BUILDING NETWORKED SYSTEMS FOR TERABIT ETHERNET

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

Qizhe Cai

August 2024

BUILDING NETWORKED SYSTEMS FOR TERABIT ETHERNET

Qizhe Cai, Ph.D.

Cornell University 2024

Over the last two decades, hardware in datacenters has shown diverging trends. On the one hand, access link bandwidth has increased rapidly, with datacenters now commonly supporting Terabit Ethernet (i.e., Ethernet with speeds above 100Gbps). Modern network hardware is capable of supporting microsecond-scale latency and multi-hundred-gigabit bandwidth. However, on the other hand, the slowdown of Moore's Law and the end of Dennard scaling have resulted in total compute capacity (the number of CPU cores multiplied by per-core performance) remaining largely stagnant. As a result, network performance bottlenecks have shifted to the host network stacks, which are responsible for processing network packets to and from applications.

The first contribution of this dissertation is to build an in-depth understanding the core challenges hindering existing host network stacks from fully leveraging modern network hardware. Our study reveals that the rapid increase in network link bandwidth has made data movement overheads (such as transferring data from NICs to CPUs) a bottleneck for scaling single-core performance. Typically, on the receiver side, after the NIC DMAs data to memory, the limited memory bandwidth leads to poor CPU efficiency, as CPUs must stall while waiting for data to be loaded from memory into CPU registers. We find out existing optimization techniques like DDIO, which enables NICs to directly read/write data from/to CPU caches, are unable to improve CPU efficiency. This is because the increase in bandwidth-delay products has outpaced the increase in

cache sizes, leading to high cache miss rates and poor CPU efficiency. With today's network stacks, multiple cores are needed to fully exploit the capabilities of Terabit network hardware. However, our study shows that using multiple cores, while saturating the link bandwidth, leads to even worse CPU efficiency compared to the single-core case. This is because host resources like cache and access link bandwidth are contended among different cores/applications.

The second contribution of this dissertation is to introduce NetChannel —a new network stack architecture that enables host network stacks to leverage network hardware without requiring application modifications. NetChannel disaggregates network stacks into multiple loosely-coupled layers, allowing each layer to scale and schedule across multiple cores independently. Using an end-to-end NetChannel realization within the Linux network stack, we demonstrate that NetChannel enables new operating points—(1) enabling a single application thread to saturate multi-hundred-gigabit access link bandwidth; (2) enabling near-linear scalability for small message processing with an increasing number of cores, independent of the number of application threads; and, (3) enabling isolation of latency-sensitive applications, allowing them to maintain μs-scale tail latency even when competing with throughput-bound applications operating at near-line rate.

This thesis leaves open several interesting directions of future research: 1) improving CPU efficiency by reducing both data movement and CPU processing overheads; 2) extending NetChannel to function in more realistic scenarios, allowing us to fully realize its potential; and 3) extending the study to understand network stack overheads, not only in terms of CPU efficiency and throughput but also network latency, as achieving low latency is also crucial for applications.

BIOGRAPHICAL SKETCH

Qizhe Cai is a Ph.D. candidate in the department of Computer Science at Cornell University, advised by Rachit Agarwal. He holds a Master of Science in Computer Science from Princeton University and a Bachelor of Science in Engineering from University of Michigan.

To my parents, Youmin and Meiyun, and my wife, Cornelia, for always trusting me and giving me unconditional support.

ACKNOWLEDGEMENTS

I cannot imagine reaching this point without the support of my family, advisor, friends, and colleagues. I am extremely fortunate to be surrounded by such kind and supportive people, who have always been there for me throughout this journey. Their unwavering encouragement and assistance have been invaluable, and I am deeply grateful for each of them.

First, I would like to thank my advisor, Rachit Agarwal. I am grateful and fortunate to have Rachit as my advisor. Rachit always tries his best to train me as a good researcher, encouraging me to tackle hard and fundamental problems, to dive deep into understanding these problems, and to always pursue driving the community forward. This experience was invaluable in my growth. Beyond research, I also learned from him about leading and taking care of teams within our group, how to socialize and interact effectively with others, and the importance of fostering a collaborative environment.

I would also like to thank my committee members: Adrian Sampson, David Shmoys, and Christos Kozyrakis, for agreeing to be part of my committee and for providing frank and insightful feedback on my research, helping to shape this dissertation. For Christos, I am also grateful for his valuable perspectives on academia, which greatly helped me prepare for entering the job market.

It would have been difficult for me to start and finish this enjoyable journey without the tremendous help of many senior people: Jennifer Rexford, Kevin Tang, Mina Tahmasbi Arashloo, Jaehyun Hwang, Vishal Shrivastav, and Praveen Kumar. For Jennifer, I am grateful for her constant encouragement and honest feedback whenever I consulted her on various matters. Mina always provided frank and honest feedback, which helped me develop critical thinking. Kevin was a constant source of support, always sending kind and warm

emails to encourage and foster me whenever I encountered difficulties. For Jaehyun, one of my long-term research collaborators, he provided invaluable support and effort when I started my research related to kernel development. Whenever people are impressed by my technical skills, it is largely due to his guidance and support. Vishal taught me how to frame research arguments effectively. I still remember our 4-hour discussion during a trip back from NYC to Ithaca, where he provided sentence-by-sentence feedback. Praveen has been a pleasure to converse with. I particularly remember our discussion about PICNIC, where he patiently provided detailed answers and insights.

I would also like to thank my friends for their support during this journey. One of my best friends, Yang Xia, always provides a fresh perspective on computer science through an interdisciplinary lens, helping me think about technology trends in a more fundamental manner. Luming Tang and Ziyun Wei, who work in different subareas such as Computer Vision and Database, have introduced me to new topics, which is one of the reasons I continued going to the office after COVID. To my undergraduate friends, Yuling Liu, Penghua Zhao, and Jinhao Ping, it is always fun to talk with you all and reminisce about the funny moments we had at Michigan. Go Blue forever!

Next, I would like to thank my colleagues and friends in the research group. For Saksham, it has been interesting to see you in the office and at home, as you are literally next door. I wish you the best of luck as you start your career as a faculty member. For Midhul, I feel very fortunate to have collaborated with you and learned a lot from you. Our deep discussions about the general trends of technology during lunch were among my favorite times in Ithaca. For Benny, you are the one who made me feel that pursuing academia is worthwhile, as I can see the tremendous value in training the next generation of researchers.

For Shreyas and Shouxu, I am confident you will become, or have already become, excellent researchers. Every piece of feedback I gave showed the great expectations I have for both of you.

I am very fortunate to have joined the Cornell Computer Science family. For Lorenzo Alvisi and Robbert van Renesse, thank you for hosting the systems seminar and giving me the opportunity to present my work in the systems group; it has been invaluable in helping me practice my research presentations. For Nate Foster, every conversation we have had has inspired me with your relaxed attitude towards life and your encouragement to join academia. For Hakim Weatherspoon, I am grateful for all the feedback you provided regarding career choices. For Rachee Singh, I still remember the invaluable feedback you gave me about career options during one of our coffee chats. For Abhishek, I am very grateful for the fresh perspectives on silicon photonics you provided, and your deep understanding always impressed me. For Gloire, seeing you work so hard in the office during our first year, always in such a happy mood, has always made me feel joyful. For Daniel, I always learn a lot from our discussions about research and life. For Yunhao, it is always fun to talk to you when you show me new work related to blockchain and share your philosophy on research and academia.

Lastly, but certainly not least, I would like to thank my family for their unwavering support. My grandma, Liuying, always expresses her happiness for every small achievement I make. My parents, Youmin and Meiyun, always trust me and stand by me. My wife, Cornelia, thank you for everything you have provided. There are so many things I want to say to my family, but somehow, I feel too shy to express all these meaningful sentiments and convey how important you are to my life in this acknowledgment.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

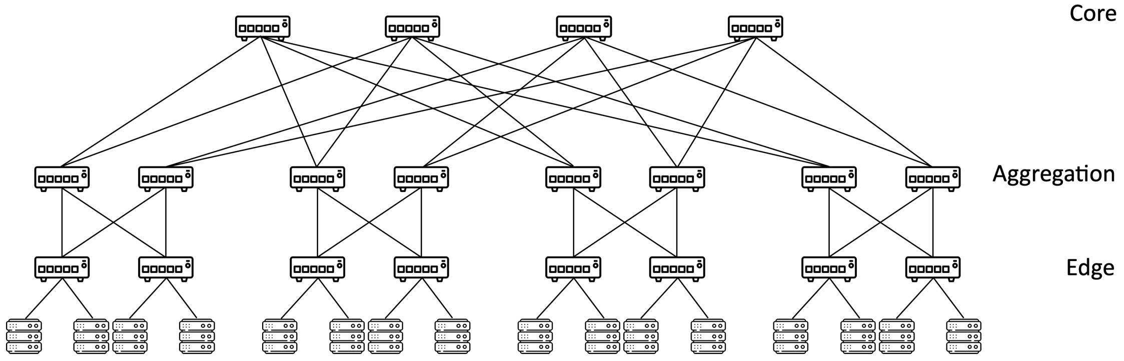


Figure 1.1: Example of a Datacenter Topology. One example of a datacenter topology is the fat-tree topology. The fat-tree topology consists of three layers of switches: core, aggregation, and edge, and supports full-bisection bandwidth [1]. Servers are directly connected to the edge switches.

Today, datacenters are the infrastructure backbone for many applications and services, ranging from machine learning [2] and big data analytics [3, 4] to fast in-memory storage [5, 6], web services [7, 8], and video conferencing [9]. These applications often operate on a very large scale, requiring robust and efficient datacenter infrastructure to support their needs. At their core, datacenters are facilities housing end hosts (or servers) and network components (e.g., network switches), which are interconnected to support the diverse needs of applications and services. One example of a datacenter topology is shown in Figure 1.1.

To ensure the good network performance, datacenter networking needs to achieve low latency and high throughput for data transfers between servers. Generally speaking, applications running in the datacenters can fall into one of two categories: batching processing applications and interactive applications. Batch processing applications require high throughput for processing data in grouped batches. For example, in Meta's deep learning recommendation model training, achieving high performance requires continuously transmitting a large volume of batched data to the training cluster within the datacenter network [10]. On the other hand, interactive applications, designed to respond to user inputs in real-time, require low latency to ensure a smooth user experience. Typically, responding to a user query may involve multiple services. High tail latency in one service can significantly impact overall performance. To maintain a smooth user experience, a service may need to achieve tail latency of 100s of microseconds, or even 10s of microseconds [11–16].

Over the last two decades, datacenter hardware technology trends have evolved differently, presenting new challenges for datacenter networking to achieve low latency and high throughput. On the one hand, access link bandwidth has increased rapidly [17], with datacenters now commonly supporting Terabit Ethernet (Ethernet with speeds over 100 Gbps) [18]. Modern network hardware is capable of supporting microsecond-scale latency and multi-hundred-gigabit bandwidth. However, the slowdown of Moore's Law and the end of Dennard scaling have caused per-core performance to be largely stagnant [17], requiring more CPU resources to be used for network packet processing. These two trends essentially have pushed network performance bottlenecks to the host network stacks.

1.1 Challenges in Host Network Stacks

The host network stack is a critical layer sitting between the network and the applications. The stack needs to handle tasks such as transferring data from/to applications, controlling transmission rates, ensuring reliable data transfer across the network, managing routing information, and sending/receiving packets through network interface cards (NICs).

Specifically, the hardware technology trends within datacenters have led to multiple challenges within host network stacks:

Poor CPU efficiency. While datacenters commonly support 100Gbps access link bandwidth, the Linux network stack, one of the widely deployed network stacks, can only achieve 40Gbps per core, even with all existing optimizations (as will be shown in Chapter 2). This results in more CPU cycles being spent on network stack processing, leaving fewer cycles available for running applications. To make things even worse, as network bandwidth continues to increase rapidly, even more CPU cycles will be required for processing network packets in the future.

High Tail Latency. As network hardware continues to evolve (e.g., evolving processing speed of network switches and increasing network link bandwidth), network round-trip time (RTT) reduces from millisecond-scale to hundreds of microseconds or even tens of microseconds [11]. On the other hand, several recent studies show that the Linux network stack suffers from millisecond-scale tail latency [11–16]. This essentially makes the packet processing latency of host network stacks a bottleneck for achieving low tail latency.

Poor resource provisioning. Given today's stagnant CPU speed and the rapid increase in network bandwidth, even after improving per-core CPU efficiency, it is likely that multiple cores will still be needed to process network packets. Ideally, the network stack is responsible for provisioning CPU cores to process

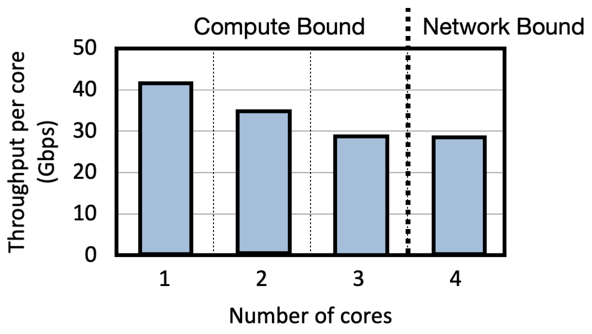


Figure 1.2: Challenges in Finding the Optimal Configuration. In this simple experiment, we run iperf [19] applications on dedicated CPU cores using a 100Gbps NIC and iperf has minimal application overhead. Even when applications are run on dedicated CPU cores, throughput-per-core degrades with an increasing number of CPU cores. As a result, it becomes very challenging for users to find the optimal configuration for achieving the desired performance, as application performance can change during runtime when other applications sharing the same server start or finish.

network packets, but today's network stacks struggle with this task. This is because, the network stack was initially designed when a single CPU core could keep up with network bandwidth, so there was no need to account for multiple cores in network packet processing. However, this is no longer the case. This limitation has shifted the burden of resource provisioning onto users, leaving them struggling to maintain optimal performance and forcing them into a tedious optimization loop as new hardware and applications emerge.

To illustrate the challenges users face in exploiting network hardware, consider the following case. Figure 1.2 illustrates the throughput-per-core that a server can achieve with an increasing number of CPU cores; we observe that throughput-per-core degrades: while a single core can achieve 40Gbps, two cores only reach 70Gbps (the reasons for this will be discussed in Chapter 2). When an application targets a specific throughput level (e.g., 80Gbps), it requires users to conduct thorough benchmarking to find the optimal setup since throughput-per-core is not stable. More importantly, this variability implies that application performance can change during runtime. For instance, even with dedicated CPU cores, the performance of an application that uses two CPU cores can further degrade from 70Gbps to 60Gbps when another application starts using additional CPU cores to transmit data over the network. This uncertainty makes it difficult for users to find the optimal configuration for efficiently using network hardware, requiring them to reconfigure systems repeatedly during runtime or overprovision resources. As network bandwidth continues to increase, requiring more CPU cores to fully leverage network hardware, managing CPU resources to meet application performance requirements becomes increasingly challenging.

Unable to provide performance isolation. To save CPU cycles, cloud providers today may co-locate multiple applications on the same CPU cores [20]. However, this approach can compromise application performance. For instance, a latency-sensitive application (e.g., interactive applications) can achieve microsecond-scale tail latency when running in isolation. But when co-located with a throughput-bound application (e.g., batch-processing applications), the tail latency inflates to millisecond-scale [21]. The network packet processing of throughput-bound applications interferes with the performance of latency-sensitive applications. Today's network stacks are unable to isolate the network packet processing of different applications when sharing the same CPU cores, leading to performance degradation.

1.2 Thesis

While network hardware can support μs-scale latency and hundreds of gigabits of bandwidth, designing end-host network stacks that can leverage these capabilities efficiently has become a key open research problem.

This dissertation dives into understanding the fundamental limitations of existing network stacks. Our study indicates: 1) With the rapid increase in access link bandwidth, data movement overheads (e.g., moving data from NICs to CPUs) become the bottleneck for scaling single-core performance; and with today's network stacks, we need multiple cores to exploit the capabilities of Terabit network hardware. 2) However, as shown in §1.1, using multiple cores further degrades the CPU efficiency. This is because host resources such as CPU caches and access link bandwidth are contended, leading to performance degradation.

