in order to transparently increase the amount of in-flight data. This is what the *TCPsplit* packet processor does [62]. In particular, the egress and the ingress servers (depicted in Figure 2.1) running *TCPsplit* will track all end-to-end TCP connections flowing through them, intercept them, sever them, and transparently create three equivalent TCP flows instead, while increasing the BDP on the TCP flow between themselves. *TCPsplit* there-

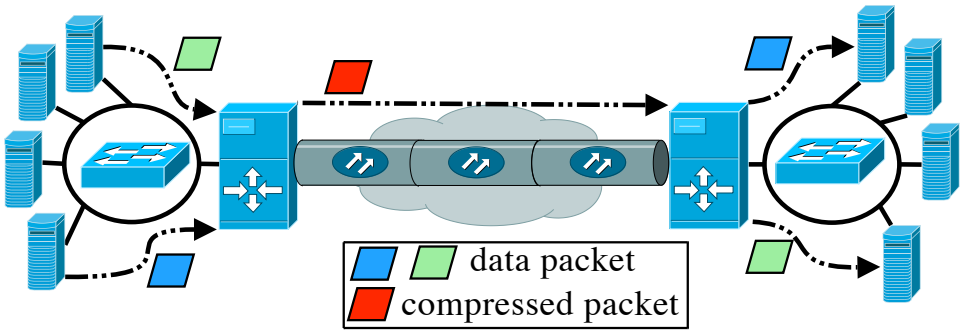


Figure 2.4: IPdedup diagram depicting traffic pattern.

fore increases the end-to-end throughput without any end-host intervention.

2.4.2 IPdedup

Next, we present a memory bound packet processor that performs a type of low-latency on-the-fly compression of the traffic that flows through it. We employ a protocol independent packet level redundancy elimination technique originally proposed by Spring et al. [199] by adapting the algorithm initially conceived by Manber [160] for finding similar files in a large file system to finding temporal similarities in a stream of packets. Manber proposed that instead of using expensive `diffs` [137] or hash functions over the entire contents of a file, Rabin fingerprints [189] can be efficiently generated instead by performing a per-byte sliding window computation. Further, since it is prohibitive to store all fingerprints for each file (for a file of size F bytes and signature size S bytes there are $(F-S+1)$ Rabin fingerprints), a set of *representative fingerprints* is selected to identify the file's content. Since Rabin fingerprints are the result of applying a cryptographic hash function, the common way of choosing the representative fingerprints is selecting the ones that have γ least significant bits zero, and hence ensuring that approximately one fingerprint out of every 2^{γ} bytes in the stream is selected.

Spring et al. [199] proposed applying Manber's fingerprinting algorithm on a pair of routers at opposite ends of a bandwidth constrained link (as depicted in Figure 2.4). Given a cache storing past packets, and an index mapping representative fingerprints to packets in the cache, the algorithm identifies contiguous strings of bytes in the current packet that are also stored in the cache. For each fingerprint matched against the cache, the matching packet is pulled out of the cache and the matched region is expanded byte-by-byte in both directions, thus obtaining the maximal region of overlap. Each of the matches within the current packet is replaced by a tuple consisting of the fingerprint, the matched region index, and the matched region length. The modified packet, now of smaller payload, is forwarded to the receiving router that will recover the original packet payload, provided that it holds a cache and an index consistent with the sending router.

This algorithm was also revisited in the context of universal packet level redundancy suppression and new redundancy-aware routing algorithms [54, 55]. Further, the same technique has since been used for different problem domains, for example to fingerprint malware [196] or to enable low bandwidth file systems [173].

Our goal in designing the *IPdedup* packet processor was to reduce memory accesses. Accordingly, we implemented the packet cache and the index both as CPU-cache friendly data structures—packet and index data is kept into fixed size arrays, avoiding linked list chaining which increases CPU-cache pollution. We provisioned each proxy with a packet cache that can hold up to roughly six seconds worth of 1Gbps data. (The six-second value was chosen in order for the cache and index together to utilize the entire RAM memory available on our commodity servers, without having to resort to slow disk accesses.) Furthermore, each proxy has a corresponding index whose size is computed based on the expected number of representative signatures per maximum transmission unit (MTU). For example, for 1500 byte MTU size TCP packets (of 1448

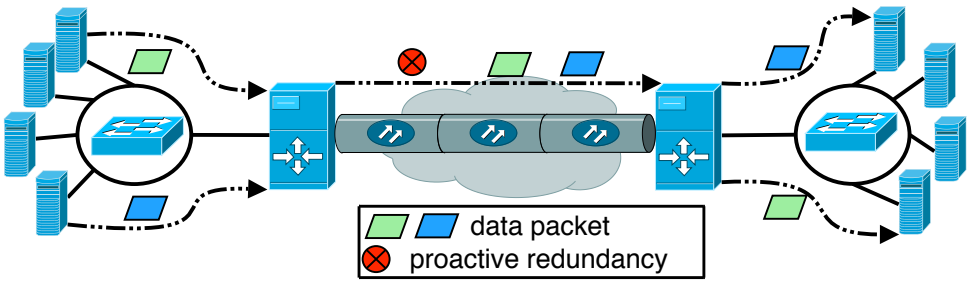


Figure 2.5: IPfec diagram depicting traffic pattern.

byte payload), 64-bit Rabin fingerprints, and $\gamma = 8$ least significant fingerprint bits set to zero, there are an expected 5.4 signatures per packet. (Since there are $(1448 - 64 + 1)$ hash values per packet, and the probability that one hash value has at least $\gamma = 8$ bits set to zero is $\frac{1}{2^8}$.) In this case, the index will be a multiple of $\lceil f \times 5.4 \times S \rceil$, where S is the capacity of the packet cache, and $f = 2$ is the hash table load factor.

2.4.3 IPfec

Finally, the *IPfec* packet processor was also designed to reduce memory accesses. *IPfec* is the dual of *IPdedup*—it adds redundancy on a path between an egress and an ingress router so as to recover quickly, without waiting for timeout and retransmissions, from packet loss events that are potentially bursty (as depicted in Figure 2.5). IPfec is essentially the implementation of the Maelstorm [63, 62] protocol—a performance enhancement proxy developed to overcome the poor performance of TCP when loss occurs on high bandwidth-delay product lambda network links. Like *IPdedup*, *IPfec* appliances work in tandem, with each appliance located at the interface between the datacenter and the high bandwidth wide area network (WAN) link.

The *IPfec* packet processors perform forward error correction (FEC) encoding over

the outbound traffic on one side and decoding over the inbound traffic on the opposite side. In Figure 2.1, for example, the *Egress* and the *Ingress* servers are running *IPfec* appliances, with the egress server encoding over all traffic originating from the left-hand side R900 and 2950 client boxes and destined for the right hand side R900 and 2950 client machines. The ingress server receives both the original Internet protocol (IP) packet traffic and the additional FEC traffic and forwards the original traffic and potentially any recovered traffic. Note that this is a symmetric pattern, each *IPfec* appliance working both as an ingress and as an egress router at the same time.

Maelstrom trades off maximum effective bandwidth to increase network performance measured in throughput: a constant overhead due to FEC masks potential future loss of packets, which reduces effective bandwidth. For example, for every 100 IP packets forwarded, Maelstrom may send three repair packets thereby reducing the available bandwidth by 3/103. Importantly, Maelstrom can be configured to tolerate packets lost in bursts while keeping the FEC overhead constant, which is useful for network provisioning and stability. The latency to recover a lost packet degrades gracefully as losses get burstier, but the FEC overhead stays constant.

2.4.4 Baseline Performance

We evaluate the low-level, in-kernel implementations of the *TCPsplit*, *IPdedup*, and *IPfec* packet processors, and provide baseline throughput measurements for the commodity server hardware they ran on. In particular, we measured the packet processors' performance on the Cornell NLR Rings experimental setup previously described in Section 2.1.1, with all commodity servers consisting of shared-bus multi-core architectures. We show that all packet processors can sustain multi-gigabit throughput, being lim-

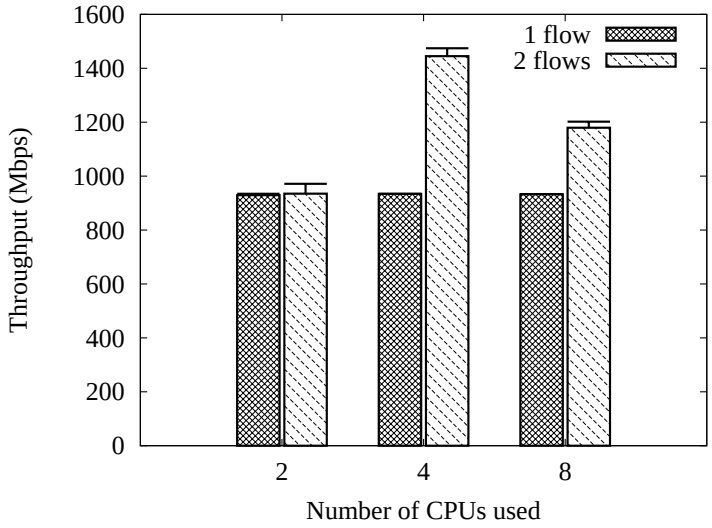


Figure 2.6: IPdedup throughput vs. number of CPUs used.

ited by the memory bandwidth. More precisely, the throughput depends on the amount of per-packet processing and memory accesses performed while saturating the memory bandwidth. For example, the *IPdedup* and *IPfec* are memory bound, with *IPdedup* performing significantly more memory accesses than *IPfec* since the index mapping signatures to previous packets is many times larger than the CPU caches. By contrast, the data structures used by *IPfec* are an order of magnitude smaller. The *TCPsplit* exerted the least amount of pressure on the memory system.

We evaluate the packet processors by issuing 120 second Iperf [204] flows (1500 byte MTU size packets) between the left hand side and the right hand side clients (Figure 2.1). All traffic flows through the egress and the ingress commodity servers. The servers are running one or a composition of the performance enhancement proxies previously described. All throughput values presented are averaged over eight independent runs; the error bars denote standard error of the mean and are always present.

Figure 2.6 shows the end-to-end TCP throughput between one and two clients sending data at around 1Gbps on the tiny path (RTT=15.9 ms), with the *IPdedup* appliances

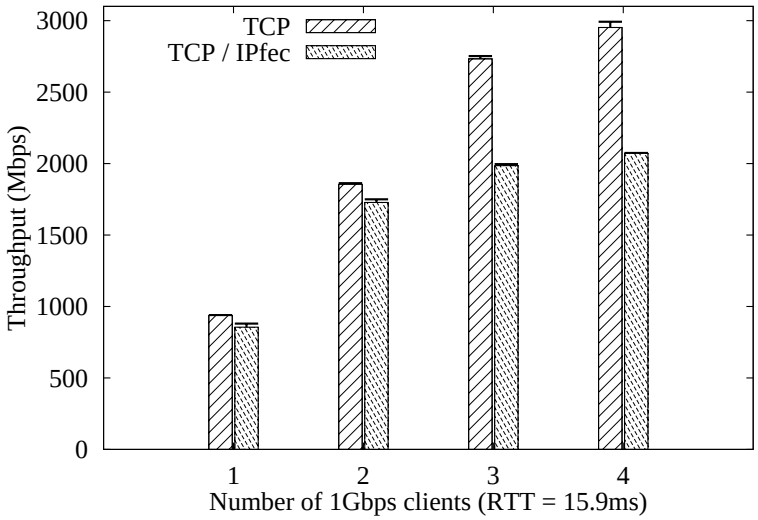


Figure 2.7: IPfec (Maelstrom) Throughput vs. number of 1Gbps clients, RTT = 15.9ms (tiny path).

performing packet-level redundancy elimination between the egress and the ingress servers. The data is a synthetic merge of a large 700MB video file, yielding a 50% redundant stream. The Figure shows the throughput plotted against the number of worker kernel threads used by the appliances. Since the machines have multiple cores, *IPdedup* is structured as several pipelines in superscalar fashion. The *IPdedup* packet processor is capable of performing as much as 49% redundancy suppression, at speeds in excess of 1.4Gbps. The value closely follows the throughput achieved by previous deduplication benchmarks that loaded packet traces in memory instead of running on live traffic [55]. Although the *IPdedup* implementation may use multiple processor cores, once the memory bandwidth is saturated, adding more processing cores degrades overall performance on the shared-bus commodity servers due to contention, as can be seen in Figure 2.6.

Figure 2.7 shows the end-to-end TCP throughput between various number of clients sending data at around 800Mbps each on the tiny path, with and without the *IPfec* appliances running on the egress and ingress commodity servers respectively. The *IPfec* ap-

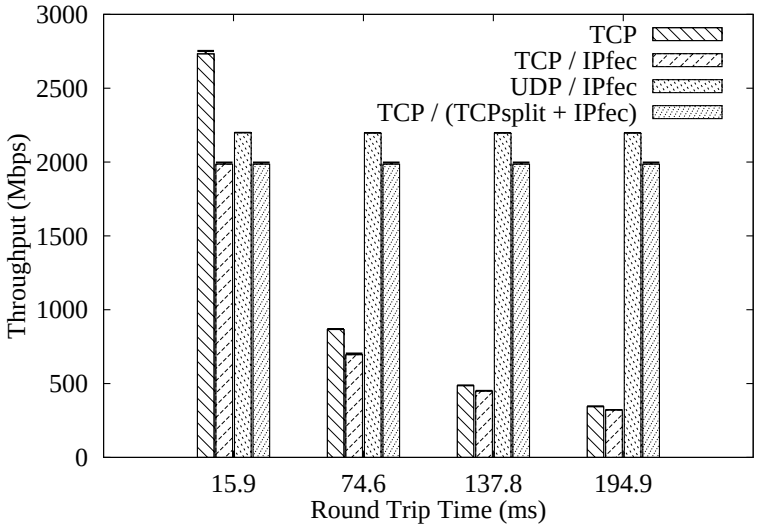


Figure 2.8: IPfec and TCPsplit Throughput vs. RTT.

pliance is capable of sustaining little more than 2Gbps aggregate throughput, given that the maximum forwarding rate is little more than 2.9Gbps (at which point the shared-bus commodity architecture saturates the memory bandwidth). Moreover, for the nominal FEC parameters used (for every $r = 8$ data packets, $c = 3$ redundant / FEC packets are sent) there is a 27% FEC protocol overhead, which means that the effective maximum goodput achievable by *IPfec* is $\left(2.9 \times \frac{r}{r+c} = 2.9 \times \frac{8}{11}\right) = 2.1$Gbps.

Finally, Figure 2.8 shows the end-to-end throughput of a 4MB TCP-window flow without any proxy, with the *IPfec* packet processor, and with a composition of *IPfec* and *TCPsplit* packet processors against the RTT of the tiny, small, medium, and large paths respectively. For comparison, we include a maximum throughput UDP flow with the *IPfec* packet processor.[1] Note that as expected, a 4MB window is sufficient to saturate the capacity of the *IPfec* packet processors over the tiny (RTT=15.9 ms) path, however it is insufficient for longer paths. By contrast, using the *TCPsplit* packet processor we

[1]The maximum throughput achieved by UDP/IPfec is slightly larger than TCP/IPfec. This is because unlike UDP, a TCP flow sends small acknowledgment packets on the return path, for which the ingress and egress routers spend CPU cycles and memory bandwidth to forward and compute the additional FEC.

achieve the same throughput irrespective of the link delay, similar to the throughput of a steady rate UDP flow—which is precisely what *TCPsplit* was designed to achieve.

2.5 Summary

In this Chapter we argued that there is a growing need for extensible router and packet processor support, and in particular, that there is a need for building software packet processors that run in user-space, on commodity servers. We also introduced the experimental testbeds used throughout the thesis, the commodity hardware employed for measurements and system evaluation, and the metrics employed in performing quantitative assessments. Finally, we provided three examples of software packet processors, and evaluated their performance. The evaluation showed that complex software packet processors can be built at a low level within the operating system's kernel layer of commodity servers to achieve high data rates. Furthermore, the evaluation provides a baseline for the expected performance equivalent user-space solutions may achieve, when running on the same commodity hardware architectures. For example, the evaluation provides a baseline for packet processors built with the new user-space abstractions introduced in Chapters 4 and 5. Importantly, the evaluation also shows that packet processors can significantly improve the performance of the communication channel between commodity end-hosts connected by high bandwidth, high latency lambda networks.

CHAPTER 3

LAMBDA NETWORKED COMMODITY SERVERS

Optical lambda networks play an increasingly central role in the infrastructure supporting globally distributed, high-performance systems and applications. Scientific, financial, defense, and other enterprise communities are deploying lambda networks for high-bandwidth, semi-private data transport over dedicated fiber optic spans between geographically dispersed data centers. Astrophysicists at Cornell University in New York receive high-volume data streams from the Arecibo Observatory in Puerto Rico or the Large Hadron Collider in Switzerland, process the data at the San Diego Supercomputer Center in California, and retrieve the results for future reference and storage at Cornell. Enterprise technology firms, such as Google and Microsoft, have begun to build proprietary networks to interconnect their data centers; this architecture balances the economics of consolidation against the benefits of end-user proximity, while increasing fault-tolerance through redundancy.

This trend will only accelerate. We are seeing a new wave of ambitious commercial networking initiatives. For example, Google recently announced a fiber-to-the-home test network [41] in the United States to deliver bidirectional bandwidth of 1 Gigabit per second (Gbps), while major Internet providers such as Verizon and Time Warner are projecting significant future improvements in consumer bandwidth. In contrast, as illustrated in Figure 3.1, the sending and receiving end-hosts themselves are approaching a (single-core) performance barrier. Thus, the future may bring high-speed networks connected to commodity machines powered by an ensemble of slow cores.

One consequence is that, while lambda networks typically have greater bandwidth than required, dedicate their transport for specific use, and operate with virtually no congestion [29] (in fact, the networks are routinely idle), commodity end-host servers

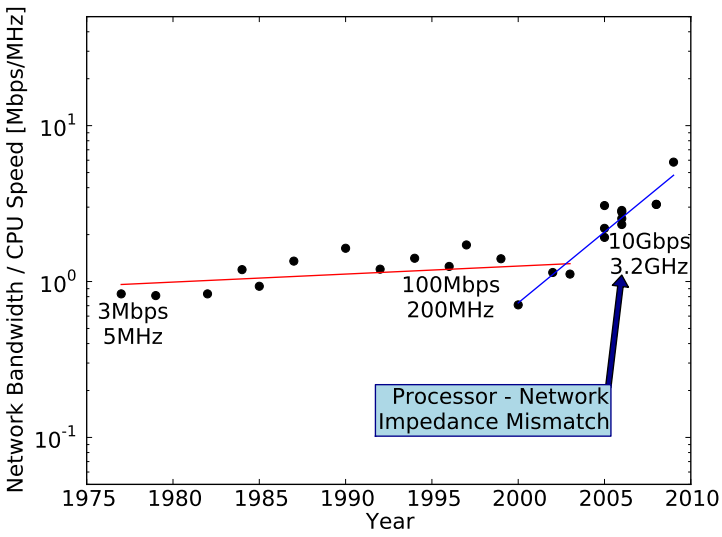


Figure 3.1: Network to processor speed ratio.

increasingly find it hard to derive the full performance they might expect [154, 46]. This can be especially frustrating because, unlike the public Internet, traffic across these semi-private lambda networks encounters seemingly ideal conditions. For example, since lambda networks operate far from the congestion threshold and employ high quality optical fiber, they should not drop any packets at all. Further, one might reasonably believe that, if traffic is sent at some regular rate well below the actual capacity of the lambda network, it will arrive intact and more or less at the same rate. In particular, if end-host Network Interface Controllers (NICs) can reliably communicate at their maximum data rates in the lab, they should similarly do so over an uncongested and lossless lambda network.

In this chapter, we show that packet loss occurs in precisely such situations. Our study reveals that, in most cases, the problem is not due to loss *within* the optical network span itself but instead arises from the interaction of lower-speed commodity end-host servers with such a high-bandwidth optical network: a kind of impedance mismatch.

This mismatch is further aggravated in situations where the bottlenecks prove to be end-host memory buses, which are generally even slower than processors. And the situation may soon worsen: end-host performance increase is expected to be achieved mostly through multicore parallelism, yet it can be a real challenge to share a network interface among multiple processor cores. One issue is contention [103], and a second is that the performance-enhancing features of modern multi-queue NICs (like Receive Side Scaling) work best only for a large number of distinct, lower bandwidth, flows.

Our goal in this chapter is not to solve this problem, but rather to shed more light on it, with the hope of informing future systems architecture research. Accordingly, we have designed a careful empirical measurement of the end-to-end behavior of a state-of-the-art high-speed optical lambda network interconnecting commodity 10 Gigabit Ethernet (10GbE) end-host servers. Our community has a long history of performing systematic measurements on many prominent networks as they have emerged, including ARPANET, its successor NSFNET [91, 134], and subsequently the early Internet [193]. However, few studies have looked at semi-dedicated lambda networks, and none consider the interactions between the high-bandwidth optical core [172] of a lambda network and 10GbE commodity end-hosts [195]. Nevertheless, this thesis shows that packet processors implementing performance enhancement protocols may be successfully employed to overcome the perfromance issues identified in this chapter.

This study in this chapter uses a new experimental networking infrastructure testbed—the *Cornell National LambdaRail (NLR) Rings*—consisting of a set of four all-optical end-to-end 10GbE paths, of different lengths (up to 15 000 km) and number of routing elements (up to 13), with ingress and egress points at Cornell University. On this testbed, the core of the network is indeed uncongested, and loss is very rare; accounting for all loss associated with sending over 20 billion packets during a 48-hour

period, we observed only one brief instance of loss in the network core, in contrast to significant packet loss observed on the end-host commodity servers themselves.

Our key findings pertain to the relation between end-to-end behavior and fine-grained configuration of the commodity end-host server:

- The size of the socket buffer and of the Direct Memory Access (DMA) ring determines the loss rate experienced by the end-host (the socket buffer and the DMA ring are depicted in Figure 3.4). Similarly, the interrupt affinity policy of the network adapter, that maps interrupts to individual processor cores upon receipt of network traffic, also affects the end-host loss distribution.
- The throughput of the ubiquitous Transmission Control Protocol (TCP) decreases as packet loss increases, and this phenomenon grows in severity as a function of both the path length (and therefore number of forwarding hops) and the window size (the TCP window size is depicted in Figure 3.4). The congestion control algorithm turns out to have only a marginal role in determining the achievable throughput.
- Batching of packets, through both kernel and NIC techniques, increases overall throughput, at the cost of disturbing any latency-sensitive measurements, such as packet inter-arrival times.

This chapter first introduces two examples of uncongested lambda networks—the TeraGrid [39] and our own Cornell NLR Rings testbed. In Section 3.2, we present and discuss our experimental results, summarized in Section 3.3.

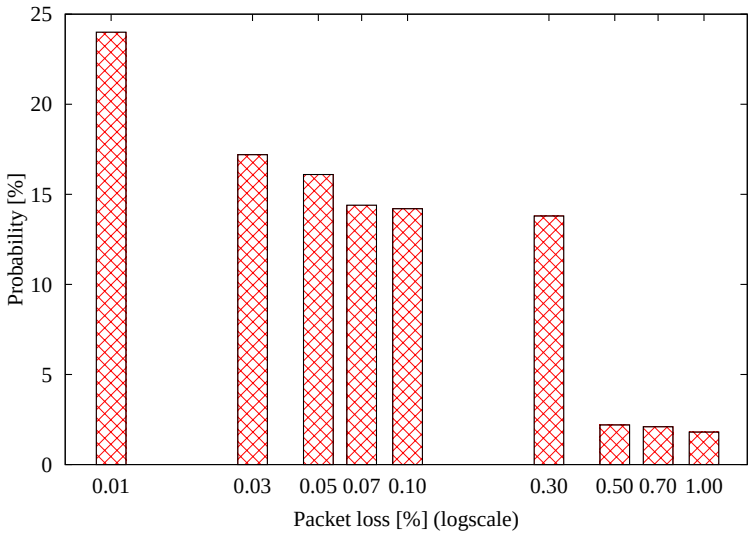


Figure 3.2: Observed loss on TeraGrid.

3.1 Uncongested Lambda Networks

Lambda networking, as defined by the telecommunications industry, is the technology and set of services directly surrounding the use of multiple optical wavelengths to provide independent communication channels along a strand of fiber optic cable [210]. In this section, we present two examples of lambda networks, namely TeraGrid [39] and the Cornell NLR Rings testbed. Both networks consist of semi-private, uncongested 10Gbps optical Dense Wavelength Division Multiplexing (DWDM) [210] or OC-192 Synchronous Optical Networking (SONET) [37] links.

3.1.1 TeraGrid

TeraGrid [39] is an optical network interconnecting ten major supercomputing sites throughout the United States. The backbone provides 30Gbps or 40Gbps aggregated throughput over 10GbE and SONET OC-192 links [172]. End-hosts, however, connect

to the backbone via 1Gbps links, hence the link capacity between each pair of end-host sites is 1Gbps.

Of particular interest is the TeraGrid monitoring framework [46]; each of the ten sites reports measurements of throughput (average data rate of successful delivery) and loss rates of User Datagram Protocol (UDP) packets performed with Iperf [204]. Every site issues a 60-second probe to every other site once an hour, resulting in a total of 90 overall measurements collected every hour. Figure 3.2 shows a histogram of percentage packet loss (on a logscale x-axis) between November 1st, 2007, and January 25th, 2008, where 24% of the measurements had 0.01% loss and a surprising 14% of the measurements had 0.10% loss. Though not shown in the Figure, after eliminating a single TeraGrid site (Indiana University) that dropped incoming packets at a steady 0.44% rate over the monitored time period, 14% of the remainder of the measurements showed 0.01% loss, while 3% showed 0.10% loss. Dialogue with TeraGrid operators revealed that the steady loss rate experienced by the Indiana University site was due to a faulty commodity network card at the end-host.

Although small, such numbers are sufficient to severely reduce the throughput of TCP on these high-latency, high-bandwidth paths [63, 180]. Conventional wisdom suggests that optical links do not drop packets. Indeed, carrier-grade optical equipment is often configured to shut down beyond bit error rates of 10^{-12} (one out of a trillion bits). However, the reliability of the lambda network is far less than the sum of its optical parts—in fact, it can be less reliable by orders of magnitude. Consequently, applications depending on protocols like TCP, which require high reliability from high-speed networks, may be subject to unexpectedly high loss rates, and hence low throughput.

Figure 3.2 shows the loss rate experienced during UDP traffic on end-to-end paths which cannot be directly generalized to TCP loss rates. It is unclear if packets were

dropped along the optical path, at intermediate devices (e.g., optical or electrical switches), or at the end-hosts. Furthermore, loss occurred on paths where levels of optical link utilization (determined by 20-second moving averages) were consistently lower than 20%, making congestion highly unlikely, a conclusion supported by the network administrators [216].

Lacking more detailed information about the specific events that trigger loss in TeraGrid, we can only speculate about the sources of the high observed loss rates. Several hypotheses suggest themselves:

Device clutter: The critical communication path between any two end-hosts consists of many electronic devices, each of which represents a potential point of failure.

End-host loss: Conventional wisdom maintains that the majority of packets are dropped when incoming traffic overruns the receiving end-host. With the NewAPI (NAPI) [25] enabled, the Linux kernel software network stack may drop packets in either of two places: when there is insufficient capacity on the receive (rx) DMA ring, and when enqueueing packets for socket delivery would breach the socket buffer limit (see Figure 3.4). In both cases, the receiver is overwhelmed and loss is observed, but they differ in the precise conditions that induce loss.

Cost-benefit of service: It may be the case that loss rates are typical of any large-scale networks, where the cost of immediately detecting and fixing failures is prohibitively high. For example, the measurements performed with the faulty network card at Indiana University persisted over at least a three month period.

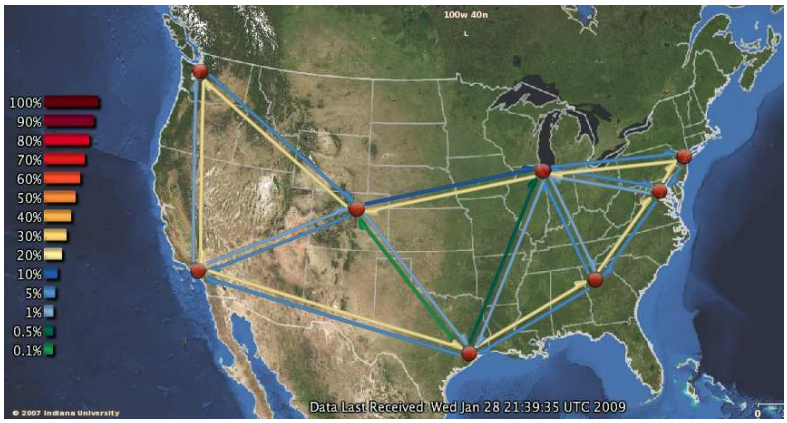


Figure 3.3: Test traffic on large NLR Ring, as observed by NLR Realtime Atlas monitor [29].

3.1.2 Cornell NLR Rings

Clearly, greater control is necessary to better determine the trigger mechanisms of loss in such uncongested lambda networks. Rather than probing further into the characteristics of the TeraGrid network, we chose instead to create our own network measurement testbed centered at Cornell University and extending across the United States; we call it the Cornell National LambdaRail (NLR) Rings testbed. In order to understand the properties of the Cornell NLR Rings, the reader should first consult the detailed description of our measurement infrastructure previously described in detail in Section 2.1.1.

Figure 3.3 illustrates the topology of the large path and highlights the network layer load on the entire NLR backbone while we performed controlled 2Gbps UDP traffic experiments over this path. Importantly, the Figure legend also demonstrates that the backbone (and our path) is uncongested. While our tests were performed, the large path, exclusive of the rest of the backbone, showed a level of link utilization of roughly 20%, corresponding directly to our test traffic.

3.2 Experimental Measurements

In this section, we use the Cornell NLR Rings testbed to answer the following questions with respect to the traffic characteristics over uncongested lambda networks:

- Under what conditions does packet loss occur, where does packet loss take place, and how is packet loss affected by NIC interrupt affinity? (Section 3.2.2)
- What is the impact of packet loss, path length, window size, and congestion control variant on TCP throughput? (Section 3.2.3)
- How does packet batching affect overall throughput and latency measurements? (Section 3.2.4)

3.2.1 Experimental Setup

Our experiments generate UDP and TCP Iperf [204] traffic between the two commodity end-host servers over all paths, i.e. between the *Ingress* and *Egress* servers depicted in Figure 2.1. We modified Iperf to report, for UDP traffic, precisely which packets were lost and which were received out of order. Before and after every experimental run, we read kernel counters on both sender and receiver that account for packets being dropped at the end-host in the DMA ring, socket buffer, or TCP window (see Figure 3.4). The default size of each receive (rx) and transmit (tx) DMA ring is 1024 slots, while the Maximum Transmission Unit (MTU) is set to the default 1500 bytes. Unless specified otherwise (Section 3.2.4), both NAPI and interrupt coalescence packet batching techniques are enabled. NAPI and interrupt coalescence are software and hardware techniques respectively, that batch process multiple received packets in a single operation in order to amortize per-packet processing overheads.

Throughout our experiments, all NLR network segments were uncongested—as a matter of fact, the background traffic over each link never exceeded 5% utilization (computed by the monitoring system [29] every 1-5 seconds). All values are averaged over multiple independent runs, and the error bars denote standard error—they are always present, most of the time sufficiently small to be invisible.

3.2.2 Packet Loss

To measure packet loss over the Cornell NLR Rings testbed, we performed many sequences of 60-second UDP Iperf runs over a period of 48 hours. We consecutively explored all paths (tiny, short, medium, and large) for data rates between 400Mbps to 2400Mbps, with 400Mbps intervals. We examined the following six different configurations of sender and receiver end-hosts (both identical in all cases): socket buffers sized at 1, 2, or 4MB; and use of either the `irqbalance` [21] daemon or static assignment of interrupts issued by the NICs to specific CPUs. The `irqbalance` daemon uses the kernel CPU affinity interface (through `/proc/irq/IRQ#/smp_affinity`) and periodically re-assigns (and balances if possible) hardware interrupts across processors in order to increase performance by spreading the load.