This dissertation also introduces NetChannel, a new host network stack architecture that allows host network stacks to manage multiple cores, fully leveraging the capabilities of modern network hardware. As shown in §1.1, today's host network stacks are unable to manage multiple cores effectively to exploit network hardware; this dissertation finds out this inability stems from today's host network stacks' dedicated, tightly integrated, and static packet processing pipelines. Typically, host resources allocated to various parts of the network stack pipelines, such as driver processing and transport layer processing, are static and cannot be shared across different pipelines. Additionally, these different processing parts are closely integrated, making it difficult to dynamically allocate resources. The dissertation demonstrates that by disaggregating host network stacks into multiple loosely-coupled layers and allowing each layer to schedule and scale independently, one can enable the host network stacks to manage multiple cores to exploit network hardware without requiring application modification.

1.3 Contributions

In this section, we summarize the key contributions of this dissertation.

1.3.1 Understanding Host Network Stack Overheads

In Chapter 2, we dive into understanding the fundamental limitations of existing network stacks, focusing on CPU efficiency. The insights from our study were surprising. Recent studies in networking and systems have explored the use of hardware offloading to improve CPU efficiency at higher network bandwidths. However, our study revealed that this approach is not a one-size-fits-all solution. The reason is that processing itself is not the only bottleneck leading to CPU inefficiency. Data movement or I/O is another important bottleneck. For example, on the receiver side, NICs DMA the received data to memory, and the CPU needs to stall while waiting for data to move from memory to CPU registers, leading to poor CPU efficiency. We observed in long-flow processing that data copy processing contributes to more than 50% of total CPU usage, and one of the reasons leading to high data copy overheads is the high data movement overheads. This results in a single core being inadequate to fully utilize the link bandwidth. To reduce this overhead, a new data path has been designed: transferring packets from NICs directly to the L3 CPU cache upon receiving them, allowing the CPU to read data from the cache instead of memory (e.g., Intel's Data Direct I/O (DDIO)). Surprisingly, even with DDIO, we observed that a single long flow experiences a high cache miss rate on the receiver side. Our investigation revealed the root cause: as bandwidth grows and CPUs struggle to keep up, the host processing latency increases, leading to the amount of in-flight data exceeding the cache's capacity. This makes DDIO less effective, resulting in higher cache miss rates and poor CPU efficiency. As the number of flows increases, CPU inefficiency further degrades due to host resource contention among flows (*e.g.* cache and access link bandwidth).

The study leaves open interesting research directions to optimize CPU efficiency for network packet processing; these may require re-architecting the network hardware and even co-designing the network hardware and software infrastructure, which will be further discussed in Chapter 4.

1.3.2 Rearchitecting Host Network Stacks for Terabit Ethernet

As discussed in §1.1, in the multi-core context, it is challenging for network operators and application users today to find the optimal configuration to exploit network hardware during runtime. We realize that this challenge arises from the current network stack's inability to fully leverage the hardware in a transparent manner, thereby shifting the burden onto network operators and users. We find out this inability stems from today's host network stack's dedicated, tightly integrated, and static packet processing pipelines. To address this challenge, in Chapter 3, we present NetChannel, a new host network stack architecture for Terabit Ethernet. NetChannel disaggregates the network stack into loosely coupled layers, enabling the dynamic scaling and scheduling of each layer across CPU cores to exploit the hardware. We demonstrated that NetChannel offers several new operating points that were previously unachievable without modifying network hardware: 1) By transparently scaling data copy processing, NetChannel empowers a single application thread to saturate multi-hundred gigabit access link bandwidth (e.g., 200Gbps). 2) Through scaling transport protocol processing, NetChannel achieves nearly linear scalability for small message processing as the number of cores increases. 3) NetChannel achieves performance isolation for latency-sensitive applications, allowing them to maintain low-latency performance even they are colocated with throughput-bound applications operating at near-line rates. Such isolation is achieved by isolating the packet processing for different types of applications.

CHAPTER 2
UNDERSTANDING NETWORK STACK OVERHEADS

Traditional end-host network stacks are struggling to keep up with rapidly increasing datacenter access link bandwidths due to their unsustainable CPU overheads. Motivated by this, our community is exploring a multitude of solutions for future network stacks: from Linux kernel optimizations to partial hardware offload to clean-slate userspace stacks to specialized host network hardware. The design space explored by these solutions would benefit from a detailed understanding of CPU inefficiencies in existing network stacks.

In this chapter, we present measurement and insights for Linux kernel network stack performance for 100Gbps access link bandwidths. Our study reveals that such high bandwidth links, coupled with relatively stagnant technology trends for other host resources (e.g., core speeds and count, cache sizes, NIC buffer sizes, etc.), mark a fundamental shift in host network stack bottlenecks. For instance, we find that a single core is no longer able to process packets at line rate, with data copy from kernel to application buffers at the receiver becoming the core performance bottleneck. In addition, increase in bandwidth-delay products have outpaced the increase in cache sizes, resulting in inefficient DMA pipeline between the NIC and the CPU. Finally, we find that traditional loosely-coupled design of network stack and CPU schedulers in existing operating systems becomes a limiting factor in scaling network stack performance across cores. Based on insights from this chapter, we will discuss implications to design of future operating systems, network protocols, and host hardware in Chapter 4.

2.1 Overview

The slowdown of Moore's Law, the end of Dennard's scaling, and the rapid adoption of high-bandwidth links have brought traditional host network stacks at the brink of a breakdown—while datacenter access link bandwidths (and resulting computing needs for packet processing) have increased by 4 – 10× over the past few years, technology trends for essentially all other host resources (including core speeds and counts, cache sizes, NIC buffer sizes, etc.) have largely been stagnant. As a result, the problem of designing CPU-efficient host network stacks has come to the forefront, and our community is exploring a variety of solutions, including Linux network stack optimizations [22–27], hardware offloads [28–31], RDMA [32–34], clean-slate userspace network stacks [12, 13, 16, 35, 36], and even specialized host network hardware [37]. The design space explored by these solutions would benefit from a detailed understanding of CPU inefficiencies of traditional Linux network stack. Building such an understanding is hard because the Linux network stack is not only large and complex, but also comprises of many components that are tightly integrated into an end-to-end packet processing pipeline.

Several recent papers present a preliminary analysis of Linux network stack overheads for short flows [16, 25, 26, 38, 39]. This fails to provide a complete picture due to two reasons. First, for datacenter networks, it is well-known that an overwhelmingly large fraction of data is contained in long flows [40–42]; thus, even if there are many short flows, most of the CPU cycles may be spent in processing packets from long flows. Second, datacenter workloads contain not just short flows or long flows in exclusion, but a mixture of different flow sizes composed in a variety of traffic patterns; as we will demonstrate, CPU characteristics change significantly with varying traffic patterns and mixture of flow sizes.

This chapter presents measurement and insights for Linux kernel network stack performance for 100Gbps access link bandwidths. Our key findings are:

High-bandwidth links result in performance bottlenecks shifting from protocol processing to data copy. Modern Linux network stack can achieve ~42Gbps throughput-per-core by exploiting all commonly available features in commodity NICs, *e.g.*, segmentation and receive offload, jumbo frames, and packet steering. While this throughput is for the best-case scenario of a single long flow, the dominant overhead is consistent across a variety of scenarios—data copy from kernel buffers to application buffers (*e.g.*, > 50% of total CPU cycles for a single long flow). One critical factor leading to such substantial data copy overheads is the inefficient data movements or I/O: the CPU has to stall while waiting for data to be loaded from memory or cache into the CPU registers. This is in sharp contrast to previous studies on short flows and/or low-bandwidth links, where protocol processing was shown to be the main bottleneck. We also observe receiver-side packet processing to become a bottleneck much earlier than the sender-side.

- *Implications.* Emerging zero-copy mechanisms from the Linux networking community [23, 24] may alleviate data copy overheads, and may soon allow the Linux network stack to process as much as 100Gbps worth of data using a single core. Integration of other hardware offloads like I/OAT [43] that transparently mitigate data copy overheads could also lead to performance improvements. Hardware offloads of transport protocols [30, 32] and userspace network stacks [16,26,35] that do not provide zero-copy interfaces may improve throughput in microbenchmarks, but will require additional mechanisms to achieve CPU efficiency when integrated into an end-to-end system. On the other hand, zero-copy technologies might not be a panacea, as single-core CPU efficiency can still be constrained by I/O bandwidth per CPU core. For example, if all the data needs to be accessed from memory, single-core throughput will be limited by the memory bandwidth per core, which is significantly less than the expected network bandwidth in the near future.

The reducing gap between bandwidth-delay product (BDP) and cache sizes leads to suboptimal throughput. To reduce data movement or I/O overheads, modern CPU support for Direct Cache Access (DCA) (*e.g.*, Intel DDIO [44]) allows NICs to DMA packets directly into L3 cache; given its promise, DDIO is enabled by default in most systems. While DDIO is expected to improve performance during data copy, rather surprisingly, we observe that it suffers from high cache miss rates (49%) even for a single flow, thus providing limited performance gains. Our investigation revealed that the reason for this is quite subtle: host processing becoming a bottleneck results in increased host latencies; combined with increased access link bandwidths, BDP values increase. This increase outpaces increase in L3 cache sizes—data is DMAed from the NIC to the cache, and for larger BDP values, cache is rapidly overwritten before the application performs data copy of the cached data. As a result, we observe as much as 24% drop in throughput-per-core.

- *Implications.* We need new mechanisms to minimize delay between packet reception and subsequent data copy to minimize cache miss rates during data copy. One possibility is to rearchitect host network stacks to enable independent scaling of processing capacity for individual layers (*e.g.*, enabling multiple cores to perform data copy for a single flow) to overcome bottlenecks. In addition, window size tuning should take into account not only traditional metrics like latency and throughput, but also L3 sizes. To further reduce host processing latency, we may also need to design new DMA pipelines, such as DMAing data closer to CPUs, into caches like L1 or L2, or even into CPU registers [45,46].

Host resource sharing considered harmful. We observe as much as 66% difference in throughput-per-core across different traffic patterns (single flow, one-to-one, incast, outcast, and all-to-all) due to undesirable effects of multiple flows sharing host resources. For instance, multiple flows on the same NUMA node (thus, sharing the same L3 cache) make the cache performance even worse—the data DMAed by the NIC into the cache for one flow is polluted by the data DMAed by the NIC for other flows, before application for the first flow could perform data copy. Multiple flows sharing host resources also results in packets arriving at the NIC belonging to different flows; this, in turn, results in packet processing overheads getting worse since existing optimizations (*e.g.*, coalescing packets using generic receive offload) lose a chance to aggregate larger number of packets. This increases per-byte processing overhead, and eventually scheduling overheads.

- *Implications.* In the Internet and in early-generation datacenter networks, performance bottlenecks were in the network core; thus, multiple flows "sharing" host resources did not have performance implications. However, for high-bandwidth networks, such is no longer the case—if the goal is to design

CPU-efficient network stacks, one must carefully orchestrate host resources so as to minimize contention between active flows. Recent receiver-driven transport protocols [47,48] can be extended to reduce the number of concurrently scheduled flows, potentially enabling high CPU efficiency for future network stacks.

The need to revisit host layering and packet processing pipelines. We observe as much as ~43% reduction in throughput-per-core compared to the single flow case when applications generating long flows share CPU cores with those generating short flows. This is both due to increased scheduling overheads, and also due to high CPU overheads for short flow processing. In addition, short flows and long flows suffer from very different performance bottlenecks—the former have high packet processing overheads while the latter have high data copy overheads; however, today's network stacks use the same packet processing pipeline independent of the type of the flow. Finally, we observe ~20% additional drop in throughput-per-core when applications generating long flows are running on CPU cores that are not in the same NUMA domain as the NIC (due to additional data copy overheads).

- *Implications.* Design of CPU schedulers independent of the network layer was beneficial for independent evolution of the two layers; however, with performance bottlenecks shifting to hosts, we need to revisit such a separation. For instance, application-aware CPU scheduling (*e.g.*, scheduling applications that generate long flows on NIC-local NUMA node, scheduling long-flow and short-flow applications on separate CPU cores, etc.) are required to improve CPU efficiency. We should also rethink host packet processing pipelines—unlike today's designs that use the same pipeline for short and long flows, achieving CPU efficiency requires application-aware packet processing pipelines.

Our study not only corroborates many exciting ongoing activities in systems, networking and architecture communities on designing CPU-efficient host network stacks, but also highlights several interesting avenues for research in designing future operating systems, network protocols and network hardware. We discuss them in Chapter 4.

Before diving deeper, we outline several caveats of our study. First, our study uses one particular host network stack (the Linux kernel) running atop one particular host hardware. While we focus on identifying trends and drawing general principles rather than individual data points, other combinations of host network stacks and hardware may exhibit different performance characteristics. Second, our study focuses on CPU utilization and throughput; host network stack latency is another important metric, but requires exploring many additional bottlenecks in end-to-end system (*e.g.*, network topology, switches, congestion, etc.); a study that establishes latency bottlenecks in host network stacks, and their contribution to end-to-end latency remains an important and relatively less explored space. Third, kernel network stacks evolve rapidly; any study of our form must fix a version to ensure consistency across results and observations; nevertheless, our preliminary exploration [49] suggests that the most recent Linux kernel exhibits performance very similar to our results. Finally, our goal is not to take a position on how future network stacks will evolve (in-kernel, userspace, hardware), but rather to obtain a deeper understanding of a highly mature and widely deployed network stack.

2.2 Preliminaries

The Linux network stack tightly integrates many components into an end-to-end pipeline. We start this section by reviewing these components (§2.2.1). We also discuss commonly used optimizations, and corresponding hardware offloads supported by commodity NICs. A more detailed description is presented in [49]. We then summarize the methodology used in our study (§2.2.2).

2.2.1 End-to-End Data Path

The Linux network stack has slightly different data paths for the sender-side (application to NIC) and the receiver-side (NIC to application), as shown in Fig. 2.1. We describe them separately.

Component	**Description**
Data copy	From user space to kernel space, and vice versa.
TCP/IP	All the packet processing at TCP/IP layers.
Netdevice subsystem	Netdevice and NIC driver operations (*e.g.*, NAPI polling, GSO/GRO, qdisc, etc.).
`skb` management	Functions to build, split, and release `skb`.
Memory de-/alloc	`skb` de-/allocation and page-related operations.
Lock/unlock	Lock-related operations (*e.g.*, spin locks).
Scheduling	Scheduling/context-switching among threads.
Others	All the remaining functions (*e.g.*, IRQ handling).

Table 2.1: CPU usage taxonomy. The components are mapped into layers as shown in Fig. 2.1.