Figure 3.5 shows our measurements of UDP packet loss, with subfigures corresponding to different combinations of socket buffer size and bound versus balanced interrupts. Each subfigure plots packet loss observed by Iperf on the receiver end-host, as a percentage of transmitted packets, for various sender data rates across each of the Cornell NLR Ring; insets provide rescaled y-axes to better view trends. Packet loss is subdivided into three components denoting the precise location where loss can occur—as depicted in Figure 3.4. In particular, loss may be a consequence of over-running the socket buffer

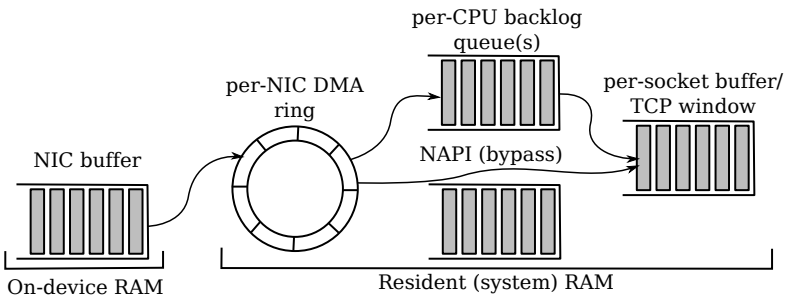
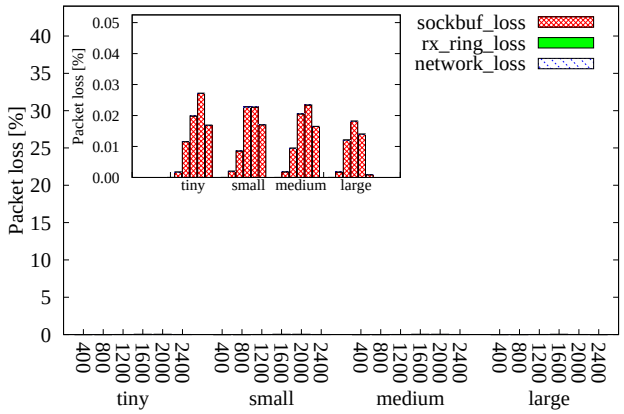


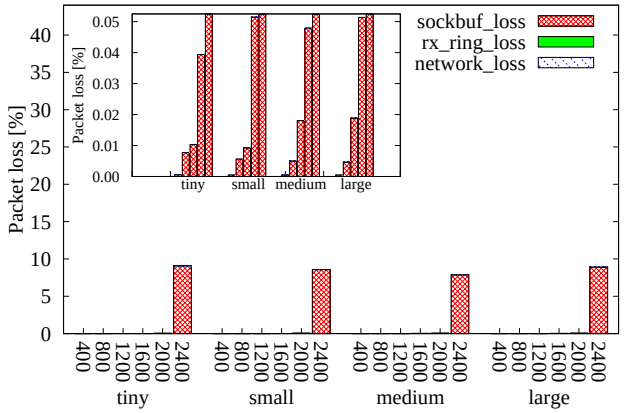
Figure 3.4: The path of a received packet through a commodity server's network stack. Packets may be dropped in either of each of the finite queues realized in memory: the NIC buffer, the DMA ring, the backlog queue, or the socket buffer / TCP window. Each queue corresponds to one kernel counter, e.g. `rx_ring_loss` is incremented when packets are dropped in the receive (rx) DMA ring. The transmit path is identical, with the edges reversed (i.e., packets travel in the opposite direction).

(`sockbuf_loss`), over-running the receive (rx) DMA ring (`rx_ring_loss`), or numerous factors within the network core (`network_loss`). Since NAPI is enabled, there is no backlog queue (to over-run) between the DMA ring and the socket buffer. Moreover, we dismiss the remaining possibilities for end-host loss for the following reasons: **i)** the sender socket buffer is never over-run during the entire 48-hour duration of the experiment—in accordance with the blocking nature of the socket API; **ii)** the sender transmit (tx) DMA ring is never over-run during the entire experiment; **iii)** neither the sender nor receiver NIC report any errors (e.g. corruption) or internal (on board) buffer over-runs throughout the experiment; **iv)** the receiver does not transmit any packets (since we used Iperf with UDP traffic).

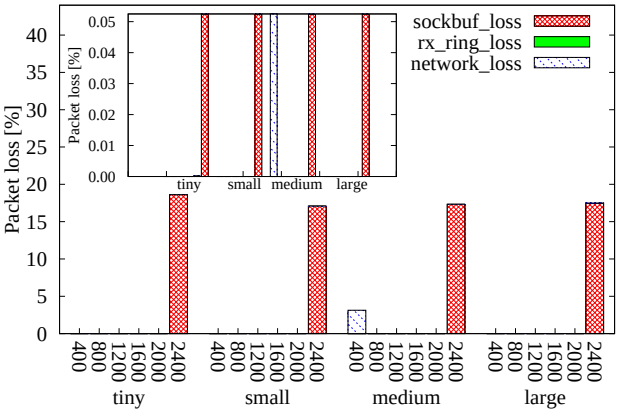
Interrupts via Irqbalance Figure 3.5(a) considers the scenario with the `irqbalance` daemon running and the socket buffer size set to 1MB. We observe *zero* loss in the network core; all loss occurs within the receiver's socket buffer. At rates beyond 2000 Mbps, irqbalance spreads the interrupts to many CPUs and the loss decreases. (Of note, omitted for space constraints, irqbalance with 2 and 4MB buffers result in zero loss for all tested data rates.)



(a) 1MB buffers, balanced interrupts



(b) 1MB buffers, bound interrupts



(c) 4MB buffers, bound interrupts

Figure 3.5: UDP loss as a function of data rate across Cornell NLR Rings: subfigures show various socket buffer sizes and interrupt options for balancing across or binding to cores; insets rescale y-axis, with x-axis unchanged, to emphasize fine features of loss.

Interrupts Bound to a Single CPU Figures 3.5(b) and 3.5(c) consider the scenario when we assign all interrupts from the NIC to a single core, with 1 and 4MB socket buffers, respectively. (The results for 2MB buffer, not shown, are identical to those of 4MB, but with $\sim 12\%$ packet loss for a sender data rate of 2400Mbps.)

There are three key points of interest. First, at 2400Mbps there is an abrupt increase in observed loss. Taking a closer look, we noticed that the receiver was experiencing *receive livelock* [169]. The onset of receive-livelock occurs when packets arrive at a rate that is larger than the interrupt processing rate. On a Linux 2.6 kernel, receive livelock can easily be observed as the network *bottom-half* cannot finish in a timely manner, and it is forced to start the the corresponding `ksoftirqd/CPU#` kernel thread. A typical interrupt service routine is split into a top and a bottom half: the top half cannot be interrupted itself and hence needs to quickly service the interrupt request, and the bottom half that performs the bulk of the operations in a deferred execution context and can be interrupted. The `ksoftirqd/CPU#` thread runs exclusively on the same CPU, and picks up the remaining work the bottom-half did not finish, acting as a rate limiter. As a result, the receive livelock occurs given that all interrupts (rx, tx, rxnobuff, etc.) were serviced by a single overwhelmed CPU—the same CPU that runs the corresponding `ksoftirqd/CPU#` and the user-mode Iperf task. The Iperf task is placed on the same CPU since the scheduler's default behavior is to minimize cache thrashing. Consequently, there is not enough CPU time remaining to consume the packets pending in the socket buffer in a timely fashion. Hence, the bigger the socket buffer, the more significant the loss, precisely as Figures 3.5(b) and (c) show.

Second, end-host packet loss increases with sender data rate, as visible in the Figure insets. Figure 3.5(b) corresponds to a relatively small buffer, 1MB, so the effect is clear. Figure 3.5(c) corresponds to a larger buffer (4MB) for which, with the exception of data

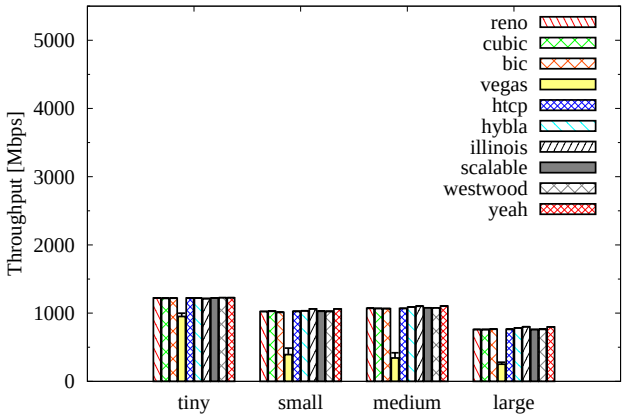
rates of 2400Mbps, there is a single negligible packet loss event along the tiny path at a data rate of 2000Mbps (almost unobservable on scale of Figure). Similarly, this trend is evident in Figure 3.5(a) (irqbalance on); however, at higher data rates, irqbalance spreads the interrupts to many different CPUs and the loss decreases.

Third, Figure 3.5(c) shows a particular event—the only loss in the core network we experienced during the entire 48-hour period, occurring on the medium path (one way latency is 68.9 ms) for a sender data rate of 400Mbps. During the course of the experiments, this was a single outlier that occurred during a single 60-second run. We believe it could have been caused by events such as NLR maintenance—we have experienced path blackouts due to various path segments being serviced, replaced, or upgraded.

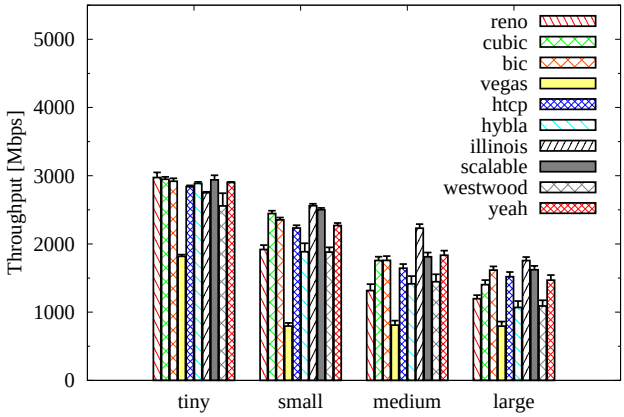
To summarize, the experiments show virtually no loss in the network core. Instead, loss occurs at the end-hosts, notably at the receiver. End-host loss is typically the result of a buffer over-run in the socket, backlog queue, or DMA ring. Unless the receiver is overloaded, a sufficiently large socket buffer prevents loss. NIC interrupt affinity to CPUs affects loss, and is pivotal in determining the end-host's ability to handle load graciously. Our experiments show that, at higher data rates, irqbalance works well (it decreases loss), whereas, at lower data rates, binding NIC interrupts to the same CPU reduces loss more than irqbalance. One benefit of binding all NIC interrupts to the same CPU stems from the fact that the driver (code and data), the kernel network stack, and the user-mode application incur less CPU cache pollution overhead.

3.2.3 Throughput

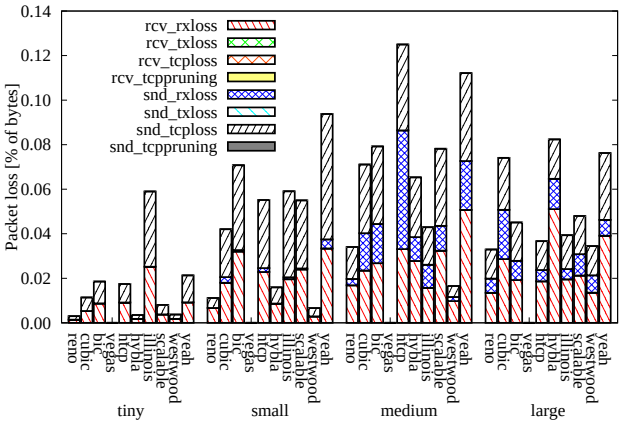
Although UDP is well suited for measuring packet loss rates and indicating where loss occurs, TCP [138] is the de-facto reliable communication protocol; it is embedded in



(a) TCP throughput for single flow



(b) TCP throughput for four concurrent flows



(c) Loss associated with four concurrent flows

Figure 3.6: TCP throughput and loss across Cornell NLR Rings: (a) throughput for single flow, (b) throughput for four concurrent flows, (c) loss associated with those four concurrent flows; TCP congestion control windows configured for each path round-trip time to allow 1Gbps of data rate per flow.

virtually every operating system's network stack. Many TCP congestion control algorithms have been proposed—Fast TCP [217], High Speed TCP [114], H-TCP [152], BIC [226], CUBIC [128], Hybla [81], Vegas [75], TCP-Illinois [156], Westwood [83], Compound TCP [202], Scalable TCP [143], YeAH-TCP [60]—and almost all have features intended to improve performance over high-bandwidth, high-latency links. The existence of so many variants indicate there is as yet no clearly superior algorithm.

To measure the achievable throughput, we used 120-second Iperf bursts to conduct a set of 24-hour bulk TCP transfer tests over all the Cornell NLR Rings; we examined all TCP variants available in the Linux kernel (except for TCP-LP, a low priority variant designed to utilize only the excess network bandwidth, and for TCP Veno, a variant designed specifically for wireless and cellular networks).

Figure 3.6(a) shows TCP throughput results for a single flow with window sizes configured with respect to each path round-trip time (RTT) to allow for a 1Gbps data rate. A higher window translates into larger amount of in-flight, not yet acknowledged data, which is necessary but not sufficient to yield high throughput on such high-latency, high-bandwidth links. In particular, a single TCP flow of 1Gbps requires a window of at least 2MB on the tiny path, 9.4MB on the short, 17.3MB on the medium, and 24.4MB on the large. Almost all TCP variants yield roughly the same throughput, with the exception of TCP Vegas that underperforms. TCP Vegas uses packet delay, rather than packet loss, as a signal for congestion and hence an indicator for the rate at which packets are sent. Furthermore, the performance of TCP Vegas depends heavily on accurate calculations of the round trip time value, which are perturbed by the packet batching techniques (see Section 3.2.4). No packet loss occurs for any of the single-flow TCP variants, yet throughput decreases for longer paths, even though the end-hosts have sufficiently large windows.

Since TCP window size is a kernel configuration parameter that requires superuser privileges for adjustment, typical user-mode applications like GridFTP [51] strive to maximize throughput by issuing multiple TCP flows in parallel to fetch / send data. To experiment with multiple flows, we issued four TCP Iperf flows in parallel in order to saturate each end-host's capacity and yield maximum throughput. Figure 3.6(b) depicts the throughput results. Note that the maximum achievable aggregate data rate is 4Gbps since each channel was configured with a window size corresponding to 1Gbps. Although the window sizes should be sufficient, the overall throughput decreases as the path length increases. Importantly, loss at the end-host does occur for multiple TCP flows. Moreover, some TCP variants yield marginally better aggregate throughput when competing with flows of the same type. The TCP throughput over the *tiny* path is identical to the maximum throughput achieved during control experiments (performed in the loopback configuration by directly linking the commodity servers with an optical patch cable), which means we reached the maximum performance for this particular commodity server.

Even though TCP is a reliable transport protocol, packet loss, albeit at the end-host, does affect performance [180]. TCP will resend any lost packets until the sender successfully acknowledges they were received. Figure 3.6(c) shows the percentage of packet loss corresponding to the TCP throughput in Figure 3.6(b). Unlike UDP loss, any analysis of TCP loss must account for retransmissions, selective and cumulative acknowledgments, different size of acknowledgments, and timeouts. Figure 3.6(c) shows percentage of loss *in bytes*, unlike UDP, for which packet count suffices since all UDP packets have identical size. Loss is reported both at the sender (denoted by `snd_`) and receiver (`rcv_`), within the DMA rings (`_txloss` and `_rxloss`), inferred by TCP itself (with `_tcploss`), and due to the inability of the user-mode process owning the socket to read the data in a timely fashion (`_tcppruning`).

Loss occurs solely in one of the following locations: the receiver's receive (rx) DMA ring (`rcv_rxloss`), loss that is then largely inferred by the sender's TCP stack (`snd_tcploss`), and finally, within the sender's receive (rx) DMA ring (`snd_rxloss`). The sender sends MTU-size (1500-byte) TCP data packets and receives TCP empty (52-byte) payload acknowledgments (ACKs), as 20-byte IP header + 20-byte TCP header + 12-byte TCP options.

There are two key observations. First, loss occurs at the end-host in the rx DMA rings—the receiver will drop inbound payload packets, while the sender will drop inbound ACK packets. Recall that the NIC is configured to a default value of 1024 slots per DMA ring. The socket buffer is essentially the TCP window; hence, it is adjusted to a large value in this experiment. Second, there are far more ACK packets (`snd_rxloss`) being lost than payload packets (`rcv_rxloss`). However, since ACKs are cumulative, TCP can afford to lose a significant portion of a window worth of ACKs on the rx DMA ring, provided that a single ACK with an appropriate (subsequent) sequence number is delivered to the TCP stack. Note that there is no loss observed by TCP Vegas since its low throughput is insufficient to induce end-host loss, a scenario identical to the one already described in Figure 3.6(a).

Our experiments show that as path length increases, more data and, importantly, more ACKs are lost since the TCP windows are enlarged to match the bandwidth delay product of the longer paths. This affects performance, and throughput decreases as the path length increases.

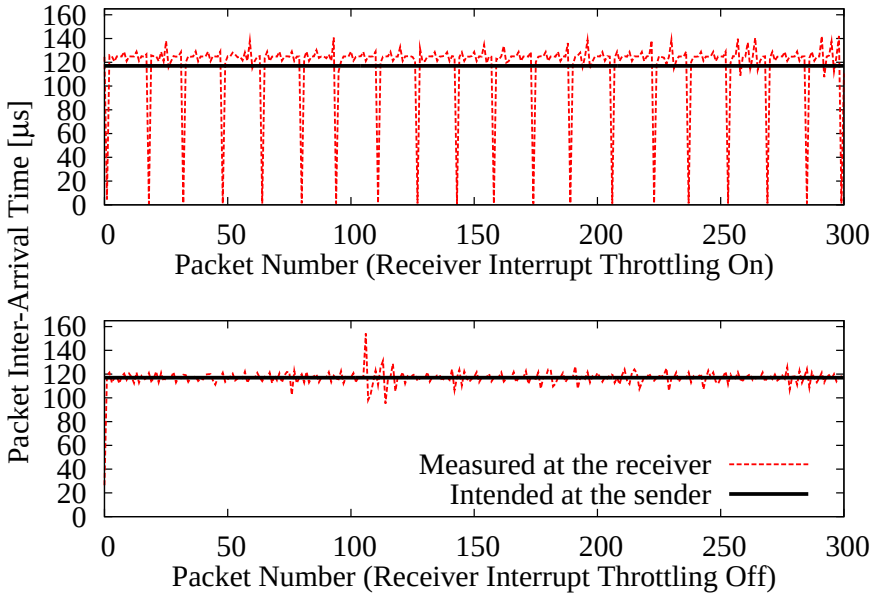


Figure 3.7: Packet inter-arrival time as a function of packet number; NAPI disabled.

3.2.4 Packet Batching

In this section, we look closely at the impact of packet batching techniques on the measurements reported above.

A CPU is notified of the arrival and departure of packets at a NIC by interrupts. The typical interrupt-driven commodity kernel, however, can find itself driven into a state in which the CPU expends all available cycles processing interrupts, instead of consuming received data. If the interrupt processing overhead is larger than the rate at which packets arrive, *receive livelock* [169] will occur. The interrupt overhead consists of two context switches plus executing the top half of the interrupt service routine. The typical solution is to batch packets by parameterizing the NIC to generate a single interrupt for a group of packets that arrive during some specified time interval. For example, Intel NICs offer an Interrupt Throttling configuration parameter that limits the maximum number of interrupts issued per second. If the device supports it, the kernel can take it one step

further by functioning in NAPI [25] mode. Instead of being exclusively interrupt-driven, a NAPI kernel is interrupt-driven at low data rates, but switches to polling at high data rates.

Packet batching techniques provide the benefit of an increase in maximum achievable bandwidth for a particular commodity architecture. For example, with NAPI and Interrupt Throttling disabled, the maximum achievable TCP throughput on our setup is approximately 1.9Gbps, in control experiments on the loopback topology with the end-host servers directly connected to each other. With NAPI enabled and Interrupt Throttling set to default parameter values, we achieved around 3Gbps throughput, as shown in Figure 3.6(b). By default, the NICs in our experiments implement Interrupt Throttling by limiting interrupts to a rate of 8000 per second.

However, this does not mean that packet batching is ideal in all scenarios, even though vanilla kernels and drivers enable batching by default. To illustrate this, consider a standard metric provided by high-end Ethernet test products [22]—the packet inter-arrival time, also known as packet dispersion. To perform this type of measurements, we patched the Linux kernel to time-stamp every received packet as early as possible (in the driver's interrupt service routine) with the CPU time stamp counter (TSC) that counts clock cycles instead of the monotonic wall-clock, thereby achieving cycle (i.e. nanosecond) resolution. Our implementation overwrites the `SO_TIMESTAMPNS` socket option to return the 64-bit value of the TSC register. For the TSC time-stamp values to be monotonic (a validity requirement), they must originate from the same CPU. This means that all NIC interrupts notifying a packet arrival must be handled by the same CPU, since received packets are time-stamped in the interrupt service routine.

Figure 3.7 shows the packet inter-arrival time for a UDP Iperf experiment consisting of a sequence of 300 packets at a data rate of 100Mbps (about one packet every 120

μs) with and without Interrupt Throttling enabled and NAPI disabled. We see that the interrupt batching superimposes an artificial structure on top of the inter-arrival times, thereby yielding spurious measurement results. This phenomenon may have significant consequences. For example, tools that rely on accurate packet inter-arrival measurements to estimate capacity or available bandwidth yield meaningless results when employed in conjunction with packet batching. The TCP Vegas variant also may be affected, since it relies on accurate packet round trip time measurements.

Although it is beyond the scope of this thesis, we have already started to investigate how to perform accurate timing measurements of packets without the interference from commodity end-host hardware and software. The Software Defined Network Adapter (SDNA) [115] is a device that transmits packets by modulating 1/10GbE software generated bitstreams onto optical fiber by means of a laser and a pulse pattern generator, and receives packets by sampling the signal with an optical oscilloscope. By relying on the precisely calibrated time-base of the optical instruments, SDNA achieves six orders of magnitude improvement in packet timing precision over endpoint measurement and three orders of magnitude relative to prior hardware-assisted solutions. Using the SDNA, we showed that as it traverses an uncongested lambda network, an input flow with packets homogeneously distributed in time becomes increasingly perturbed to such an extent that, within a few hops, the egress flow has become a series of minimally-spaced packet chains. Interestingly, this phenomenon occurs irrespective of the input flow data rate, and may cause packet loss at the commodity end-host server.

3.2.5 Summary of Results

Our experiments answer two general questions with respect to uncongested lambda network traffic. First, we show that loss occurs almost exclusively at the end-hosts as opposed to within the network core, typically a result of the receiver being over-run. Second, we show that measurements are extremely sensitive to the configuration of the commodity end-hosts. In particular, we show that:

- UDP loss is dependent upon both the size of socket buffers and DMA rings as well as the specifics of interrupt affinity in the end-host network adapters.
- TCP throughput decreases with an increase in packet (data and acknowledgment) loss and an increase in path length. Packet loss also increases with an increase in TCP window size. The congestion control algorithm is only marginally important in determining the achievable throughput, as most TCP variants are similar.
- Built-in kernel NAPI and NIC Interrupt Throttling improve throughput, although they are detrimental for latency sensitive measurements. This reinforces the conventional wisdom that there is no "one-size-fits-all" set of parameters, and careful parameter selection is necessary for the task at hand.

Although this chapter limits itself to measurement, we should note that this thesis proposes a practical way to overcome poor end-to-end performance. In particular, we show that a packet processor perimeter middlebox (or a performance enhancement proxy) can significantly improve end-to-end throughput in the face of packet loss. We achieved this through a combination of Forward Error Correction (FEC) [63] at line speed and TCP segment caching which transparently stores and re-transmits dropped TCP segments without requiring a sender retransmission to travel across the entire net-

work to reach the destination [62]. Additionally, we greatly increased both the performance and reliability of wide-area storage using such a technique [215].

3.3 Discussion and Implications

In this chapter, we used our Cornell National LambdaRail Ring testbed to methodically probe the end-to-end behavior of 10GbE networks connected to commodity end-host servers to send and receive traffic. Surprisingly, we observed significant penalties in end-host performance and end-to-end dependability in this scenario, consistently measuring packet loss at the receiving end-host even when traffic was sent at relatively low data rates. Moreover, such effects were readily instigated by subtle (and often default) configuration issues of these end-hosts—socket buffer size, TCP window size, NIC interrupt affinity, and status of various packet batching techniques, with no single configuration alleviating observed problems for all scenarios. Interestingly, there was no difference between the congestion control variants employed.

As optical networking data rates continue to outpace clock speeds of commodity servers, more end-to-end applications will invariably face similar issues. This empirical study confronts the difficulty of reliably and consistently maximizing the performance of such networks, which brings forth two consequences pertinent to this thesis. First, new network protocols are required and are being constantly developed to overcome the performance issues faced by commodity servers communicating over such high-bandwidth networks. Therefore, support for building efficient packet processors is crucial to fully utilize the networking substrate of modern datacenters. Second, the study reveals that user-space applications, like the Iperf used in this study, built on top of conventional operating system abstractions, like the socket, and running on commodity servers suffer

from significant performance penalties when communicating over a lambda network. Conventional packet processing applications rely on the same operating systems abstractions (namely, the socket—although packet processors use raw sockets instead of endpoint ones), therefore new packet processing abstractions are required in order to build high-performance packet processing applications that run on commodity servers.

This thesis continues by introducing new operating system abstractions that enable developers to build high-speed packet processing network protocols that run in software, in user-space, on commodity servers, and ultimately improve the performance of datacenter communication.

CHAPTER 4

PACKET PROCESSING ABSTRACTIONS I: OVERCOMING OVERHEADS WITH FEATHERWEIGHT PIPES

Enterprise networks of all kinds are increasingly dependent upon technologies that enable real-time data processing at line rates, for tasks such as lawful intercept, targeted advertisement, malware detection, copyright enforcement, and traffic shaping / analysis [131, 57]. Examples include deep packet inspectors (DPI), wide-area performance enhancement proxies (PEP) [72, 207], protocol accelerators, overlay routers, multimedia servers, security appliances, intrusion detection systems (IDS), and network monitors, to name a few.

Today, such applications are expensive, and implemented as in-kernel software running over dedicated hardware. For example, commercially available PEP, DPI, and WAN Optimization appliances [31, 36, 27, 12, 9, 32] typically cost tens to hundreds of thousands of dollars and offer a rigid proprietary technology solution. As high-speed networked applications become more pervasive, the ability to implement them inexpensively with user-space software on commodity platforms becomes increasingly vital. Writing and debugging such applications in user-space is simple and fast, and running them on commodity machines can slash deployment and maintenance costs by an order of magnitude. Unfortunately, such packet processing applications that run on modern OSes and multi-core hardware are rarely able to saturate modern networks [168, 93].

The key barrier to running high-speed networked applications on commodity hardware and OSes is the memory subsystem (i.e. the "memory wall")—multi-core systems create tremendous demand for memory bandwidth [194, 95, 132]. (For a detailed exposition of the architecture of a modern commodity server refer to Appendix A.) In one sense, this is an old problem; for example, the scientific computing community has ex-

tensively dealt with memory bandwidth issues, which led to the development of domain-specific optimization schemes like ATLAS [219]. The advent of multi-core platforms has ensured that the memory subsystem is a bottleneck for a wide range of applications in general settings—including high-speed network-facing systems—for which existing data partitioning techniques do not work well.

Further, modern commodity OSes exacerbate the effects of the memory barrier for networked applications. Network packets follow a tortuous path from the device into the memory of a user process. Packets are copied and placed on queues multiple times, memory is allocated and paged in, locks are acquired, and processes are blocked and awakened, incurring the cost of system calls and context switches. The critical path of each packet has a dramatic effect upon processing throughput and latency, to the point where it is nearly impossible to write user-space applications that perform non-trivial per-packet processing and can saturate gigabit links. Worse, such processing may be less effective on a multi-core machine than on a single processor due to excessive contention. For example, it has been shown that commodity network stacks are slow and bulky due to general contention for shared data structures in the kernel, and run poorly on multi-core machines [194, 121, 74].

This chapter reports on the implementation of the *featherweight pipes* (fwP)—a fast, streamlined communication channel between the kernel and user-space. The fwP is custom designed to enable the execution of high-speed packet processing application in user-space, while leveraging multi-core hardware. The fwP provides the convenience of standard user-mode processes—safety, ease of development / debugging, and large memory address spaces—while achieving levels of performance previously available only with specialized, dedicated appliances.

Unlike previous high-performance networking approaches [182, 118, 56, 209, 208,

69, 106, 181], zero-copy is *not* the center-piece. Interestingly, current multi-core / SMP systems have sufficient aggregate cycles to spare for copying data at least once more, especially if the data is resident in cache [95]. Instead, our work in this chapter focuses on mitigating the memory pressure issued by multiple cores and DMA (Direct Memory Access) devices. In particular, we focus on reducing memory bus traffic—cross-core cache coherency messages and host main memory access. We borrow from early pioneering work concepts such as streamlined I/O data paths, short-circuiting the conventional network stack, and shared circular buffers between the kernel and user-space.

Furthermore, we chose to target a very narrow class of network-centric applications, namely packet processors. This enabled us to provide a limited, yet extremely fast multi-core aware packet processing application programming interface (API). This new API does not supersede nor replace the extant conventional socket API, instead it coexists with it. Moreover, fwP is relatively easy to deploy and use on commodity systems—unlike U-Net [208] and its variants, fwP does not require specialized hardware and can be installed via a simple modular addition and a kernel patch (fwP also requires a kernel compilation) to commodity Linux, and runs over ubiquitous Ethernet interconnects.

The contributions of this chapter are as follows:

- We show that memory pressure, and in general the memory subsystem, is the fundamental bottleneck a shared-bus multi-core machine faces when performing per-packet processing at line speed.
- We present a novel kernel to user-space communication channel, enabling packet exchange with minimal kernel involvement and overhead. The featherweight pipes channel is aggressively designed to mitigate memory pressure and contention costs, while leveraging the power of multi-core processors.
- We implement and evaluate a security appliance, a protocol accelerator, an IPsec

gateway and an overlay router with fwP. Each has comparable performance to in-kernel implementations and significantly greater performance than applications developed with user-space libraries such as libpcap [38, 221] that rely on conventional operating abstractions like the raw socket.

4.1 Challenges

Developing high-performance networked applications in user-space on a commodity OS is a difficult and challenging task. The key problem in modern multi-core machines stems from the fact that the memory subsystem is slower than the processors (refer to the von Neumann bottleneck in Appendix A) and cannot cope with the demands of multiple cores and DMA devices. The pressure on the memory subsystem results primarily from three sources: the *"memory wall," data contention*, and *raw bus contention*. The memory wall describes the current integrated circuit status quo in which the data transfer rate between memory and CPUs is significantly smaller than the rate at which a CPU operates, data contention is when multiple cores introduce sharing overheads by accessing data concurrently, while raw bus contention is when many devices issue requests to the memory controller(s). All are exacerbated by inefficient OS design yielding excessive memory accesses at inopportune times. Moreover, network stacks are subject to other sources of overhead that impact packet processing latency and indirectly stress the memory subsystem even further—namely, *blocking*, *system calls*, and *context switches*. In this section we examine these overheads and define a road-map to mitigating them.

4.1.1 Overheads

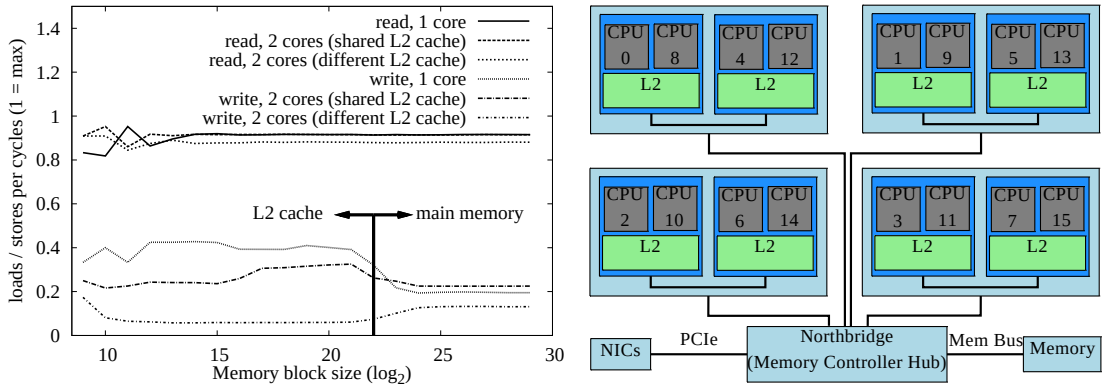
Memory Wall and Data Contention

Although recent emphasis has shifted away from CPU clock speed [201], there remains a serious mismatch between processor clock rates and memory latencies. This disparity has forced chip architects to explore a number of performance-enhancing optimizations. However, none of them eliminate the underlying problem.