Sender-side. When the sender-side application executes a `write` system call, the

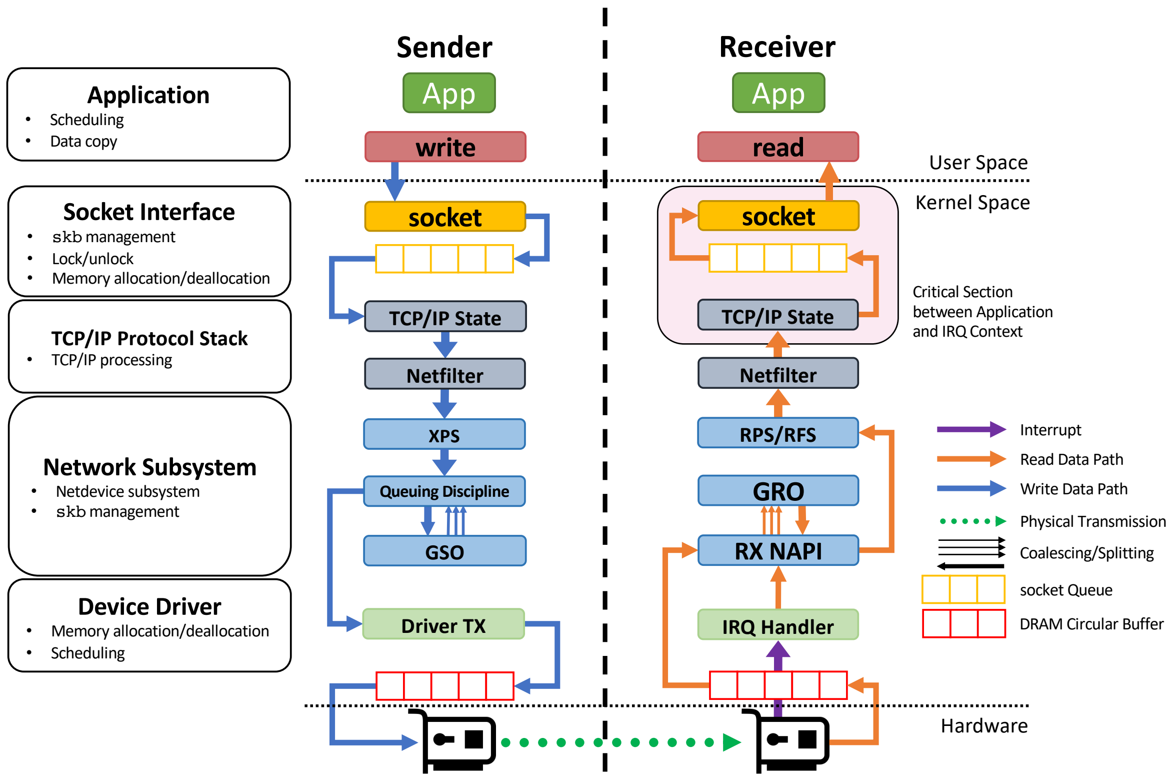


Figure 2.1: Sender and receiver-side data path in the Linux network stack. See §2.2.1 for description.

kernel initializes socket buffers (skbs). For the data referenced by the skbs, the kernel then performs data copy from the userspace buffer to the kernel buffer. The skbs are then processed by the TCP/IP layer. When ready to be transmitted (*e.g.*, congestion control window/rate limits permitting), the data is processed by the network subsystem; here, among other processing steps, skbs are segmented into Maximum Transmission Unit (MTU) sized chunks by Generic Segmentation offload (GSO) and are enqueued in the NIC driver Tx queue(s). Most commodity NICs also support hardware offload of packet segmentation, referred to as TCP segmentation offload (TSO); see more details in [49]. Finally, the driver processes the Tx queue(s), creating the necessary mappings for the NIC to DMA the data from the kernel buffer referenced by skbs. Importantly, almost all sender-side processing in today's Linux network stack is performed at the same core as the application.

Receiver-side. The NIC has a number of Rx queues and a per-Rx queue page-pool from which DMA memory is allocated (backed by the kernel `pageset`). The NIC also has a configurable number of Rx descriptors, each of which contains a memory address that the NIC can use to DMA received frames. Each descriptor is associated with enough memory for one MTU-sized frame.

Upon receiving a new frame, the NIC uses one of the Rx descriptors, and DMAs the frame to the kernel memory associated with the descriptor. Ordinarily, the NIC DMAs the frame to DRAM; however, modern CPUs have support for Direct Cache Access (DCA) (*e.g.*, using Intel's Data Direct I/O technology (DDIO) technology [44]) that allows NIC to DMA the frames directly to the L3 cache. DCA enables applications to avoid going to DRAM to access the data.

Asynchronously, the NIC generates an Interrupt ReQuests (IRQ) to inform the driver of new data to be processed. The CPU core that processes the IRQ is selected by the NIC using one of the hardware steering mechanisms; see Table 2.2 for a summary, and [49] for details on how receiver-side flow steering techniques work. Upon receiving an IRQ, the driver triggers NAPI polling [50], that provides an alternative to purely interrupt-based network layer processing—the system busy polls on incoming frames until a certain number of frames are received or a timer expires[1]. This reduces the number of IRQs, especially for high-speed networks where incoming data rate is high. While busy polling, the driver allocates an `skb` for each frame, and makes a cross reference between the `skb` and the kernel memory where the frame has

[1]These NAPI parameters can be tuned via `net.core.netdev_budget` and `net.core.netdev_budget_usecs` kernel parameters, which are set to 300 and 2ms by default in our Linux distribution.

Mechanism	**Description**
Receive Packet Steering (RPS)	Use the 4-tuple hash for core selection.
Receive Flow Steering (RFS)	Find the core that the application is running on.
Receive Side Steering (RSS)	Hardware version of RPS supported by NICs.
accelerated RFS (aRFS)	Hardware version of RFS supported by NICs.

Table 2.2: Receiver-side flow steering techniques.

been DMAed. If the NIC has written enough data to consume all Rx descriptors, the driver allocates more DMA memory using the page-pool and creates new descriptors.

The network subsystem then attempts to reduce the number of `skbs` by merging them using Generic Receive Offload (GRO), or its corresponding hardware offload Large Receive Offload (LRO); see discussion in [49]. Next, TCP/IP processing is scheduled on one of the CPU cores using the flow steering mechanism enabled in the system (see Table 2.2). Importantly, with aRFS enabled, all the processing (the IRQ handler, TCP/IP and application) is performed on the same CPU core. Once scheduled, the TCP/IP layer processing is performed and all in-order `skbs` are appended to the socket's receive queue. Finally, the application thread performs data copy of the payload in the `skbs` in the socket receive queue to the userspace buffer. Note that at both the sender-side and the receiver-side, data copy of packet payloads is performed only once (when the data is transferred between userspace and kernel space). All other operations within the kernel are performed using metadata and pointer manipulations on `skbs`, and do not require data copy.

2.2.2 Measurement Methodology

In this subsection, we briefly describe our testbed setup, experimental scenarios, and measurement methodology.

Testbed setup. To ensure that bottlenecks are at the network stack, we setup a testbed with two servers directly connected via a 100Gbps link (without any intervening switches). Both of our servers have a 4-socket NUMA-enabled Intel Xeon Gold 6128 3.4GHz CPU with 6 cores per socket, 32KB/1MB/20MB L1/L2/L3 caches, 256GB RAM, and a 100Gbps Mellanox ConnectX-5 Ex NIC connected to one of the sockets. Both servers run Ubuntu 16.04 with Linux kernel 5.4.43. Unless specified otherwise, we enable DDIO, and disable hyper-threading and IOMMU in our experiments.

Experimental scenarios. We study network stack performance using five standard traffic patterns (Fig. 2.2)—single flow, one-to-one, incast, outcast, and all-to-all—using workloads that comprise long flows, short flows, and even a mix of long and short flows. For generating long flows, we use a standard network benchmarking tool, iPerf [19], which transmits a flow from sender to receiver; for generating short flows, we use netperf [51] that supports ping-pong style RPC workloads. Both of these tools perform minimal application-level processing, which allows us to focus on performance bottlenecks in the network stack (rather than those arising due to complex interactions between applications and the network stack); many of our results may have different characteristics if applications were to perform additional processing. We also study the impact of in-network congestion, impact of DDIO and impact of IOMMU. We use Linux's default congestion control algorithm, TCP Cubic, but also study impact of dif-

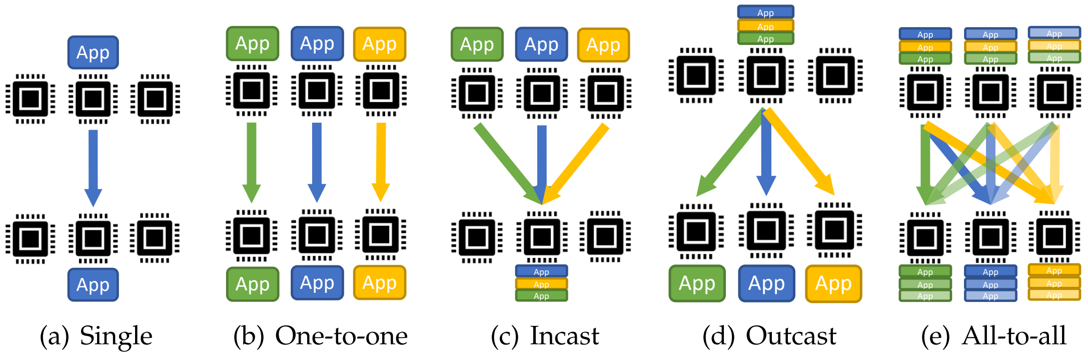


Figure 2.2: Traffic patterns used in our study. (a) Single flow from one sender core to one receiver core. (b) One flow from each sender core to a unique receiver core. (c) One flow from each sender core, all to a single receiver core. (d) One flow to each receiver core all from a single sender core. (e) One flow between every pair of sender and receiver cores.

ferent congestion control protocols. For each scenario, we describe the setup inline.

Performance metrics. We measure total throughput, total CPU utilization across all cores (using `sysstat` [52], which includes kernel and application processing), and throughput-per-core—ratio of total throughput and total CPU utilization at the bottleneck (sender or receiver). To perform CPU profiling, we use the standard sampling-based technique to obtain a per-function breakdown of CPU cycles [53]. We take the top functions that account for ~95% of the CPU utilization. By examining the kernel source code, we classify these functions into 8 categories as described in Table 2.1.

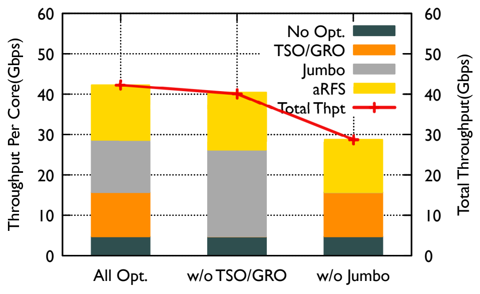
2.3 Linux Network Stack Overheads

We now evaluate the Linux network stack overheads for a variety of scenarios, and present detailed insights on observed performance.

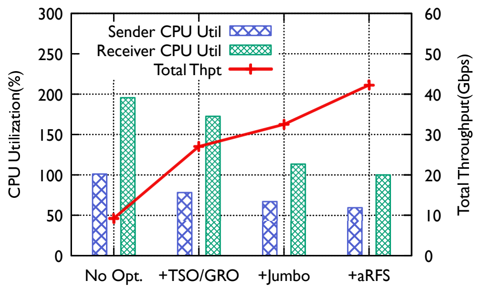
2.3.1 Single Flow

We start with the case of a single flow between the two servers, each running an application on a CPU core in the NIC-local NUMA node. We find that, unlike the Internet and early incarnations of datacenter networks where the throughput bottlenecks were primarily in the core of the network (since a single CPU was sufficient to saturate the access link bandwidth), high-bandwidth networks introduce new host bottlenecks even for the simple case of a single flow.

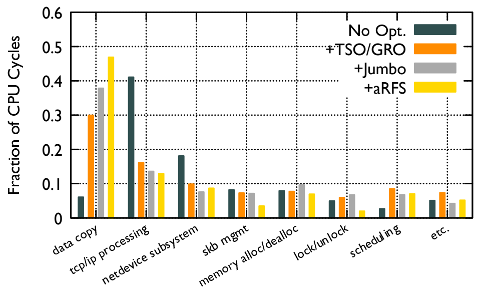
Before diving deeper, we make a note on our experimental configuration for the single flow case. When aRFS is disabled, obtaining stable and reproducible measurements is difficult since the default RSS mechanism uses hash of the 4-tuple to determine the core for IRQ processing (§2.2.1). Since the 4-tuple can change across runs, the core that performs IRQ processing could be: (1) the application core; (2) a core on the same NUMA node; or, (3) a core on a different NUMA node. The performance in each of these three cases is different, resulting in non-determinism. To ensure deterministic measurements, when aRFS is disabled, we model the worst-case scenario (case 3): we explicitly map the IRQs to a core on a NUMA node different from the application core. For a more detailed analysis of other possible IRQ mapping scenarios, see [49].



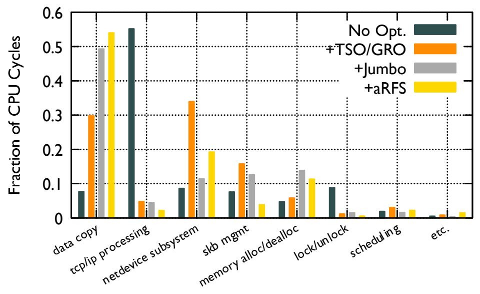
(a) Throughput-per-core (Gbps)



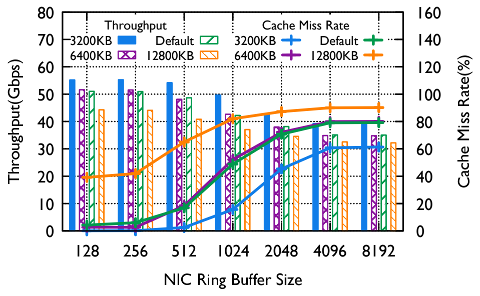
(b) CPU utilization (%)



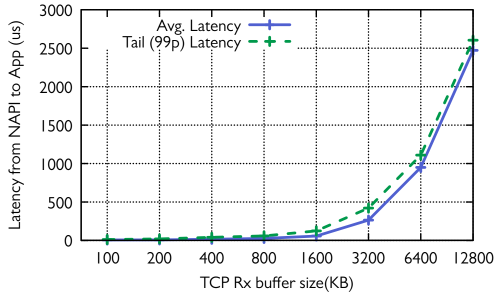
(c) Sender CPU breakdown



(d) Receiver CPU breakdown



(e) Cache miss rate (%)



(f) Latency from NAPI to start of data copy

Figure 2.3: Linux network stack performance for the case of a single flow. (a) Each column shows throughput-per-core achieved for different combinations of optimizations. Within each column, optimizations are enabled incrementally, with each colored bar showing the incremental impact of enabling the corresponding optimization. **(b)** Sender and Receiver total CPU utilization as all optimizations are enabled incrementally. Independent of the optimizations enabled, receiver-side CPU is the bottleneck. **(c, d)** With all optimizations enabled, data copy is the dominant consumer of CPU cycles. **(e)** Increase in NIC ring buffer size and increase in TCP Rx buffer size result in increased cache miss rates and reduced throughput. **(f)** Network stack processing latency from NAPI to start of data copy increases rapidly beyond certain TCP Rx buffer sizes. See §2.3.1 for description.

A single core is no longer sufficient. For 10 – 40Gbps access link bandwidths, a single thread was able to saturate the network bandwidth. However, such is no longer the case for high-bandwidth networks: as shown in Fig. 2.3(a), even with all optimization enabled, Linux network stack achieves throughput-per-core of ~42Gbps[2]. Both Jumbo frames[3] and TSO/GRO reduce the per-byte processing overhead as they allow each `skb` to bring larger payloads (up to 9000B and 64KB respectively). Jumbo frames are useful even when GRO is enabled, because the number of `skbs` to merge is reduced with a larger MTU size, thus reducing the processing overhead for packet aggregation in software. aRFS, along with DCA, generally improves throughput by enabling applications on the NIC-local NUMA node cores to perform data copy directly from L3 cache.

Receiver-side CPU is the bottleneck. Fig. 2.3(b) shows the overall CPU utilization at sender and receiver sides. Independent of the optimizations enabled, receiver-side CPU is the bottleneck. There are two dominant overheads that create the gap between sender and receiver CPU utilization: (1) data copy and (2) `skb` allocation. First, when aRFS is disabled, frames are DMAed to remote NUMA memory at the receiver; this causes data copy across different NUMA nodes, increasing per-byte data copy overhead as the CPU stalls while waiting for data to load from remote NUMA. This is not an issue on the sender-side since the local L3 cache is warm with the application send buffer data. Enabling aRFS alleviates this issue reducing receiver-side CPU utilization by as much as 2×

[2]We observe a maximum throughput-per-core of upto 55Gbps, either by tuning NIC Rx descriptors and TCP Rx buffer size carefully (See Fig. 2.3(e)), or using LRO instead of GRO (See [49]). However, such parameter tuning is very sensitive to the hardware setup, and so we leave them to their default values for all other experiments. Moreover, the current implementation of LRO causes problems in some scenarios as it might discard important header data, and so is often disabled in the real world [54]. Thus we use GRO as the receive offload mechanism for the rest of our experiments.