The first optimization has been to aggressively increase cache sizes across all levels. This technique continues to be used to date, and has been used since the early CPU designs. Unfortunately, large caches are useless in the face of data contention. For example, thread-safe symmetric multiprocessing (SMP) code acquiring locks will bounce the lock variable itself between CPUs.

Another notable change was to limit complexity, unless it results in a significant increase in SMP performance. For example, deep pipeline (20+ stage) CPUs, like the Intel Pentium 4, were replaced in subsequent processor generations by more "reasonable" (14 stage) designs since pipeline flushes and stalls caused by branch mispredictions and atomic instructions became prohibitively expensive. An example of atomic instructions are atomic memory updates (e.g., instructions that store into a memory location and are prefixed by the `lock` assertion on Intel x86 architectures). Such atomic instruction support is typically implemented by expensive locking of the memory bus or freezing the cache—should the memory being accessed be resident in the cache. Importantly, SMP code makes extensive use of atomic instructions to implement synchronization primitives that are necessary for ensuring the correctness of concurrent executions.

The most noteworthy advancement in recent years has been placing multiple cores



(a) Raw cache performance

(b) Four socket quad core (Xeon) machine

Figure 4.1: Four socket quad core (Xeon) cache performance and architecture.

on the same silicon die, and was motivated by several factors [178]: energy and cooling constraints of current semiconductor technology have limited the frequency scaling of single processors, instruction level parallelism (ILP) cannot be easily extracted any longer from programs, and the complexity required to implement ILP on the chip yields diminishing returns and fundamentally caps the performance of CPUs. Placing many simple cores on the same die instead means that each core can scale well since the physical constraints such as silicon real-estate and frequency caps due to wiring are loosened. Likewise, multi-core chips potentially simplify and reduce the costs of the MESI (Modified, Exclusive, Shared, Invalid) cache coherency protocol, and in particular, expensive read for ownership (RFO) inter-processor bus messages.

However, none of these techniques eliminate the underlying problem. Today, virtually any commodity machine is a real SMP system, and hence kernel code, and in particular the network stack, runs simultaneously on multiple cores. Conventional wisdom, and Amdahl's law, states that when adding processors to a system, the benefit grows at most linearly, while the cost (memory / bus contention, cache coherency, serialization, etc.) grows quadratically. To make matters worse, the network stack is littered with shared data, locks, memory barriers and queues that are usually implemented as

linked lists—a data structure notorious for cache thrashing.

To harness the power of multi-core processors, it is vital to reduce the load on the memory subsystem. In particular, solutions must keep cores busy for relatively long periods of time working on independent tasks that touch little or no shared data. Ideally, solutions would have one writer per cache line and use as few locks as possible.

To get a rough idea of the costs of cache contention, we ran a simple experiment, in which a block of data is concurrently read or written by threads pinned to distinct cores. Figure 4.1 (a) shows the results for the quad core Intel Xeon X5355 (Core) system presented in Section 2.2. The system has 8MB of L2 cache, but since the X5355 bundles a pair of two dual core CPUs on the same chip as depicted in Figure 4.1 (b), the effective size of the L2 cache for any one core is 4MB. We measured the number of load / store operations per cycle for all power of two block sizes ranging from 512 bytes to 512MB. This particular CPU can service a single load operation per cycle, hence a ratio of one is ideal for a memory-bound workload. There are no spinlocks or memory barriers (e.g. bus `lock` prefix), hence the system achieves the upper bound.

Note that read performance does not degrade, even when the data no longer fits in the L2 cache. The lack of performance degradation is due to the prefetching mechanism of the Xeon CPU (since data is traversed in ascending memory address order).[1] There is a small degradation in read performance when the cores do not share the caches, since the MESI protocol has to bounce the cache lines between the cores.

In contrast, write performance is significantly worse. Moreover, for cores that do not share the L2 cache, the performance is in fact worse when touching data in the L2 cache than when going directly to main memory (the L2 cache size is 4MB = 2^{22} on the

[1] Intel Xeon processors are sold to the higher-end server market—by contrast, for the lower end Intel Core 2 processor, the read performance plummets as soon as the data block is larger than the L2 cache.

x-axis). Cores that share the L2 cache exhibit (slightly) better performance.

Given this analysis, consider one of the most basic locking primitives of a SMP kernel—the *spinlock*. Although an indispensable interlocking building block of all modern operating system kernels, when contended for, spinlocks are detrimental for performance since the lock intrinsically has multiple writers, which means the writer cores enter expensive RFO cache coherency cycles. Moreover, a spinlock spends most of the time performing costly atomic updates (e.g. the spinlock built for the Linux kernel version 2.6.28 performs `lock`; `decb` instructions).

Raw Bus Contention

Consider the path of a typical packet entering a modern commodity operating system. The packet is first copied from the wire / NIC FIFO buffer into the kernel host memory—this happens by means of a DMA transfer, and memory pressure, e.g., memory bus contention (Figure 4.1 (b)), is still exerted even if the main CPU(s) is not utilized. The packet is then copied from the kernel buffer into a user-space buffer—this operation may block if a page fault is taken while copying. If conventional packet-capture and packet redirection mechanisms are used, like libpcap [38, 221] or libipq [24], the kernel buffer is duplicated before being forwarded or delivered to the appropriate socket; this (third) copy is enqueued for independent delivery into user-space. A packet follows the same sequence in reverse when the application sends data on the wire.

System Calls and Blocking

System calls have been a major source of overhead for high-performance applications running within user-space. Practically all conventional existing approaches to moving

data between kernel and user-space involve one or more system calls. The recently-introduced fast system call hardware instructions (SYSCALL/SYSENTER and SYSRET/SYSEXIT) reduce this cost, provided that all parties support them (CPU, OS, and library). For example, on a Pentium 4, the SYSENTER `getpid` system call is three times faster than the traditional variant based on a trap (or a synchronous exception). However, issuing one system call per packet remains rather expensive, especially when dealing with massive amounts of data available in short time frames. This is exacerbated by the blocking nature of the standard application programming interfaces (APIs) for packet send / receive (e.g. the socket API), which means that the kernel typically performs potentially expensive task wake-up / resume operations for every system call.

Scheduling / Context Switches

Boundary crossings and in particular context switches are also expensive operations—consider for example the boundary crossings involving the receipt of a network frame: hardware interrupt handler to softirq to context switch to return from system call. When processes block for resources (e.g., to time-share a CPU), they are context-switched out, when the resources become available, they are switched back in. In addition, the process incurs scheduling delays that depend on the scheduling policy used. Moreover, context switching often performs an expensive translation lookaside buffer (TLB) flush, and further has a latent cost in terms of cache-pollution.

4.1.2 Design Goals

Consequently, we designed an abstraction (the fwP) with the following objectives:

- Minimize data and raw bus contention across multiple cores with a three pronged approach: using cache friendly data structures, sharing L2 caches between producer and consumers, and streamlining the communication path and bypassing the conventional network stack.
- Attain near linear speedup for network processing applications in multi-core environments.
- Employ traditional overhead reduction techniques:
  - Block the user-space process only if there are no new packets to be processed.
  - Not use system calls for sending and receiving packets unless blocking / signaling is required.
  - Minimize context switches and rescheduling when receiving and sending packets.

More precisely, fwP tackles the overheads in the following fashion:

- The fwP design bypasses the conventional network path, uses cache-friendly data structures, and has a *single* spinlock per communication channel, used both in kernel and user-mode. By contrast, while processing a packet, the conventional network stack acquires multiple spinlocks.
- fwP reduces packet copies, although it is *not* a zero-copy but a 1-copy system—arguably more efficient for commodity OSes with protection domains [94, 96] running over Ethernet NICs [73]. To reduce raw bus contention, we perform only the necessary minimum amount of work in the kernel network stack—a shared global resource and potential hot spot—thereby conserving the memory bandwidth resources for the user-space application [192].

- fwP is designed to exchange data *without* issuing any system calls, while blocking occurs *solely* if there are no pending packets.
- fwP uses thread core-placement that minimizes context switches and rescheduling when exchanging packets, while at the same time reducing the impact of the cache coherency cross-talk.

4.2 Multi-Core and the fwP Design

Ahmdal's Law is more relevant than ever in a multi-core environment—in order to gain near linear benefits / speedup with the number of cores, we must reduce the overheads due to the data / bus contention. The featherweight pipes design takes advantage of multi-core CPUs while reducing memory pressure / contention. We bypass the conventional network stack, and provide a fast streamlined shared memory communication channel between the kernel and user-space. We reduce context switches significantly, and perform cache-aware placement of tasks on cores, thereby reducing the cache-coherency and memory contention penalties.

In this section we start by describing the mechanisms that comprise fwP, discuss some optimizations, and show an example of a security appliance developed using the fwP API. Next, we delve into technical details pertaining to the fwP internals and discuss some of the limitations. Finally, we summarize how fwP takes advantage of multi-core CPUs in order to maximize performance.

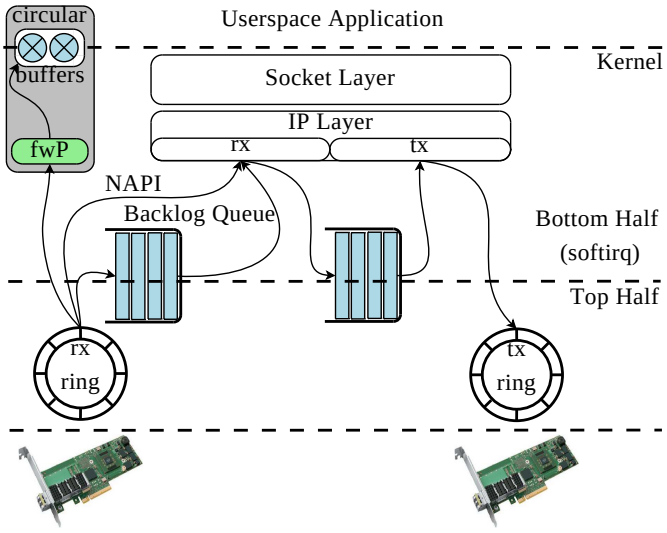
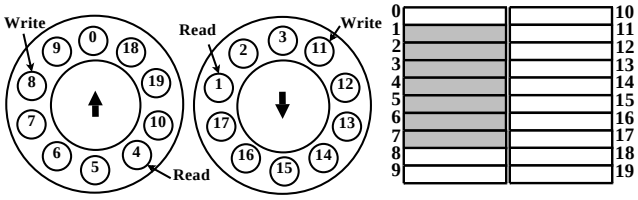


Figure 4.2: Linux network stack path of a packet forwarded between interfaces.



(a) "inbound and outbound" channels (b) memory layout

Figure 4.3: fwP buffers and shared memory layout.

4.2.1 fwP

The Conventional Path of a Packet

Figure 4.2 depicts the journey of a network packet through the Linux kernel network stack. Upon receipt of an Ethernet frame, the network interface card (NIC) performs a DMA cycle to place the frame directly in host kernel memory (rx ring in Figure 4.2), before issuing an interrupt. Alternatively, a single interrupt is issued to signal the arrival of several packets in a row if the NIC and/or the kernel performs batching (e.g. NAPI [169]). The network driver acknowledges the interrupt, enqueues the received

frame on a backlog queue, and enables a deferred execution context, or a *bottom half*, that will process the frame at a later time. (If NAPI mode is enabled, the backlog queue is bypassed, and frames are accumulated directly on the DMA rx ring.) In Linux terminology, the *top half* of a driver is the code that executes in the interrupt handler, while the *bottom half* runs outside it. To maximize IO performance, it is important that the top half execute as quickly as possible, deferring work for later completion by the bottom half, since the bottom half runs with all interrupts enabled to allow other interrupts to be serviced in a timely fashion.

In most cases, as with the networking stack, the bulk of the work is done in the bottom half. The Linux kernel implements the entire network stack using *softirq* bottom-halves, which are non-reentrant, interruptible contexts of higher priority than tasks, with no backing process control block. (A softirq is thus considered to be an interruptible interrupt context.) The softirq picks up every packet and pushes them along a battery of layers (e.g. IP and socket layers) before copying them into buffers provided by the user-space application and waking up the user-space task if necessary.

Instead of using the traditional network software stack, we designed a more direct, streamlined path for the packets, bypassing the conventional, bulky network stack and placing the frames on a fwP channel, as shown on the left-hand side of Figure 4.2. The packets are demultiplexed at an early point to avoid synchronization hot spots, and relieve memory pressure in general.

fwP channel and API

The fwP channels are a fast, streamlined communication channels between the kernel and the user-space. A fwP channel is *a pair* of unidirectional circular buffers residing on

Table 4.1: fwP API. All functions take a parameter indicating the direction of the buffer (in / out) and like traditional IPC, the IPC_NOWAIT flag indicating if the task should block or not while issuing the operation.

fwP channel interface:	
`fw_crte`	Create a fwP channel by applying a template over the shared buffer.
`fw_rcv`	Copy data from next channel read slot onto a buffer. Similar to the POSIX msgrcv. It advances the read index. Equivalent to `fw_get_r` + `fw_put_r`.
`fw_snd`	Copy data from a buffer onto next channel write slot. Similar to the POSIX msgsnd. It advances the write index. Equivalent to `fw_get_w` + `fw_put_w`.
`fw_flip`	Atomically swap a non-empty read slot from one direction with an empty write slot in the other direction. Typically used for user-space in-place buffer modification and return into the kernel.

zero-copy **fwP channel interface:**	
`fw_get_r`	Get index of the next channel read slot. Slot can be manipulated in place. Used in conjunction with `fw_put_r`. Read index is *not* advanced.
`fw_put_r`	Release the next channel read slot. Used in conjunction with `fw_get_r`. Read index is advanced.
`fw_get_w`	Get index of the next channel write slot. Slot can be manipulated in place. Used in conjunction with `fw_put_w`. Write index is *not* advanced.
`fw_put_w`	Release the next channel write slot. Used in conjunction with `fw_get_w`. Write index is advanced.
`fw_buf`	Compute buffer pointer based on index.

a shared memory region, one channel for data going from the kernel to the user-mode task and the other channel for data going in the opposite direction. Figure 4.3 shows the data structure template over the shared memory region, while Figure 4.4 depicts the memory layout of a task. The kernel places data on the "inbound" slot and fetches data from the "outbound" slot, advancing the appropriate read and write pointers. In turn, the user-mode task performs the dual operations—fetching data from the "inbound" slot and placing data on the "outbound" slot.

In contrast to conventional circular buffers [222, 99, 100, 94, 213], fwP channels are different in that they consist of a tandem of tightly coupled circular buffers holding *position independent* pointers to frame buffers, all of which are placed on a shared memory region. This allows operations like atomic pointer swapping between the rings, therefore enabling zero-copy receive, in-place processing, and forwarding of packets. Moreover, by using fixed size channel buffers we avoid chains of linked-list queues which would degrade cache performance.

The fwP channel has the same memory pages mapped by both the kernel and the user-space task, in a fashion somewhat similar to the System V IPC (Inter Process Communication) shared memory mechanism. However, the pages are proactively faulted and pinned into physical memory, and are charged to the user-space task's address space—hence the channel resources are automatically reclaimed at process `exit`.

We overload the `shmget`, `shmat` and `shmdt` system calls to create, attach, and release the channel shared memory region, and we altered the process control block to maintain a list of fwP channels.

The fwP API described in Table 4.1 was designed to allow data exchange without issuing system calls—every function has a fast path on which no system calls are performed. The slow path of the API functions may need to block the current task, e.g. if a channel is empty, or conversely may need to wake up tasks that have been waiting for events. We implemented two system calls, `fw_wakeup` and `fw_sleep`, to perform this functionality. Each direction of the channel is protected against concurrent accesses by a per-channel spinlock embedded within the fwP channel data structure. In particular, `fw_wakeup` and `fw_sleep` are called holding the appropriate spinlock—the *only* synchronization point on the path between the fwP bottom half and user-space. (The current API fast path & spinlock design requires that the Linux kernel be preemptive, i.e. compiled with `CONFIG_PREEMPT` support.)

Optimizations

While blocked, a process does not get work done. Moreover, a process blocking while taking page faults can have disastrous performance, especially if it needs to process large volumes of network traffic in a timely fashion. Ideally, a task would only block

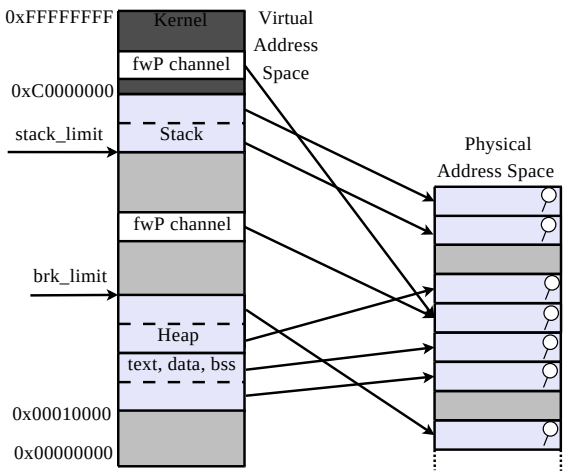


Figure 4.4: Memory layout of a (32bit) process / task.

when performing initializations (e.g. calling `shmget`, `shmat`), or in situations where the system has no more incoming data, or no room to store outgoing packets.

Consequently, we provide an optional system call, `ffork`, that allows a task to *proactively* request and commit needed memory resources up front, trading higher start-up costs for low and predictable runtime latencies. Note that if start-up and warm-up performance is not an issue, `ffork` need not be used. To enable this behavior without breaking the `malloc` and sister functions, we modified the `brk` and `mmap` system calls.

Unlike the conventional `fork`, `ffork` takes in two parameters—the sizes of the heap and the stack. These values are enforced as limits for the address space of the newly forked task. (Memory resources are bounded and typically known in advance for applications that process a continuous transient stream of data.) The task's break value is first copied into a shadow break point (held into the process control block) and then set to the limit. `ffork` then proceeds to walk the entire address space, including the text, data, heap and stack and pin it into memory, shown in Figure 4.4.

A process may use any of `mmap` and `brk` to solicit an address space increase. While

`brk` increases the heap in a contiguous manner, `mmap` can return a memory region mapped anywhere in the virtual address space. To simplify memory region accounting and eliminate a kernel synchronization point, we modified the `mmap` system call to prevent a `ffork`'ed task from increasing its address space. Library allocators work by using a combination of `brk` and `mmap` system calls to request for address space increase; if one method fails they default to the other. The library allocator is thus forced to use the `brk` system call exclusively. When this happens, the `brk` system call uses the shadow break value to satisfy requests without blocking, returning a region of memory that is already resident.

There is a subtle difference between our approach and the `mlock/mlockall` system calls that lock current and future pages of a process in resident memory. After a `mlock/mlockall` is issued, subsequent `malloc` requests may still block, and the caller process may also take expensive page faults in the future—not to mention the fact that page faults continue to be taken automatically whenever the stack grows.

Application Example

Figure 4.5 shows pseudo code for a security application developed using fwP. The figure illustrates a number of key characteristics. First, the program is written in C, in a conventional fashion. Second, a relatively small amount of familiar setup code is required to create an application: the pseudo code in the figure serves as a template for creating real packet-processing applications. Third, standard compilers and debuggers can be used to build and test this code.

```
1: #include "fwp.h"
2:
3: int main(void) {
4:   int heap_sz = 64*ONE_MEG;
5:   int stack_sz = 8*ONE_MEG;
6:
7:   // Fork process and pin in memory
8:   child_pid = ffork(heap_sz, stack_sz);
9:   if(child_pid == 0) {
10:
11:   // shared memory region with kernel
12:   id = shmget(IPC_PRIVATE, map_len,
13:         (SHM_FWIPC | IPC_CREAT));
14:   map = shmat(id, 0, SHM_FWIPC);
15:
16:   // apply buffer template
17:   chan = fw_crte(map);
18:
19:   // register filter w/ kernel module
20:   filter.proto = IPPROTO_TCP;
21:   filter.dst_port = htons(80);
22:   fw_register(id, chan, &filter);
23:
24:   for(ever) {
25:    // Manipulate buffer in place
26:    peek_idx = fw_get_r(chan, IN, 0);
27:    buf = fw_buf(chan, peek_idx);
28:
29:    if(cmpsig(buf, signature)!=NULL){
30:      // cmpsig can be memmem
31:      alert_msg = get_alert_msg(buf);
32:
33:      // Release slot, do NOT forward
34:      fw_put_r(chan, IN);
35:
36:      fw_snd(chan, OUT, alert,
37:             len(alert), flags);
38:      syslog(LOG_INFO, "%s", alert);
39:
40:    }else {
41:       // Forward packet by moving
42:       // it from IN to OUT channel
43:       fw_flip(chan, IN);
44:    }}}
45: exit(0);
46:}
```

Figure 4.5: Pseudo code for a security fwP application.

4.2.2 fwP Under the Hood

Packet Demultiplexing

Up to now, we have described the system as if only one task were running. However, to use many cores, a system will have many potentially multi-threaded tasks running in parallel, each using one or more fwP channels. (We modified the `clone` system call such that threads share the fwP channel descriptors.) Accordingly, the fwP *bottom half* (left side of Figure 4.2) demultiplexes received IP datagrams before placing them on appropriate channels.

The fwP bottom half listens for registration requests from fwP user-mode tasks. A registration request consists of a fwP channel identifier, as returned by the `shmget`

syscall, and a rudimentary packet filter / classifier rule—for example in Figure 4.5 lines 20 and 21 indicate that the protocol is TCP, and destination port is 80.

For each received IP packet, fwP seeks a match with a filter in the order in which filters were registered, and upon success places the packet on the corresponding channel. The fwP bottom half creates one kernel thread per fwP channel used to receive packets from user-space, route and enqueue them for transmission.

Core Affinity

The fwP bottom half creates one kernel thread per fwP channel so as to perform thread binding per core in a flexible, memory, cache-aware, and dynamic fashion. In particular, the kernel and user-mode threads touching data on the same fwP channels are placed on cores that share the L2 caches to reduce the cache coherency penalties. Moreover, the threads are exclusively assigned to cores, thereby preventing tasks from being re-scheduled on different CPUs—this reduces scheduling overheads since producer and consumer tasks work in parallel on different cores.

Limitations

In our current system, multiple threads cannot use the zero-copy API (Table 4.1) concurrently, since they could alter the circular buffer indices in an inconsistent fashion. If multiple threads are to be used, there are generally two options: each thread registers their private fwP channel with the same filter, in which case the kernel module will perform a round robin load balancing over the channels interested in packets of the same type, or data can be copied from the shared fwP channel onto user-space buffers using `fw_rcv`.

The fwP API was specialized for a narrow breed of applications and may not fit all packet processing types. For example, while developing TCP and UDP stacks on top of the fwP API we found that using the zero-copy functions was almost impossible for tasks like TCP and IP reassembly—we let the kernel stack perform the latter before placing the datagrams on channel buffers.

4.2.3 Taking Advantage of Multi-Core CPUs

The overall goal of our system is maximize throughput by having multiple cores run concurrently, while at the same time minimize the pressure exerted over the memory subsystem. The fwP achieves this by:

- Providing a streamlined path for packets by sidestepping the conventional IP network stack. Packets are demultiplexed at an early point, bypassing the default, bulky in-kernel network stack, and hence we avoid synchronization hot spots, and relieve memory pressure in general.
- Using shared fixed size fwP channel buffers. We avoid cache-degrading chains of linked-list queues and reduce system call overheads. This essentially allows the application to exchange data with the kernel without issuing system calls. Moreover, our narrow interface minimizes blocking overheads.
- Using page flipping between unidirectional circular buffers to reduce additional copies.
- Placing threads that touch data on the fwP channels on cores that share the L2 cache, to reduce the cache coherency penalties. Moreover, this technique significantly reduces the scheduling overheads, since producers and consumers work simultaneously in parallel on different cores.

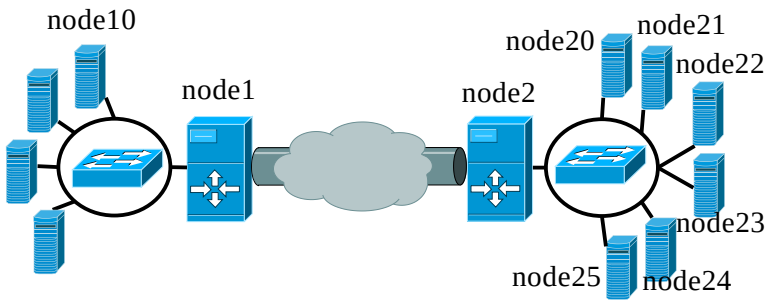


Figure 4.6: The Emulab (DETER) experimental topology.

4.3 Experimental Evaluation

We evaluated our system on the DETER Emulab [220] testbed using the topology depicted in Figure 4.6. Each machine is a Dell PowerEdge 2850 with single 3.0GHz Pentium 4 Xeon processors (with Hyperthreading enabled), 2 GB of memory and 5 Intel PRO/1000 MT adapters, running the 2.6.20-16 Linux kernel patched with fwP support. We emulate inter-node link latencies between nodes 1 and 2, if any, using Emulab's traffic shaping mechanisms. Throughout this section we shall refer to nodes in the topology by name, as shown in Figure 4.6.

All links have a capacity of 1Gbps. This topology is intended to represent two interacting datacenters—node1 and node2 at the perimeter of their respective datacenters would typically be running packet processor applications like protocol accelerators and security appliances.

Unless mentioned otherwise, we ran 120 second end-to-end Iperf [204] TCP flows (with the Reno congestion control variant) with traffic flowing through nodes 1 and 2. Whenever possible, we compare fwP with in-kernel and user-space, potentially multi-threaded, implementations of the same functionality. Out of the conventional user-space alternatives, the `pcap` [38, 221] (Packet CAPture) mechanism

has been the only viable out-of-the-box user-space option since at high data rates, `libipq/libnetfilter_queue` [24] consistently crashed with a "Failed to receive netlink message" fault. Note that Click-userspace [147] is built atop pcap.

The fwP consists of 1897 lines of kernel patch code, 1508 lines of kernel module code and some 4488 lines of library code, of which 832 are AES / IPSec ports.

We begin the evaluation by showing that real world applications, like a deep packet inspection security tool, an overlay router, a protocol accelerator, and an IPsec gateway can be implemented in user-space with fwP and vastly outperform conventionally-built (i.e., pcap based) user-space counterparts. We then report the results of several micro-benchmarks that reveal the overheads of modern commodity operating systems and libraries. We show the extent to which the featherweight pipes mechanism mitigate memory pressure. Most importantly, our experiments show that we can take advantage of emerging multi-core architectures.

4.3.1 Real World Applications

We show four examples that demonstrate how developers can write user-space applications while at the same time reducing memory pressure and overheads, thus allowing the applications to take advantage of multi-core chips. In particular, we show that:

- A deep packet inspection security appliance built with fwP and using a single core reduces conventional overheads significantly.
- The fwP interface is flexible and can be extended to provide a highly efficient multi-send operation, enabling a user-space gigabit overlay router.

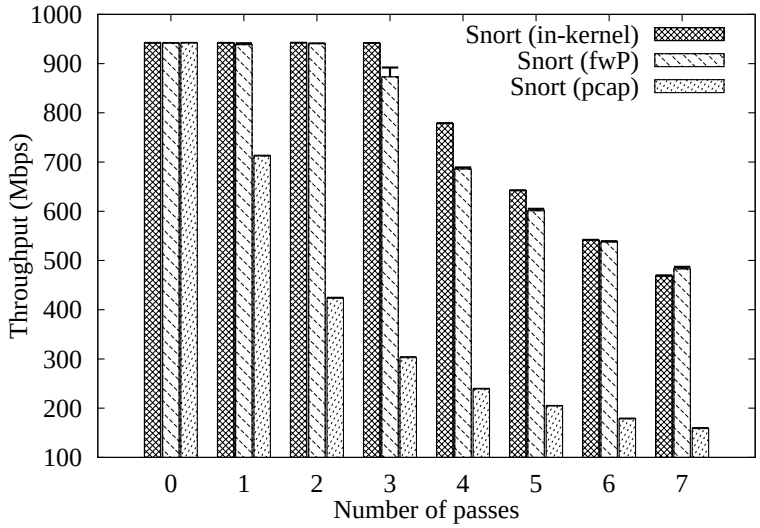


Figure 4.7: Snort deep packet inspection throughput.

- An IPsec gateway may use independent tasks to take advantage of multiple cores, reducing memory contention and achieving near linear scalability.
- Finally, putting it all together, we show that a complex, multi-threaded cooperative protocol accelerator achieves the same performance as the equivalent hand-tuned, optimized, in-kernel counterpart.

Deep Packet Inspection

Deep Packet Inspection technologies enable network operators to throttle, inspect and shape protocol data in real time on potentially high-volume backbone networks. Network operators would benefit from the ability to deploy such technology over cost-effective commodity machines. Instead, they purchase expensive hardware appliances able to perform various degrees of traffic inspection and modulation. For example, the $800,000 PacketLogic PL10000 can operate full-duplex at 40Gbps.

We have built a simple DPI over the skeleton from Figure 4.5 that sifts through all

IP traffic and checks for a set of signatures (e.g., malware signatures). Our implementation is fairly straightforward, in that each packet received is matched against a set of signatures, one signature at a time; there is a single thread of control. For all practical purposes, it is precisely the `byte_test` functionality of the Snort [198] intrusion prevention and detection system.

Figure 4.7 shows the throughput plotted against the number of signatures, i.e. the number of memory-passes per packet. Zero passes means that packets are simply forwarded through. To memory-stress all variants, we do not compile and collapse all the signatures into one big non-deterministic finite automata (NFA) that accepts when any of the signatures is matched. The fwP implementation vastly outperforms the pcap implementation. The in-kernel implementation performs slightly better than the fwP since the signature matching is done early in softirq context.

Interestingly, the fwP version slightly outperforms the in-kernel implementation for seven passes. This is due to the fact that the softirq implementation spends too much time processing each packet and does not clean up fast enough the DMA rx ring (NAPI, Figure 4.2). As a result, packets are dropped, and the low priority ksoftirqd kernel thread picks up the remaining bottom half processing. By contrast, the fwP implementation has a producer (the fwP bottom-half) that cleans up the rx ring, and a consumer (the user-space application) that performs the matching, each running on distinct cores simultaneously.

Gigabit Overlay Multicast Router

Overlay networks have been employed as an effective alternative to IP multicast for efficient point to multipoint data dissemination across the entire Internet [148]. To prove

the versatility of fwP, we built a high performance multicast router that can operate at gigabit speeds from user-space. The key functionality within such a device is the ability to forward an incoming packet to multiple destinations rapidly and efficiently.

While the fwP API is designed for forwarding packets efficiently, it does not intrinsically support multi-destination sends; however, its simple design allows extended functionality of this nature to be easily layered above the basic interface. To support multi-destination sends, we implement a new operation called a *splace* over the fwP API. The user-mode task places the original packet and a set of destination addresses on the fwP buffer. The kernel helper module then iteratively grafts each destination address on the original packet, recomputes the checksums and enqueues the packet for transmission. As a result, the addition of the splace layer above the basic fwP API allows a task to batch multiple sends of a packet in a single non-blocking operation.

We implemented a simple IP and UDP multicast relay and we deployed it on the topology in Figure 4.6 such that node2 will multicast all IP inbound traffic to nodes 20, 21, 22, 23, 24, and 25. Node1 produces a steady stream of UDP data at 150Mbps. For this experiment we built our own version of "Iperf" in python so as to report the bandwidth at every node of the multicast stream.

Table 4.2 contains the experimental results, with the first column indicating the number of forwarding destinations, and the last column indicating the average throughput per receiver node. Ints/sec and cs/sec represent interrupts and context switches per second. The CPU utilization is split between the time spent by the in-kernel producer context (marked $\Uparrow$), the fwP user-mode context and the in-kernel consumer context respectively (marked $\Downarrow$). Note that the fwP task spends less CPU cycles as the number of forwarding destinations increases then the kernel thread which performs the the multi-send.

Table 4.2: Gigabit IP multisend metrics per stream.

Receivers	cpu(%) ⇑	cpu(%) fwP	cpu(%) ⇓	ints/sec	cs/sec	Mb/s
1	9	7	8	20318	65331	175.4
3	9	11	16	27629	73727	175.1
5	9	11	16	32801	75910	167.3
6	11	12	38	34703	79575	155.4

IPSec

Next we built a multi-threaded version of IPsec using both fwP and libpcap. Our solution implements AES encryption in Cipher-block Chaining (CBC) [45] mode of operation (128 bit key). Our experiments focused on steady-state performance, the key-establishment protocol is not included as it runs outside our framework.

The experiments ran on the Cornell NLR Rings topology in loopback mode (see Section 2.1.1) identical to the one presented in Figure 4.6. The fwP packet processing elements were deployed on the four-way 2.4 GHz Xeon E7330 quad-core Dell PowerEdge R900 servers with 32GB RAM. Since each R900 server was equipped with only two NetXtreme II BCM5708 Gigabit Ethernet cards, the maximum forwarding data rate achievable is 1Gbps. We did not introduce any link delay—the one way latency between any two adjacent nodes is roughly 40μs. Figure 4.8 reports the throughput in Mbps against the number of threads performing the CBC AES encryption; as usual, error bars denote standard error.

The three lines correspond to the fwP implementation, and two pcap implementations, one in which the pcap (main) producer loop places all packets on a shared queue for several consumer threads to process, and a second one in which the producer places packets on a private queue for each consumer thread. The pcap implementations use standard pthreads, semaphores and futexes (fast mutexes) to implement queues. We

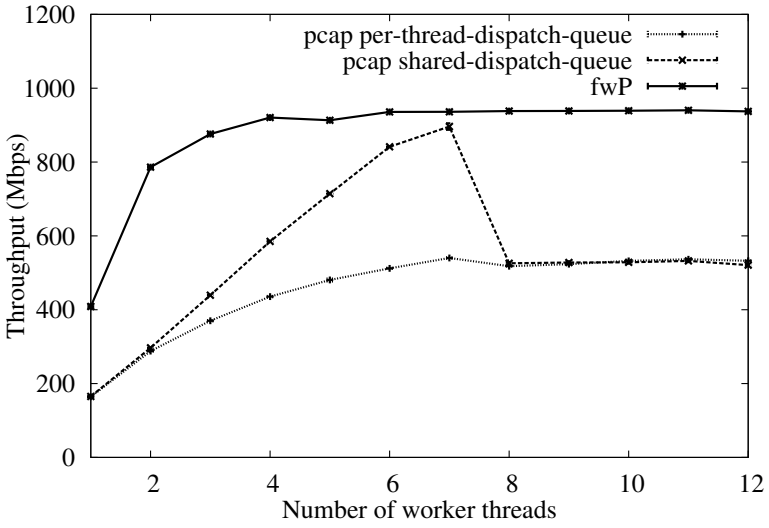


Figure 4.8: IPsec throughput vs. worker threads.

also experimented with spinlocks instead of futexes to accelerate the pcap implementation, however the performance was marginally worse, hence we report the best results we were able to achieve. The fwP implementation forks a process, that in turn clones a set of worker threads via the `clone` system call. Each worker thread maps a fwP buffer and registers for receiving all IP packets—the fwP kernel bottom half will round-robin load balance packets for identical filter rules. (Recall from Section 4.2.2 that fwP is inherently designed to use individual buffers per thread.)

The graph shows that the single-threaded fwP implementation achieves 2.5 times the throughput of the best pcap implementation. With two threads, the fwP version comes close to line (1Gbps) speed. Note that the pcap implementation with a shared queue performs better than the implementation with a private queue per worker thread—this is due to the fact that when the private queues become empty the producer must wake its worker threads. We see that beyond seven worker threads both pcap implementations yield the same throughput (there are eight distinct CPUs, sufficient for the producer and seven worker threads). With the producer thread pegged to a CPU the behavior was the

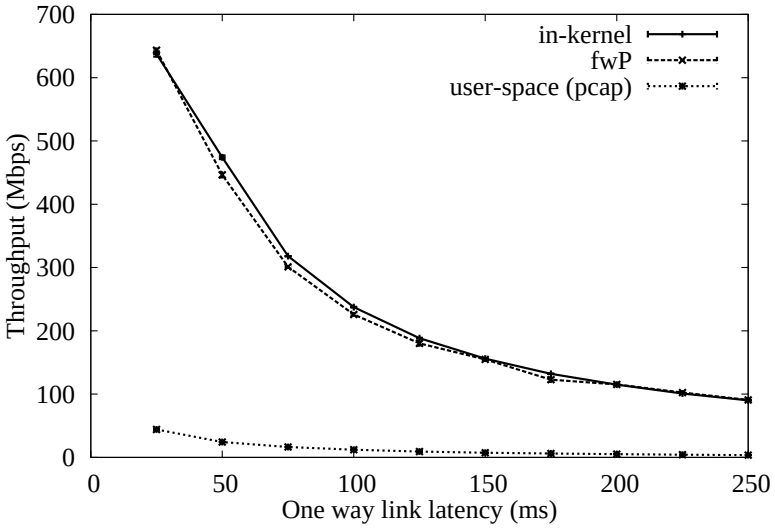


Figure 4.9: Maelstrom implementations throughput.

same, hence we attribute this degradation to increased scheduling and contention (e.g. schedule out a thread that already holds a lock thus preventing threads that are waiting for the lock to be released from making progress as well).

The takeaway from this experiment is that developers who use a conventional network stack must be intimately aware of impedance matching issues, especially when using threads and synchronization. By contrast, the fwP implementation easily reaches the maximum achievable throughput (1Gbps in this setup) with almost as little as two cores, while the pcap version plateaus at just 530Mbps.

The Maelstrom Protocol Accelerator

Maelstrom [63] is a performance enhancement proxy developed to overcome the poor performance of TCP when loss occurs on high bandwidth / high latency network links. Maelstrom appliances work in tandem, each appliance located at the perimeter of the network and facing a LAN on one side and a high bandwidth / high latency WAN link on

the opposite side. The appliances perform forward error correction (FEC) encoding over the egress traffic on one side and decoding over the ingress traffic on the opposite side. In Figure 4.6, for example, node1 and node2 are running Maelstrom appliances, with node1 encoding over all traffic originating from node10 and destined for, say node20. Node2 receives both the original IP traffic and the additional FEC traffic and forwards the original traffic and potentially any recovered traffic to node20. Note that this is a symmetric pattern, each Maelstrom appliance working both as an ingress and as an egress router at the same time.

The existing hand-tuned, in-kernel version of Maelstrom is about 8432 lines of C code. It is self contained with few calls into the exported base of kernel symbols. In contrast, the fwP implementation required just 496 lines of user-space C code (not counting libraries or the fwP kernel bottom half). Figure 4.9 shows the throughput as a function of the round trip time with a pair of Maelstrom appliances running on nodes 1 and 2, on the Emulab testbed. The graph shows that the fwP implementation yields almost *identical* performance to the in-kernel version. We also compared with a libpcap implementation; the fwP version is more than an order of magnitude faster. Moreover, these results continue to hold in the presence of message loss on the WAN link.

Note that the throughput shown in the figure is the end-to-end measured goodput between node10 and node20. Maelstrom introduces additional FEC traffic between nodes 1 and 2. In particular, for the parameters we ran the experiments with 27% of the bandwidth is allotted to FEC traffic (i.e., for every $r = 8$ packets we send $c = 3$ additional FEC packets), hence the goodput is at most 686Mbps. This is precisely the performance obtained: the fwP version of Maelstrom reduces memory contention / pressure and is thus running as fast as possible on the given hardware, being constrained by network bandwidth—1Gbps in this case—and not processing power.

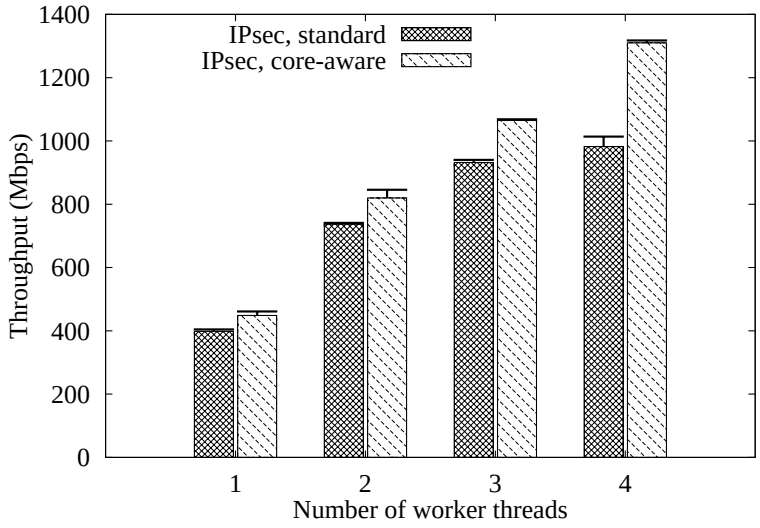


Figure 4.10: fwP IPsec throughput, 2x1Gbps links.

4.3.2 Microbenchmarks

In the previous section we showed four examples of user-space packet processing appliances built with fwP while taking advantage of multi-core chips. In this section we use instrumentation and profiling to look at fine-grained performance characteristics. In particular, we investigate how does fwP scale and what are the limitations, next we quantify the baseline performance of receiving a packet, and finally we examine how well does fwP mitigate memory contention.

Multi-Core Scaling and Limitations

To test the limits of fwP we modified the local topology from Figure 4.6 so that there are two independent 1Gbps channels between nodes 10 and 20.

As a baseline, forwarding 2Gbps of packets through the fwP workqueue bottom half pegs one CPU to 50% usage, not accounting for cycles spent during the top half, while

Table 4.3: Packet delivery delay (μs).

delay	pcap		fwP (wq)		fwP (all)	
	run1	run2	run1	run2	run1	run2
avg	70.4	59.5	6.8	7.0	22.3	22.5
stdev	114.3	98.0	28.0	6.9	62.2	76.7

achieving the maximum line throughput of 1876Mbps (for TCP-reno), or 938Mbps on each channel. If packets are routed through a user-mode fwP application without performing any processing, the maximum achievable throughput is about 1621Mbps, about 810Mbps on each channel, and 1721Mbps if threads are intelligently assigned on cores (as described in Section 4.2.2).

Figure 4.10 shows how fwP IPsec scales given more network capacity. The take-away is two-fold. First, core-aware placement boosts the performance of fwP: the plot shows an average 12.49% improvement in end-to-end throughput for a single worker thread, and 33.39% for four worker threads. Importantly, the throughput gap increases with the number of cores. Second, although we can process data at 1300Mbps, to breach 1800Mbps we would need faster memory architecture and / or a few more cores.

The Path of a Packet

Both fwP and pcap implementations can sustain routing packets throughput at 1Gbps without additional processing. The end-to-end throughput on the Emulab testbed can be seen in Figure 4.7, for zero passes on the x-axis, the left most set of bars. However, the pcap implementation spends more CPU cycles, therefore it has fewer cycles to spare for additional per-packet computation.

In addition to CPU cycles, another metric is per-packet delivery latency. We measured the time to deliver a packet, percolating through the network stack, from the mo-

Table 4.4: Memory bus transactions / cpu cycles.

a) fwP

binary	ratio
vmlinux	.015
fw_test	.0015
bnx2	.007
fw_mod	.0065
total	.01004

b) pcap

binary	ratio
vmlinux	.014
pcap_test	.00388
bnx2	.0059
af_packet	.0046
total	.0104

ment it is time-stamped by the kernel (by the NIC driver) until it is delivered to the user-space application. Table 4.3 shows the statistics for a few separate Iperf runs. Each measurement is over about 10.2 million packets, i.e. the number of packets processed in a 120 second Iperf run.

For fwP we report both the end to end delay (marked "all") and the delay since the packet is handled by the dispatching workqueue (marked "wq")—this is where the bulk of the fwP work is done. Due to design tradeoffs, we added another level of indirection, namely enqueueing packets from the fwP softirq bottom half onto a workqueue (essentially a queue serviced by a task)—unlike softirqs, tasks can block and be bound to cores. The delay penalty incurred is shown in Table 4.3 as the difference between the columns marked "all" and "wq." The table illustrates that the latency for fwP to deliver the packet from the NIC to user-space is about a third that of pcap. The high variance for both fwP and pcap is due to scheduling delays.

Memory Pressure

The design of the fwP aimed to reduce memory pressure / contention. We use `oprofile` to measure the overheads imposed on the memory subsystem by the IPsec appliance built both with fwP and with pcap.

Table 4.5: Load ratio (L1d loads / cpu cycles).

a) fwP

binary	ratio
vmlinux	.1567
fw_test	.691
bnx2	.1
fw_mod	.148
total	.2712

b) pcap

binary	ratio
vmlinux	.145
pcap_test	.5529
bnx2	.0935
af_packet	.134
total	.377

Oprofile [179] is a statistical profiler that can measure CPU events by a sampling technique. It does so by taking advantage of existing CPU performance counters. Each CPU has a limited number of counters that can be configured to decrement every time some internal event occurs (e.g. L2 cache miss). When the counter reaches zero, an interrupt is issued, and the handler will gather statistics, like the event count and the position of the instruction pointer. Each counter can be initialized to some arbitrary *overrun* value, therefore it is possible to gather more accurate results at the expense of more overhead.

The profiling runs consist of consecutive fwP and pcap IPsec tests with a single worker thread, as described in section 4.3.1. Care must be taken when choosing the events to profile and the methodology employed. Absolute values are misleading, for example the number of branch mispredictions in a function is meaningless unless we know how often that function is called. To compare two programs, we use the ratios of two events sampled at the same frequency (overrun value is 6000).

We report on all symbols: the kernel marked as *vmlinux*, device driver as *bnx2*, the bottom half mechanism marked as *fw_mod* for the fwP kernel module and *af_packet* for the traditional network stack and test binary marked as *fw_test* and *pcap_test*. We omit irrelevant symbols, like all other user-mode processes and kernel drivers (however, they are represented by the *total* field).

Table 4.6: Store ratio (L1d stores / cpu cycles).

a) fwP

binary	ratio
vmlinux	.0837
fw_test	.194
bnx2	.039
fw_mod	.0569
total	.111

b) pcap

binary	ratio
vmlinux	.072
pcap_test	.41
bnx2	.0361
af_packet	.064
total	.236

Table 4.7: Pipeline flushes / number of instructions retired.

a) fwP

binary	ratio
vmlinux	.000121
fw_test	.000015
bnx2	.000166
fw_mod	.000155
total	.000054

b) pcap

binary	ratio
vmlinux	.00015
pcap_test	.00009
bnx2	.000122
af_packet	.00011
total	.000096

We start by comparing the number of memory transactions on the bus, per CPU cycles (while not halted). The memory transaction counter aggregates the number of burst read and write-back transactions, the number of RFO (read for ownership) transactions and the number of instruction fetch transactions, all between the L2 cache and the external bus. Table 4.4 contains the results. In both cases the kernel is the dominant component. We can see that the fwP test program issues about 2.58 less transactions than the pcap test program.

Taking a closer look, Tables 4.5 and 4.6 show the load and store rates for fwP and pcap: fwP performs about 28% less load operations per cycle (in total) with respect to pcap and roughly half the number of stores per cycle (in total); therefore reducing the

Table 4.8: RFO / memory bus transactions (total).

version	ratio
fwP	.1244
fwP (core-aware)	.105
pcap	.1087

pressure on the memory subsystem and yielding an end-to-end throughput 2.5 times higher. A high load / store ratio means the execution is bound by memory.

Table 4.7 shows the ratio between the CPU(s) pipeline flushes and the number of retired instructions. The fwP version performs half the total pipeline flushes per instruction retired the pcap does—a fair improvement.

Finally, Table 4.8 shows that when smart core placement is used, fewer RFO transactions are issued on the bus, and the scheduler spends less time balancing tasks. In contrast, with smart core placement disabled, we noticed that besides copying data—to and from the fwP buffer and duplicating the sk_buff for pcap—the kernel spends most time in the scheduler, a significant part of which is re-balancing tasks. The pcap version performs less RFO transactions, since during this IPsec test we used a single thread, and the conventional network stack typically wakes up the `af_packet` receiver on the same CPU.

4.4 Experience

To appreciate the value of the fwP abstraction, it may be helpful to understand our own experience developing Maelstrom. The Maelstrom box has to run over high-speed optical links while performing XOR computations over the packets that pass through it—hence, it's imperative that it be able to run at line rates on a gigabit link. Our first prototype of Maelstrom was a user-space application that used libpcap (on FreeBSD) to pick up packets and process them—it ran at a disappointing 60 Mbps on a 1 Gbps link, beyond which it began to experience buffer overflows and data losses.

In the ensuing six months we dropped Maelstrom into the kernel and optimized it

extensively, resulting in versions that ran at 250 Mbps, 400 Mbps and finally 940 Mbps. In the course of optimizing our kernel implementation, we learned several useful things about building high-performance networked systems. In particular, we were able to categorize the various sources of overheads for such a system, as well as abstract out the functionality required by a packet processing application. This led to the idea of the Revolver IPC.

With fwP, we were able to code a Maelstrom implementation in a single day—and it ran with zero optimization at 900+ Mbps of output data rate. And most importantly, we were able to code multiple different applications—such as the security packet inspector that scanned packets for suspicious signatures, a gigabit overlay router and an IPsec proxy—within hours. Each of these applications ran without any optimization at line-speed. Given that the majority of a developer's time is arguably spent optimizing code, and this is especially true for high-performance networked systems, we believe this result to be a significant advance in the state-of-the-art.

4.5 Summary

In this Chapter we show how to overcome the overheads incurred by packet processing applications that run on commodity shared bus multi-core systems. In doing so, we have distilled the lessons learned and the prototype engineering process into a new operating system abstraction—fwP. We demonstrate the efficacy of fwP by building various packet processors that can improve the inter-datacenter communication while operating at multi-gigabit speeds. In the next chapter, we leverage the knowledge we gained to generalize and improve upon the fwP abstraction, and ultimately extend its scope and applicability.

CHAPTER 5

PACKET PROCESSING ABSTRACTIONS II: HARNESSING THE PARALLELISM OF MODERN HARDWARE WITH NETSLICES

Newly emerging non-uniform memory access (NUMA) architectures that use point-to-point interconnects reduce the memory pressure since concurrent memory accesses from different CPUs may be routed on different paths to distinct memory controllers instead of being issued over a shared bus to the same memory controller. However, NUMA architectures do not eliminate the memory contention caused by CPU cores that are concurrently accessing shared data and peripheral (e.g., memory mapped) devices [103]. In other words, NUMA architectures mitigate the *raw bus contention*, but not the *data contention* (see Section 4.1), whereas the *memory wall* remains an issue. Therefore, the conventional raw socket—the de-facto abstraction exported by operating systems to enable packet processing in user-space—remains highly inefficient and unable to fully take advantage of the emerging hardware, like multi-core NUMA architectures.

In Chapter 4 we have shown that packet processors running on shared-bus commodity servers are unable to take advantage of the available spare CPU cores, due to the conventional software network stack that overloads the memory subsystem. After revealing the intrinsic overheads of the operating systems' network stack, we introduced fwP—a new operating system abstraction that mitigates these overheads—and demonstrated how fwP can be utilized to build high-performance software packet processors in user-space that outperform conventional counterparts.

Although fwP does mitigate memory contention in general, it was not designed specifically for NUMA architectures (fwP was designed prior to the emergence of commodity NUMA servers). Moreover, fwP was not designed to mitigate the contention exerted by multiple CPU cores accessing the same high-speed 10GbE network adapters,

nor was it designed to interface with modern multi-queue 10GbE NICs. Hardware multi-queue NIC support has been introduced since a single CPU core is not sufficiently fast to drive a 10GbE NIC, whereas the contention and synchronization overhead of multiple CPU cores accessing the same NIC is prohibitive [103]. Another drawback of fwP is that it is somewhat invasive since it alters the core operating system kernel interface and functionality. This severely limits fwP's portability and maintainability, since a core kernel patch is required before fwP is deployed, unlike regular device drivers that are self-contained in a kernel extension, or module, and can be loaded at runtime without requiring a custom interface with the core kernel. Moreover, fwP presents an unfamiliar and awkward programming interface (e.g. the `fw_flip` operation).

In this chapter we report on the design and implementation of NetSlice—a new fully portable operating system abstraction that enables linear performance scaling with the number of CPU cores while processing packets in user-space. We achieves this through an efficient, end-to-end, raw communication channel akin to the raw socket, that leverages modern hardware. NetSlice performs *spatial partitioning* (i.e., exclusive, non-overlapping, assignment) of the CPU cores, memory, and multi-queue Network Interface Controller (NIC) resources at coarse granularity, to aggressively reduce overall memory and interconnect contention.

NetSlice tightly couples the hardware and software resources involved in packet processing. The spatial partitioning effectively offers the illusion of a battery of independent, isolated SMP machines working in parallel with near zero contention. At the same time, each individual NetSlice partition was designed to provide a fast, lightweight, streamlined path for packets between the NICs and the user-space raw endpoint. Moreover, the NetSlice application programming interface (API) exposes fine-grained control over the hardware resources, and also provides efficient batched send / receive opera-

tions. NetSlice is completely modular and easy to deploy, requiring only a simple kernel extension that can be loaded at runtime, like any regular device driver. As a result it is highly portable since it does not depend on any special hardware. By contrast, fwP requires the kernel to be patched and recompiled prior to loading a portable module.

We show that complex user-space packet processors built with NetSlice—like a protocol accelerator and an IPsec gateway—closely match the performance of state-of-the art, high-performance, in-kernel RouteBricks [103] variants. Moreover, NetSlice packet processors scale linearly with the number of cores and operate at nominal 10Gbps network line speeds, vastly exceeding alternative user-space implementations that rely on the conventional raw socket.

The contributions of the work in this chapter are as follows:

- We argue that the conventional raw socket is ill-suited for packet processing applications.
- We propose NetSlice—a new operating system abstraction for developing packet processors in user-space that can leverage modern hardware.
- We evaluate NetSlice and show that it enables linear throughput scaling with the number of cores, closely following the performance of state-of-the-art in-kernel variants.
- We provide a drop-in replacement for the conventional raw socket that requires only a simple kernel extension which can be loaded at runtime.

The rest of the chapter is structured as follows. Section 5.1 expands on the motivation behind user-mode packet processors. Section 5.2 details the NetSlice design and implementation while Section 5.3 presents our evaluation.

5.1 The Case Against The RAW Socket: Where Have All My CPU Cycles Gone?

Refer to Appendix B for details pertaining to the path of a packet through a modern, in-kernel, network stack.

Operating systems abstractions for packet processing in user-space are overly general, and in need of an overhaul. The issue stems from the fact that the entire network stack handles the raw socket in the same fashion it handles a regular endpoint (TCP or UDP) socket—essentially taking the least common denominator between the two. However, unlike TCP or UDP sockets, a raw socket is different in that it manipulates the entire traffic seen by the host. Given today's network capabilities, such traffic is sufficient to easily overwhelm a host that uses raw sockets. We argue that applications are unable to take advantage of modern hardware since:

1. The raw socket abstraction is too general and provides the user-mode application with no control over the physical resources.
2. Although simple and common to all types of sockets, the socket API is largely inefficient.
3. The conventional network stack is not end-to-end. In particular, the hardware and software resources that are involved in packet processing are loosely coupled, which results in increased contention.
4. Likewise, the conventional network stack was built for the general / common case. This renders the path taken by a packet between the NIC and the user-space raw endpoint unnecessarily expensive.

Engler et al. [110] have similarly argued for an end-to-end approach, and against the high cost introduced by high-level abstractions. A fixed set of high level abstractions has been known to **i)** hurt application performance, **ii)** hide information from applications, and **iii)** limit the functionality of applications. The conventional (raw) socket is such an example—it offers a single, arguably ossified, API which abstracts away the path taken by a packet between the NIC and the application, thus providing no control over the hardware resources utilized, which is why applications fail to perform.

Next, we expand on the four above claims. First, the socket API does not provide tight control over the physical resources involved in packet processing. For example, the user-mode application has no control over the path taken by a packet between some NICs queue and the raw endpoint. Second, although providing a simplified interface, the socket API is largely inefficient. For example, it requires a system call for every packet send / receive operation (the asynchronous I/O interface is currently only used for file operations, since it does not support ordering—equally important for both TCP send/receive and UDP send operations).

Third, the network stack is not end-to-end. For example, the raw socket endpoint is loosely coupled with the network stack by virtue of the user-mode task it belongs to. Since processing is performed in a separate protection domain, an additional cost is incurred due to packet copies between address spaces, cache pollution, context switches, and scheduling overheads. Importantly, the cost depends on the CPU affinity of the user-mode task relative to the corresponding in-kernel network stack that processed the packets in the first place. In general, there are several choices where the user-mode task may run with respect to the in-kernel network stack:

- **Same-core:** In lockstep on the same CPU with the in-kernel network stack.
- **Hyperthread:** Concurrently on a peer hyperthread of the CPU that runs the

in-kernel network stack, if one is available.

- **Same-chip:** Concurrently on a CPU that shares the Last Level Cache (LLC), e.g. L3 for Nehalem.
- **Different-chip:** Concurrently on a CPU that belongs to a different packaging socket / silicon die.

The first option is ideal in terms of cache performance, however one has to consider the cost of frequent context switches and the impedance mismatch between the in-kernel network stack running in softirq context (a type of bottom half), at a strictly higher priority than user-mode tasks, and the user-mode task. If the user-mode task is not time-shared sufficient CPU cycles to clear the socket buffers in a timely fashion, packets will be dropped.

If hyperthreads are available, the second option may be ideal. However, hyperthreads need to be simultaneous, i.e. the CPU can fetch instructions from multiple threads in a single cycle. Hyperthreads are not ideal if they work on separate data (i.e. at different physical locations in memory), since they would split all shared cache levels into half. However, if hyperthreads work on shared data, e.g. the packets passed between a user-mode task and the in-kernel network stack, then this scenario has the potential of also reducing cache misses beyond the LLC. Alternatively, two CPUs may only share the LLC and still reduce the number of cache misses. The final option is sub-optimal, since every packet would induce an additional LLC cache miss.

By default, however, the kernel scheduler dynamically chooses on which CPU to run the user-mode task, constantly re-evaluating its past decision, and potentially migrating the task onto a different CPU. Although the user-space application is able to choose a CPU affinity to request on which CPUs to run, the socket interface provides no insight

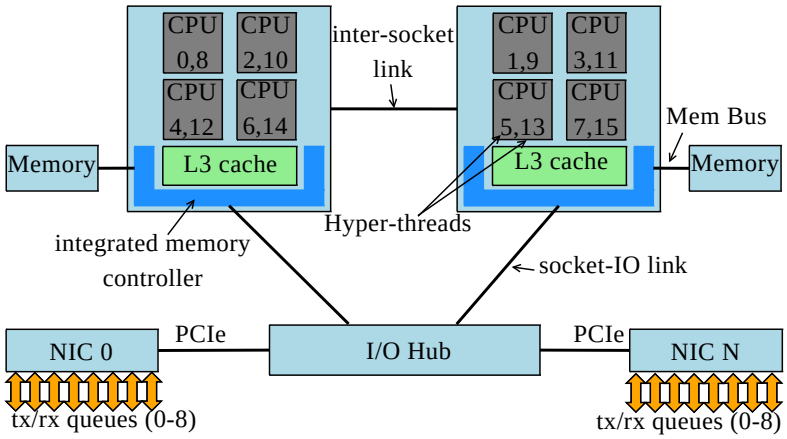


Figure 5.1: Nehalem cores and cache layout.

into what the placement should be. The socket traffic may have been handled by the in-kernel network stack on any of the available CPUs. Worse, the raw socket receives traffic from all queues of every NIC, that is handled by all (interrupt receiving) CPUs, thus increasing the contention overhead. Further, in such a case, there is no optimal CPU placement for the task.

Fourth, and final, the in-kernel network stack is overly general, bulky, and unnecessarily expensive. To illustrate this, consider a user-space application processing the entire traffic by means of raw sockets. For the system depicted in Figure 5.1, in order to utilize the available CPU cores, boilerplate solutions either use several raw sockets in parallel, one per process / thread, or a single raw socket and load balance traffic to several worker threads.

If several raw sockets are used in parallel, each received packet is processed by protocol handlers as many times as there are raw sockets, and a *copy* of the packet is delivered to each of the raw sockets. Moreover, the original packet is also passed to the default in-kernel IP layer. To implement a packet processor in user-space, an additional firewall rule is needed that instructs the kernel to needlessly drop the packet. Berkeley

Packet Filters (BPF) can be installed on each raw socket in an attempt to disjointly split the traffic, however:

- BPF filters are expensive, and they scale poorly with an increase in the number of sockets [116].
- Writing non-overlapping filters for all possible traffic patterns is hard at best, and requires a priori knowledge of traffic characteristics, not to mention the complexity of handling traffic imbalances. Filters may be installed at runtime, by reacting to the traffic patterns, however, installing filters on the fly at rates around 10Gbps is not feasible [223].
- Without understanding the NICs opaque hash function that classifies flows to queues we are unable to predict the CPU that will be executing the kernel network stack, hence filters may exacerbate interference (e.g. cache misses). Such predictions are only possible if the interrupts from queues are issued in a deterministic fashion, and if the classification function is itself deterministic. The issue is further aggravated by using NICs from different vendors, which implement different classification functions (in our experience this is true of the Intel and Myricom 10GbE NICs).

Alternatively, a single raw socket may be used to load balance and quickly dispatch traffic to several worker threads. In this scenario, there are two potential contention spots. First, between the in-kernel network stacks running on all (interrupt receiving) CPUs and the dispatch task, and second, between the dispatch task and the worker threads (we evaluate this scenario in Section 5.3).

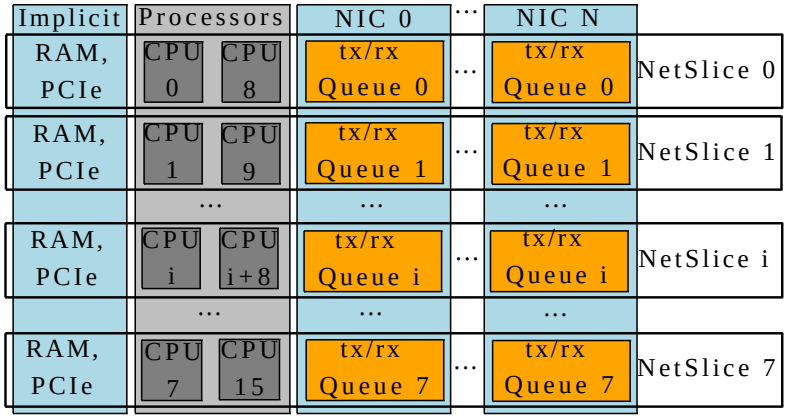


Figure 5.2: NetSlice spatial partitioning example.

5.2 NetSlice

We argue that user-mode processes need end-to-end control over the entire path taken by packets, all the way from the NICs to the applications and back. NetSlice relies on a four pronged approach to provide an efficient end-to-end OS abstraction for packet processing in user-space. First, NetSlice spatially partitions the hardware resources at coarse granularity to reduce interference / contention. Second, the NetSlice API provides the application with fine-grained control over the hardware resources. Third, NetSlice provides a streamlined path for packets between the NICs and user-space. Fourth, NetSlice exports a rich and efficient API.

The core of the NetSlice design consists of spatial partitioning of the hardware resources involved in packet processing. In particular, we provide an array of independent packet processing execution contexts that "slice" the network traffic to exploit parallelism and minimize contention. We call such an execution context a NetSlice. Each NetSlice is end-to-end [192], tightly coupling all software and hardware components from the NICs to the CPUs executing the in-kernel network stack and the user-mode task.

A NetSlice packet processing execution context consists of one transmit (tx) and one receive (rx) queue per attached NIC, and two (or more) tandem CPUs. Modern high speed (10GbE) NICs support a configurable number of tx/rx queues, usually larger than the number of cores. Importantly, a NIC queue belongs to a single NetSlice context. While the NIC queues and CPU cores are resources explicitly partitioned by NetSlice, each execution context also consists of implicit resources, like a share of the physical memory, PCIe bus bandwidth, etc. The tandem CPUs are sharing at the very least the LLC; NetSlice defaults to using hyperthreads if available. NetSlice automatically binds the tx/rx queues of each context to issue interrupts exclusively to one of the peer CPUs in the context—we call this the *k-peer* CPU; we call the other CPU(s) the *u-peer* CPU(s). The in-kernel (NetSlice) network stack executes on the k-peer CPU, while the user-mode task that utilizes NetSlice runs on the u-peer CPU. If a NetSlice has more than two CPUs, several threads may execute concurrently in user-mode.