[3]Using larger MTU size (9000 bytes) as opposed to the normal (1500 bytes).

(right-most bar in Fig. 2.3(b)) compared to the case when no optimizations are enabled; however, CPU utilization at the receiver is still higher than the sender. Second, when TSO is enabled, the sender is able to allocate large-sized `skb`s. The receiver, however, allocates MTU-sized `skb`s at device driver and then the `skb`s are merged at GRO layer. Therefore, the receiver incurs higher overheads for `skb` allocation.

Where are the CPU cycles going? Figs. 2.3(c) and 2.3(d) show the CPU usage breakdowns of sender- and receiver-side for each combination of optimizations. With none of the optimizations, CPU overheads mainly come from TCP/IP processing as per-`skb` processing overhead is high (here, `skb` size is 1500B at both sides[4]). When aRFS is disabled, lock overhead is high at the receiver-side because of the socket contention due to the application context thread (`recv` system call) and the interrupt context thread (softirq) attempting to access the same socket instance.

These packet processing overheads are mitigated with several optimizations: TSO allows using large-sized `skb` at the sender-side, reducing both TCP/IP processing and Netdevice subsystem overheads as segmentation is offloaded to the NIC (Fig. 2.3(c)). On the receiver-side, GRO reduces the CPU usage by reducing the number of `skb`s, passed to the upper layer, so TCP/IP processing and lock/unlock overheads are reduced dramatically, at the cost of increasing the overhead of the network device subsystem where GRO is performed (Fig. 2.3(d)). This GRO cost can be reduced by 66% by enabling Jumbo frames as explained above. These reduced packet processing overheads lead to throughput improvement, and the main overhead is now shifted to data copy, which

[4]Linux kernel 4.17 onwards, GSO is enabled by default. We modified the kernel to disable GSO in "no optimization" experiments to evaluate benefits of `skb` aggregation.

takes almost 49% of total CPU utilization at the receiver-side when GRO and Jumbo frames are enabled. A major reason for high data copy overheads is data movement or I/O, unlike short flows where performance bottlenecks are due to per-packet CPU processing, as prior work shows [16]. Here, the main issue is the CPU stalling while waiting for data to load from memory, causing high CPU overheads.

Once aRFS is enabled, co-location of the application context thread and the IRQ context thread at the receiver leads to improved cache and NUMA locality. The effects of this are two-fold:

1. Since the application thread runs on the same NUMA node as the NIC, it can now perform data copy directly from the L3 cache (DMAed by the NIC via DCA) to reduce CPU stalling time.
2. `skbs` are allocated in the softirq thread and freed in the application context thread (once data copy is done). Since the two are co-located, memory deallocation overhead reduces. This is because page free operations to local NUMA memory are significantly cheaper than those for remote NUMA memory.

Even a single flow experiences high cache misses. Although aRFS allows applications to perform data copy from local L3 cache, we observe as much as 49% cache miss rate in this experiment. This is surprising since, for a single flow, there is no contention for L3 cache capacity. To investigate this further, we varied various parameters to understand their effect on cache miss rate. Among our experiments, varying the maximum TCP receive window size, and the number of NIC Rx descriptors revealed an interesting trend.

Fig. 2.3(e) shows the variation of throughput and L3 cache miss rate with varying number of NIC Rx descriptors and varying TCP Rx buffer size[5]. We observe that, with increase in either of the number of NIC Rx descriptors or the TCP buffer size, the L3 cache miss increases and correspondingly, the throughput decreases. We have found two reasons for this phenomenon: (1) BDP values being larger than the L3 cache capacity; and (2) suboptimal cache utilization.

To understand the first one, consider an extreme case of large TCP Rx buffer sizes. In such a case, TCP will keep BDP worth of data in flight, where BDP is defined as the product of access link bandwidth and latency (both network and host latency). It turns out that large TCP buffers can cause a significant increase in host latency, especially when the core processing packets becomes a bottleneck. In addition to scheduling delay of IRQ context and application threads, we observe that each packet observe large queueing behind previous packets. We measure the delay between frame reception and start of data copy by logging the timestamp when NAPI processing for an `skb` happens, and the timestamp when the data copy of it starts, and measure the difference between the two. Fig. 2.3(f) shows the average and 99th percentile delays observed with varying TCP Rx buffer size. As can be seen, the delays rise rapidly with increasing TCP Rx buffer size beyond 1600KB. Given that DCA cache size is limited[6], this increase in latency has significant impact: since TCP buffers and BDP values are large, NIC always has data to DMA; thus, since the data DMAed by the NIC is not promptly copied to userspace buffers, it is evicted from the cache when NIC performs subsequent DMAs (if the NIC runs out of Rx descriptors, the driver replenishes the NIC Rx descriptors during NAPI polling). As a result, cache misses increase and throughput reduces. When TCP buffer sizes are large

[5]The kernel uses an auto-tuning mechanism for the TCP Rx socket buffer size with the goal of maximizing throughput. In this experiment, we override the default auto-tuning mechanism by specifying an Rx buffer size.

[6]DCA can only use 18% (~3 MB) of the L3 cache capacity in our setup.

enough, this problem persists independent of NIC ring buffer sizes.

To understand the second reason, consider the other extreme where TCP buffer sizes are small but NIC ring buffer sizes are large. We believe cache misses in this case might be due to an imperfect cache replacement policy and/or cache's complex addressing, resulting in suboptimal cache utilization; recent work has observed similar phenomena, although in a different context [55, 56]. When there are a large number of NIC Rx descriptors, there is a correspondingly larger number of memory addresses available for the NIC to DMA the data. Thus, even though the total amount of in-flight data is smaller than the cache capacity, the likelihood of a DCA write evicting some previously written data increases with the number of NIC Rx descriptors. This limits the effective utilization of cache capacity, resulting in high cache miss rates and low throughput-per-core.

Between these two extremes, both of the factors contribute to the observed performance in Fig. 2.3(e). Indeed, in our setup, DCA cache capacity is ~3MB and hence TCP buffer size of 3200KB and fewer than 512 NIC Rx descriptors (512×9000 bytes $\approx$ 4MB) delivers the optimal single-core throughput of ~55Gpbs. An interesting observation here is that the default auto-tuning mechanism used in the Linux kernel network stack today is unaware of DCA effects, and ends up overshooting beyond the optimal operating point.

DCA limited to NIC-local NUMA nodes. In our analysis so far, the application was run on a CPU core on the NIC-local NUMA node. We now examine the impact of running the application on a NIC-remote NUMA node for the same single flow experiment. Fig. 2.4 shows the resulting throughput-per-core and L3 cache miss rate relative to the NIC-local case (with all optimizations enabled in

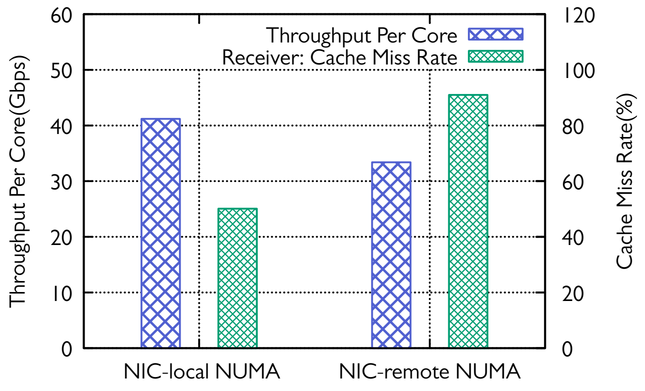
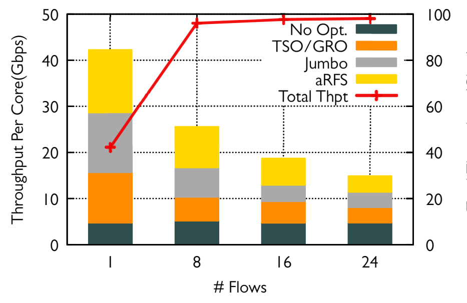


Figure 2.4: Linux network stack performance for the case of a single flow on NIC-remote NUMA node. When compared to the NIC-local NUMA node case, single flow throughput-per-core drops by ~20%.

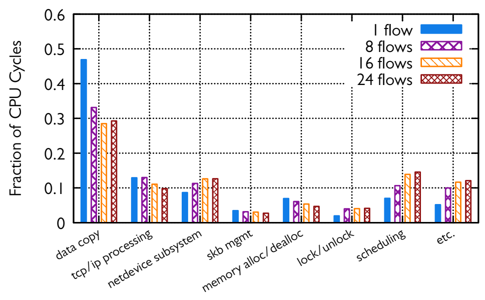
both cases). When the application runs on NIC-remote NUMA node, we see a significant increase in L3 cache miss rate and ~20% drop in throughput-per-core. Since aRFS is enabled, the NIC DMAs frames to the target CPU's NUMA node memory. However, because the target CPU core is on a NIC-remote NUMA node, DCA is unable to push the DMAed frame data into the corresponding L3 cache [44]. As a result, cache misses increase and throughput-per-core drops.

2.3.2 Increasing Contention via One-to-one

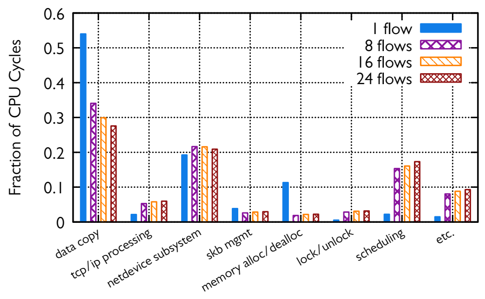
We now evaluate the Linux network stack with higher contention for the network bandwidth. Here, each sender core sends a flow to one unique receiver core, and we increase the number of core/flows from 1 to 24. While each flow still has the entire host core for itself, this scenario introduces two new challenges compared to the single-flow case: (1) network bandwidth becomes saturated as multiple cores are used; and (2) flows run on both NIC-local and NIC-remote NUMA nodes (our servers have 6 cores on each NUMA node).



(a) Throughput-per-core (Gbps)



(b) Sender CPU breakdown



(c) Receiver CPU breakdown

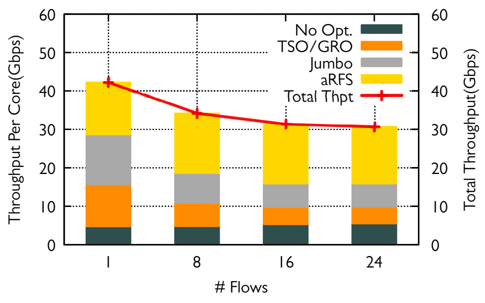
Figure 2.5: Linux network stack performance for one-to-one traffic pattern. (a) Each column shows throughput-per-core achieved for different number of flows. At 8 flows, the network is saturated, however, throughput-per-core decreases with more flows. **(b, c)** With all optimizations enabled, as the number of flows increase, the fraction of CPU cycles spent in data copy decreases. On the receiver-side, network saturation leads to lower memory management overhead (due to better page recycling) and higher scheduling overhead (due to frequent idling). The overall receiver-side CPU utilizations for x= 1, 8, 16 and 24 cases are, 1, 3.75, 5.21 and 6.58 cores, respectively. See §2.3.2 for description.

Similar to §2.3.1, to obtain deterministic measurements when aRFS is disabled, we explicitly map IRQs for individual applications to a unique core on a different NUMA node.

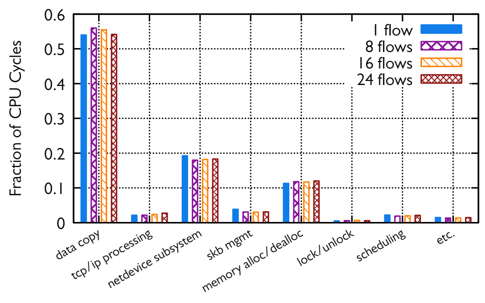
Host optimizations become less effective with increasing number of flows. Fig. 2.5(a) shows that, as the number of flows increases, throughput-per-core decreases by 64% (i.e., 15Gbps at 24 flows), despite each core processing only a single flow. This is because of reduced effectiveness of all optimizations. In particular, when compared to the single flow case, the effectiveness of aRFS reduces by as much as 75% for the 24-flow case; this is due to reduced L3 cache locality for data copy for NIC-local NUMA node cores (all cores share L3 cache), and also due to some of the flows running on NIC-remote NUMA nodes (that cannot exploit DCA, see §2.3.1, Fig. 2.4). The effectiveness of GRO also reduces: since packets at the receiver are now interleaved across flows, there are fewer opportunities for aggregation; this will become far more prominent in the all-to-all case, and is discussed in more depth in §2.3.5.

Processing overheads shift with network saturation. As shown in Fig. 2.5(a), at 8 flows, the network link becomes the bottleneck, and throughput ends up getting fairly shared among all cores. Fig. 2.5(c) shows that bottlenecks shift in this regime: scheduling overhead increases and memory management overhead decreases. Intuitively, when the network is saturated, the receiver cores start to become idle at certain times—threads repeatedly go to sleep while waiting for data, and wake up when new data arrives; this results in increased context switching and scheduling overheads. This effect becomes increasingly prominent with increase in number of flows (Fig. 2.5(b), Fig. 2.5(c)), as the CPU utilization per-core decreases.

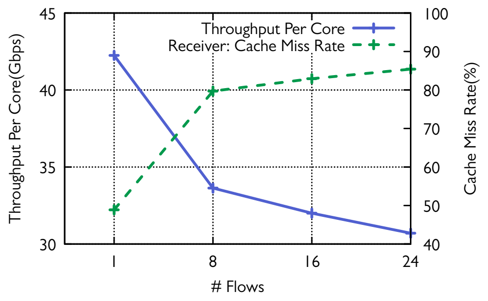
To understand reduction in memory alloc/dealloc overheads, we observe that the kernel page allocator maintains per-core `pageset` that includes a certain number of free pages. Upon an allocation request, pages can be fetched directly from the `pageset`, if available; otherwise the global free-list needs to be accessed (which is a more expensive operation). When multiple flows share the access link bandwidth, each core serves relatively less amount of traffic com-



(a) Throughput-per-core (Gbps)



(b) Receiver CPU breakdown



(c) L3 cache miss rate (%)

Figure 2.6: Linux network stack performance for incast traffic pattern. (a) Each column shows throughput-per-core for different number of flows (receiver core is bottlenecked in all cases). Total throughput decreases with increase in the number of flows. **(b)** With all optimizations enabled, the fraction of CPU cycles used by each component does not change significantly with number of flows. See [49] for sender-side CPU breakdown. **(c)** Receiver-side cache miss rate increases with number of flows, resulting in higher per-byte data copy overhead, and reduced throughput-per-core. See §2.3.3 for description.

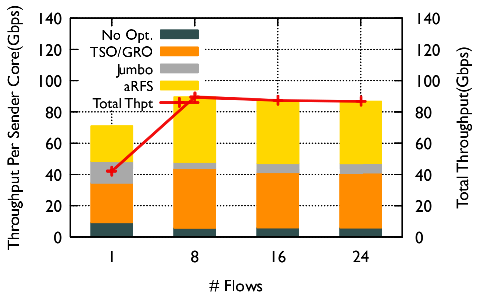
pared to the single flow case. This allows used pages to be recycled back to the `pageset` before it becomes empty, hence reducing the memory allocation overhead (Fig. 2.5(c)).

2.3.3 Increasing Receiver Contention via Incast

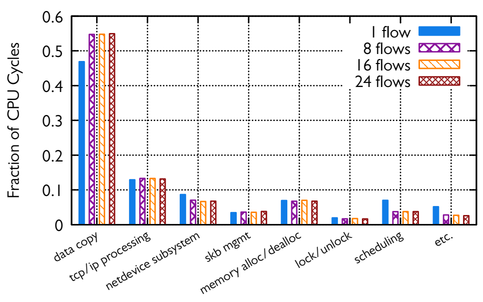
We now create additional contention at the receiver core using an incast traffic pattern, varying number of flows from 1 to 24 (each using a unique core at the sender). Compared to previous scenarios, this scenario induces higher contention for (1) CPU resources such as L3 cache and (2) CPU scheduling among application threads. We discuss how these changes affect the network processing overheads.

Per-byte data copy overhead increases with increasing flows per-core. Fig. 2.6(a) shows that throughput-per-core decreases with increase in number of flows, observing as much as ∼19% drop with 8 flows when compare to the single-flow case. Fig. 2.6(b) shows that the CPU breakdown does not change significantly with increasing number of flows, implying that there is no evident shift in CPU overheads. Fig. 2.6(c) provides some intuition for the root cause of the throughput-per-core degradation. As number of flows per core increases at the receiver side, applications for different flows compete for the same L3 cache space resulting in increased cache miss rate (the miss rate increases from 48% to 78%, as the number of flows goes from 1 to 8.). Among other things, this leads to increased per-byte data copy overhead and reduced throughput-per-core. As shown in Fig. 2.6(c), the increase in L3 cache miss rate with increasing flows correlates well with degradation in throughput-per-core.

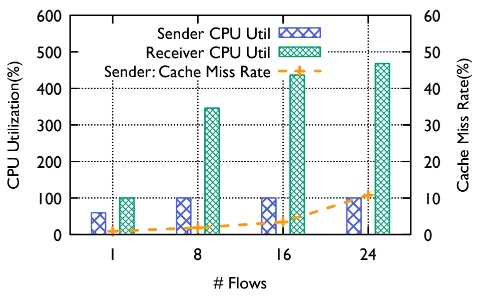
Sender-driven nature of TCP precludes receiver-side scheduling. Higher cache contention observed above is the result of multiple active flows on the same core. While senders could potentially reduce such contention using careful flow scheduling, the issue at the receiver side is fundamental: the sender-



(a) Throughput-per-core (Gbps)



(b) Sender CPU breakdown



(c) CPU utilization (%)

Figure 2.7: Linux network stack performance for outcast traffic pattern. (a) Each column shows throughput-per-*sender*-core achieved for different number of flows, that is the maximum throughput sustainable using a single sender core (we ignore receiver core utilization here). Throughput-per-sender-core increases from 1 to 8 flows, and then decreases as the number of flows increases. **(b)** With all optimizations enabled, as the number of flows increases from 1 to 8, data copy overhead increases but does not change much when the number of flows is increased further. Refer to [49] for receiver-side CPU breakdown. **(c)** For 1 flow, sender-side CPU is underutilised. Sender-side cache miss rate increases slightly as the number of flows increases from 8 to 24, increasing the per-byte data copy overhead, and there is a corresponsing decrease in throughput-per-core. See §2.3.4 for description.

driven nature of the TCP protocol precludes the receiver to control the number of active flows per core, resulting in unavoidable CPU inefficiency. We believe receiver-driven protocols [47, 48] can provide such control to the receiver, thus enabling CPU-efficient transport designs.

2.3.4 Increasing Sender Contention via Outcast

All our experiments so far result in receiver being the bottleneck. To evaluate sender-side processing pipeline, we now use an outcast scenario where a single sender core transmits an increasing number of flows (1 to 24), each to a unique receiver core. To understand the efficiency of sender-side processing pipeline, this subsection focuses on throughput-per-*sender*-core: that is, the maximum throughput achievable by a single sender core.

Sender-side processing pipeline can achieve up to 89**Gbps per core.** Fig. 2.7(a) shows that, with increase in number of flows from 1 to 8, throughput-per-sender-core increases significantly enabling total throughput as high as ~89Gbps; in particular, throughput-per-sender-core is 2.1× when compared to throughput-per-receiver-core in the incast scenario (§2.3.3). This demonstrates that, in today's Linux network stack, sender-side processing pipeline is much more CPU-efficient when compared to receiver-side processing pipeline. We briefly discuss some insights below.

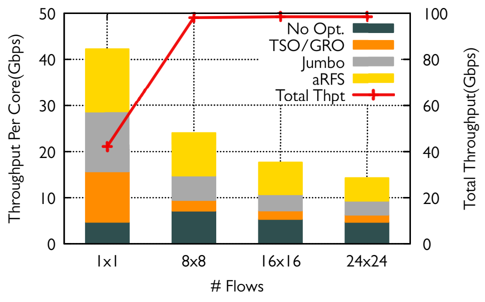
The first insight is related to the efficiency of TSO. As shown in Fig. 2.7(a), TSO in the outcast scenario contributes more to throughput-per-core improvements, when compared to GRO in the incast scenario (§2.3.3). This is due to two reasons. First, TSO is a hardware offload mechanism supported by the NIC; thus, unlike GRO which is software-based, there are no CPU overheads associated with TSO processing. Second, unlike GRO, the effectiveness of TSO does not degrade noticeably with increasing number of flows since data from applications is always put into 64KB size `skbs` independent of the number of flows. Note that Jumbo frames do not help over TSO that much compared to the previous cases as segmentation is now performed in the NIC.

Second, aRFS continues to provide significant benefits, contributing as much as ~46% of the total throughput-per-sender-core. This is because, as discussed earlier, L3 cache at the sender is always warm: while cache miss rate increases slightly with larger number of flows, the absolute number remains low (~11% even with 24 flows); furthermore, outcast scenario ensures that not too many flows compete for the same L3 cache at the receiver (due to receiver cores distributed across multiple NUMA nodes). Fig. 2.7(b) shows that data copy continues to be the dominant CPU consumer, even when sender is the bottleneck.

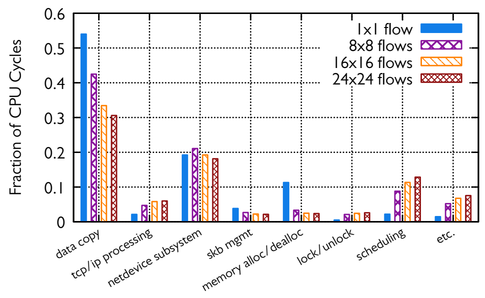
2.3.5 Maximizing Contention with All-to-All

We now evaluate Linux network stack performance for all-to-all traffic patterns, where each of x sender cores transmit a flow to each of the x receiver cores, for x varying from 1 to 24. In this scenario, we were unable to explicitly map IRQs to specific cores because, for the largest number of flows (576), the number of flow steering entries requires is larger than what can be installed on our NIC. Nevertheless, even without explicit mapping, we observed reasonably deterministic results for this scenario since the randomness across a large number of flows averages out.

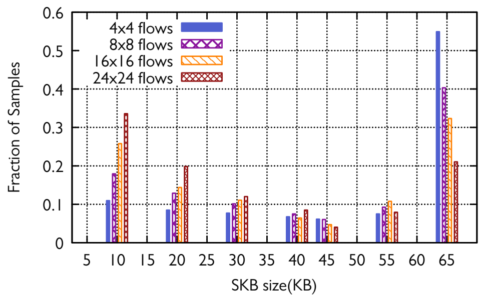
Fig. 2.8(a) shows that throughput-per-core reduces by ~67% going from 1×1 to 24×24 flows, due to reduced effectiveness of all optimizations. The benefits of aRFS drop by ~64%, almost the same as observed in the one-to-one scenario (§2.3.2). This is unsurprising, given the lack of cache locality for cores in non-NIC-local NUMA nodes, and given that cache miss rate is already abysmal (as



(a) Throughput-per-core (Gbps)



(b) Receiver CPU breakdown



(c) skb size distribution

Figure 2.8: Linux network stack performance for all-to-all traffic pattern. (a) Each column shows throughput-per-core achieved for different number of flows. With 8×8 flows, the network is fully saturated. Throughput-per-core decreases as the number of flows increases. **(b)** With all optimizations enabled, as the number of flows increase, the fraction of CPU cycles spent in data copy decreases. On the receiver-side, network saturation leads to lower memory management overhead (due to better page recycling) and higher scheduling overhead (due to frequent idling and greater number of threads per core.). TCP/IP processing overhead increases due to smaller skb sizes. The overall receiver-side CPU utilizations for x= $1 \times 1, 8 \times 8, 16 \times 16$ and 24×24 are $1, 4.07, 5.56$ and 6.98 cores, respectively. See [49] for sender-side CPU breakdown. **(c)** The fraction of 64KB skbs after GRO decreases as the number of flows increases because the larger number of flows prevent effective aggregation of received packets. See §2.3.5 for description.

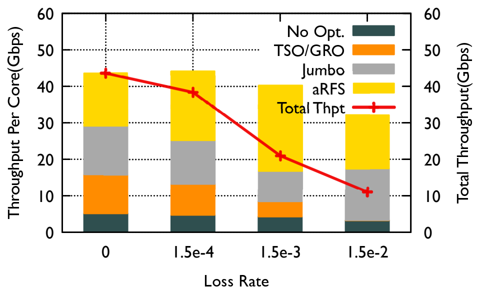
discussed in §2.3.2). Increasing the number of flows per core on top of this does not make things worse in terms of cache miss rate.

Per-flow batching opportunities reduce due to large number of flows. Similar to the one-to-one case, the network link becomes the bottleneck in this scenario, resulting in fair-sharing of bandwidth among flows. Since there are a large number of flows (*e.g.*, 24×24 with 24 cores), each flow achieves very small throughput (or alternatively, the number of packets received for any flow in a given time window is very small). This results in reduced effectiveness of optimizations like GRO (that operate on a per-flow basis) since they do not have enough packets in each flow to aggregate. As a result, upper layers receive a larger number of smaller `skb`s, increasing packet processing overheads.

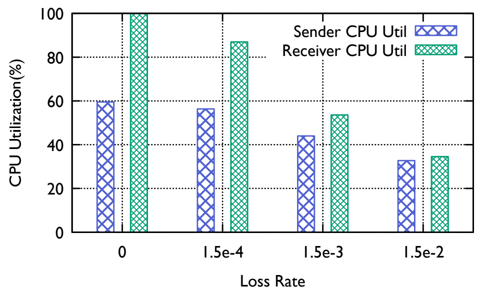
Fig. 2.8(c) shows the distribution of `skb` sizes (post-GRO) for varying number of flows. We see that as the number of flows increase, the average `skb` size reduces, leading to our argument above about the reduced effectiveness of GRO. We note that the above phenomenon is not unique to the all-to-all scenario: the number of flows sharing a bottleneck resource also increase in the incast and one-to-one scenarios. Indeed, this effect would also be present in those scenarios, however the total number of flows in those cases is not large enough to make these effects noticeable (max of 24 flows in incast and one-to-one versus 24×24 flows in all-to-all).

2.3.6 Impact of In-network Congestion

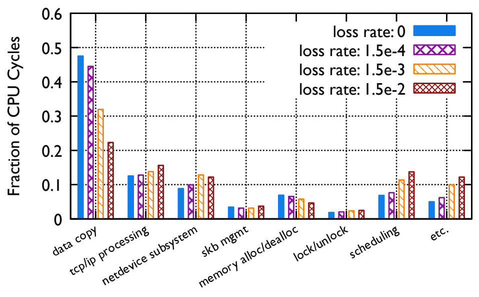
In-network congestion may lead to packet drops at switches, which in turn impacts both the sender and receiver side packet processing. In this subsection,



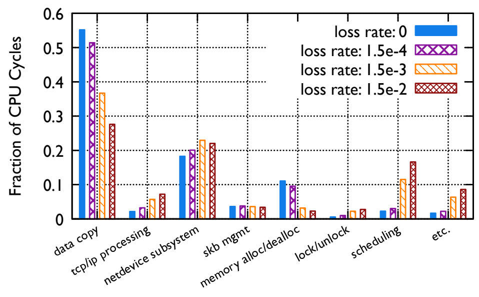
(a) Throughput-per-core (Gbps)



(b) CPU Utilization



(c) Sender CPU breakdown



(d) Receiver CPU breakdown

Figure 2.9: Linux network stack performance for the case of a single flow, with varying packet drop rates. (a) Each column shows throughput-per-core achieved for a specific packet drop rate. Throughput-per-core decreases as the packet drop rate increases. **(b)** As the packet drop rate increases, the gap between sender and receiver CPU utilisation decreases because the sender spends more cycles for retransmissions. **(c, d)** With all optimizations enabled, as the packet drop rate increases, the overhead of TCP/IP processing and netdevice subsystem increases. See §2.3.6 for description.

we study the impact of such packet drops on CPU efficiency. To this end, we add a network switch between the two servers, and program the switch to drop packets randomly. We increase the loss rate from 0 to 0.015 in the single flow scenario from §2.3.1, and observe the effect on throughput and CPU utilization at both sender and receiver.

Impact on throughput-per-core is minimal. As shown in Fig. 2.9(a) the throughput-per-core decreases by ~24% as the drop rate is increased from 0 to

0.015. Fig. 2.9(b) shows that the receiver-side CPU utilization decreases with increasing loss rate. As a result, the total throughput becomes lower than throughput-per-core, and the gap between the two increases. Interestingly, the throughput-per-core slightly increases when the loss rate goes from 0 to 0.00015. We observe that the corresponding receiver-side cache miss rate is reduced from 48% to 37%. This is because packet loss essentially reduces TCP sending rate, thus resulting in better cache hit rates at the receiver-side.

Figs. 2.9(c) and 2.9(d) show CPU profiling breakdowns for different loss rates. With increasing loss rate, at both sender and receiver, we see that the fraction of CPU cycles spent in TCP, netdevice subsystem, and other (etc.) processing increases, hence leading to fewer available cycles for data copy.

The minimal impact is due to increased ACK processing. Upon detailed CPU profiling, we found increased ACK processing and packet retransmissions to be the main causes for increased overheads. In particular:

- At the receiver, the fraction of CPU cycles spent in generating and sending ACKs increases by 4.87× (1.52% → 7.4%) as the loss rate goes from 0 to 0.015. This is because, when a packet is dropped, the receiver gets out-of-order TCP segments, and ends up sending duplicate ACKs to the sender. This contributes to an increase in both TCP and netdevice subsystem overheads.
- At the sender, the fraction of CPU cycles spent in processing ACKs increases by 1.45× (5.79% → 8.41%) as the loss rate goes from 0 to 0.015. This is because the sender has to process additional duplicate ACKs. Further, the fraction of CPU spent in packet retransmission operations increases by 1.34%. Both of these contribute to an increase in TCP and netdevice subsystem overheads,

while the former contributes to increased IRQ handling (which is classified under "etc." in our taxonomy).

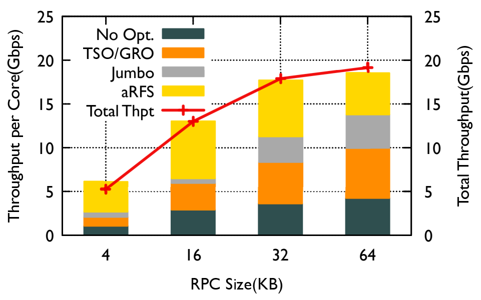
Sender observes higher impact of packet drops. Fig. 2.9(b) shows the CPU utilization at the sender and the receiver. As drop rates increase, the gap between sender and receiver utilization decreases, indicating that the increase in CPU overheads is higher at the sender side. This is due to the fact that, upon a packet drop, the sender is responsible for doing the bulk of the heavy lifting in terms of congestion control and retransmission of the lost packet.

2.3.7 Impact of Flow Sizes

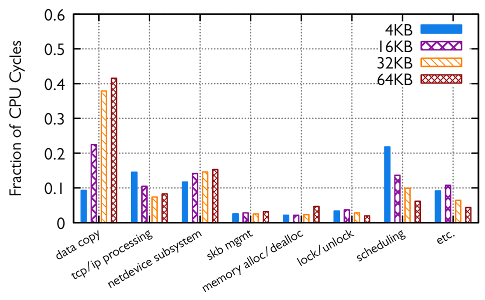
We now study the impact of flow sizes on the Linux network stack performance. We start with the case of short flows: a ping-pong style RPC workload, with message sizes for both request/response being equal, and varying from 4KB to 64KB. Since a single short flow is unable to bottleneck CPU at either the sender or the receiver, we consider the incast scenario—16 applications on the sender send ping-pong RPCs to a single application on the receiver (the latter becoming the bottleneck). Following the common deployment scenario, each application uses a long-running TCP connection.