There are as many NetSlices as there are CPU tandems. For our experimental setup depicted in Figure 5.1, NetSlice partitions resources as depicted in Figure 5.2. Every NIC is configured with eight tx/rx queues, associating the i^{th} tx/rx queue of every NIC (e.g. NICs 0 and 1 in Figure 5.1) with tandem pairs consisting of CPUs i (k-peer) and $i+8$ (u-peer). Each NIC issues interrupts signaling events pertaining to the i^{th} queue to the i^{th} CPU exclusively. Through this technique, no two k-peer CPUs will handle packets on the same NIC queue, thus eliminating the costs of contention like locking, cache coherency, and cache misses. This scheme that binds NIC queues to CPUs was previously evaluated for 1Gbps NICs [70] and is the keystone to RouteBrick's individual forwarding element scaling (RouteBricks relied on Click [147] which uses a polling driver instead of the conventional interrupt driven one, however, NAPI and Interrupt Coalescence achieve the same effect).

Further, NetSlice exposes fine-grained control over the hardware resources of the entire packet processing execution context to the user-mode application. For example, NetSlice provides control over which CPU the in-kernel (NetSlice) network stack is executing with respect to the user-mode application to take advantage of the physical cache layout. The added control is key to minimizing inter-CPU contention in general, and cache misses and cache coherency penalties in particular.

Importantly, the path a packet takes through each NetSlice execution context is streamlined, bypassing the default, bulky, in-kernel general purpose network stack. NetSlice hijacks the packets at an early stage subsequent to DMA reception and before it would have been handed off to the network stack. Next it performs minimal processing while in kernel context executing on the k-peer CPU, and then passes the packets to the user-space application to be processed in overlapped (pipelined) fashion, on the u-peer CPU. Notably, on an entire NetSlice path there is a single spinlock being used per send / receive direction. The spinlock is specialized for synchronization between the communicating execution contexts, namely between a bottom half and a task context.

While the NetSlice API provides tight control over physical resources, it also supersedes and extends the ossified socket API, which, although providing a simplified interface, is largely inefficient. Instead of requiring a system call for every packet send or receive, the NetSlice API supports batched operations to amortize the cost associated with protection domain crossings. At the same time, the NetSlice API is backwards compatible with the conventional BSD socket API. In particular, the API supports conventional library `read/write` operations with precisely the same semantics as the ones used on file or socket descriptors.

```
1:  #include "netslice.h"
2:
3:  struct netslice_iov {
4:    void *iov_base;
5:    size_t iov_len; /* capacity */
6:    size_t iov_rlen;/* returned length */
7:  } iov[IOVS];
8:
9:  struct netslice_rw_multi {
10:   int flags;
11: } rw_multi;
12:
13: struct netslice_cpu_mask {
14:  cpu_set_t k_peer;
15:  cpu_set_t u_peer;
16: } mask;
17:
18: fd = open("/dev/netslice-1", O_RDWR);
19:
20: rw_multi.flags = MULTI_READ |
21:      MULTI_WRITE;
22: ioctl(fd, NETSLICE_RW_MULTI_SET,
23:      &rw_multi);
24:
25: ioctl(fd, NETSLICE_CPUMASK, &mask);
26: sched_setaffinity(getpid(),
27:      sizeof(cpu_set_t), &mask.u_peer);
28:
29: for (i = 0; i < IOVS; i++) {
30:   iov.iov_base = malloc(MTU_LARGE);
31:   iov.iov_len = MTU_LARGE;
32: }
33: if (mlockall(MCL_CURRENT) < 0)
34:   exit_fail_msg("mlockall");
35:
36: for (;;) {
37:   ssize_t cnt, wcnt = 0;
38:   if ((cnt = read(fd, iov, IOVS)) < 0)
39:       exit_fail_msg("read");
40:
41:   for (i = 0; i < cnt; i++) {
42:       /* iov_rlen marks bytes read */
43:       scan_pkg(iov[i].iov_base,
44:            iov[i].iov_rlen);
45:   }
46:   /* forward the packets back */
47:   do {
48:       size_t wr_iovs;
49:       /* write iov_rlen bytes */
50:       wr_iovs = write(fd, &iov[wcnt],
51:            cnt-wcnt);
52:       if (wr_iovs < 0)
53:           exit_fail_msg("write");
54:       wcnt += wr_iovs;
55:   } while (wcnt < cnt);
56: }
```

Figure 5.3: One NetSlice (1^{st}) batched read/write example.

5.2.1 NetSlice Implementation and API

The NetSlice API is UNIX-like, as elegant as the file interface, and as flexible as the `ioctl` mechanism. User-mode applications perform conventional file operations using the familiar API, e.g. open / read / write / poll over each slice, which map to corresponding operations over the per-NetSlice data flows. For example, a conventional read operation will return the next available packet, block if no packet is available, or return `-EAGAIN` if there are no packets available and the device was opened with the `O_NONBLOCK` flag set. In fact, we implemented the NetSlice abstraction as a set of character devices with the same major number and N minor numbers—one minor number

for each of the *N* slices.

The `ioctl` mechanism was sufficient to provide NetSlice additional control and API extensions. For example, the `NETSLICE_CPUMASK ioctl` request returns the mask of the tandem CPUs, thus enabling the current user-mode task fine control over the CPU it runs atop. The `NETSLICE_TX_CSUM_SET ioctl` request allows the user-mode application to offload the kernel module to perform TCP, IP, both or no checksum computation. The in-kernel NetSlice stack in turn has the knowledge to enable hardware specific offload computation to spare CPUs from unnecessarily spending cycles.

The `NETSLICE_RW_MULTI_SET ioctl` request is of particular interest. Once set, the user-mode application can use the read / write calls to send and receive an array of datagrams encoded within the parameters. This is fundamentally different than the `readv` / `writev` calls which can only perform scatter-gather of a *single* datagram (or packet) per call, which means that using `readv` / `writev` a system call is to be issued for every packet. Batched packet receive and send operations are instrumental in mitigating the overheads of issuing a system call per operation. At the same time, it reduces per packet locking overheads, e.g. spinlock induced cycle waste and cache coherency overheads, between the user-mode task while executing system calls and the in-kernel NetSlice network stack.

Figure 5.3 shows an example of application code using NetSlice batched read / write for a naïve deep packet inspection tool. Commenting out lines 44 through 48, the application forwards packets acting as a regular router. The array of buffers are passed to the read and write functions encoded in `netslice_iov` structures. The example consists of a single NetSlice (the 1^{st} NetSlice) hence the application will only receive packets classified to be handled by the 1^{st} queue of each NIC. To handle the entire traffic, the example can be easily extended to accommodate all available queues using either an

equal number of threads or separate processes.

For outgoing packets, the routing decision is performed by default within the in-kernel NetSlice stack. However, the `NETSLICE_NOROUTE_SET` `ioctl` request allows applications to perform routing in user-space (the chosen output interface is encoded within the parameters of the write call). If the hardware decides which NIC rx queue to place the received packets onto, the software is responsible for selecting an outbound NIC queue to transmit packets on. For the conventional network stack, the kernel or the device driver is responsible for implementing this functionality. NetSlice provides a specialized classification "virtual function" that overrides any driver or kernel provided hash functions (we update the `select_queue` function pointer of every `net_device` structure). The NetSlice classification function ensures that packets belonging to a particular NetSlice context are placed solely on the tx queues associated with the context. Unlike driver provided (e.g. `myri10ge` driver's `myri10ge_select_queue`) or the kernel's default `simple_tx_hash` classification function, the NetSlice classification function is considerably cheaper, consisting of three load operations, one arithmetic, and one bitwise mask operation.

Note that instead of a character device, we could have implemented NetSlice by extending the socket interface and adding a new `PF_PACKET` (e.g. `SOCK_RAW`) socket type. Instead, the current approach allowed us to seamlessly commandeer received packets immediately after reception. By contrast, a new `PF_PACKET` socket does not curtail the default network stack, nor does it prevent the kernel from performing additional processing per packet (e.g. pass packets through all relevant protocol handlers).

5.2.2 Discussion

The reader familiar with the large body of user-space networking work [182, 118, 56, 209, 208, 69] may rightly have a sense of *déjà vu*. Nevertheless, we point out that NetSlice is in fact orthogonal to past work that relocated the networking stack into the user-space—user-space networking may very well be built on top of NetSlice, however the converse does not hold. Leveraging modern hardware resources is at the core of the NetSlice design, whereas most user-space networking approaches predate multi-core CPUs, multi-queue NICs, and in fact predate conventional in-kernel SMP network stacks.

Although we could have, we did not implement the network stack encapsulation to replace endpoint sockets. In our experience, TCP and UDP sockets using the conventional in-kernel network stack implementation still perform sufficiently well, to date. Moreover, given that a typical host may have an arbitrarily large number of concurrent TCP and UDP connections, it is not clear that user-space networking, even built over NetSlice, would perform better than the current network stack.

Unlike most prior user-space networking solutions, we chose not to implement zero-copy / copy avoidance [106, 181] for the reasons described below:

- With the advent of modern NUMA architectures like the Intel Nehalem, the bottleneck for network I/O has shifted away from the raw memory bandwidth [103]. In particular, if for shared bus architectures, both *raw bus contention* and *data contention* were the main sources of overhead, only *data contention* remains a source of overhead for NUMA architectures, along with a bound on the available number of CPU cycles. Performance scaling may thus predictably follow from the performance increase with each processor generation. With NetSlice, we provide

a mechanism to minimize data contention and harness the aggregate spare cycles of multi-core CPUs for packet copies, while the data already resides in the LLC.

- Most zero-copy techniques introduce new interfaces that are incompatible with the conventional socket API. By contrast, NetSlice extends the socket API while maintaining backwards compatibility.
- Typical zero-copy techniques are hardware dependent, thus requiring significant driver porting effort. By contrast, NetSlice is completely portable, requiring a single modular addition, instead of one for each possible device driver. Moreover, zero-copy can be expensive on modern commodity hardware due to the cost of memory management (e.g. on-demand page mapping and un-mapping).
- Alternatively, solutions may preemptively set aside a fixed physical memory region from which all packet (`sk_buff`) payloads are allocated [69]. However, this solution is deficient, since it over-commits and pins down large physical memory spaces for which ultimately the user-space processes are responsible and trusted to relinquish.
- The OS needs to ensure tight impedance matching between the top-half interrupt service routine and the user-mode task. This would require invasive scheduler modifications [96, 73] we chose to avoid.

To summarize, a zero-copy designs for NetSlice would reduce the number of CPU cycles spent on performing a copy per network packet. However, a zero-copy design would also render NetSlice significantly less portable and would require each device driver to be rewritten and maintained, whereas the current design is fully portable. Moreover, a few spare CPU cores are sufficient to compensate for the cost of packet copies, provided that data contention does not increase. (Section 5.3 shows that NetSlice scales linearly with the number of cores, therefore data contention is indeed minimized.) Given

the current trend in semiconductor technology of placing an increasing number of CPU cores per silicon chip with each new generation, we believe that the cost of pursuing a zero-copy design is not justified at this time.

5.3 Evaluation

In this section we evaluate software packet processors running NetSlice against the state-of-the-art user-space and in-kernel equivalent implementations. In particular we have ported packet processors to run over RouteBricks [103] forwarding elements, as well as to run in user-space using the Packet CAPture (pcap) library [38]. Pcap is implemented on top of the conventional raw (`PF_PACKET`) sockets. We also linked the pcap applications with Phil Wood's libpcap-mmap library [222], which uses the memory mapping functionality of `PF_PACKET` sockets (known as `PACKET_MMAP`). A kernel built with the `PACKET_MMAP` flag copies each packet onto a circular buffer before adding it to the socket's queue. The circular buffer is shared between the kernel and the user-space in order to reduce the numbers of system calls. Note that `PACKET_MMAP` sockets do not provide zero-copy receive—a packet is copied the same number of times as with a traditional socket. During our experiments, we set the circular buffer size to the maximum possible (`PCAP_MEMORY=max`)—a limit imposed by the (physically contiguous) kernel memory allocator.

NetSlice requires a simple modular kernel extension that can be loaded at runtime, like any other device driver. NetSlice consists of 1739 lines of kernel module code and 2981 lines of user-space applications—e.g. a router, an IPsec gateway, and a protocol accelerator, of which 839 lines are AES / IPSec ports.

Our evaluation tries to answer the following questions:

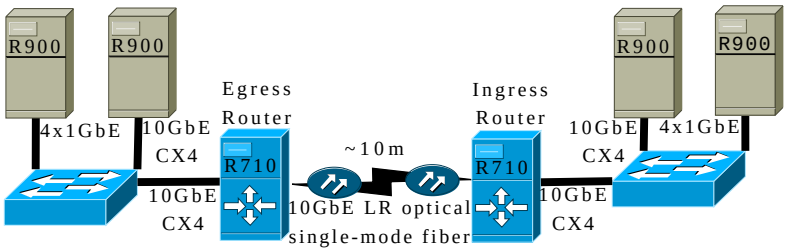


Figure 5.4: Experimental evaluation physical topology.

- What is the performance of NetSlice with respect to the state-of-the-art, for both routing and IPsec?
- How efficient is the streamlining of individual NetSlices? To quantify this scenario, we funnel all traffic to be handled by a single NIC queue. Since there is no interference from the other CPUs and NIC resources, we evaluate a single communication channel in isolation. We refer to this scenario as receive-livelock [169], even though it is not strictly identical to the original definition.
- What is the benefit of NetSlice batched send / receive operations?
- What is the performance of various choices of NetSlice peer CPUs placement?
- How does NetSlice scale with the number of cores?
- Can complex packet processors built with NetSlice deliver the advertised performance increase?

5.3.1 Experimental Setup

We used the Cornell NLR Rings testbed topology 2.1.1 in the loopback configuration, with several modifications. In particular, we deployed the testbed as depicted in Figure 5.4, with four Dell PowerEdge R900 machines serving as end-hosts that generate and receive traffic. The traffic is aggregated by two Cisco Catalyst 4948 series switches,

instead of the original HP ProCurve 2900-24G switches, before being routed through a pair of identical Dell PowerEdge R710 machines. We refer to the R710 machines as the egress and the ingress routers. The egress and the ingress routers run various packet processors, e.g. NetSlice, or RouteBricks [103].

Each R900 machine is a four socket 2.40GHz quad core Xeon E7330 (Penryn) with 6MB of L2 cache and 32GB of RAM—the E7330 is effectively a pair of two dual core CPUs packaged on the same chip, each with 3MB of L2 cache. By contrast, the R710 machines are dual socket 2.93GHz Xeon X5570 (Nehalem) with 8MB of shared L3 cache and 12GB of RAM, 6GB connected to each of the two CPU sockets. The Nehalem CPUs support hardware threads, or hyperthreads, hence the operating system manages a total of 16 processors. Each R710 machine is equipped with two Myri-10G NICs, one CX4 10G-PCIE-8B-C+E NIC and one 10G-PCIE-8B-S+E NIC with a 10G-SFP-LR transceiver. Figure 5.1 depicts the R710 internal structure, including the two NICs.

The egress router is connected to the ingress router through a 10 meter single-mode fiber optic patch cable, and each router is connected to the corresponding switch through a 6 meter CX4 cable. Two of the R900 machines are each equipped with an Intel 82598EB 10-Gigabit CX4 NIC, while the other two R900 machines are connected to the switches through all of their four Broadcom NetXtreme II BCM5708 Gigabit Ethernet NICs. We use the additional R900 machines, although the egress and ingress routers only have one 10GbE connection on each side, since a single R900 machine with a 10GbE interface is unable to receive (in the best configuration) more than roughly 5Gbps worth of MTU size (1500 byte packets) traffic. The packet rate (pps) for the R710 router with the Myricom 10GbE NIC is roughly the same for small (64 byte) and MTU size packets. The same observation applies for the R900 client with the Intel 10GbE NIC.

RouteBricks altered the NIC driver to increase the packet rate by performing batched DMA transfers of small packets. We have not implemented this feature yet—it is not clear this is possible on our Myricom NICs.

Unless specified otherwise, we generate traffic between the R900 machines with Netperf [28] that consists of MTU size UDP packets at line rate (10Gbps). The machines run the Linux kernel version 2.6.28-17; we use the myri10ge version 1.5.1 driver for the Myri-10G NICs and the ixgbe version 2.0.44.13 driver for the Intel NICs. Both drivers support NAPI and are configured with factory default interrupt coalescence parameters. To enable RouteBricks, we modified the myri10ge driver to work in polling mode with Click (we used Linux kernel version 2.6.24.7 with Click, the most recent version supported).

All values presented are averaged over multiple independent runs, between as low as eight and as high as 32 runs; the error bars denote standard error of the mean and are always present, although most of the time they are sufficiently small to be virtually invisible.

5.3.2 Forwarding / Routing

Figure 5.5 shows the UDP payload throughput for the most basic functionality—packet routing with MTU size packets. We compare the NetSlice implementation with the default in-kernel routing, a RouteBricks implementation, and with the best configurations of pcap user-space solutions. In the absence of receive-livelock, NetSlice forwards packets at nominal line rate (roughly 9.7Gbps for MTU packet size and MAC layer overhead), as do the kernel and RouteBricks routing. However, the best pcap variants top off at about 2.25Gbps. There is virtually no difference between regular pcap and

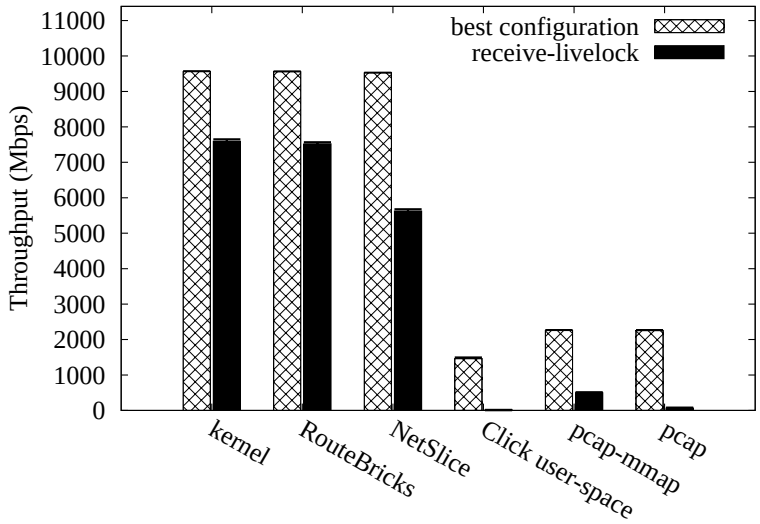


Figure 5.5: Packet routing throughput.

pcap-mmap, while Click user-space does in fact perform worse.

For each case, the Figure shows the additional scenario we previously described as receive-livelock, when all traffic is sent to and handled by a single NIC queue. During receive-livelock, the kernel achieves 7.59Gbps, while NetSlice achieves 74% of the kernel throughput, while pcap-mmap achieves one fifteenth of the throughput achieved by NetSlice, and about 7.6 times better than regular pcap. As expected, in-kernel variants perform better since routing is performed at an early stage, and less CPU work is wasted per dropped packet [169].

The take-away is that the NetSlice kernel to user-space communication channel is highly efficient, even when a single CPU is used and receive-livelock ensues. Moreover, using more than a single NetSlice easily sustains line rate—currently, our clients are not able to generate more than 10Gbps worth of MTU-size packet traffic.

To quantify the performance benefits of the NetSlice batching send / receive operation, we measured throughput for a single NetSlice. Figure 5.6 shows the packet

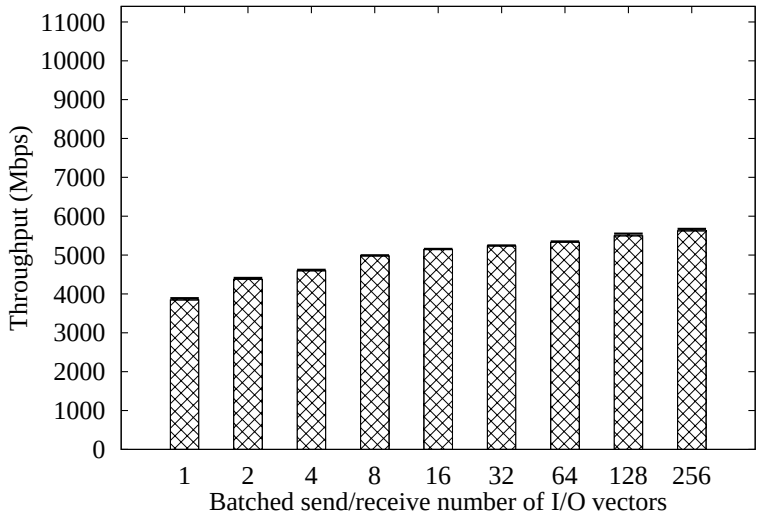


Figure 5.6: Routing throughput for a single NetSlice performing batched send / receive operations.

forwarding throughput for various number of I/O vectors (i.e., the number of packets sent/received during a single system call), in geometric expansion. The Figure shows a 46.2% increase in aggregate throughput from singleton send / receive operations to 256 batched I/O vectors shuttled between user-space and the kernel in a single operation, even though the kernel uses the fast system call processor instructions (`SYSCALL/SYSENTER`).

Next, we evaluate the importance of the u-peer CPU placement. User-space processing takes place on the u-peer CPU as part of the spatial partitioning that isolates individual NetSlices. We used a single NetSlice to stress one communication channel that handles all traffic in isolation. This means that only two tandem CPU cores are utilized, hence the experiment only accounts for direct interference (e.g. cache coherency, cache misses due to pollution) between the tandem CPUs of a single NetSlice—additional indirect interference is expected in the general case. Figure 5.7 shows the throughput given various core placement choices and the number of I/O vectors used for batched

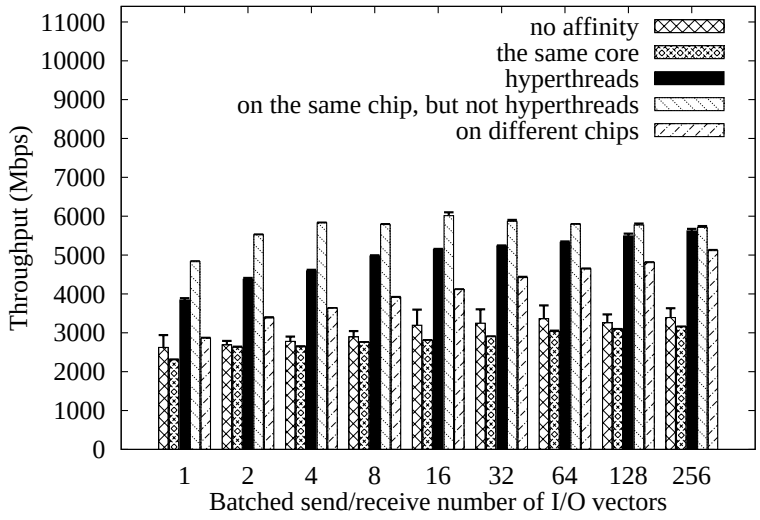


Figure 5.7: Routing throughput for a single NetSlice and different choice of u-peer CPU placement.

operations. There are several key observations. First, if the user-mode task does not use the CPU affinity as instructed by NetSlice, the default choice made by the OS scheduler is suboptimal. Moreover, the high error bars imply that the kernel does not attempt to perform smart task placement. Indeed, the Linux scheduler is primitive in that it typically moves a task on the runqueue of a different CPU only if the current CPU is deemed congested.

The second observation is that using the same CPU core for both in-kernel and user-space processing performs the worst—there are simply not enough cycles to counter the excessive overheads introduced by the context switches. Additionally, there is an impedance mismatch between the task context and the in-kernel processing that happens in a softirq context and is of strictly higher priority than the task. This scenario is complicated further by the kernel's per-CPU `ksoftirqd` threads that are spawned to act as rate-limiters during receive-livelock scenarios in order to give the task the chance to process packets.

The third observation is that same-chip and hyperthread placement outperform the scenario in which the user-space processing happens on a different chip. This is consistent with the memory hierarchy—i.e. accessing the shared L3 cache is faster than accessing data over the QuickPath Interconnect inter-socket link. However, the gap between same-chip and different-chip data access decreases considerably with the increase in the number of I/O vectors. This is because the QuickPath Interconnect link is a packet oriented point-to-point channel, which takes advantage of message passing optimizations like batching and pipelined processing. Interestingly, batched processing also improves the performance of user-space processing on the hyperthread—presumably because the hardware threads still contend for functional units (like ALUs) within the (shared) physical CPU core.

The best case is when the peer CPUs are on the same chip yet are not hyperthreads. However, the Figure shows the scenario in which a single NetSlice is used, hence only the peer CPUs are utilized, all the remaining cores are idle. In the general case, such a placement choice is only viable when there is a lower number of NetSlices than there are available CPUs. By default, NetSlice performs user-space processing on the sibling hyperthread, if one is available. Moreover, having two sibling hyperthreads work on different NetSlices would split the cache levels (higher than the LLC) into half.

5.3.3 IPsec

Next we experiment with IPsec encryption with 128 bit key (typically used by VPNs)—a CPU intensive task. We implemented AES encryption in Cipher-block Chaining (CBC) [45] mode of operation. Our experiments focused on steady-state performance and the key-establishment protocol is not included in the evaluation.

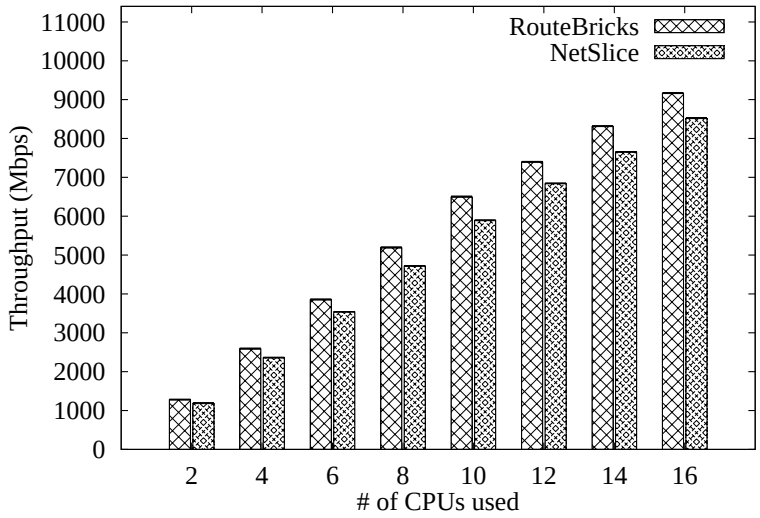


Figure 5.8: IPsec throughput scaling with the number of CPUs (there are two peer CPUs per NetSlice).

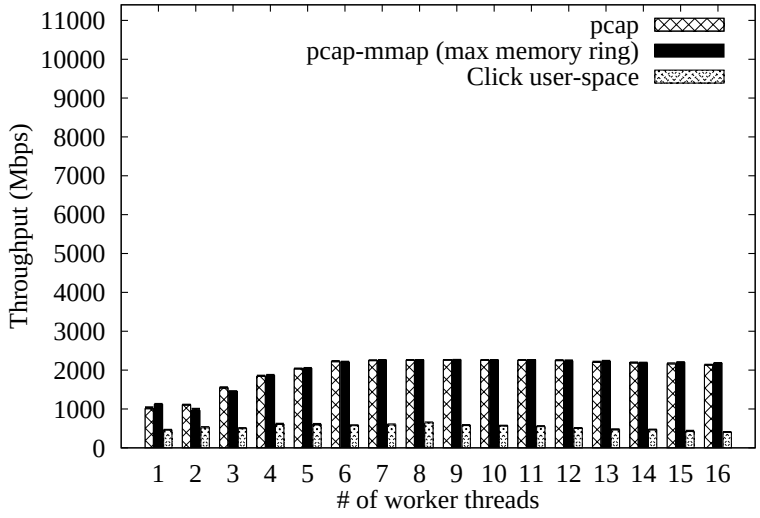


Figure 5.9: IPsec throughput for user-space / raw socket.

We use the IPsec application to evaluate how NetSlice scales with the number of cores. IPsec accelerators typically need all the CPU cycles they can spare and two NetSlices proved sufficient to forward all the 10Gbps MTU-size traffic that our testbed was able to generate. Figure 5.8 shows NetSlice scaling linearly with the number of CPUs,

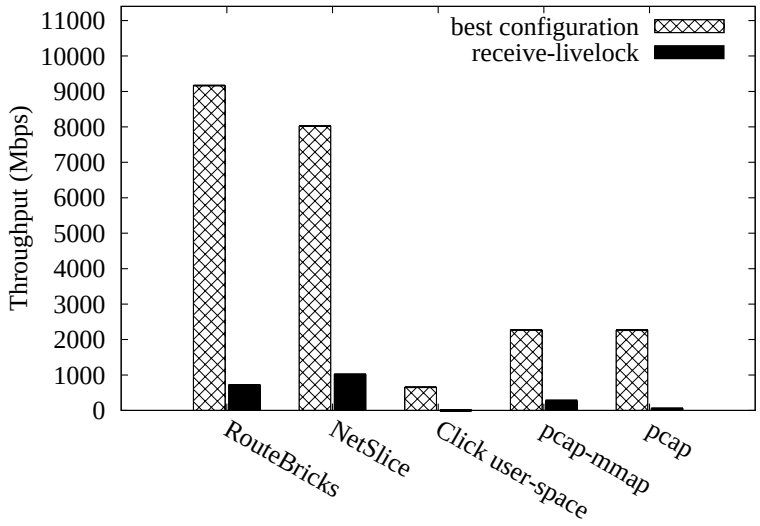


Figure 5.10: IPsec throughput.

closely following RouteBricks. RouteBricks tops off at 9157Mbps, about 600Mbps shy of achieving nominal line rate. NetSlice tops off at about 8011Mbps. We expect both NetSlice and RouteBricks to continue to scale linearly given more cores. By contrast, Figure 5.9 shows that the best of the userspace variants (with a dispatch thread load balancing packets to threads bound to CPUs exclusively) using pcap scale poorly. Hence, they are unable to take advantage of the current technology trend towards placing many independent cores on the same silicon die.

Figure 5.10 shows IPsec throughput results for the best configurations of NetSlice, RouteBricks, and pcap user-space solutions. For each case, the Figure also shows the additional scenario we previously described as receive-livelock (all traffic is sent to and handled by one NIC queue). First, notice that the pcap variants top off at about 2250Mbps in the common case, with poor performance during receive-livelock. Nevertheless, as with routing, the pcap-mmap does outperform conventional pcap during receive-livelock. By contrast, NetSlice, like RouteBricks, vastly outperforms the user-space variants. NetSlice yields better throughput than RouteBricks during receive-

livelock. This is because during receive-livelock, all traffic is routed to a single NIC queue, which RouteBricks handles with a single CPU in kernel mode, whereas NetSlice handles with a pair of CPUs, one running in kernel-mode and one running the user-mode task.

The take away is that NetSlice scales with the number of available cores as good as the in-kernel RouteBricks implementation does. By contrast, user-space implementations that use conventional raw sockets scale poorly. Finally, during a CPU intensive task, NetSlice and RouteBricks vastly outperform the best user-space configurations that rely on conventional raw sockets.

5.3.4 The Maelstrom Protocol Accelerator

Maelstrom [63] is a performance enhancement proxy developed to overcome the poor performance of TCP when loss occurs on high bandwidth / high latency network links. Maelstrom appliances work in tandem, each appliance located at the perimeter of the network and facing a LAN on one side and a high bandwidth / high latency WAN link on the opposite side. The appliances perform forward error correction (FEC) encoding over the egress traffic on one side and decoding over the ingress traffic on the opposite side. In Figure 5.4, for example, the egress and the ingress routers are running Maelstrom appliances, with the egress router encoding over all traffic originating from the clients on the same LAN and destined for the clients on the LAN behind the ingress router. The ingress router receives both the original IP traffic and the additional FEC traffic and forwards the original traffic and potentially any recovered traffic to original destination nodes. In reality, each Maelstrom appliance works both as an encoder and as an decoder at the same time.

The existing hand-tuned, in-kernel version of Maelstrom is about 8432 lines of C code. It is self contained with few calls into the exported kernel base symbols. By contrast, the NetSlice implementation required 934 lines of user-space C code, not counting the NetSlice kernel module or the 263 lines of hash-table implementation. For MTU size packets, NetSlice achieves a goodput of 6993.69 ± 35.7Mbps for a throughput of 8952.04 ± 37.25Mbps. For the nominal FEC parameters we used (for every r=8 packets we send c=3 additional FEC packets), there is a 27.27% overhead, and we achieve close to maximum effective goodput—6993.69Mbps $\times \left(1 + \frac{c}{r+c}\right) =$ 6993.69Mbps $\times \left(1 + \frac{3}{11}\right) =$ 8901Mbps.