We also evaluate the impact of workloads that comprise of a mix of both long and short flows. For this scenario, we use a single core at both the sender and the receiver. We run a single long flow, and mix it with a variable number of short flows. We set the RPC size of short flows to 4KB.

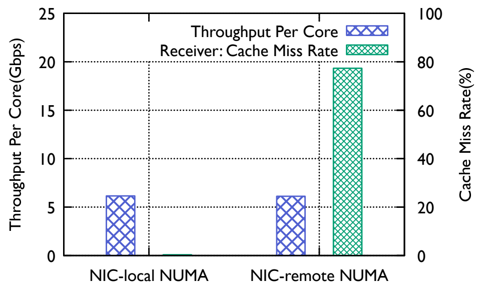
DCA does not help much when workloads comprise of extremely short flows.



(a) Throughput-per-core (Gbps)



(b) Server CPU breakdown



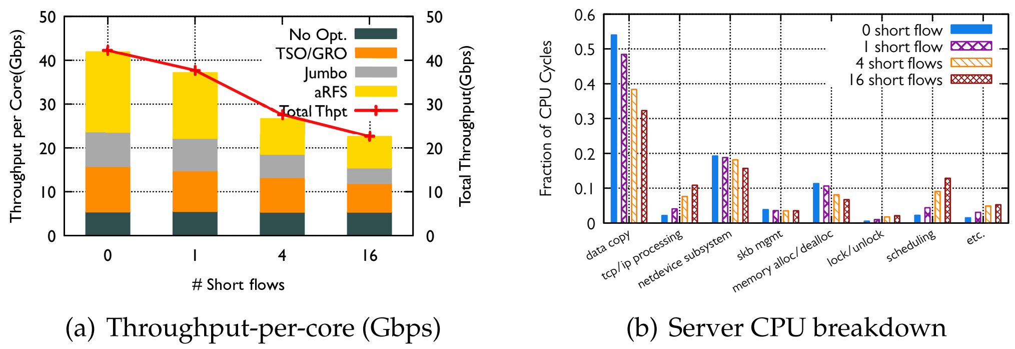
(c) NIC-remote NUMA effect (4KB)

Figure 2.10: Linux network stack performance for short flow, 16:1 incast traffic pattern, with varying RPC sizes. (a) Each column shows throughput-per-core achieved for a specific RPC size. Throughput-per-core increases with increasing RPC size. For small RPCs, optimizations like GRO do not provide much benefit due to fewer aggregation opportunities. **(b)** With all optimizations enabled, data copy quickly becomes the bottleneck. The server-side CPU was completely utilized for all scenarios. See [49] for client-side CPU breakdown. **(c)** Unlike long flows, no significant throughput-per-core drop is observed even when application runs on NIC-remote NUMA node core at the server. See §2.3.7 for description.

Fig. 2.10(a) shows that, as expected, throughput-per-core increases with increase in flow sizes. We make several observations. First, as shown in Fig. 2.10(b), data copy is no longer the prominent consumer of CPU cycles for extremely small flows (*e.g.*, 4KB)—TCP/IP processing overhead is higher due to low GRO effectiveness (small flow sizes make it hard to batch `skbs`), and scheduling overhead is higher due to ping-pong nature of the workload causing applications to repeatedly block while waiting for data. Second, data copy not being the dominant consumer of CPU cycles for extremely short flows results in DCA not contributing to the overall performance as much as it did in the long-flow case: as shown in Fig. 2.10(c), while NIC-local NUMA nodes achieve significantly lower cache miss rates when compared to NIC-remote NUMA nodes, the difference in throughput-per-core is only marginal. Third, while DCA benefits reduce for extremely short flows, other cache locality benefits of aRFS still apply: for example, `skb` accesses during packet processing benefit from cache hits. However, these benefits are independent of the NUMA node on which the applications runs. The above three observations suggest interesting opportunities for orchestrating host resources between long and short flows: while executing on NIC-local NUMA nodes helps long flows significantly, short flows can be scheduled on NIC-remote NUMA nodes without any significant impact on performance; in addition, carefully scheduling the core across short flows sharing the core can lead to further improvements in throughput-per-core.

We note that all the observations above become relatively obsolete even with slight increase in flow sizes—with just 16KB RPCs, data copy becomes the dominant factor and with 64KB RPCs, the CPU breakdown becomes very similar to the case of long flows.

Mixing long and short flows considered harmful. Fig. 2.11(a) shows that, as expected, the overall throughput-per-core drops by ~43% as the number of short flows colocated with the long flow is increased from 0 to 16. More importantly, while throughput-per-core for a single long flow and 16 short flows is ~42Gbps (§2.3.1) and ~6.15Gbps in isolation (no mixing), it drops to ~20Gbps and ~2.6 Gbps, respectively when the two are mixed (48% and 42% reduction for long



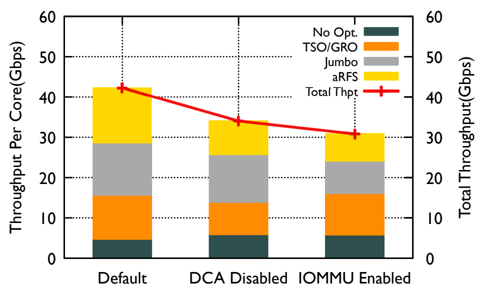
(a) Throughput-per-core (Gbps) (b) Server CPU breakdown

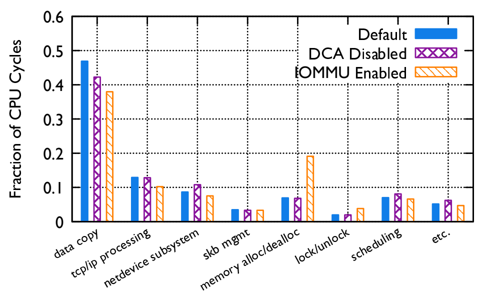
Figure 2.11: Linux network stack performance for workloads that mix long and short flows on a single core. **(a)** Each column shows throughput-per-core achieved for different number of short flows colocated with a long flow. Throughput-per-core decreases with increasing number of short flows. **(b)** Even with 16 flows colocated with a long flows, data copy overheads dominate, but TCP/IP processing and scheduling overheads start to consume significant CPU cycles. The server-side CPU was completely utilized for all scenarios.; refer to [49] for client-side CPU breakdown. See §2.3.7 for description.

and short flows). This suggests that CPU-efficient network stacks should avoid mixing long and short flows on the same core.

2.3.8 Impact of DCA

All our experiments so far were run with DCA enabled (as is the case by default on Intel Xeon processors). To understand the benefits of DCA, we now rerun the single flow scenario from §2.3.1, but with DCA disabled. Fig. 2.12(a) shows the throughput-per-core without DCA relative to the scenario with DCA enabled (Default), as each of the optimizations are incrementally enabled. Unsurprisingly, with all optimizations enabled, we observe a 19% degradation in throughput-per-core when DCA is disabled. In particular, we see a ~50% reduction in the effectiveness of aRFS; this is expected since disabling DCA reduces


(a) Throughput-per-core (Gbps)


(b) Sender CPU breakdown

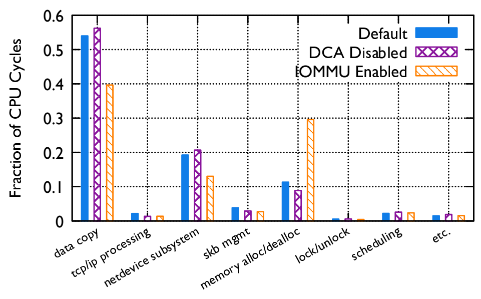

(c) Receiver CPU breakdown

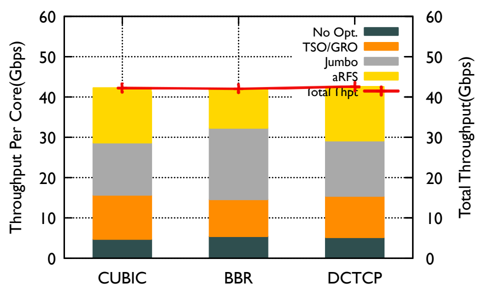
Figure 2.12: Impact of DCA and IOMMU on Linux network stack performance. (a) Each column shows throughput-per-core achieved for different DCA and IOMMU configurations: Default has DCA enabled and IOMMU disabled. Either of disabling DCA or enabling IOMMU leads to decrease in throughput-per-core. **(b, c)** Disabling DCA does not cause a significant shift in CPU breakdown. Enabling IOMMU causes a significant increase in memory management overheads at both the sender and the recever. See §2.3.8 and §2.3.9 for description.

the data copy benefits of NIC DMAing the data directly into the L3 cache. The other benefits of aRFS (§2.3.1) still apply. Without DCA, the receiver-side remains the bottleneck, and we do not observe any significant shift in the CPU breakdowns at sender and receiver (Figs. 2.12(b) and 2.12(c)).

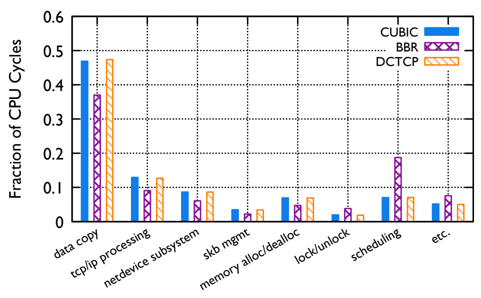
2.3.9 Impact of IOMMU

IOMMU (IO Memory Management Unit) is used in virtualized environments to efficiently virtualize fast IO devices. Even for non-virtualized environments, they are useful for memory protection. With IOMMU, devices specify virtual addresses in DMA requests which the IOMMU subsequently translates into physical addresses while implementing memory protection checks. By default, the IOMMU is disabled in our setup. In this subsection, we study the impact of IOMMU on Linux network stack performance for the single flow scenario (§2.3.1).

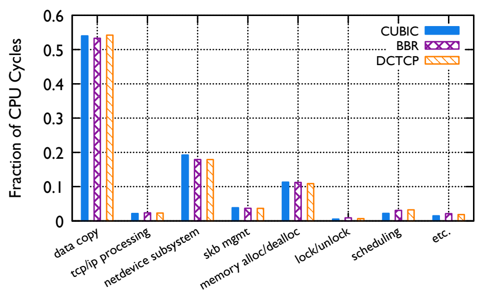
The key take-away from this subsection is that IOMMU, due to increased memory management overheads, results in significant degradation in network stack performance. As seen in Fig. 2.12(a), enabling IOMMU reduces throughput-per-core by 26% (compared to Default). Figs. 2.12(b) and 2.12(c) show the core reason for this degradation: memory alloc/dealloc becoming more prominent in CPU consumption at both sender and receiver (now consuming 30% of CPU cycles at the receiver). This is because of two additional per-page operations required by IOMMU: (1) when the NIC driver allocates new pages for DMA, it has to also insert these pages into the device's pagetable (domain) on the IOMMU; (2) once DMA is done, the driver has to unmap those pages. These two additional per-page operations result in increased overheads.



(a) Throughput-per-core (Gbps)



(b) Sender-side CPU breakdown



(c) Receiver CPU breakdown (IOMMU enabled)

Figure 2.13: Impact of congestion control protocols on Linux network stack performance. (a) Each column shows throughput-per-core achieved for different congestion control protocols. There is no significant change in throughput-per-core across protocols. **(b, c)** BBR causes a higher scheduling overhead on the sender-side. On the receiver-side, the CPU utilization breakdowns are largely similar. For all cases, receiver-side core is fully utilized for all protocols. See §2.3.10 for description.

2.3.10 Impact of Congestion control protocols

Our experiments so far use TCP CUBIC, the default congestion control algorithm in Linux. We now study the impact of congestion control algorithms on network stack performance using two other popular algorithms implemented in Linux, BBR [57] and DCTCP [42], again for the single flow scenario (§2.3.1). Fig. 2.13(a) shows that choice of congestion control algorithm has minimal impact on throughput-per-core. This is because, as discussed earlier, receiver-side is the core throughput bottleneck in high-speed networks; all these algorithms being “sender-driven”, have minimal difference in the receiver-side logic. Indeed, the receiver-side CPU breakdowns are largely the same for all protocols (Fig. 2.13(c)). BBR has relatively higher scheduling overheads on the sender-side (Fig. 2.13(b)); this is because BBR uses pacing for rate control (with qdisc) [58], and repeated thread wakeups when packets are released by the pacer result in increased scheduling overhead.

2.4 Summary

In this chapter, we have demonstrated that recent adoption of high-bandwidth links in datacenter networks, coupled with relatively stagnant technology trends for other host resources (*e.g.*, core speeds and count, cache sizes, etc.), mark a fundamental shift in host network stack bottlenecks. Using measurements and insights for Linux network stack performance for 100Gbps links, our study highlights several avenues for future research in designing CPU-efficient host network stacks, which will be presented in Chapter 4. These are exciting times for networked systems research—with emergence of Terabit Ethernet, the bottlenecks outlined in this study are going to become even more prominent, and it is only by bringing together operating systems, computer networking and computer architecture communities that we will be able to design host network stacks that overcome these bottlenecks. We hope this chapter will enable a deeper understanding of today’s host network stacks, and will guide the design of not just future Linux kernel network stack, but also future network and host hardware.

CHAPTER 3

REARCHITECTING HOST NETWORK STACKS FOR TERABIT ETHERNET

Dedicated, tightly integrated, and static packet processing pipelines in today's most widely deployed network stacks preclude them from fully exploiting capabilities of modern hardware.

In this chapter, we present NetChannel, a disaggregated network stack architecture for μs-scale applications running atop Terabit Ethernet. NetChannel 's disaggregated architecture enables independent scaling and scheduling of resources allocated to each layer in the packet processing pipeline. Using an end-to-end NetChannel realization within the Linux network stack, we demonstrate that NetChannel enables new operating points—(1) enabling a single application thread to saturate multi-hundred gigabit access link bandwidth; (2) enabling near-linear scalability for small message processing with number of cores, independent of number of application threads; and, (3) enabling isolation of latency-sensitive applications, allowing them to maintain μs-scale tail latency even when competing with throughput-bound applications operating at near-line rate.

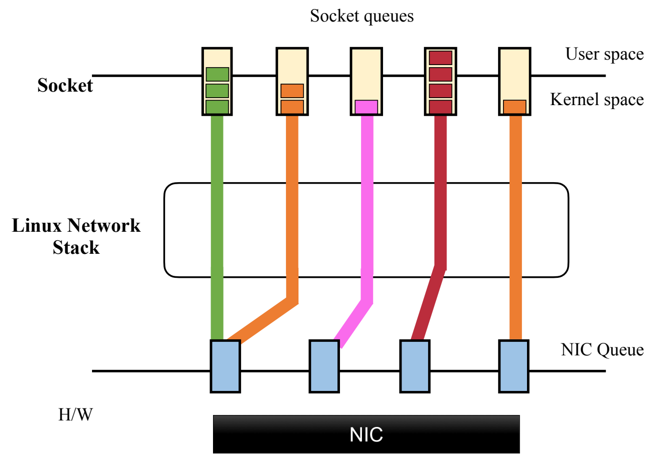
3.1 Overview

In discussions about future host network stacks, there is widespread agreement that, despite its great success, today's Linux network stack is seriously deficient along one or more dimensions. Some of the most frequently cited flaws include its inefficient packet processing pipeline [12, 17, 20, 26, 35], its inability to isolate latency-sensitive and throughput-bound applications [11, 59, 60], its rigid and complex implementation [30], its inefficient transport protocols [47, 61, 62], to name a few. These critiques have led to many interesting (and exciting!) debates on various design aspects of the Linux network stack: interface (*e.g.*, streaming versus RPC [59, 63–65]), semantics (*e.g.*, synchronous versus asynchronous I/O [65–67]), and placement (*e.g.*, in-kernel versus userspace versus hardware [20, 30]).

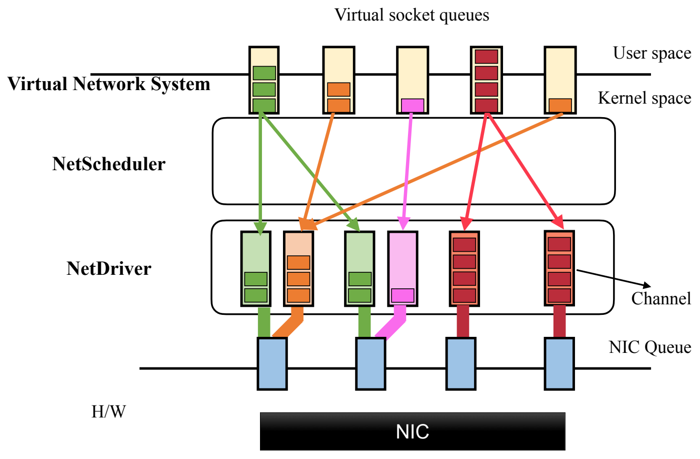
This chapter demonstrates that many deficiencies of the Linux network stack are *not* rooted in its interface, semantics and/or placement, but rather in its core architecture[1]. In particular, since the very first incarnation, the Linux network stack has offered applications the same "pipe" abstraction designed around essentially the same rigid architecture:

- **Dedicated pipes:** each application and/or thread submits data to one end of a dedicated pipe (sender-side socket) and the network stack attempts to deliver the data to the other end of that dedicated pipe (receiver-side socket);
- **Tightly-integrated packet processing pipeline:** each pipe is assigned its own socket, has its own independent transport layer operations (congestion control, flow control, etc.), and is operated upon by the network subsystem completely independently of other coexisting pipes;
- **Static pipes:** the entire packet processing pipeline (buffers, protocol processing, host resource provisioning, etc.) is determined at the time of pipe cre-

[1]One exception is per-core performance, which indeed depends on its interface, semantics and placement. This chapter is not about per-core performance of network stacks—our architecture is agnostic to the interface, semantics, and placement of network stacks.



(a) Linux network stack



(b) NetChannel design

Figure 3.1: NetChannel architecture overview. **(a)** The Linux network stack architecture uses dedicated, tightly-integrated, and static packet processing pipelines. **(b)** NetChannel disaggregates the packet processing pipeline into three loosely-coupled layers: applications interact with the network stack using a Virtual Network System (VNS) layer that enables data movement between application buffers and kernel buffers while maintaining correctness of interface semantics; NetDriver abstracts away the network and remote servers as a multi-queue device using a `channel` abstraction, and performs dynamic resource scheduling for individual `channels`; NetScheduler performs fine-grained multiplexing and demultiplexing (as well as scheduling) of data between VNS buffers and individual NetDriver `channels`. More discussion in §3.3.

ation, and remains unchanged during the pipe lifetime, again, independent of other pipes and dynamic resources availability at the host.

Such dedicated, tightly-integrated and static pipelines were well-suited for the Internet and early-generation datacenter networks—since performance bottlenecks were primarily in the network core, careful allocation of host resources (compute, caches, NIC queues, etc.) among coexisting pipes was unnecessary. However, rapid increase in link bandwidths, coupled with relatively stagnant technology trends for other host resources, has now pushed bottlenecks to hosts [17, 20, 26, 30, 35, 44, 62]. For this new regime, our measurements in §3.2 show that dedicated, tightly-integrated and static pipelines are now limiting today's network stacks from fully exploiting capabilities of modern hardware that supports μs-scale latency and hundred(s) of gigabits of bandwidth. Experimenting with new ideas has also become more challenging: performance patches have made the tightly-integrated pipelines so firmly entrenched within the stack that it is frustratingly hard, if not impossible, to incorporate new protocols and mechanisms. Unsurprisingly, existing network stacks are already at the brink of a breakdown and the emergence of Terabit Ethernet will inevitably require rearchitecting the network stack. Laying the intellectual foundation for such a rearchitecture is the goal of this chapter.

The NetChannel architecture. NetChannel disaggregates the tightly-integrated packet processing pipeline in today's network stack into three loosely-coupled layers (Fig. 3.1)[2].

[2]In the hindsight, NetChannel is remarkably similar to the Linux storage stack architecture [68, 69]. This similarity is not coincidental—for storage workloads, bottlenecks have always been at the host, and the "right" architecture has evolved over years to both perform fine-grained resource allocation across applications, and to make it easy to incorporate new storage technologies.

Applications interact with a Virtual Network System (VNS) layer that offers standardized interfaces, *e.g.*, system calls for streaming and RPC traffic. Internally, VNS enables data transfer between application buffers and kernel buffers, while ensuring correctness for the interface semantics (*e.g.*, in-order delivery for the streaming interface). The core of NetChannel is a NetDriver layer that abstracts away the network and remote servers as a multi-queue device using a `channel` abstraction. In particular, the NetDriver layer decouples packet processing from individual application buffers and cores: data read/written by an application on one core can be mapped to one or more `channels` *without breaking application semantics*. Each `channel` implements protocol-specific functionalities (congestion and flow control, for example) independently, can be dynamically mapped to one of the underlying hardware queues, and the number of `channels` between any pair of servers can be scaled independent of number of applications running on these servers and the number of cores used by individual applications. Between the VNS and NetDriver layers is a NetScheduler layer that performs fine-grained multiplexing and demultiplexing (as well as scheduling) of data from individual cores/applications to individual `channels` using information about individual core utilization, application buffer occupancy and `channel` buffer occupancy.

NetChannel benefits. The primary benefit of NetChannel is to enable new operating points for existing network stacks without any modification in existing protocol implementations (TCP, DCTCP, BBR, etc.). These new operating points are a direct result of NetChannel's disaggregated architecture: it not only allows independent scaling of each layer (that is, resources allocated to each layer), but also flexible multiplexing and demultiplexing of data to multiple `channels`. We provide three examples. First, for short messages where throughput is bottlenecked by network layer processing overheads [26, 35], NetChannel allows unmodified (even single-threaded) applications to scale throughput near-linearly with number of cores by dynamically scaling cores dedicated to network layer processing. Second, in the extreme case of a single application thread, NetChannel can saturate multi-hundred gigabit links by transparently scaling number of cores for packet processing on an on-demand basis; in contrast, the Linux network stack forces application designers to write multi-threaded code to achieve throughput higher than tens of gigabits per second [17]. As a third new operating point, we show that fine-grained multiplexing and demultiplexing of packets between individual cores/applications and individual `channels` enabled by NetChannel, combined with a simple NetScheduler, allows isolation of latency-sensitive applications from throughput-bound applications: NetChannel enables latency-sensitive applications to achieve μs-scale tail latency (as much as 17.5× better than the Linux network stack), while allowing bandwidth-intensive applications to use the remaining bandwidth near-perfectly.

NetChannel also has several secondary benefits that relate to the extensibility of network stacks. For instance, NetChannel alleviates the painful process of applications developers manually tuning their code for networking performance (*e.g.*, number of threads, connections, sockets, etc.) in increasingly common case of multi-tenant deployments[3]. NetChannel also simplifies experimentation with new designs (protocols and/or schedulers) without breaking legacy hosts—implementation of a new transport protocol (*e.g.*, dcPIM [61],

[3]Libraries and/or schedulers outside the network stack (*e.g.*, gRPC) may be able to offer this benefit to some extent, but in many multi-tenant deployments, they may not have the global visibility of all applications, system-level metrics like CPU utilization, and in particular, network-layer metrics like congestion information to effectively offer such a benefit. Nevertheless, our point here is not that similar benefits cannot be achieved using other systems, but rather that one of the most widely used network stacks, Linux, is limited by its architecture in offering such a benefit.

pHost [47] or Homa [48]) in NetChannel is equivalent to writing a new "device driver" that realizes only transport layer functionalities without worrying about functionalities in other layers of the stack like data copy, isolation between latency-sensitive and throughput-bound applications, CPU scheduling, load balancing, etc. Thus, similar to storage stacks (that have simplified evolution of new hardware and protocols via device drivers, and have simplified writing applications with different performance objectives via multiple coexisting block layer schedulers, etc.), NetChannel would hopefully lead to a broader and ever-evolving ecosystem of network stack designs. To that end, we have open-sourced NetChannel for our community; the implementation of NetChannel is available at `https://github.com/Terabit-Ethernet/NetChannel`.

What this chapter is *not* about. Going back to our starting point, there have been a lot of interesting and exciting recent debates on various design aspects of network stacks including their interface, semantics and placement. These are important discussions, but are tangential to our goals.

First, NetChannel architecture is complementary to recent efforts in improving per-core (or, per-connection) performance of the Linux kernel network stack (*e.g.*, zero-copy mechanisms [23, 24], and the new `io_uring` interface [67])—we will demonstrate, in §3.5, that applications using the `io_uring` interface can also benefit from the NetChannel architecture. Given that single-core CPU speeds have long been saturated, simple calculations show that saturating multi-hundred gigabit access link bandwidths would necessitate using multiple cores. NetChannel architecture thus focuses on enabling new design points to enable applications to share and exploit all host resources (*e.g.*, multiple cores, NIC queues, multi-hundred gigabit bandwidth, etc.).

Second, NetChannel architecture is independent of where the network stack is placed—in-kernel, userspace or hardware; one could very well implement NetChannel design on top of a microkernel-style userspace stack [15, 16, 20]. We choose the Linux kernel simply because of its maturity, stability, and widespread deployment; we leave it to future work to explore integration of NetChannel with userspace and hardware network stacks.

3.2 Motivation

In this section, we demonstrate that the dedicated, tightly-integrated, and static packet processing pipelines in today's host network stacks lead to deficiencies along multiple dimensions. We perform measurements for the Linux network stack, with various transport designs including TCP and its multi-path extension MPTCP [70], various system interfaces (standard `read/write` interface and `io_uring`), various packet processing optimization techniques (*e.g.*, packet coalescing and packet steering), and different isolation mechanisms. We start by describing our measurement setup (§3.2.1), and then highlight several limitations of today's Linux network stack (§3.2.2) using these measurements. Our key findings are:

- *Static and dedicated* packet processing pipelines preclude applications from fully utilizing host CPU resources. Even with all optimizations enabled, a single TCP long flow fails to saturate a 100Gbps link (achieving a maximum of ~60Gbps) even when ample CPU cores are available. We find that, at ~60Gbps, one of the cores at the receiver side becomes the bottleneck (specifically, the core where the application runs), and today's network stacks provide no way to dynamically scale the compute resources allocated to the packet processing pipeline during runtime (even if there are other free CPU cores available). Multipath extensions are no different—while MPTCP enables a single TCP connection to utilize multiple network paths, processing at the host is still bound to a single CPU core leaving the bottleneck unchanged.

- *Static and dedicated* packet processing pipelines also preclude Linux from dynamically scaling the number of connections when network layer processing becomes the bottleneck for short messages (*e.g.*, RPCs). MPTCP also fails to achieve scalability for network layer processing of short messages due to the same reason as above.
- *Tightly-integrated* nature of packet processing pipelines lead to poor performance isolation when Latency-sensitive (L-apps) and Throughput-bound (T-apps) applications are co-located. When L-apps and T-apps share a core, today's network stacks provide no mechanism to steer L-app packet processing and T-app packet processing to different cores—this results in high tail latency for L-apps due to head-of-line blocking; we observe as much as 37× increase in L-apps tail latency during such contention.

3.2.1 Measurement Setup

We set up a testbed using two servers directly connected with a 100Gbps link so as to push bottlenecks to the host network stack. Each server has 4 NUMA nodes with 8 CPU cores per-NUMA node. Direct Cache Access (Intel's Data Direct I/O (DDIO) [44]) is enabled and configured to use the maximum possible number of L3 cache ways. We use Linux kernel v5.6 for TCP and MPTCP kernel

Mechanism	Description
TSO (Tx)	TCP Segmentation Offload. Offloads the segmentation of "big" packets (up to 64KB) into frames to the NIC.
GRO (Rx)	Generic Receive Offload. Aggregates MTU-sized frames into a "big" packet (up to 64KB) before passing them to the TCP/IP layer.
Jumbo Frames (Tx/Rx)	Using larger MTU size (9000B)
aRFS (Rx)	accelerated Receive Flow Steering. Steers received frames to the core that the application is running on.
DCA (Rx)	Direct Cache Access. Allows NICs to DMA receiving frames directly to the processor's L3 cache.

Table 3.1: Packet processing optimization techniques used in many modern network stacks.

v0.95 for MPTCP[4]. To emulate multi-pathing of the MPTCP kernel over a single 100Gbps link, we increase the number of subflows manually.

To understand performance bottlenecks for long flows and short messages, we use a long-lived TCP connection for transmitting stream data and 4KB RPCs respectively, avoiding extra overheads of creating/destroying connections. When measuring the performance interference between T-apps and L-apps, for T-apps, we generate long flow traffic similar to Iperf [19], and for L-apps, we use a ping-pong style RPC workload (message size for both request/response is 4KB).

We measure total throughput of long flows and short messages; to better understand performance bottlenecks, we also perform CPU profiling and classify kernel functions into different categories as in prior work [17]. We also measure throughput-per-core of T-apps and P99.9 tail latency of L-apps for co-located

[4]The full implementation of MPTCP has been maintained separately from the upstream kernel using different version numbers. As of now, the most recent version of MPTCP kernel is based on Linux kernel 4.19 [70]. Efforts to push MPTCP into the upstream kernel are still in progress [71].

T-apps and L-apps.

3.2.2 Limitations of Existing Kernel Stack

We use three measurement scenarios to showcase limitations of today's Linux network stack.

(1) Static and dedicated pipeline ⇒ lack of scalability for long flows. We run an application that transmits a stream of data through a single TCP socket using standard read/write system calls. Fig. 3.2(a) shows the network stack is unable to saturate the 100Gbps link for this application, even after enabling various prevalent offload and aggregation-based optimizations like Jumbo Frames, TSO, GRO, aRFS (see Table 3.1), new upcoming interfaces like `io_uring`, and multiple subflows for MPTCP. Jumbo Frames and TSO/GRO help reduce the per-packet processing overheads since they allow the processing pipeline to operate on larger size packet buffers (or `skbs`). aRFS allows NICs to steer received frames to the application core; along with DCA, it generally improves throughput by performing data copy directly from the L3 cache when applications are running on the cores in the same NUMA node as the NIC. For a more detailed discussion of these optimizations and their impact, refer to [17].

To dig deeper, we perform CPU profiling as shown in Fig. 3.2(b); our results suggest that, despite DCA being enabled, the core bottleneck is data copy from kernel to userspace *at the receiver-side* consistent with observations in recent work [17]. Data copy is performed on the application core that executes the `recv()` system calls. Additionally, with aRFS enabled, interrupts (IRQs) are steered to the application core, and hence other network layer processing such as TCP/IP, Net-device subsystem (netdev), etc., also happens on the application core.

We find that by disabling aRFS and manually steering interrupts to a different core on the same NUMA node as the application, the network stack achieves slightly higher throughput of ~60Gbps (Fig. 3.2(a) second bar). Part of the processing (TCP/IP, netdev, etc.) is now offloaded to a different CPU core, hence freeing up more CPU cycles for data copy on the application core.

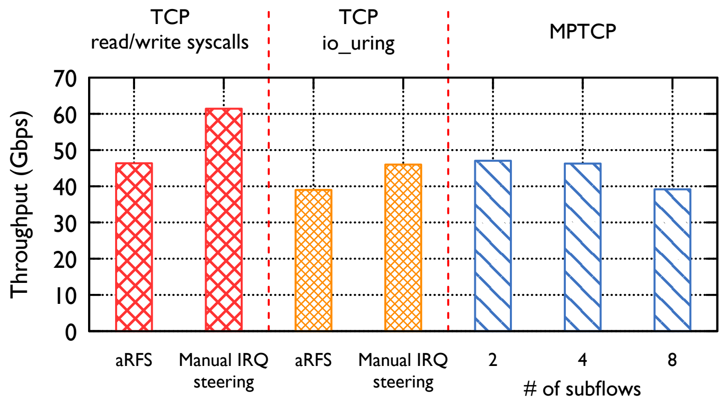
Even with Linux's recent `io_uring` [67] feature, with aRFS and with manual NUMA-local IRQ steering (bars 3 and 4 in Fig. 3.2(a) respectively), we find that the total throughput does not improve—in particular, data copy remains the dominant overhead (bars 3 and 4 in Fig. 3.2(b))[5]. In fact, we observe a slight degradation in total throughput with `io_uring` because it dispatches some of the socket receive calls to a separate kernel thread which contends with the application thread for the common socket lock.