Thus we have shown that highly complex protocol accelerators may be built atop NetSlice. Such performance enhancement proxies run on commodity components and scale with the number of cores to achieve line rates.

5.4 Discussion and Limitations

Building upon fwP, the NetSlice operating system abstraction enables high-speed scalable packet processors in user-space, while fully taking advantage of modern hardware resources, like CPU cores and multi-queue NICs. We demonstrate NetSlice by showing that complex user-space packet processors can scale linearly with the number of cores and operate at nominal 10Gbps line speeds. Furthermore, unlike fwP, NetSlice is portable, requiring only a simple modular addition at runtime like any device driver, and it does not introduce an unfamiliar programming interface—instead, the NetSlice API is backwards compatible with the socket API.

Nevertheless, the NetSlice design does have several drawbacks. First, NetSlice does not employ zero-copy techniques to achieve portability, hence processors spend cycles

to copy packets between the kernel and the user-space protection domains, albeit only once per packet and while packets are already in the last level of cache (LLC). Nevertheless, as we argued in Section 5.2.2, Moore's Law works in our favor since the current semiconductor trend is to place an increasing number of cores on the same die. In particular, a few spare CPU cores are sufficient to compensate for the cost of packet copies, provided that data contention is kept to a minimum, which NetSlice does (see Section 5.3).

Second, the NetSlice design relies on the hardware capabilities of modern 10GbE network adapters, namely the hardware multi-queue support. More precisely, NetSlice relies on the NICs ability to classify inbound (i.e., received) packets onto hardware queues in a deterministic fashion, without the intervention from the host CPUs, or at least without the host CPU intervention on the critical data path. For both devices we employed in our experiments, the classification is performed by immutable opaque functions, while for the Intel 10GbE adapter, the classification appears to be nondeterministic. The nondeterminism prevents us from building (efficient) packet processing middleboxes that work in tandem, like the ones exemplified in Section 2.4, while the inability to change the functions potentially leads to pathological traffic imbalance. For example, all the compressed traffic between the *IPdedup* packet processors (see Section 2.4) is classified to be received on the same hardware queue by the Myri-10G NICs. This occurs because the *IPdedup* implementation encodes that a packet was diff-compressed in the protocol field of the IP datagrams. Since the Myri-10G NIC does not recognize the IP protocol, it will default to placing the traffic always on the first queue. (The NIC would otherwise use the protocol field to infer other header fields used in the classification hash function, e.g. TCP or UDP port numbers.)

Nevertheless, we believe that multi-queue NIC technology will continue to mature,

ultimately providing a means for loadable classification hash-functions. Our judgment is based on the fact that multi-queue NIC support is aggressively driven by the current foray of virtualization technologies into the datacenter environment as a means to increase the network performance of guest virtual machines.

5.5 Summary

The end of CPU frequency scaling is ushering in a world of slow cores and fast networks. The immediate impact of this trend is seen on networked systems, like packet processors, that need to process data at wire speeds. In this Chapter we demonstrated how to achieve high data rates while performing packet processing in user-space. In particular, we demonstrate how to reduce memory contention overheads and how to leverage the parallelism intrinsic of modern hardware, like multi-core processors and multi-queue network interfaces, to improve application packet processing performance. We present NetSlice—a new operating system abstractions that embodies these techniques, and we demonstrate that complex user-space packet processors built with NetSlice can scale linearly with the number of cores and operate at nominal 10Gbps line speeds. Consequently, developers may use NetSlice to build general purpose packet processing performance enhancement protocols to ultimately improve the datacenter communication.

CHAPTER 6

RELATED WORK

6.1 Network Measurements and Characterization

There has been a tremendous amount of work aimed at characterizing the Internet at large by analytical modeling, simulation, and empirical measurements. Measurements, in particular, have covered a broad range of metrics, from end-to-end packet delay and packet loss behavior [71, 89], to packet dispersion (spacing) experienced by back-to-back packets [82], packet inter-arrival time [134], per-hop and end-to-end capacity, end-to-end available bandwidth, bulk transfer capacity, achievable TCP throughput, and other general traffic characteristics [91]. However, there has been little work aimed at characterizing uncongested semi-private or dedicated networks [195], like modern optical lambda networks.

The need for instruments with which to perform such measurements has led to the development of a myriad of tools [82, 141, 104, 105, 139, 190]. These tools are typically deployed in an end-to-end fashion for convenience and often embody a tradeoff between intrusiveness and accuracy [186]. For example, some tools rely on self-induced congestion, while others rely on relatively small probes consisting of packet pairs or packet trains. Tools like these have become essential and provide a solid foundation for measurements; for example, we have saved significant time by working with (and extending) the existing Iperf [204].

Internet measurements provide a snapshot of the characteristics of the network at the time the measurements are performed. For example, in its early days, the Internet was prone to erratic packet loss, duplication, reordering, and the round-trip time delays were

observed to vary over a wide range of values [193]. Today, none of these issues remain, although other challenges have emerged.

Historically, networks have been characterized as they became available—ARPANET, its successor, NSFNET [91, 134], and the early Internet [193] have all been the focus of systematic measurements. Murray et al. [172] compared end-to-end bandwidth measurement tools on the 10GbE TeraGrid backbone, while Bullot et al. [80] evaluated the throughput of various TCP variants by means of the standard Iperf, over high-speed, long-distance production networks of the time (from Stanford to Caltech, to University of Florida, and to University of Manchester over OC-12 links of maximum throughput of 622Mbps)—similar to the experiments in Section 3.2.3.

However, unlike our experiments, Bullot et al. [80] focused on throughput and related metrics, like the stability (in terms of throughput oscillations), and TCP behavior while competing against a sinusoidal UDP stream. Although disregarding loss patterns and end-host behavior, the authors did provide insight into how the `txqueuelen` parameter (i.e., the capacity of the backlog queue between the IP layer and the DMA rx ring—currently made obsolete by NAPI) affects throughput stability. In particular, larger values of the `txqueuelen` are correlated with more instability. An equally interesting observation was that reverse-cross-traffic affects some TCP variants more than others, since they alter ACK delivery patterns (e.g. ACK compression due to queueing or loss). It is also worth noting that the authors performed a set of tentative TCP performance measurements on 10Gbps links, using jumbo (9000-byte) frames.

By contrast, relatively few works have investigated the effect of traffic patterns on end-hosts and their ability to handle such traffic, especially when connected to uncongested lambda networks. Mogul et al. [169] investigated the effect of high data rate traffic on the end-host, noting that a machine would live-lock and spend all available

cycles while handling the interrupt service routine as a result of packets being received, only to drop these packets at the subsequent layer, and hence fail to make forward progress. Consequently, NAPI [25] and on-board NIC Interrupt Throttling have been widely adopted, to the point where they are enabled by default in vanilla kernels. On the other hand, an interesting study looked at how "interrupt coalescence" (produced by NAPI possibly in conjunction with Interrupt Throttling) hinders active measurement tools that rely on accurately estimating packet dispersion to measure capacity and available bandwidth [187]. Since the packets were time-stamped in user-space, context switches at the receiver cause similar behavior as packet batching.

6.2 High-speed Long-distance Transport

TCP/IP is the de-facto standard for unicast communication, especially in datacenters, where it makes up 99.91% of all network traffic [50]. However, TCP/IP does not work well on networks with high bandwidth delay products, a fact that has been established through analytical proofs as well as practical experience [150, 180]. Consequently, the last decade has seen much research aimed at developing viable TCP/IP variants and alternatives for high-speed long-distance networks. Some approaches like XCP [142] have brought good results but require significant changes to the routing infrastructure. Other approaches like TCP Vegas [75] or more recently FAST TCP [217] use delay as a congestion signal. Still some approaches attempt to differentiate between loss caused by congestion and non-congestion events, to enable TCP/IP to react appropriately by cutting back on the window size only when congestion occurs [183, 112, 76, 50]. In a similar vein, other TCP variants replace the 'Additive Increase Multiplicative Decrease' (AIMD) curve with more sophisticated control curves. For example, the TCP BIC [226] protocol window resizing mechanisms alleviate problems that commodity TCP/IP has

with preserving fairness across flows with different window sizes and different RTTs. TCP CUBIC [128] further improves on BIC's fairness properties, factoring in the passage of wall-clock time while resizing the window to prevent the RTT of a flow from influencing the growth of the congestion window.

A large body of research in high-speed data transfer has emerged from the efforts of the e-science and grid computing communities, which were early adopters of high-bandwidth long-haul networks for shuttling large data-sets between supercomputing sites [77, 98, 113, 175]. A Grid is a set of distributed, networked, middleware-enabled computing, storage, and visualization resources [144]. A LambdaGrid is a Grid in which lambdas form end-to-end connections (lightpaths) among computing resources [77]. Many of these solutions replace TCP/IP with application-level protocols that operate above UDP. For instance, SABUL [125] and Tsunami [166, 211] are application-level flow control protocols that use UDP for the data plane and TCP for control plane. Similarly, RBUDP [133] is an aggressive 'blast' protocol that attempts to send data over UDP at the maximum constant rate the network will allow. An interesting alternative solution that does not require changing or eliminating commodity TCP/IP involves striping application-level flows across multiple parallel TCP/IP flows, an idea first described in PSockets [197].

Performance enhancing proxies were proposed in RFC 3135 [72] as a means to enhance TCP/IP performance on specialized networks—such as wireless networks, satellite networks, or long-distance links—without changing the end-host protocol stack [207, 85, 171, 167].

6.3 Intra-datacenter Transport

Within a datacenter, new network architectures and protocols have also been recently proposed: DCTCP [50], VL2 [123], Monsoon [124], SEATTLE [145], Portland [177], DCell [127], BCube [126]. These are designed to solve recently identified problems, such incast onset [102, 88, 185, 206] and increased round-trip time estimates measured within virtual machine instances [212] (TCP acknowledgment offloading [140]).

6.4 Packet Processors

In practice, developers can implement high-speed packet processing applications today in one of five ways. None of these explicitly take advantage of multi-core chips, nor do they reduce memory pressure:

1. As runtime loadable modules—Writing kernel code demands a great degree of sophistication from the developer. While modules allow for rapid compilation and deployment of new functionality, they do not alleviate the difficulties of kernel development.
2. As packet filters [73, 164]—Packet filters are extremely fast and highly optimized, but are severely limited in the set of things they can do. Filters are typically stateless and building arbitrarily complex functionality within them is almost impossible.
3. As safe kernel extensions—This requires using an extensible kernel like SPIN [67] where the OS is written in a type-safe language. Most developers writing high-performance applications are constrained to use commodity OSes and / or conventional languages like C.

4. Using an embedded / real-time OS—Many commodity OSes provide a real-time branch (e.g. RTAI [191] or PREEMPT_RT [188]), used mostly in settings where predictable real-time responses are required (e.g. to control traffic lights or industrial laser welder). However, such OSes typically achieve real-time responsiveness via strict scheduling mechanisms that sacrifice high throughput. Packet batching techniques like NAPI [25] and device interrupt coalescence that significantly increase network throughput are incompatible with such an operating system.

5. Using user level device drivers—This approach had little traction within mainstream OSes, dismissed for severe performance degradation due to system call, context switch, data copying, and event signaling overheads. Recent attempts [153] try to alleviate the overheads of data copying, without addressing the remaining issues.

A large body of research literature is also relevant to building high-speed packet processors in software:

Symmetric Multiprocessors (SMP) Early work on packet-level parallelism [174] has found its way into commodity OSes like Linux in the form of interrupt balancing and the per-CPU network stack. However, it has been well known that large scale cache coherent—potentially NUMA—multiprocessors require careful operating system design, or else bottlenecks prevent the systems from reaching their performance potential. Indeed, operating systems like Tornado/K42 [205, 149] have been carefully designed to minimize contention by clustering and replicating key kernel data structures, and by employing intricate scheduling algorithms that, for example, take NUMA locality into account.

More recently, there have been several research efforts that aimed at redesigning the

OS from the ground up in order to effectively exploit the emerging and now ubiquitous multi-core architectures. Corey [74] is an ExoKernel-like OS within which shared kernel data structures and kernel intervention are kept to a minimum, while applications are given explicit control over the sharing of resources. This allows the Corey kernel to perform finer grained locking of highly accessed data structures, like process memory regions. The Barrelfish research operating system [66] explores how to structure the OS as a distributed system in order to best utilize future multi- and many-core, potentially heterogeneous systems. Similarly, the Helios [176] operating system tackles building and tuning applications for heterogeneous systems through satellite kernels. Satellite kernels export a uniform set of OS abstractions across all CPUs and communicate one with another by means of explicit message passing instead of relying on a cache coherent memory system. The Tessellation OS [155] introduces a "nano-visor" to enforce strict spatial and temporal resource multiplexing between library OSes. To ensure resource isolation, the Tessellation OS envisions hardware support for resources that have been traditionally hard to share, like caches and memory bandwidth.

By contrast, fwP focuses on the network stack and high-performance packet processing, and has the orthogonal goal of minimizing memory pressure, including the effects of locking due to cache coherency. Like the Tessellation OS, NetSlice performs spatial partitioning of resources at coarse granularity, in particular, it partitions the CPU, memory, and multi-queue network interface controller (NIC). However, the NetSlice partitioning is domain specific, and the performance isolation need not be strongly enforced, instead it is implicit by the design of the NetSlice abstraction itself.

Zero-Copy Architectures Historically, there have been a large number of zero-copy user-space network stacks proposed [182, 118, 56, 209, 208, 69, 106, 181]. Their general approach was to eliminate the OS involvement on the communication path, and

virtualize the NIC while providing direct, low-level access to the network. Some of these approaches relied on hardware support. For example, U-Net [208] and its commercial successor VIA [108, 61], required a communications co-processor capable of demultiplexing packets into user-space buffers, and an on-board MMU (Memory Management Unit) to perform RDMA (Remote DMA) [52]. The former is not available on typical commodity Fast Ethernet NICs, and as a result U-Net/FE [218] requires an extra copy for every received frame, as well as a system call to send every packet. A further obstacle to this approach is the inability of current modern IOMMUs to handle page faults [117]. VIA over Gigabit Ethernet has enjoyed success in the cluster computing community, though still relying on a specialized chipset [61].

Other techniques relied on virtual memory and page protection techniques to improve performance by avoiding unnecessary copies. Fbufs [106] introduced copy avoidance techniques across protection boundaries by using immutable buffers, and IO-Lite [181] described the composition of these buffers. Brustoloni et al. [78, 79] used virtual memory mechanisms in conjunction with a set of new techniques, like input-disabled page-out, to provide high performance I/O with with emulated copy semantics over the `write/read` API. However, on demand memory mapping of shared buffers is particularly tricky and can be unnecessarily expensive; as a result, such techniques are yet to be adopted by mainstream kernels.

Importantly, zero-copy techniques were proposed in the context of single processor machines and do not explicitly handle memory contention. Additionally, they were targeted at general end-host applications and did not allow the zero-copy interface-to-interface forwarding central to packet processing applications. By contrast, fwP performs an additional copy in L2 cache, taking advantage of locality while harnessing the aggregate horsepower of multiple cores. NetSlice is orthogonal to past work that

relocated the network stack into user-space; further, NetSlice does not use zero-copy techniques since currently, as our evaluation shows, they were not necessary.

Packet Routing / Processing Software routers Achilles' heel has been, and continues to be, the low performance with respect to their hardware counterparts. Nevertheless, recent efforts, like RouteBricks [103], have shown that modern multi-core architectures and multi-queue NICs are well suited for building low-range software routers, albeit in kernel-space. RouteBricks relies on a cluster of PCs fitted with Nehalem multi-core CPUs and multi-queue NICs, connected through a k-degree butterfly interconnect. Packets are forwarded / routed at aggregate rates of 24.6Gbps per PC, however, the interconnect routing algorithm introduces packet re-ordering.

Internally, RouteBricks uses the Click [147] modular router—an elegant framework for building functionality from smaller building blocks arranged in a flow graph. However, Click is aimed at building routers and does not easily express general packet processing; e.g., it cannot support global state that extends across building blocks. NetTap [69] was specifically designed to support building packet processing applications like routers and bridges in user-space. NetTap modified the BSD kernel to allocate all *mbufs* (the data structure holding frames) from a single pinned region, which can be mapped into application address space; however, the interface fails to address synchronization between modern SMP kernel and user-space threads and safe *mbuf* reclaiming.

Active networks [203] proposed that routers run arbitrary code as instructed by untrusted packets—for example, packets may carry code that would decide how they should be routed. By contrast, fwP and NetSlice both require a trusted user to alter the packet processing performed by a router.

Both fwP and NetSlice can be effectively used to provide rapid prototyping of Open-

Flow [165] forwarding elements. For example, the current reference NetFPGA [157] implementation is limited to four 1GbE interfaces, whereas NetSlice is only limited by the number of CPUs and PCIe connections a commodity server can support. Moreover, developers need not have intimate Verilog knowledge, or worry about details such as gateware real-estate.

In general, software routers are implemented within the kernel, early in the network stack and below the (raw) socket interface. Full blown software routers like RouteBricks [103] may require distributed coordination algorithms to decide interconnect forwarding paths [49]. By contrast, fwP and NetSlice provides support for user-space implementation of individual packet processing units, independent of interconnects. Therefore, complex packet processing logic, like multi-dimensional packet classification [159] or the RouteBricks' distributed coordination may be seamlessly built using fwP and NetSlice (note that fwP and NetSlice elements do not introduce packet reordering, which may otherwise severely cripple the performance of TCP).

Packet Capture Tools The `PACKET_MMAP` socket option [33] is an extension to raw `PF_PACKET` sockets that allows receiving packets onto a size configurable circular buffer mapped by the user-space application. Optionally, packets may also be enqueued on the socket buffer, subject to the kernel global limits. The user-space application can then poll the arrival of new packets, at which point packets can be received without the cost of issuing an additional system call per packet. The same net effect of this is achieved by the NetSlice batching read operation, however unlike NetSlice batched transmit, `PACKET_MMAP` sockets do not offer the same support for outbound packets, which means they do not facilitate interface-to-interface forwarding. Moreover, it is important to note that `PACKET_MMAP` sockets do not provide zero-copy receive—each packet is copied the same number of times as with a traditional socket. Importantly,

PACKET_MMAP sockets suffer the same performance debilitating symptoms as regular raw sockets our evaluation shows in Section 5.3.

Zero-copy packet capturing tools like nCap [100] map the NICs descriptor rings and frame buffers into the application's address space. There are several issues with this approach—it requires a device-specific rewrite of the device driver, which in turn has to coordinate with a user-space library. Also, there is no efficient, general way of delivering events to the application (e.g. interrupt delivery). Likewise, since nCap takes exclusive control of the interface, demultiplexing packets to applications is expensive (requiring copy or memory re-mapping). Moreover, it needs the tight cooperation of the kernel scheduler, otherwise the user-space application may miss DMA traffic [73, 96].

In general, packet capture tools are not designed for building packet processing applications and interface-to-interface forwarding is an inefficient operation, often requiring a copy and a system call per message, and are excessively costly in a multi-core environment due to locking and scheduling effects.

Inter-process Communication (IPC) Circular buffers have been used for decades to relay data between producers and consumers, typically in cases where dynamic memory allocation is infeasible; e.g., NIC ring buffers, hypervisor ring buffers [213, 65], beltway buffers [94] for packet filters [73], and so on. The fwP buffers are different than vanilla circular buffers in that they consist of a tandem of tightly coupled circular buffers holding pointers to frame buffers. This enables operations like atomic pointer swapping between the rings, thus enabling zero-copy receive, in-situ processing, and forwarding of packets.

Lightweight RPC [68] proposed a mechanism that used a stack handoff technique for passing arguments, using a call-by-value return semantics, where the caller thread

continued in the callee context. This is not always an option when the kernel does an upcall into user-space. Also, the assumption was that most communication is simple, involving few arguments and little data, which is hardly the case with network frames.

CHAPTER 7

FUTURE WORK AND CONCLUSION

7.1 Future Work

The network has become a core component of the datacenter ecosystem, some may go as far as claiming that the "network *is* the distributed operating system." In the future, we wish to provide datacenter operators and developers with systems and services that allow them to control and leverage the next-generation network of datacenters. We expect these systems to harness the modern commodity hardware designs, like multi-core processors and high speed network adapters, and leverage lambda network pathways that connect peer datacenters. Furthermore, due to the enormous scale at which a network of datacenters operates, power awareness and fault-tolerance will be first-class design principles. We plan to start by addressing the following key challenges: **(i)** How to leverage the new packet processing abstractions to extend and improve the datacenter network substrate? **(ii)** How to perform "ground-truth" accurate network measurements to reveal the underlying characteristics of lambda networks? **(iii)** How to utilize the networking substrate effectively, that is currently power-hungry yet largely underutilized? **(iv)** How to tolerate varying degrees of faults automatically, within and across datacenters? **(v)** How to strike a balance between processing speed and energy utilization for the applications resident in the datacenter?

Extensible router and packet processing support is key to enabling and improving the next generation datacenter networking. Accordingly, we plan to provide NetSlice as an alternative, more flexible, forwarding element for emerging network technologies—like OpenFlow [165], RouteBricks [103], and PortLand [177]—which depend heavily on extensible router or extensible switch support. Currently, OpenFlow and PortLand rely

on NetFPGA [157] hardware for their reference implementations, whereas RouteBricks relies on the in-kernel Click [147] software modular router to perform packet forwarding. Using NetSlice instead would ease the developer burden considerably, and it would provide much needed fault-isolation. For example, with RouteBricks we plan to replace Click forwarding elements with NetSlice forwarding elements, while maintaining the original distributed coordination algorithm that decides packet forwarding paths along the interconnect. Furthermore, there are a large number of user-space applications and tools that are currently built using the raw socket. Such tools would greatly benefit from the performance improvement provided by the NetSlice and fwP abstractions. A notable example is the Packet CAPture (PCAP [38]) library that is the de-facto interface used by developers in general to build various flavors of network protocol and traffic analyzers [221, 198]. In the near future, we plan to provide support for the PCAP library built atop NetSlice. This would enable conventional applications that relied on PCAP to be seamlessly ported and take advantage of NetSlice in a transparent fashion.

Existing Internet measurement methods and conventional wisdom does not always apply in the newly emerging fiber-optic world. As we have shown in Chapter 3, even in an uncongested lambda network, packets can be lost, inter-packet spacing disrupted, and other issues arise. Only by understanding the causes of these phenomena can we develop accurate network models that would enable the development of new protocols to overcome the problems. Yet, existing tools, measurements, and experimental methodologies do not provide the high-resolution timings needed for creating those accurate models. In fact, end-host network measurement software on commodity hardware had completed masked the phenomena that we observed and now attribute to causing packet loss in a lightly loaded fiber-optic network [115, 163]. In order to characterize lambda networks and understand how the physical layer impacts the performance of the rest of the networking stack, we have recently designed and developed a software defined

network adapter (SDNA) [115] apparatus. The SDNA is a high-precision instrumentation apparatus to enable generation of extremely precise network traffic flows, and their capture and analysis. With SDNA, we can generate and acquire analog traces in real-time directly off optical fiber, using typical physics laboratory equipment (oscilloscopes, lasers, etc.) and an off-line software (post-/pre-)processing stack. SDNA achieves six orders of magnitude improvement in timing precision over existing software end-host measurements and two to three orders of magnitude relative to prior hardware-assisted solutions. We plan to conduct network experiments using SDNA in various controlled environments and diverse network settings, like the Cornell NLR Rings testbed, to answer interesting questions such as: Under what conditions does a steady stream of packets degenerate into packet convoys and what is the expected length of packet convoys based on the length of a path in an uncongested lambda network?

Energy utilization in modern datacenters has recently received an enormous amount of attention because of the growing importance of leaner operational budgets and long-term environmental impact. We also plan to explore power saving opportunities across the entire spectrum of components, and at various layers. For example, current network protocols and equipment were not designed with power awareness in mind, hence they are not always efficient. Particularly, wired networks, like 10GbE, send bit-streams constantly on the wire at the maximum symbol rate, consisting of mostly *idle symbols* and the occasional data packet sandwiched in between. This means that network elements are expending energy to encode symbols that are useless idles into frames, and likewise expend energy to decode frames into symbols that are predominantly useless idles.[1] The SDNA apparatus allows us to rapidly prototype and implement in software new physical medium dependent (PHY) layers. We intend to leverage the SDNA apparatus to investigate and explore the feasibility of alternative power efficient PHY layers.

[1]Except for the time intervals during which packets are sent at a sufficiently high data rate so as to have small inter-frame gaps consisting of idle symbols.

The sheer scale at which a network of datacenter operates makes failures a common and frequent event [119, 97, 87]. However, mirroring or replicating data at a remote location is highly sensitive to the latency between the sites. Consequently, many organizations trade off the safety of their data for performance. To offer stronger guarantees on data reliability without sacrificing performance, we designed and developed the Smoke and Mirrors File System (SMFS) [215]. The SMFS design relies on a packet processing middlebox at the perimeter of the site, or datacenter, which further increases the reliability of the high-speed optical link by introducing proactive redundancy (e.g., Forward Error Correction [63]) at the network level in addition to the transmitted data. Furthermore, the middlebox sends feedback to end-hosts which makes it possible for a file system (or other applications) to respond to clients as soon as enough recovery data has been transmitted to insure that the desired safety level has been reached (e.g., if the link's reliability is the same as that of a local disk). Using this new mirroring mode, which we named *network-sync*, SMFS effectively hides the latency penalty incurred over long distance links due to the limit on the wave propagation speed of a transmission medium. We are beginning to think on ways to enable heavier weight fault-tolerant protocols [84, 151, 86] that can similarly benefit from using packet processing middleboxes to improve performance.

Since power consumption in the datacenter is of paramount importance, we are currently designing a storage system for the datacenter that aims to save power by significantly reducing the number of disks that are kept spinning. We plan to leverage the intrinsic parallelism of commodity servers to achieve high levels of performance within a given power envelope. More precisely, we are exploring how to generalize NetSlice's spatial partitioning approach to disk hardware resources—*DiskSlice.* In particular, we envisage a storage system to have different slices, or DiskSlice execution contexts, working independently on different data while accessing separate hardware resources. This

paradigm naturally fits a RAID mirroring scheme, with cross-slice communication occurring off the critical path—typically only during the rebuild phase of the RAID array, or during journal log cleanup. Our approach is to use DiskSlice to overlay a log abstraction over a fault-tolerant mirrored multi-disk storage array. Consider for example such a storage system on top of ten disks, of which five are used as primaries and five as mirrors (typical of a RAID 1 mirroring scheme without parity or striping). A log structured storage system writes only to the log head, hence it continuously writes to the same two disks for long periods of time (for as long as it takes to fill these disks). Therefore, DiskSlice has the opportunity to power down the four remaining mirror disks. Read requests are served from the primary disks that are still powered up, trading off read throughput for power savings. Further, periodically cleaning up the tail of the log, which requires spinning up the mirror disks, is not on the critical path, so the storage system may incur the large latency penalty hit due to powering up disks on demand. Importantly, the storage system ensures stable read and write throughput even during log cleanup. In particular, stable write throughput is achieved since the head and the body of the log are placed on different disks, while stable read throughput is ensured by serving reads from one mirror while the other is being cleaned.

7.2 Conclusion

The modern datacenter has taken the center-stage as the dominant computing platform that powers most of today's consumer online services, financial, military, and scientific application domains. Further, datacenters are becoming a commodity themselves, and are increasingly being networked with each other through high speed optical networks for load balancing (e.g. to direct requests to closest datacenter and provide a localized service) and fault tolerance (e.g. to mirror data for protection against disaster scenarios).

Consequently, virtually every operation within and between datacenters relies on the networking substrate, making the network a first class citizen.

Despite the fact that the network substrate is such an indispensable part of the datacenter, applications are unable to harness it to achieve the expected levels of performance. For example, although long distance optical lambda networks have sufficient bandwidth, are dedicated for specific use, and operate with virtually no congestion [29], end-hosts and applications find it increasingly harder to derive the full performance they might expect [154, 46, 150, 180]. The end-to-end characteristics of commodity servers communicating via high-bandwidth, uncongested, and long distance optical networks—typical of inter-datacenter communication—have not been well understood [163].

Consequently, in order to enhance the network substrate, new network protocols are being constantly developed to stitch together commodity operating systems, previously designed for conventional stand-alone servers, and to provide efficient large-scale coordinated services. However system support for building network protocols in general, and packet processing network protocols in particular, continues to lag. Developers are typically forced into trading off the programmability and extensibility of commodity computing platforms (e.g. general purpose servers with several network interfaces) for the performance (in terms of data and packet rate) of dedicated, hardware solutions.

The emerging network of datacenters, and the datacenters themselves, require proper abstractions that allow developers to build network protocols which power the next generation of online services. This thesis takes a step towards addressing this gap. First, the thesis provides a study of the properties of commodity end-host servers connected over high bandwidth, uncongested, and long distance lambda networks. We identify scenarios associated with loss, latency variations, and degraded throughput at the attached commodity end-host servers. Interestingly, we show that while the network core

is indeed uncongested and loss in the core is very rare, significant loss is observed at the end-hosts themselves—a scenario that is both common and easily provoked. Therefore, conventional applications find it increasingly harder to derive the expected levels of performance when communicating over high-speed lambda networks.

Second, the thesis shows how packet processors may be used to improve the performance of the datacenter's communication layer. Further, we show that these performance enhancement packet processors can be built in software to run on the commodity servers resident in the datacenter and can sustain high data / packet rates.

Third, and last, the thesis extends the operating system with new abstractions that developers can use to build high-performance packet processing protocols in user-space, without incurring the performance penalty that conventional abstractions engender. Importantly, these new operating system abstractions allow applications to achieve high data rates by leveraging the parallelism intrinsic of modern hardware, like multi-core processors and multi-queue network interfaces. In particular, we demonstrate that the performance of packet processors build with these abstractions scales linearly with the number of available processor cores.

APPENDIX A

BACKGROUND ON COMMODITY PROCESSOR ARCHITECTURES

Commodity servers, and most general and special purpose computers are designed based on the *von Neumann architecture* [135]. In the von Neumann architecture (depicted in Figure A.1), the central processing unit (CPU) uses a single separate memory store—typically volatile random access memory (RAM)—to hold both instructions of programs and the data the programs operate on. Peripheral devices are also separate from the processor and are accessed by the CPU either through port-mapped or memory-mapped input/output (I/O). Port-mapped I/O requires CPUs to use a distinct category of instructions for performing I/O, whereas memory-mapped I/O allows CPUs to read and write to devices by simple memory access at pre-assigned addresses. Most modern devices have memory-mapped controllers. This strict separation between the processor and the memory has led to the von Neumann bottleneck [214, 59], which is expressed as the limited throughput (or data transfer rate) between the CPU and the memory, when compared to the total amount of memory.

In practice, the von Neumann bottleneck has been aggravated by the physical limitations of current semiconductor technology: the inability of memory access times to keep up with CPU frequency increases. With every new processor generation, single CPU speed and memory size have both increased at a rate that far outpaced the increase in (external) memory throughput. Therefore, the data transfer rate between memory and CPU is significantly smaller than the rate at which a CPU operates, which severely limits the efficiency of programs that access memory often, e.g. programs that perform basic instructions (little computation) on large volumes of data, especially if the data is only needed for brief periods of time. In such cases, the CPU will continuously stall (i.e. idly wait) while data is being transferred to and from the main memory. This problem has

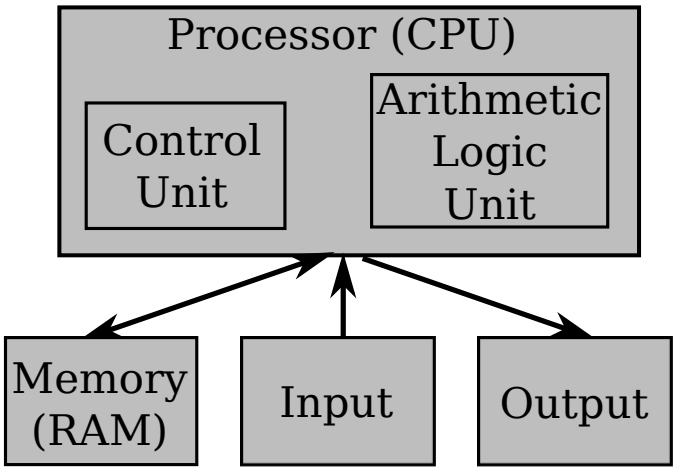


Figure A.1: Diagram of von Neumann architecture.

been named the *memory wall*.

There are several common mechanisms that alleviate the memory wall bottleneck. However, the fundamental problem remains. The most common mechanism has been to provide caches that are smaller than main memory and also are faster to access than main memory is. Typically there are a hierarchy of caches, with the smallest cache being the fastest and the closest to the CPU and the largest cache being the slowest and the farthest . (Currently, caches are situated on-chip, but in the past off-chip caches were commonplace.) For example, modern CPUs may have a split level-1 (L1) cache with a capacity for 32KB of instructions and 32KB of data, a unified 256KB L2 cache, and a unified 8MB L3 cache. Split data and instruction paths—as implemented by typical L1 caches (and reminiscent of the *Harvard architecture* [135])—are also a mechanism for improving the CPU-to-memory throughput, as are pipelined, superscalar, out-of-order execution logic components, and complex branch prediction units.

For example, a pipelined processor design, depicted in Figure A.2, splits the processor operation into stages, and each stage works on one instruction at a time, per time unit (or cycle). This technique increases the throughput of instructions retired (that have

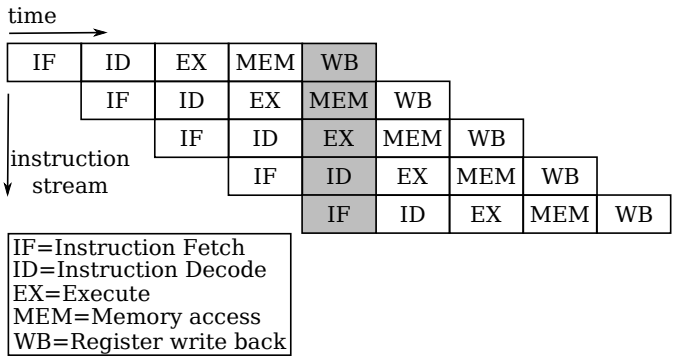


Figure A.2: Canonical five-stage pipeline in a processor. Shown in the gray (shaded) column, the earliest instruction in the WB (register write back) stage, and the latest instruction being fetched (IF).

been executed) per cycle, by utilizing the functional units in parallel, on subsequent instructions. A superscalar processor may execute more than one instruction during a clock cycle, by simultaneously dispatching multiple instructions to redundant functional units. For example, a simple superscalar design is achieved by duplicating the pipeline stages in Figure A.2. Additionally, an out-of-order execution engine may dispatch instructions that were fetched more recently instead of older ones, based on the availability of the instructions' input operands. By preventing the pipeline from stalling, due to input operand unavailability, the CPU throughput, in terms of retired instructions per cycle, is increased.

However, instruction level parallelism techniques (ILP) like superscalar, pipelined, and out-of-order execution have themselves hit the *ILP wall*—the increasing difficulty of extracting sufficient parallelism from a single instruction stream to keep a single processor from stalling. Moreover, single CPU frequency scaling has ceased due to the physical limitations of current semiconductor technology (the transistor leakage current increase which leads to excessive energy usage and heat dissipation that in turn requires large amounts of energy for cooling)—also known as the *power wall*. For that reason, the industry has shifted to relying on (von Neumann) systems with multiple processor

cores (currently on the same chip) to continue to ride Moore's Law. Moore's Law states that the number of transistors that can be placed inexpensively on an integrated circuit has doubled approximately every two years.

Though in the past, symmetric multiprocessor (SMP) systems (systems equipped with two or more *identical* CPUs) were uncommon for commodity computing devices—being used mostly by high-end computing systems—the recent advancements in semiconductor technology, and the power wall, made them affordable. Moreover, the current trend of placing multiple CPU cores on the same chip means that a SMP system can be built in an integrated fashion at a lower cost. For example, the Intel Nehalem commodity processors used in the experimental setup described in Section 5.3 are each equipped with two quad-core CPUs (i.e. there are two distinct CPU chips, and each chip has four independent processing cores). Further, certain CPUs provide several *virtual* hardware processors per core (a technique called *hyperthreading*). With hyperthreading, each core may provide a pair of virtual processors by duplicating certain sections of the processor—those that store the architectural state, e.g. the registers as defined by the instruction set—while sharing the main functional / execution units (e.g. the floating point unit) and all levels of cache. Should hyperthreads not contend for the same functional resources, and should the instruction fetch unit be capable of issuing more than one instruction per cycle, hyperthreads may execute simultaneously. There is a tradeoff in that all hyperthreads of the same core would be executing on the same shared caches, potentially increasing the cache miss rate for all via cache pollution.

Nevertheless, the problem with such commodity multicore architectures is that they do not alleviate the von Neumann bottleneck, instead they aggravated it further. This is because in an SMP system, multiple processors are competing for the bandwidth to the same memory banks, hence multiple CPUs may starve for data at the same time. For

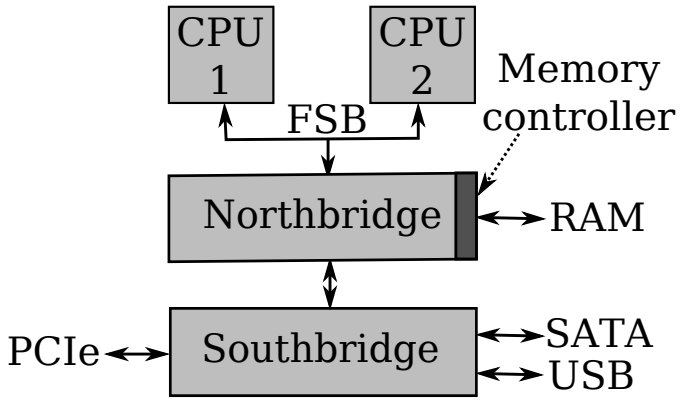


Figure A.3: Diagram of dual-CPU commodity system with front-side bus (FSB). The peripheral devices are connected to the Southbridge, which acts as an I/O hub. The Southbridge is in turn connected to the Northbridge. The figure depicts the Southbridge with the following interfaces: Peripheral Component Interconnect Express (PCIe) bus (e.g. to attach high speed 10GbE network cards), Serial Advanced Technology Attachment (SATA) bus (e.g. to attach mass storage devices such as hard disk drives), and Universal Serial Bus (USB).

example, until recently, the most common commodity architecture connected all processors with the memory through a single *shared-bus*, named the *front-side bus (FSB)*—Figure A.3 depicts such an architecture with two processors.[1] The memory controller (in this case a single entity within the Northbridge) serializes the accesses amongst the many CPUs that compete for the FSB's bandwidth. Furthermore, since more CPU chips have distinct cache hierarchies, a *cache coherency* protocol must also be implemented over the FSB, so that all processors have a consistent view of the entire physical memory address space. While a single multi-core chip implements internally the cache coherency mechanism, multiple chips are required to participate in a coordinated protocol. Describing the precise cache coherency protocols most commonly used today (i.e. the Modified-Exclusive-Shared-Invalid, or MESI, protocol) is beyond the scope of this dissertation, however, the underlying mechanism ensures that if a CPU modifies a memory

[1] This is a simplistic description that does not consider the chipset's North and South-bridge by which the CPUs are connected to memory banks as well as peripheral devices. Nevertheless, these details do not alter the argument.

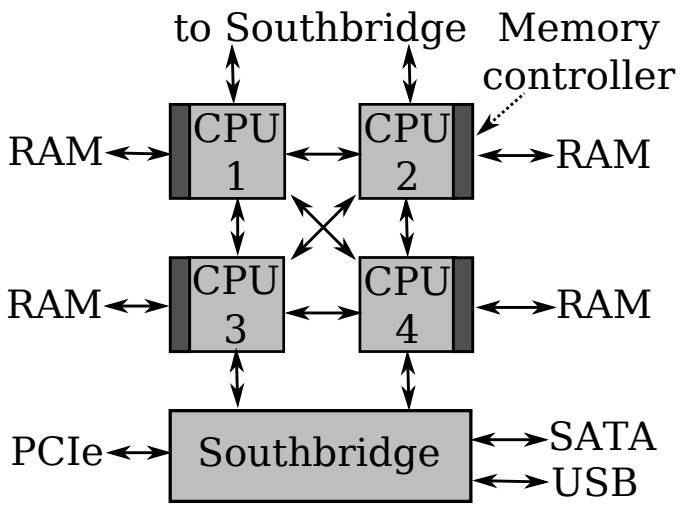


Figure A.4: Diagram of quad-CPU commodity system with integrated memory controllers (note the Northbridge is lacking) and point-to-point interconnects between the processors. The figure depicts the Southbridge (also known as the I/O hub) with the following interfaces: Peripheral Component Interconnect Express (PCIe) bus (e.g. to attach high speed 10GbE network cards), Serial Advanced Technology Attachment (SATA) bus (e.g. to attach mass storage devices such as hard disk drives), and Universal Serial Bus (USB).

location in its local cache, then the new value is propagated to all other caches that contain the same memory location.

The cache coherency protocol is run implicitly by modern hardware, for all but a rare few, esoteric, experimental processors, thus consuming FSB bandwidth whenever multiple processors access the same memory locations. Consequently, a performance penalty for running the cache coherency protocol is incurred every time two or more processors coordinate amongst themselves. This is due to the fact that shared memory is the principal mechanism by which processors communicate. (Inter-processor interrupts, or IPIs, are another mechanism, however they are used much more infrequently and for very specific operating system tasks, like translation lookaside buffer (TLB) shootdowns.)

Most recently, commodity multicore architectures (like Intel Nehalem) have adopted a technique that multiprocessor supercomputing designs of the 1980s and 1990s have used to improve CPU-to-memory throughput. In particular, each physical processor is equipped with separate memory banks and an integrated on-chip memory controller to avoid the performance penalty incurred when multiple processors access the same memory. This technique, depicted in Figure A.4, is called non-uniform memory access (NUMA), and it has the potential to mitigate the memory contention amongst CPUs, provided that each CPU works on data that resides in its nearby physical memory. To access memory that is "remote" to a CPU, NUMA architectures employ additional hardware (and sometimes software as well) to shuttle data between memory banks. Accessing remote memory to a CPU is a slower operation, and may slow down the remote CPU as well, since each CPU is integrated with a corresponding memory controller. Furthermore, virtually all NUMA architectures are cache-coherent (although, notably, Intel's Single-chip Cloud Computer research microprocessor was recently developed to explore a design that forgoes hardware cache coherency [92]), therefore there exists additional hardware support for maintaining memory consistent amongst caches. Typically, a single mechanism is used both to move data between CPUs and to keep their respective caches consistent, and recent commodity implementations (like Intel's QuickPath Interconnect or AMD's HyperTransport) do so by using *point-to-point* (and packet oriented) links between the distinct cache controllers.

APPENDIX B

NETWORK STACK PRIMER

We describe the path of a network packet, from the time it is received by a modern (interrupt driven) NIC until it is delivered to user-space applications by a conventional SMP network stack. We also briefly discuss the conventional mechanisms available for building packet processors in user-space—the raw socket and BSD Packet Filter [164] (BPF) / Linux Socket Filter (LSF). (There are typically two types of raw sockets that can be used to build packet processors, namely the `PF_PACKET` and the `SOCK_RAW`.) The raw socket and the BPF are the underlying mechanisms traditional applications like software routers and packet capture libraries like `tcpdump` / `pcap` [38] are built with.

Due to the *power wall* (see Appendix A for a comprehensive description), major processor vendors have focused on increasing the number of independent CPU cores per silicon chip, instead of increasing the performance of individual processor cores. By contrast, the data rates of network adapters have continued to increase at an exponential rate—for example, 10GbE commodity NICs are now commonplace. This means that a single core handling traffic at line speed from a single high speed interface has few, if any, cycles to spare. Device interrupt coalescence, NIC offload capabilities, like large receive offload (LRO), generic receive offload (GRO), TCP segmentation offload (TSO), and kernel NAPI [25] support prevent the early onset of receive-livelock [169] (too many packets overwhelm the receiver which becomes unable to do useful work). However, they do not solve the underlying problem, namely how to take advantage of the aggregate CPU cycles of all available cores to service network traffic from multiple fast network interfaces. For example, if many CPUs are used to service the same NIC resources, the contention overheads would be prohibitively expensive [103].

Consequently, multi-core scaling has driven vendors to introduce multi-queue NIC

hardware. A NIC with multi-queue capabilities can present itself as a virtual set of M individual NICs (for some maximum arbitrary value of M, as specified by hardware design). The typical multi-queue NIC has the ability to classify the inbound traffic through an opaque hardware "hashing function" to determine the corresponding destination receive (rx) queue for each individual packet (the hashing typically ensures that packets belonging to the same flow, e.g. TCP, are classified into the same queue). Once a destination rx queue is chosen, the NIC transfers the packets via Direct Memory Access (DMA), before issuing message signaled interrupts (MSI/MSI-X) to prompt a receive event solely for the chosen rx queue. If the hardware decides which NIC rx queue to place the received packets onto, the software is responsible for classifying packets to be placed on NIC transmit (tx) queues. The kernel uses a driver-specific hash function for classification if one is provided, or the generic `simple_tx_hash` function otherwise. In the latter case, the kernel makes no assumption about the underlying device capabilities, hence the classification may be suboptimal.

Upon receiving a packet, the NIC issues an interrupt to some CPU—for clarity, we omit batched processing techniques like NAPI [25] or NIC Interrupt Throttling. The interrupted CPU executes the interrupt service routine, also known as the top-half, which performs minor book-keeping (e.g. enqueue received packet, update NIC memory mapped registers), schedules a corresponding bottom-half execution context to run, and terminates. The bottom-half runs exclusively on the CPU that executed the top-half, hence there are as many bottom-halves of same type running concurrently as there are CPUs receiving interrupts.

Next, the in-kernel network stack passes each packet through two lists of protocol handlers, the first list contains handlers for generic packets, while the second list contains handlers for specific packet types, e.g. Internet Protocol (IP) packets. Protocol

handlers register themselves either at kernel startup time or when some particular socket type is created. For example, issuing a `socket(PF_PACKET, socket_type, protocol)` system call registers the socket's default handler either to the specific, or the generic list (if the protocol field is `ETH_P_ALL`). Each of the handlers proceeds to perform additional processing, e.g. TCP or UDP demultiplexing, enqueues the packet for user-space delivery, and wakes up the receiving application before returning. Additionally, the received packets may be filtered based on various criteria by installing BPF filters on the sockets.

APPENDIX C

GLOSSARY OF TERMS

- **Address space:** The range of discrete addresses, each of which may correspond to physical or virtual memory, disk sector, peripheral device, or other logical or physical entity. See also *virtual memory, process, kernel, disk storage, random access memory, peripheral device.*
- **Amdahl's law:** The speedup of a program using multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program. In particular, if a fraction P of the overall computation can be sped up through parallelism by a factor of S, then the overall speedup of the entire computation, including the fraction $(1-P)$ that cannot be sped up (parallelized), is $\frac{1}{(1-P)+\frac{P}{S}}$.
- **Application program:** Computer program that runs in user-space. See also *computer program, user-space.*
- **Application programming interface (API):** Interface implemented by a computer program which enables it to interact with other computer programs. See also *computer program*.
- **Availability:** Fraction of the time a service / system can promptly respond to requests.
- **Berkeley socket:** Endpoint abstraction for inter-process communication, most commonly for communication across computer networks. The berkeley socket application programming interface (API) is the de facto standard for network sockets. See also *inter-process communication, application programming interface, computer networks, network socket, unix domain socket, endpoint.*
- **Blocking:** See *process blocking*.

- **Cache:** A smaller, faster memory which stores copies of the data for the most frequently used main memory locations. See also *processor, main memory, random-access memory.*
- **Cache coherency:** A mechanism by which reads and writes to the same main memory locations are kept consistent throughout all caches of different processors in a symmetric multiprocessing system. See also *cache, MESI protocol, processor, symmetric multiprocessing.*
- **Cache pollution:** The scenario in which a computer program loads data into a CPU cache which causes useful data to be evicted, thus causing a performance penalty when the evicted data needs to be loaded back in the cache. See also *cache, program, process, processor, main memory.*
- **Cache thrashing:** Cache pollution scenario in which main memory is accessed in a pattern that leads to multiple main memory locations competing for the same cache lines, resulting in excessive cache misses. This is problematic in a symmetric multiprocessor system, where different CPUs engage into a potentially expensive cache coherency protocol. See also *cache pollution, cache coherency, main memory, processor.*
- **Commodity:** A mass-produced unspecialized product for which there is demand, but which is supplied without qualitative differentiation across a market. It is a fungible product, meaning the same irrespective of who produces it. See also *commodity computing, commodity computer systems.*
- **Commodity computer systems:** Computer systems manufactured by multiple various vendors, incorporating standardized commodity components. Standardizing commodity components promotes lower costs and less differentiation. See also *commodity, commodity computing.*

- **Commodity computing:** Computing performed on commodity computer systems. See also *commodity, commodity computer systems.*
- **Communication channel:** A physical transmission medium such as a wire, or a logical connection over a multiplexed medium such as a radio channel. See also *transmission medium.*
- **Communication endpoint:** The entity on one end of a communication channel, or transport connection. See also *communication channel, OSI model (transport layer).*
- **Communication protocol:** A formal description of message formats and the rules for exchanging those messages by the parties involved in the communication. Protocols can be defined as the rules governing the syntax, semantics, and timing (synchronization) of communication. See also *communication channel, communication endpoint.*
- **Computer network:** A collection of computers and devices connected by communications channels. See also *communication channel.*
- **Computer program:** A sequence of instructions written to perform a specified task for a computer system.
- **Congestion control:** Controlling the flow of data transmission between two nodes when congestion has occurred along the path between the nodes, either by oversubscribed processing or link capabilities of the intermediate nodes and network segments. See also *data transmission, flow-control, network segment, node, link.*
- **Context switch:** The process by which the state of a CPU is stored and restored, so as the execution of a process can be interrupted and resumed from the same point at a later time. This enables asynchronous event delivery (e.g., signaled

through interrupts) and allows multiple processes to share the same physical CPU. See also *processor, process, time sharing, interrupt.*

- **Central processing unit (CPU):** The component part of the computer system that carries out the instructions of a computer program. See also *computer program, symmetric multiprocessing.*
- **Data:** Opaque sequence of bytes.
- **Data fault tolerance:** Ability to tolerate computer system failure without loss of data. See also *data replica, redundancy, permanent data loss*.
- **Data link:** The means of connecting one location to another for the purpose of transmitting and receiving digital information.
- **Data replica:** An entire copy of a data object. See also *data* and *redundancy*.
- **Data transmission:** The physical transfer of data (a digital bit stream) over a point-to-point or point-to-multipoint communication channel. See also *communication channel, data.*
- **Device driver:** Computer program that interfaces with a hardware device, most commonly it exists as a kernel extension module that runs in kernel-space (i.e. the same address space as the core kernel functionality). See also *kernel, kernel/user-space, virtual memory.*
- **Disk storage:** General storage mechanism, in which data are digitally recorded by various electronic, magnetic, optical, or mechanic methods on a surface layer deposited on one or more planar, round, and rotating platters. See also *data.*
- **Endpoint:** See *communication endpoint.*
- **Error correction and control:** Techniques that enable reliable delivery of data over unreliable communication channels. See also *forward error correction, communication channel.*

- **Fault tolerance:** See *data fault tolerance.*
- **Flow-control:** The process of managing the rate of data transmission between two nodes to prevent a fast sender from overwhelming a slow receiver. See also *node, data transmission, congestion control.*
- **Forward error correction (FEC):** A system of error control for data transmission, whereby the sender adds (carefully selected) redundant data to its messages. Also known as an error-correction code. See also *redundancy, data transmission.*
- **Forwarding:** The act of relaying packets from one network segment to another by nodes in a computer network. See also *network segment, packet switched networks, routing, node*.
- **Goodput:** The application level throughput, excluding protocol overhead and retransmitted data packets. See also *throughput, communication protocol.*
- **Internet Socket:** See *network socket*.
- **Interrupt:** An asynchronous signal indicating the need for attention from the CPU, or a synchronous event in a (software) computer program indicating the need for a change in execution (e.g., an exceptional condition like invalid memory access). See also *processor, trap.*
- **Inter-process communication (IPC)** Data exchange between two or more processes. Processes may be running on one or more computers connected by a network. See also *data, process, socket, unix domain socket, network socket, computer network.*
- **Kernel:** The most basic component of operating systems, providing the lowest abstraction layer for hardware resources, like the process, the address space, and inter-process communication. See also *operating system, kernel/user-space, process, inter-process communication, address space.*

- **Kernel/user-space:** Conventional operating systems segregate virtual memory into kernel-space and user-space. Kernel-space is strictly reserved for running the kernel, kernel extensions, and loadable device drivers, whereas user-space is the (virtual) memory area wherein all user mode applications execute in. The address space segregation is typically enforced by memory management hardware. See also *kernel, virtual memory, memory management hardware, paging.*
- **Kernel-space:** See *kernel/user-space*.
- **Lambda networking:** Technology and set of services directly surrounding the use of multiple optical wavelengths to provide independent communication channels along a strand of fiber optic cable. See *computer network, communication channel, transmission medium, data link.*
- **Link:** See *data link.*
- **Main memory:** Storage that is directly accessible by the processor. The processor directly addresses the locations of the main memory to read instructions and to access data. See also *processor, random-access memory, cache, computer program, virtual memory.*
- **Maximum transmission unit (MTU):** The size in bytes of the largest protocol data unit. See also *communication protocol, packet switched network, Open Systems Interconnect (OSI) model.*
- **Memory:** See *main memory.*
- **Memory management hardware:** Computer hardware component that handles accesses to main memory requested by CPUs. See also *processor, virtual memory, main memory.*
- **MESI (Modified, Exclusive, Shared, Invalid) protocol:** The most common cache coherency protocol that marks each cache line with one of the Modified, Ex-

clusive, Shared, and Invalid states. Each state determines whether the cache line may be directly read or written by the CPU, or if a fresh copy has to be fetched instead. The protocol performs state machine transitions for each cache line based on inputs from the current CPU instruction (e.g., read/load or write/store) and from other CPUs (e.g., if a broadcast *Read For Ownership (RFO)* message is issued by a CPU requesting all other copies of the same cache line to be invalidated). See also *cache coherency, cache, processor, symmetric multiprocessing.*

- **Middlebox:** Proxy network agent designed to perform some sort of packet processing, for example to improve the end-to-end performance of certain communication protocols, such as Transmission Control Protocol (TCP). See also *proxy, performance enhancement proxy, packet processor, router.*
- **Multi-core:** A multi-core processor is an integrated circuit that contains two or more individual processors (called cores), typically integrated onto a single silicon circuit die. See also *processor, symmetric multiprocessing.*
- **Multi-queue NIC:** A network interface controller/card (NIC) that is capable of virtualizing the input and output channels (or queues). Inbound packet traffic is classified in hardware before being placed on the corresponding queue, whereas the device driver controls which queue outbound traffic is placed on. See also *network interface card/controller (NIC), packet switched network, device driver.*
- **Network:** See *computer network.*
- **Network interface card/controller (NIC):** A hardware device that interfaces a computer system with a computer network. See also *computer network, peripheral device.*
- **Network link:** See *data link.*
- **Network node:** A connection point, either a redistribution point or a communication endpoint. See also *computer network, endpoint, router, routing, forwarding.*

- **Network packet:** A finite sequence of bytes the network traffic is split / grouped into, and on which forwarding elements operate on. See also *forwarding* and *packet switched networks.*
- **Network protocol:** See *communication protocol.*
- **Network segment:** A portion of a computer network wherein every device communicates using the same physical layer. See also *OSI model (physical layer).*
- **Network socket:** The endpoint of an inter-process communication channel across a computer network. Network sockets typically implement the berkeley sockets application programming interface (API). See also *socket, inter-process communication, computer network, endpoint, communication channel, application programming interface.*
- **Network throughput:** The average rate of successful data delivery over a communication channel. See also *data, communication channel, data transmission.*
- **Network traffic:** Data in a network. See also *network, data, network packet.*
- **Node:** See *network node.*
- **Open Systems Interconnect (OSI) model:** A way to subdivide a communications system into smaller parts called layers. Each layer is a collection of conceptually similar functions that provide services to the layer above it, while utilizing services from the layer below it. The OSI model comprises the following layers:

  Layer 1: Physical Layer The physical layer (PHY) consists of the basic hardware transmission technologies of a network. It defines the means of transmitting raw bits rather than logical data packets over a physical link connecting network nodes. The bit stream may be grouped into code words or symbols and converted to a physical signal that is transmitted over a hardware transmission medium. The Ethernet physical layers are representative

physical layers, for example, 1000BASE-T is the standard for gigabit Ethernet over copper wiring. See also *network nodes, transmission medium.*

Layer 2: Data Link Layer The data link layer is the protocol layer which transfers data between adjacent network nodes in a network or between nodes on the same network segment. It provides the functional and procedural means to transfer data between network entities and to detect and possibly correct errors that may occur in the physical layer. Ethernet is an example of data link protocol.

Layer 3: Network Layer The network layer is responsible for routing packets delivery including routing through intermediate routers. It provides the functional and procedural means of transferring variable length data sequences from a source to a destination host via one or more networks while maintaining the quality of service requested by the layer on top. The Internet Protocol (IPv4) is the most prominent example of a network layer.

Layer 4: Transport Layer The transport layer provides transparent transfer of data between endpoints, providing reliable data transfer services to the upper layers. It controls the reliability of a given link through flow-control, segmentation/desegmentation, and error control. See also *link, data transmission, endpoint, flow-control, packet, packet segmentation, error correction and control.*

- **Operating system (OS):** The set of system software programs that multiplex and provide access to the computer hardware. The OS also provides the abstractions and interfaces for users and user applications to control the computer, acting as an intermediary between application programs and the computer hardware. See also *kernel, application programs, user-space, kernel-space.*
- **Packet:** See *network packet.*

- **Packet processor:** A forwarding element, like a router, that performs additional per-packet processing. See also *forwarding, node, router, packet, packet switched networks.*
- **Packet segmentation:** The act of splitting a stream of data into packets. See also *packet switched networks, packet.*
- **Packet switched networks:** Computer networks in which the nodes perform packet switching. See also *computer network, packet switching, node.*
- **Packet switching:** Digital networking communications method that groups all transmitted data—regardless of content, type, or structure—into suitably-sized blocks, called packets. See also *forwarding, packet*.
- **Paging:** Memory management technique that divides the virtual address space of a process into pages—i.e., blocks of contiguous virtual memory addresses. See also *main memory, virtual memory.*
- **Peripheral device:** Device attached to a host computer, expanding the host's capabilities, but not part of the core computer architecture. See also *device driver.*
- **Permanent data loss:** The scenario in which data can no longer be retrieved or reconstructed from the information within the system. See also *data, fault tolerance, forward error correction.*
- **Performance enhancement proxy (PEP):** Network agent designed to improve the end-to-end performance of certain communication protocols, such as Transmission Control Protocol (TCP). See also *proxy, middlebox, router.*
- **Process:** The operating system abstraction representing an instance of a computer program that is being executed. Whereas a computer program is a passive collection of instructions, a process is the actual execution of those instructions. See also *computer program, kernel, operating system.*

- **Process blocking:** A process that is waiting for some event, such as a signal (e.g., due to synchronization) or an I/O operation to complete. See also *process, process waiting, context switch.*.
- **Process synchronization:** A mechanism that ensures that two or more concurrently (or time-shared) executing processes do not execute specific portions (or critical sections) of a program at the same time. If one process has begun to execute a critical section of a program, any other processes trying to execute the same section must wait until the first process finishes. See also *process, time sharing, symmetric multiprocessing, context switch.*
- **Process waiting:** A process that is loaded into main memory or is swapped on secondary storage and is awaiting execution on a CPU (awaits to be context switched onto a CPU). See also *process, context switch, main memory, swapping.*
- **Processor:** See *central processing unit (CPU).*
- **Program:** See *computer program.*
- **Protection domain:** See *virtual memory, user-space.*
- **Proxy:** A proxy is an network node entity that acts as an intermediary between parties that communicate over a channel. For example, a proxy may break an end-to-end connection into multiple connections to use different parameters to transfer data across the different legs. See also *performance enhancement proxy, middlebox, communication channel, network node.*
- **Random-access memory (RAM):** Computer data storage realized to date by integrated circuits, that allows data to be accessed in any order (i.e., at random) with virtually the same access time. See also *main memory.*
- **RAW socket:** A socket abstraction that allows direct sending and receiving of network packets in bulk by applications, sidestepping protocol encapsulation, if

any. RAW sockets implement the berkeley socket application programming interface. See also *socket, network packet, application programming interface.*

- **Read for ownership (RFO):** See *MESI protocol.*
- **Redundancy:** Duplication of data in order to reduce the risk of permanent data loss. See also *replication, permanent data loss.*
- **Replication:** Duplication of data in order to reduce the risk of permanent loss, by creating entire, identical, copies of the original data. See also *data, redundancy, permanent data loss, data replica.*
- **Router:** A device that interconnects two or more packet-switched computer networks, and selectively forwards packets of data between them. Each packet contains address information that a router can use to determine if the source and destination are on the same network, or if the data packet must be transferred from one network to another. When multiple routers are used in a large collection of interconnected networks, the routers exchange information about target system addresses, so that each router can build up a (routing) table showing the preferred paths between any two systems on the interconnected networks. See also *packet, computer network, routing, packet switching, forwarding, routing table.*
- **Routing:** The process of selecting paths in a network along which to send network traffic. See also *packet switching.*
- **Routing table:** A data structure stored in a router that lists the routes to particular network destinations. See also *router, routing, packet, packet switching, forwarding.*
- **Socket:** See *berkeley socket*.
- **Swapping:** Memory management scheme by which a computer stores and retrieves data from secondary storage (e.g., a disk) for use in main memory. See

also *main memory, disk storage, virtual memory, paging.*

- **Symmetric multiprocessing (SMP):** A multiprocessor computer hardware architecture where two or more identical processors are connected to the same resources (like memory, disk, network interfaces) and are usually under the control of a single operating system instance. See also *multi-core, operating system.*
- **System call:** The interface between the operating system and a user-space process, by which the process requests a service from the operating system's kernel—the process does not have the permission to perform the service by itself. See also *application programming interface, operating system, process, kernel, trap.*
- **Throughput:** See *network throughput*.
- **Time sharing:** A method by which multiple processes share common resources, such as a CPU, by multiplexing them in time. See also *context switch, process, CPU.*
- **Translation lookaside buffer (TLB):** A CPU cache that the memory management hardware uses to improve the virtual address translation speed. See also *processor, memory management hardware, cache, virtual memory, paging, main memory.*
- **Transmission medium:** A material substance which can propagate energy waves.
- **Trap:** A type of synchronous interrupt typically caused by an exception condition (e.g. invalid memory access or division by zero) in a process. See also *interrupt, processor, process.*
- **Userland:** All application software, including libraries, that runs in user-space. See also *user-space, kernel/user-space.*

- **Unix domain socket:** Data inter-process communication endpoint between processes resident on the same (i.e., on the local) node. See also *inter-process communication, socket, process, node.*
- **User-space:** See *kernel/user-space*.
- **Virtual memory:** A memory management technique that virtualizes a computer system's hardware memory devices (RAM and disk storage). The technique allows programs to treat the memory as a single, contiguous, large address space that is private from any other programs (but is not private from the kernel). See also *program, address space, paging, swapping, random access memory, main memory, disk storage.*

BIBLIOGRAPHY

[1] Amazon Elastic Compute Cloud (EC2). http://aws.amazon.com/ec2/.

[2] Amazon Simple Storage Service (Amazon S3). http://aws.amazon.com/s3/.

[3] Amazon Virtual Private Cloud (Amazon VPC). http://aws.amazon.com/vpc/.

[4] Amazon Web Services. http://aws.amazon.com/.

[5] AppScale. http://appscale.cs.ucsb.edu/.

[6] Beowulf Clusters. http://www.beowulf.org/.

[7] Cisco Carrier Routing System (CRS-1). http://www.cisco.com/en/US/products/ps5763/.

[8] Cisco Catalyst 6500 Series. http://www.cisco.com/en/US/products/hw/switches/ps708/.

[9] Citrix, application delivery infrastructure. http://www.citrix.com/.

[10] DAG Network Monitoring Cards. http://www.endace.com/dag-network-monitoring-cards.html.

[11] Eucalyptus. http://open.eucalyptus.com/.

[12] F5 WAN Delivery Products. http://www.f5.com/.

[13] FlexiScale. http://flexiscale.com/.

[14] Global Environment for Network Innovations (GENI). http://www.geni.net/.

[15] GoGrid. http://gogrid.com/.

[16] Google App Engine. http://code.google.com/appengine/.

[17] Google News. http://news.google.com/.

[18] GoogleDocs. http://docs.google.com/.

[19] Heroku. http://heroku.com/.

[20] Internet2. http://www.internet2.edu/.

[21] Irqbalance. http://www.irqbalance.org/.

[22] Ixia. http://www.ixiacom.com/.

[23] Juniper Networks: Open IP Service Creation Program (OSCP). http://www-jnet.juniper.net/us/en/company/partners/open-ip/oscp/.

[24] libipq/libnetfilter_queue. http://www.netfilter.org/projects/libnetfilter_queue/.

[25] NAPI. http://www.linuxfoundation.org/.

[26] National LambdaRail. http://www.nlr.net/.

[27] Netequalizer Bandwidth Shaper. http://www.netequalizer.com/.

[28] Netperf. http://netperf.org/.

[29] NLR PacketNet Atlas. http://atlas.grnoc.iu.edu/atlas.cgi?map_name=NLR%20Layer3.

[30] Office Web Apps. http://office.microsoft.com/web-apps/.

[31] Packeteer WAN optimization solutions. http://www.packeteer.com/.

[32] PacketLogic Hardware Platforms. http://www.proceranetworks.com/.

[33] PF_RING. http://www.ntop.org/PF_RING.html.

[34] Rackspace Cloud. http://www.rackspacecloud.com/.

[35] RightScale. http://rightscale.com/.

[36] Riverbed Networks: WAN Optimization. http://www.riverbed.com/results/solutions/optimize/.

[37] SONET. http://www.sonet.com/.

[38] tcpdump/libpcap. http://www.tcpdump.org/.

[39] Teragrid. http://teragrid.org/.

[40] The Sun Modular Data Center. http://www.sun.com/products/sunmd/s20/.

[41] Think big with a gig: Our experimental fiber network. http://googleblog.blogspot.com/2010/02/think-big-with-gig-our-experimental.html.

[42] TOP500 Supercomputing Sites. http://www.top500.org/.

[43] Vyatta series 2500. http://vyatta.com/downloads/datasheets/vyatta_2500_datasheet.pdf.

[44] Windows Azure Platform. http://www.microsoft.com/windowsazure/.

[45] RFC 3602, The AES-CBC Cipher Algorithm and Its Use with IPsec, 2003.

[46] TeraGrid Performance Monitoring. https://network.teragrid.org/tgperf/, 2005.

[47] Cisco opening up IOS. http://www.networkworld.com/news/2007/121207-cisco-ios.html, 2007.

[48] Internet Archive. http://www.archive.org/, 2009.

[49] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. *SIGCOMM Comput. Commun. Rev.*, 38(4):63–74, 2008.

[50] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitu Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. DCTCP: Efficient Packet Transport for the Commoditized Data Center. In *SIGCOMM*, 2010.

[51] W. Allcock, J. Bester, J. Bresnahan, A. Chervenak, L. Liming, and S. Tuecke. GridFTP: Protocol extensions to FTP for the Grid. *GGF Document Series GFD*, 20, 2003.

[52] AMD. AMD I/O Virtualization Technology Specification, 2007.

[53] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485, New York, NY, USA, 1967. ACM.

[54] Ashok Anand, Archit Gupta, Aditya Akella, Srinivasan Seshan, and Scott Shenker. Packet caches on routers: the implications of universal redundant traffic elimination. *SIGCOMM Comput. Commun. Rev.*, 38(4):219–230, 2008.

[55] Ashok Anand, Vyas Sekar, and Aditya Akella. SmartRE: an architecture for coordinated network-wide redundancy elimination. In *SIGCOMM '09: Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, pages 87–98, New York, NY, USA, 2009. ACM.

[56] Darrell C. Anderson, Jeffrey S. Chase, Syam Gadde, Andrew J. Gallatin, Kenneth G. Yocum, and Michael J. Feeley. Cheating the I/O bottleneck: network storage with Trapeze/Myrinet. In *Proceedings of the USENIX Annual Technical Conference*, 1998.

[57] Nate Anderson. Deep packet inspection meets Net neutrality, CALEA. *Ars Technica*, July 2007.

[58] Thomas E. Anderson, David E. Culler, David A. Patterson, and the NOW team. A case for now (networks of workstations). *IEEE Micro*, 15(1):54–64, 1995.

[59] John Backus. Can Programming be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. *ACM Turing award lectures*, page 1977, 2007.

[60] Andrea Baiocchi, Angelo P. Castellani, and Francesco Vacirca. YeAH-TCP: Yet Another Highspeed TCP. In *PFLDnet 2007: Fifth International Workshop on Protocols for Fast Long-Distance Networks*, 2007.

[61] Pavan Balaji, Piyush Shivam, and Pete Wyckoff. High Performance User Level Sockets over Gigabit Ethernet. In *Cluster Comp.*, 2002.

[62] Mahesh Balakrishnan, Tudor Marian, Ken Birman, Hakim Weatherspoon, and Lakshmi Ganesh. Maelstrom: Transparent Error Correction for Communication between Data Centers. *To appear in IEEE/ACM Transactions on Networking (ToN)*, 2010.

[63] Mahesh Balakrishnan, Tudor Marian, Ken Birman, Hakim Weatherspoon, and Einar Vollset. Maelstrom: Transparent error correction for lambda networks. In *Proceedings of NSDI*, 2008.

[64] Roger Barga. Cloud Computing (Invited Keynote Talk). In *IEEE Ninth International Conference on Peer-to-Peer Computing (P2P'09)*, 2009.

[65] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *SOSP*, 2003.

[66] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 29–44, New York, NY, USA, 2009. ACM.

[67] 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 *Proceedings of SOSP*, 1995.

[68] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight RPC. *ACM TOCS*, 8(1):37–55, 1990.

[69] Stephen Blott, José Brustoloni, and Cliff Martin. NetTap: An Efficient and Reliable PC-Based Platform for Network Programming. In *Proceedings of OPENARCH*, 2000.

[70] Raffaele Bolla and Roberto Bruschi. Pc-based software routers: high performance and application service support. In *PRESTO '08: Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow*, 2008.

[71] Jean-Chrysotome Bolot. End-to-end packet delay and loss behavior in the internet. In *SIGCOMM '93: Conference proceedings on Communications architectures, protocols and applications*, pages 289–298, New York, NY, USA, 1993. ACM.

[72] J. Border, M. Kojo, J. Griner, G. Montenegro, and Z. Shelby. Performance En-

hancing Proxies Intended to Mitigate Link-Related Degradations. RFC 3135, Network Working Group, 2001.

[73] Herbert Bos, Willem de Bruijn, Mihai Cristea, Trung Nguyen, and Georgios Portokalidis. FFPF: Fairly fast packet filters. *In Proceedings of OSDI*, 2004.

[74] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. Corey: an operating system for many cores. In *Proceedings of OSDI*, 2008.

[75] Lawrence S. Brakmo and Larry L. Peterson. TCP Vegas: end to end congestion avoidance on a global Internet. *IEEE Journal on selected Areas in communications*, 13:1465–1480, 1995.

[76] Stefano Bregni, Senior Member, Davide Caratti, and Fabio Martignon. Enhanced Loss Differentiation Algorithms for Use in TCP Sources over Heterogeneous Wireless Networks. In *Proceedings of IEEE GLOBECOM 2003*, pages 666–670, 2003.

[77] Maxine D. Brown. Introduction: Blueprint for the future of high-performance networking. *Commun. ACM*, 46(11):30–33, 2003.

[78] José Carlos Brustoloni and Peter Steenkiste. Effects of buffering semantics on I/O performance. In *Proceedings OSDI*, 1996.

[79] José Carlos Brustoloni, Peter Steenkiste, and Carlos Brustoloni. User-Level Protocol Servers with Kernel-Level Performance. In *Proceedings of the IEEE Infocom Conference*, pages 463–471, 1998.

[80] Hadrien Bullot, R. Les Cottrell, and Richard Hughes-Jones. Evaluation of advanced TCP stacks on fast long-distance production networks. In *Proceedings of the International Workshop on Protocols for Fast Long-Distance Networks*, 2004.

[81] Carlo Caini and Rosario Firrincieli. TCP Hybla: a TCP enhancement for heterogeneous networks. *International Journal of Satellite Communications and Networking*, 22, 2004.

[82] Robert L. Carter and Mark E. Crovella. Measuring bottleneck link speed in packet-switched networks. *Perform. Eval.*, 27-28:297–318, 1996.

[83] Claudio Casetti, Mario Gerla, Saverio Mascolo, M. Y. Sanadidi, and Ren Wang.

TCP Westwood: end-to-end congestion control for wired/wireless networks. *Wirel. Netw.*, 8(5):467–479, 2002.

[84] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, 2002.

[85] Rajiv Chakravorty, Sachin Katti, Jon Crowcroft, and Ian Pratt. Flow Aggregation for Enhanced TCP over Wide-Area Wireless. In *Proc. IEEE INFOCOM*, pages 1754–1764, 2003.

[86] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[87] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):1–26, 2008.

[88] Yanpei Chen, Rean Griffith, Junda Liu, Randy H. Katz, and Anthony D. Joseph. Understanding TCP incast throughput collapse in datacenter networks. In *WREN '09: Proceedings of the 1st ACM workshop on Research on enterprise networking*, pages 73–82, New York, NY, USA, 2009. ACM.

[89] Israel Cidon, Asad Khamisy, and Moshe Sidi. Analysis of Packet Loss Processes in High-Speed Networks. *IEEE Transactions on Information Theory*, 39:98–108, 1991.

[90] Cisco Systems. Buffers, Queues, and Thresholds on the Catalyst 6500 Ethernet Modules, 2007.

[91] Kimberly Claffy, George C. Polyzos, and Hans-Werner Braun. Traffic Characteristics of the T1 NSFNET Backbone. In *Proceedings of INFOCOM*, 1993.

[92] Intel Corp. Single-chip Cloud Computer. `http://techresearch.intel.com/articles/Tera-Scale/1826.htm`, 2010.

[93] Patrick Crowley, Marc E. Fluczynski, Jean-Loup Baer, and Brian N. Bershad. Characterizing processor architectures for programmable network interfaces. In *Proceedings of the 14th international conference on Supercomputing*, 2000.

[94] Willem de Bruijn and Herbert Bos. Beltway Buffers: Avoiding the OS Traffic Jam. In *Proceedings of Infocom 2008*, April 2008.

[95] Willem de Bruijn and Herbert Bos. Model-T: Rethinking the OS for terabit speeds. In *Proceedings of the HSN2008, co-located with Infocom 2008*, April 2008.

[96] Willem de Bruijn and Herbert Bos. PipesFS: fast Linux I/O in the UNIX tradition. *SIGOPS Oper. Syst. Rev.*, 42(5):55–63, 2008.

[97] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, 2007.

[98] Tom DeFanti, Cees de Laat, Joe Mambretti, Kees Neggers, and Bill St. Arnaud. TransLight: a global-scale LambdaGrid for e-science. *Commun. ACM*, 46(11):34–41, 2003.

[99] Luca Deri. Improving passive packet capture: beyond device polling. In *SANE*, 2004.

[100] Luca Deri. nCap: wire-speed packet capture and transmission. In *E2EMON Workshop*, 2005.

[101] DETER Network Security Testbed. `http://www.isi.deterlab.net/`.

[102] Prajjwal Devakota and A. L. Narasimha Redd. Performance of Quantized Congestion Notification in TCP Incast scenarios in data centers. In *IEEE MASCOTS*, 2010.

[103] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proceedings of SOSP*, 2009.

[104] Constantinos Dovrolis, Parameswaran Ramanathan, and David Moore. What Do Packet Dispersion Techniques Measure? In *Proceedings of INFOCOM*, 2001.

[105] Constantinos Dovrolis, Parameswaran Ramanathan, and David Moore. Packet-dispersion techniques and a capacity-estimation methodology. *IEEE/ACM Trans. Netw.*, 12(6):963–977, 2004.

[106] Peter Druschel and Larry L. Peterson. Fbufs: a high-bandwidth cross-domain transfer facility. *SIGOPS Operating Systems Review*, 1993.

[107] Ita Dukkipati, Masayoshi Kobayashi, Rui Zhang-shen, and Nick Mckeown. Processor sharing flows in the internet. In *Thirteenth International Workshop on Quality of Service (IWQoS '05)*, 2005.

[108] Dave Dunning, Greg Regnier, Gary McAlpine, Don Cameron, Bill Shubert, Frank Berry, Anne Marie Merritt, Ed Gronke, and Chris Dodd. The Virtual Interface Architecture. *Micro*, 1998.

[109] Emulab - Network Emulation Testbed. `http://www.emulab.net/`.

[110] DR Engler, MF Kaashoek, and J. O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of SOSP*, 1995.

[111] William H. Whitted et al. Modular data center, October 2007.

[112] Tae eun Kim, Songwu Lu, and Vaduvur Bharghavan. Improving congestion control performance through loss differentiation. In *ICCCN '99: The 8th International Conference on Computer Communications and Networks*, 1999.

[113] Aaron Falk, Ted Faber, Joseph Bannister, Andrew Chien, Robert Grossman, and Jason Leigh. Transport protocols for high performance. *Commun. ACM*, 46(11):42–49, 2003.

[114] S. Floyd. HighSpeed TCP for Large Congestion Windows. RFC 3649, Network Working Group, December, 2003.

[115] Daniel A. Freedman, Tudor Marian, Jennifer H. Lee, Ken Birman, Hakim Weatherspoon, and Chris Xu. Exact temporal characterization of 10 Gbps optical wide-area network using high precision instrumentation. In *Proceedings of the 10th Internet Measurement Conference (IMC' 10)*, November 2010.

[116] Masanobu Yuhara Fujitsu, Masanobu Yuhara, Brian N. Bershad, Chris Maeda, J. Eliot, and B. Moss. Efficient packet demultiplexing for multiple endpoints and large messages. In *Proceedings of the 1994 Winter USENIX Conference*, 1994.

[117] Patrick Geoffray. A Critique of RDMA. `http://www.hpcwire.com/features/17886984.html`, 2006.

[118] Patrick Geoffray, Loïc Prylli, and Bernard Tourancheau. BIP-SMP: high performance message passing over a cluster of commodity SMPs. In *Proceedings of the ACM/IEEE conference on Supercomputing*, 1999.

[119] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.

[120] George Gilder. The Information Factories. Wired Magazine, Issue 14.10, October 2005.

[121] Corey Gough, Suresh Siddha, and Ken Chen. Kernel Scalability – Expanding the Horizon Beyond Fine Grain Locks. In *Proceedings of the Linux Symposium*, pages 153–165, June 2007.

[122] Albert Greenberg, James Hamilton, David A. Maltz, and Parveen Patel. The cost of a cloud: research problems in data center networks. *SIGCOMM Comput. Commun. Rev.*, 39(1):68–73, 2009.

[123] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: a scalable and flexible data center network. In *SIGCOMM '09: Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, pages 51–62, New York, NY, USA, 2009. ACM.

[124] Albert Greenberg, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. Towards a next generation data center architecture: scalability and commoditization. In *PRESTO '08: Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow*, pages 57–62, New York, NY, USA, 2008. ACM.

[125] Yunhong Gu and Robert L. Grossman. Sabul: A transport protocol for grid computing. *J. Grid Comput.*, 1(4):377–386, 2003.

[126] Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, Yongguang Zhang, and Songwu Lu. BCube: a high performance, server-centric network architecture for modular data centers. In *SIGCOMM '09: Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, pages 63–74, New York, NY, USA, 2009. ACM.

[127] Chuanxiong Guo, Haitao Wu, Kun Tan, Lei Shi, Yongguang Zhang, and Songwu Lu. Dcell: a scalable and fault-tolerant network structure for data centers. In *SIGCOMM '08: Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, pages 75–86, New York, NY, USA, 2008. ACM.

[128] Sangtae Ha, Injong Rhee, and Lisong Xu. CUBIC: a new TCP-friendly high-speed TCP variant. *SIGOPS Oper. Syst. Rev.*, 42(5):64–74, 2008.

[129] Thomas J. Hacker, Brian D. Athey, and Brian Noble. The End-to-End Performance Effects of Parallel TCP Sockets on a Lossy Wide-Area Network. In *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*, page 314, Washington, DC, USA, 2002. IEEE Computer Society.

[130] Thomas J. Hacker, Brian D. Noble, and Brian D. Athey. The effects of systemic packet loss on aggregate TCP flows. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–15, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[131] Andreas Haeberlen, Rodrigo Rodrigues, Krishna Gummadi, and Peter Druschel. Pretty good packet authentication. In *HotDep*, 2008.

[132] James Hamilton. Internet-Scale Service Efficiency (Keynote Talk). In *Large Scale Distributed Systems & Middleware (LADIS)*, 2008.

[133] Eric He, Jason Leigh, Oliver Yu, and Thomas A. DeFanti. Reliable Blast UDP: Predictable High Performance Bulk Data Transfer. In *CLUSTER '02: Proceedings of the IEEE International Conference on Cluster Computing*, page 317, Washington, DC, USA, 2002. IEEE Computer Society.

[134] Steven A. Heimlich. Traffic characterization of the NSFNET national backbone. *SIGMETRICS Perform. Eval. Rev.*, 18(1):257–258, 1990.

[135] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[136] Mark D. Hill and Michael R. Marty. Amdahl's law in the multicore era. *IEEE COMPUTER*, 2008.

[137] James W. Hunt and M. Douglas McIlroy. An Algorithm for Differential File Comparison. Computer Science Technical Report 41, Bell Laboratories, 1976.

[138] V. Jacobson. Congestion avoidance and control. *SIGCOMM Comput. Commun. Rev.*, 25(1):157–187, 1995.

[139] Manish Jain and Constantinos Dovrolis. End-to-end available bandwidth: measurement methodology, dynamics, and relation with TCP throughput. *IEEE/ACM Tr. Net.*, 11(4):537–549, 2003.

[140] Ardalan Kangarlou, Sahan Gamage, Ramana Rao Kompella, and Dongyan Xu.

vSnoop: Improving TCP Throughput in Virtualized Environments via Acknowledgement Offload. In *IEEE SC*, 2010.

[141] Rohit Kapoor, Ling-Jyh Chen, Li Lao, Mario Gerla, and M. Y. Sanadidi. CapProbe: a simple and accurate capacity estimation technique. *SIGCOMM Comp. Comm. Rev.*, 34(4):67–78, 2004.

[142] Dina Katabi, Mark Handley, and Charlie Rohrs. Congestion control for high bandwidth-delay product networks. In *SIGCOMM '02: Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 89–102, New York, NY, USA, 2002. ACM.

[143] Tom Kelly. Scalable TCP: Improving Performance in Highspeed Wide Area Networks. *ACM SIGCOMM Computer Communication Review*, 33:83–91, 2002.

[144] Carl Kesselman and Ian Foster. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, November 1998.

[145] Changhoon Kim, Matthew Caesar, and Jennifer Rexford. Floodless in SEATTLE: A Scalable Ethernet Architecture for Large Enterprises. In *SIGCOMM '08: Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, pages 3–14, New York, NY, USA, 2008. ACM.

[146] E. Kohler, M. Handley, and S. Floyd. Datagram Congestion Control Protocol (DCCP). RFC 4340, IETF, March, 2000.

[147] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click modular router. *TOCS*, 2000.

[148] D. Kostić, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: high bandwidth data dissemination using an overlay mesh. *ACM SIGOPS Operating Systems Review*, 37(5), 2003.

[149] Orran Krieger, Marc Auslander, Bryan Rosenburg, Robert W. Wisniewski, Jimi Xenidis, Dilma Da Silva, Michal Ostrowski, Jonathan Appavoo, Maria Butrico, Mark Mergen, Amos Waterland, and Volkmar Uhlig. K42: building a complete operating system. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2006.

[150] T. V. Lakshman and Upamanyu Madhow. The performance of TCP/IP for networks with high bandwidth-delay products and random loss. *IEEE/ACM Trans. Netw.*, 5(3):336–350, 1997.

[151] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[152] Douglas Leith and Robert Shorten. H-TCP: TCP for high-speed and long-distance networks. In *Second Workshop on Protocols for FAST Long-Distance Networks*, 2004.

[153] Ben Leslie, Peter Chubb, Nicholas Fitzroy-dale, Stefan Gtz, Charles Gray, Luke Macpherson, Daniel Potts, Yueting Shen, and Gernot Heiser. User-level device drivers: Achieved performance. *Journal of Computer Science and Technology*, 20, 2005.

[154] David A. Lifka. Director, Center for Advanced Computing, Cornell University. *Private Communication*, 2008.

[155] Rose Liu, Kevin Klues, Sarah Bird, Steven Hofmeyr, Krste Asanović, and John Kubiatowicz. Tessellation: Space-Time Partitioning in a Manycore Client OS. In *HotPar*, 2009.

[156] Shao Liu, Tamer Başar, and R. Srikant. TCP-Illinois: A loss- and delay-based congestion control algorithm for high-speed networks. *Perform. Eval.*, 65(6-7):417–440, 2008.

[157] John W. Lockwood, Nick McKeown, Greg Watson, Glen Gibb, Paul Hartke, Jad Naous, Ramanan Raghuraman, and Jianying Luo. NetFPGA–An Open Platform for Gigabit-Rate Network Switching and Routing. In *Proceedings of the IEEE International Conference on Microelectronic Systems Education*, 2007.

[158] Lucian Popa and Ion Stoica and Sylvia Ratnasamy. Rule-based Forwarding (RBF): improving the Internet's flexibility and security. In *Proc. of workshop on Hot Topics in Networks (HotNets-VIII)*, 2009.

[159] Yadi Ma, Suman Banerjee, Shan Lu, and Cristian Estan. Leveraging Parallelism for Multi-dimensional Packet Classification on Software Routers. In *Proceedings of ACM SIGMETRICS*, 2010.

[160] Udi Manber. Finding similar files in a large file system. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 1–10, 1994.

[161] Tudor Marian, Mahesh Balakrishnan, Ken Birman, and Robbert van Renesse. Tempest: Soft State Replication in the Service Tier. In *Proceedings of the 38th*

Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN-DCCS '08), June 2008.

[162] Tudor Marian, Ken Birman, and Robbert van Renesse. A Scalable Services Architecture. In *Proceedings of the 25th IEEE Symposium on Reliable Distributed Systems (SRDS 2006)*. IEEE Computer Society, 2006.

[163] Tudor Marian, Daniel Freedman, Ken Birman, and Hakim Weatherspoon. Empirical Characterization of Uncongested Lambda Networks and 10GbE Commodity Endpoints. In *Proceedings of the 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN-PDS '10)*, June 2010.

[164] Steven McCanne and Van Jacobson. The BSD packet filter: a new architecture for user-level packet capture. In *Proceedings of USENIX*, 1993.

[165] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, 2008.

[166] Mark R. Meiss. Tsunami: A High-Speed Rate-Controlled Protocol for File Transfer. `http://steinbeck.ucs.indiana.edu/~mmeiss/papers/tsunami.pdf`.

[167] M. Meyer, J. Sachs, and M. Holzke. Performance evaluation of a TCP proxy in WCDMA networks. *Wireless Communications, IEEE*, 10(5):70 – 79, oct 2003.

[168] Andrew G. Miklas, Stefan Saroiu, Alec Wolman, and Angela Demke Brown. Bunker: A Privacy-Oriented Platform for Network Tracing. In *Proceedings of NSDI*, 2009.

[169] Jeffrey C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Trans. Comput. Syst.*, 15(3):217–252, 1997.

[170] Biswanath Mukherjee. *Optical WDM Networks*. Springer, February 2006.

[171] L. Munoz, M. Garcia, J. Choque, R. Aguero, and P. Mahonen. Optimizing Internet flows over IEEE 802.11b wireless local area networks: a performance-enhancing proxy based on forward error correction. *Communications Magazine, IEEE*, 39(12):60 –67, dec 2001.

[172] M. Murray, S. Smallen, O. Khalili, and M. Swany. Comparison of End-to-End

Bandwidth Measurement Tools on the 10GigE TeraGrid Backbone. In *Proceedings of GRID*, 2005.

[173] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 174–187, New York, NY, USA, 2001. ACM.

[174] Erich M. Nahum, David Yates, James Kurose, and Don Towsley. Performance issues in parallelized network protocols. *In Proceedings of OSDI*, 2004.

[175] Harvey B. Newman, Mark H. Ellisman, and John A. Orcutt. Data-intensive e-science frontier research. *Commun. ACM*, 46(11):68–77, 2003.

[176] Edmund B. Nightingale, Orion Hodson, Ross McIlroy, Chris Hawblitzel, and Galen Hunt. Helios: heterogeneous multiprocessing with satellite kernels. In *Proceedings of the 22nd symposium on Operating systems principles*, 2009.

[177] Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat. PortLand: a scalable fault-tolerant layer 2 data center network fabric. In *SIGCOMM '09: Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, pages 39–50, New York, NY, USA, 2009. ACM.

[178] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. In *Proceedings of ASPLOS-VII*, 1996.

[179] OProfile. `http://oprofile.sourceforge.net/`, 2008.

[180] Jitendra Padhye, Victor Firoiu, Don Towsley, and Jim Kurose. Modeling TCP throughput: a simple model and its empirical validation. *SIGCOMM Comp. Comm. Rev.*, 28(4):303–314, 1998.

[181] V.S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: a unified I/O buffering and caching system. *ACM TOCS*, 2000.

[182] Scott Pakin, Mario Lauria, and Andrew Chien. High performance messaging on workstations: Illinois fast messages (FM) for Myrinet. In *Proceedings of the ACM/IEEE conference on Supercomputing*, 1995.

[183] Christina Parsa and J.J. Garcia-Luna-Aceves. Differentiating congestion vs. random loss: A method for improving tcp performance over wireless links. In *IEEE WCNC2000*, pages 90–93, 2000.

[184] JoAnn M. Paul and Brett H. Meyer. Amdahl's law revisited for single chip systems. *International Journal of Parallel Programming*, 35(2):101–123, 2007.

[185] Amar Phanishayee, Elie Krevat, Vijay Vasudevan, David G. Andersen, Gregory R. Ganger, Garth A. Gibson, and Srinivasan Seshan. Measurement and Analysis of TCP Throughput Collapse in Cluster-based Storage Systems. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, San Jose, CA, February 2008.

[186] R. S. Prasad, M. Murray, C. Dovrolis, and K. Claffy. Bandwidth Estimation: Metrics, Measurement Techniques, and Tools. *IEEE Network*, 17:27–35, 2003.

[187] Ravi Prasad, Manish Jain, and Constantinos Dovrolis. Effects of Interrupt Coalescence on Network Measurements. In *Proceedings of the PAM Workshop*, 2004.

[188] PREEMPT_RT. `http://lwn.net/Articles/146861`, 2005.

[189] Michael Oser Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[190] Vinay J. Ribeiro, Rudolf H. Riedi, Richard G. Baraniuk, Jiri Navratil, and Les Cottrell. pathChirp: Efficient Available Bandwidth Estimation for Network Paths. In *Proceedings of the PAM Workshop*, 2003.

[191] RTAI. `https://www.rtai.org/`, 2008.

[192] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Trans. Comput. Syst.*, 2(4), 1984.

[193] D. Sanghi, A. K. Agrawala, O. Gudmundsson, and B. N. Jain. Experimental Assessment of End-to-end Behavior on Internet. In *Proceedings of IEEE INFOCOM*, 1993.

[194] Leah Shalev, Julian Satran, Eran Borovik, and Muli Ben-Yehuda. IsoStack–Highly Efficient Network Processing on Dedicated Cores. In *USENIX ATC '10: USENIX Annual Technical Conference*, 2010.

[195] Stephen C. Simms, Gregory G. Pike, and Doug Balog. Wide Area Filesystem Performance using Lustre on the TeraGrid. In *Teragrid Conference*, 2007.

[196] Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage. Automated worm fingerprinting. In *In OSDI*, pages 45–60, 2004.

[197] H. Sivakumar, S. Bailey, and R. L. Grossman. PSockets: the case for application-level network striping for data intensive applications using high speed wide area networks. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 37, Washington, DC, USA, 2000. IEEE Computer Society.

[198] SNORT. `http://www.snort.org/`, 2008.

[199] Neil T. Spring and David Wetherall. A protocol-independent technique for eliminating redundant network traffic. In *Proceedings of ACM SIGCOMM*, pages 87–95, 2000.

[200] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. Stream Control Transmission Protocol (SCTP). RFC 2960, IETF, 2000.

[201] Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005.

[202] Kun Tan and Jingmin Song. A compound TCP approach for high-speed and long distance networks. In *Proc. IEEE INFOCOM*, 2006.

[203] David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden. A Survey of Active Network Research. *IEEE Communications Magazine*, 35:80–86, 1997.

[204] A. Tirumala, F. Qin, J. Dugan, J. Ferguson, and K. Gibbs. Iperf–The TCP/UDP bandwidth measurement tool, 2004.

[205] Ronald C. Unrau, Orran Krieger, Benjamin Gamsa, and Michael Stumm. Experiences with locking in a NUMA multiprocessor operating system kernel. In *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, 1994.

[206] Vijay Vasudevan, Amar Phanishayee, Hiral Shah, Elie Krevat, David G. Andersen, Gregory R. Ganger, Garth A. Gibson, and Brian Mueller. Safe and effec-

tive fine-grained TCP retransmissions for datacenter communication. *SIGCOMM Comput. Commun. Rev.*, 39(4):303–314, 2009.

[207] Dimitris Velenis, Dimitris Kalogeras, and Basil S. Maglaris. SaTPEP: A TCP Performance Enhancing Proxy for Satellite Links. In *NETWORKING '02: Proceedings of the Second International IFIP-TC6 Networking Conference on Networking Technologies, Services, and Protocols*, pages 1233–1238, London, UK, 2002. Springer-Verlag.

[208] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-Net: a user-level network interface for parallel and distributed computing. *Proceedings SOSP*, 1995.

[209] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th annual international symposium on Computer architecture*, 1992.

[210] Steven Wallace. Lambda Networking. Advanced Network Management Lab, Indiana University.

[211] Steven Wallace. Tsunami File Transfer Protocol. In *PFLDNet 2003: First International Workshop on Protocols for Fast Long-Distance Networks*, 2003.

[212] Guohui Wang and T. S. Eugene Ng. The Impact of Virtualization on Network Performance of Amazon EC2 Data Center. In *IEEE INFOCOM*, 2010.

[213] Andrew Warfield, Steven Hand, Keir Fraser, and Tim Deegan. Facilitating the development of soft devices. In *Proceedings of the USENIX Annual Technical Conference*, 2005.

[214] Edsger W.Dijkstra. A review of the 1977 Turing Award Lecture by John Backus. `http://userweb.cs.utexas.edu/~EWD/transcriptions/EWD06xx/EWD692.html`.

[215] Hakim Weatherspoon, Lakshmi Ganesh, Tudor Marian, Mahesh Balakrishnan, and Ken Birman. Smoke and Mirrors: Shadowing Files at a Geographically Remote Location Without Loss of Performance. In *Proceedings of FAST*, February 2009.

[216] P. Wefel. Network Engineer, National Center For Supercomputing Applications (NCSA), University of Illinois. *Private Communication*, 2007.

[217] David X. Wei, Cheng Jin, Steven H. Low, and Sanjay Hegde. FAST TCP: motivation, architecture, algorithms, performance. *IEEE/ACM Trans. Netw.*, 14(6):1246–1259, 2006.

[218] Matt Welsh, Anindya Basu, and Thorsten Von Eicken. ATM and Fast Ethernet Network Interfaces for User-level Communication. In *3rd Intl. Symp. on High Performance Computer Architecture*, 1997.

[219] R. Clint Whaley and Antoine Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience*, 35(2), 2005.

[220] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *Proceedings of OSDI*, 2002.

[221] Wireshark. `http://www.wireshark.org/`, 2008.

[222] Phil Wood. libpcap-mmap, Los Alamos National Labs, 2008.

[223] Zhenyu Wu, Mengjun Xie, and Haining Wang. Swift: A fast dynamic packet filter. In *Proceedings of NSDI*, 2008.

[224] Bartek Wydrowski, Lachlan L. H. Andrew, and Moshe Zukerman. Maxnet: A congestion control architecture for scalable networks. *IEEE Communications Letters*, 6:512–514, 2002.

[225] Yong Xia, Lakshminarayanan Subramanian, Ion Stoica, and Shivkumar Kalyanaraman. One more bit is enough. *IEEE/ACM Trans. Netw.*, 16(6):1281–1294, 2008.

[226] Lisong Xu, Khaled Harfoush, and Injong Rhee. Binary increase congestion control (BiC) for fast long-distance networks. In *Proceedings of IEEE INFOCOM*, 2004.

[227] Yueping Zhang, Derek Leonard, and Dmitri Loguinov. Jetmax: Scalable max-min congestion control for high-speed heterogeneous networks. *Comput. Netw.*, 52(6):1193–1219, 2008.