With MPTCP, we observe that using aRFS gives the best possible throughput. Independent of the number of subflows, all the processing happens on the core where the application runs. The total throughput is reduced when the number of subflows increases due to the increase in the amount of network layer processing. Without aRFS, interrupts for subflows get mapped to arbitrary CPU cores potentially on different NUMA nodes resulting in poor throughput[6].

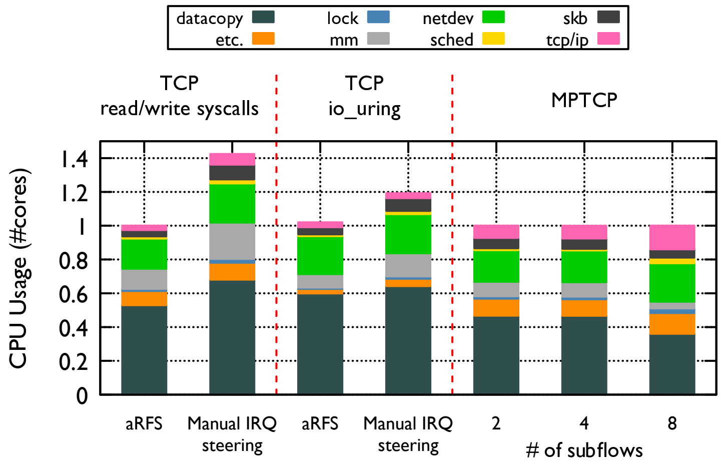
Overall, independent of the configuration used, the packet processing

[5]While `io_uring` enables "zero-copy" of the metadata associated with socket operations, payload data is copied as usual. While there is ongoing work on exploiting `io_uring` for zero-copy send [72], we are not aware of any work on zero-copy receive (which is usually the bottleneck [17]).

[6]We were not able to manually steer subflow IRQs to different cores in the same NUMA using 4-tuples, because the kernel determines the source ports of subflows at runtime.



(a) Total throughput (Gbps)



(b) Receiver-side CPU breakdown

Figure 3.2: Static pipeline of the Linux network stack (using both TCP and MPTCP) fails to saturate 100**Gbps access link bandwidths** since it is unable to scale compute resources allocated to packet processing pipelines. More discussion in §3.2.2.

pipeline of today's network stack is static (both data copy and network layer processing are bound to a single core). As a result, it is unable to dynamically scale resources allocated to a packet processing pipeline to utilize the full network link bandwidth, despite the availability of idle CPU cores. Even if data copy were to be eliminated (via zero-copy mechanisms [23,24]), simple calculations show that the network stack will not be able to saturate emerging multi-hundred gigabit links using a single core, as the packet processing pipeline is

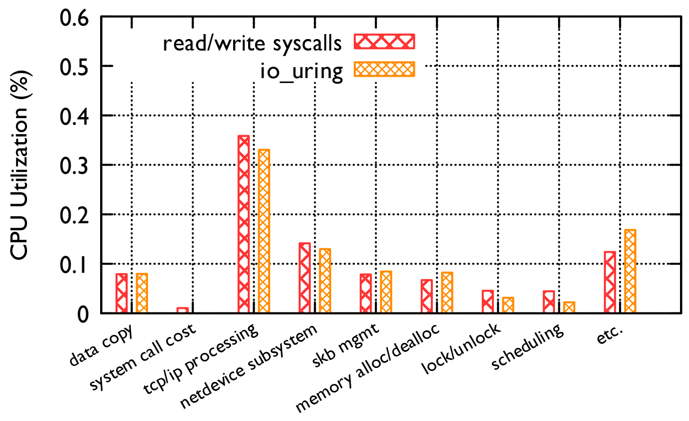


Figure 3.3: In today's Linux network stack, sender-side network layer processing (including protocol and netdevice subsystem processing) overheads are the bottleneck for short message processing. More discussion in §3.2.2.

still bound to one core. The requirement of multi-core processing is, thus, inevitable.

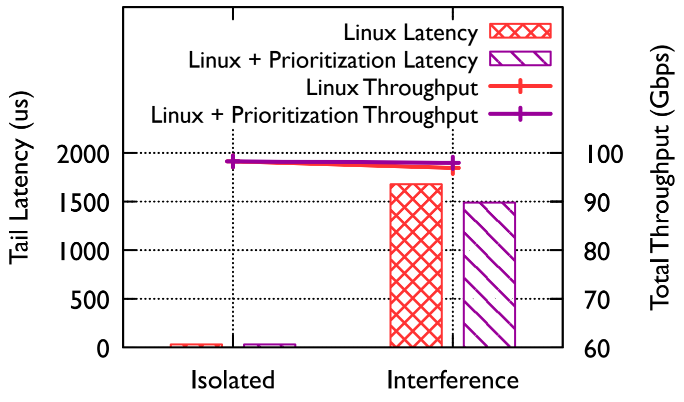
(2) Static and dedicated pipeline ⇒ lack of scalability for short message processing. We run a client application which sends 4KB RPC requests to a server. The sustained throughput using a single socket is roughly ~ 9.88Gbps. We find that the sender-side is the bottleneck. The sender-side CPU breakdown in Fig. 3.3, shows that TCP/IP processing and netdevice subsystem processing are the dominant overheads, accounting for almost a half of the total CPU cycles used. Once again, all of this processing is bound to a single CPU core and is unable to dynamically scale even if additional CPU cores are available—a result of the static nature of today's network stack.

We tried this experiment using `io_uring` as well, but observed no improvement in throughput (achieving a maximum of 8.5Gbps; there is a small degradation in throughput [73, 74])). This is unsurprising given that system call cost and context-switch overheads (included in scheduling)—the main overheads that `io_uring` is supposed to minimize—account for a very tiny fraction of CPU cycles in this scenario, as seen in Fig. 3.3. MPTCP cannot help in this case for two reasons. First, since sender-side network layer processing still happens on the application core, running multiple subflows on a single core does not help. Second, MPTCP cannot dynamically scale the number of subflows at runtime based on the CPU load.

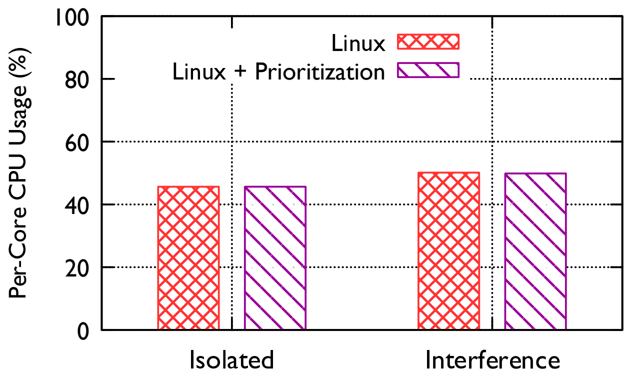
While the application could overcome the network layer processing bottleneck by sending data over multiple sockets from different threads, it is difficult for application developers to estimate how many sockets they will need, especially in multi-tenant and virtualized deployments [17]. Moreover, the application and userspace libraries linked to the application (e.g., gRPC) have limited information about congestion in the network and utilization of CPU cores, making it hard to make informed packet scheduling decisions across sockets. An ideal networking stack should dynamically and transparently allocate new connections on idle cores and multiplex packets to different connections so that the application can achieve higher throughput without manually managing the number of connections. Further, this should happen only when the throughput is limited by CPU, not by congestion control, to maintain protocol-side properties such as TCP-friendliness.

(3) Tightly-integrated pipeline ⇒ lack of performance isolation. To understand performance interference between L-apps and T-apps when they are co-located, we run 1 L-app and 8 T-apps on the same NUMA node (number of applications > number of cores). We do not pin the applications to specific cores, thus allowing the CPU scheduler to dynamically move applications across all of the cores within the NUMA node. We enable all optimizations including aRFS.

Fig. 3.4(a) (Linux) shows the tail latency (99.9th percentile) of the L-app and



(a) P99.9 latency (μs) and total throughput (Gbps)



(b) Per-Core CPU Usage

Figure 3.4: Tightly integrated pipeline of the Linux network stack fails to provide μs-scale isolation for L-apps when they are co-located with T-apps. L-apps can suffer from extremely high tail latency. More discussion in §3.2.2.

the overall throughput achieved, when the applications are run in isolation (Isolated) versus when they are co-located (Interference). In the latter case, the L-app experiences a 37× inflation in tail latency, relative to when it is run in isolation. This dramatic inflation in tail latency is due to the tight integration of network layer processing with the application cores. Since there are more applications than cores, it is inevitable that at certain points in time, the L-app will share a CPU core with one or more T-apps. When this happens the kernel runs the corresponding network layer processing for both applications on the same core, hence causing interference—the network layer processing for the L-app can get blocked behind network layer processing for the T-app, causing inflation in tail latency.

Using prioritization techniques to prioritize L-app traffic does not solve the problem. Prioritization can be performed at two layers today—(1) transmission of L-app packets can be prioritized at the qdisc [75] layer on the sender-side using the pfifo_fast scheduling policy [76]; and, (2) the L-app process can be prioritized at the CPU scheduler (on both sender and receiver-side) by setting the niceness value of L-app's processes to -20 (the highest CPU scheduling priority in Linux's CFS scheduler [77]). Despite applying both of these prioritization techniques, as shown in Fig. 3.4(a) (Linux + Prioritization), we observe no noticeable improvement in the tail latency of the L-app. qdisc prioritization does not help because there is not much queueing at the qdisc layer to begin with. This is because of the TCP Small Queue (TSQ) [78] feature which limits the number of in-flight bytes at the qdisc layer in order to minimize bufferbloat. CPU scheduling prioritization does not help for two reasons. First, a majority of the network layer processing happens in IRQ threads (Rx packet processing at the receiver-side, and TSQ processing at the sender-side) whose scheduling is not impacted by the priority of application threads. Second, even if there were a mechanism to prioritize IRQ processing, it would not fully solve the problem, as IRQ processing is non-preemptive in nature; thus, L-apps could still get blocked by T-apps.

Since prioritization mechanisms are not effective, the only way to achieve isolation is by separating the network layer processing for L-apps and T-apps onto separate cores. However, due to tight-integration of the processing pipeline with application cores, today's network stack is unable to do so—indeed, as shown in Fig. 3.4(b), the average per-core CPU utilization is only 40%–50% in the above experiments. An ideal network stack should allow separation and isolation of network layer processing for L-apps and T-apps, even if the two classes of applications are sharing the same CPU core.

3.3 NetChannel Design

As discussed earlier, the dedicated, tightly-integrated, and static packet processing pipelines in today's Linux network stack leads to several limitations. To resolve this, NetChannel disaggregates the pipeline by rearchitecting the network stack into three layers: (1) Virtual Network System (VNS) layer, (2) NetDriver layer, and (3) NetScheduler layer.

The VNS layer (discussed in §3.3.1) provides interfaces to applications (e.g., socket, RPC) while ensuring correctness of the interface semantics. These interfaces are "virtual", since unlike in today's Linux network stack, they only buffer data from/to applications and are disaggregated from the rest of the packet processing pipeline. At the bottom, the NetDriver layer (discussed in §3.3.2) abstracts the network as a multi-queue "device" through a generic `channel` abstraction exposed to the upper layer. Decoupling application interfaces (in the VNS layer) from `channels` (in the NetDriver layer), enables flexible and fine-grained multiplexing/demultiplexing and scheduling of data between the two. This multiplexing/demultiplexing is controlled by the NetScheduler layer (discussed in §3.3.3) which enables pluggable schedulers that can be designed to achieve various objectives including dynamic scaling of the packet processing pipeline across CPU cores, and performance-isolation of L-apps from T-apps.

3.3.1 Virtual Network System (VNS) Layer

The VNS layer offers application interfaces while maintaining the necessary semantics of each interface. In order to support unmodified applications, NetChannel supports the standard POSIX socket interface through *virtual sockets*. From the application's perspective, these virtual sockets have the exact same semantics as normal sockets, i.e., reliable in-order delivery between endpoints. As usual, applications can interact with these sockets using standard system calls (e.g., connect, send, recv, epoll) or even using `io_uring` [67]. While VNS also supports other interfaces such as RPCs, we focus our discussion here on the socket interface since it has the strongest requirements in terms of semantics.

Ensuring correctness of interface semantics. Each virtual socket internally maintains a pair of Tx and Rx buffers. When applications send/receive data to/from virtual sockets, data is copied from/to userspace to/from the virtual socket Tx/Rx buffers. Data in the virtual socket Tx buffer is forwarded to the NetDriver layer for transmission, while data received over the network is forwarded from the NetDriver layer to the virtual socket Rx buffer. While we can rely on the network transport (underlying the `channels` in NetDriver layer) to guarantee reliable delivery, VNS needs to do some book-keeping to ensure in-order delivery of bytes between a pair of virtual sockets. This is because, as we will discuss in §3.3.2, data from a virtual socket can be multiplexed/demultiplexed to/from more than one underlying `channel` in the NetDriver layer, in which case, it is possible for data to arrive in the virtual socket

API method	Arguments	Description
create()	metadata	Create a new channel instance
destroy()	channel	Destroy a channel instance
enqueue()	channel, List<packet buffer>, metadata	Enqueue data for transmission through a given channel. Given as a list of packet buffers along with metadata.
dequeue()	channel, count	Dequeue upto count bytes of data received through a given channel. Returns a list of packet buffers along with metadata.

Table 3.2: Summary of core channel API calls that need to be implemented by network drivers. For a more exhaustive list, see [79]. The metadata argument in both create and enqueue calls is opaque from the perspective of the channel interface, and can be used to encode transport-specific information such as host address and port number.

Rx buffer out-of-order. To that end, on the sender side, VNS embeds a sequence number (§3.4) in each data packet representing its order within the virtual socket stream. On the receiver-side, it can then use these sequence numbers to ensure data is delivered in-order—packets are buffered in the virtual socket Rx buffer until they are next in-sequence.

Decoupling data copy from application threads. VNS also maintains per-core worker threads for data copy between userspace and the kernel (for interfaces that require it). Data copy operations for virtual sockets can be divided into smaller parts (each with a target buffer address and length) and distributed across worker threads of multiple cores. This enables utilizing multiple cores to scale data copy processing for throughput-bound applications with long flows.

3.3.2 NetDriver Layer

The NetDriver layer abstracts away the network and remote servers as a multi-queue device and exposes `channels` which are analogous to queues of this device. In this subsection, we discuss the `channel` abstraction, NetDriver mechanisms for decoupling network layer processing from sockets, NetDriver mechanisms to enable ease of integration of new network transport designs, and other details relevant to managing buffer overflow and avoiding head-of-line blocking.

Channel abstraction. In NetDriver, each `channel` consists of a pair of Tx/Rx queues, and an instance of an independent network layer processing pipeline of an underlying network transport that implements functionalities such as reliable delivery, flow control, and congestion control (for example, in the case of TCP, a `channel` would map to a single underlying TCP connection). The `channel` API (Table 3.2) is simple and generic enabling it to encapsulate a wide range of transports. In particular, it can support both connection-oriented stream-based transports (e.g., TCP, DCTCP [42]), and connection-less message-oriented transports (pHost [47], Homa [48] or dcPIM [61]). For example, in the case of the former, upon a `channel` API create call, a connection can be created. Subsequently, arbitrary chunks of data in the stream can be transmitted through `channel` enqueue calls. In the case of the latter, no connection will be created during `channel` creation, and individual messages can directly be transmitted through the `channel` enqueue call (passing destination information in the metadata).

Decoupled network layer processing. `channels` in NetDriver are decoupled from instances of virtual interfaces (e.g., virtual sockets) in the VNS layer. This architectural choice enables *decoupling network layer processing from individual sockets and cores* that applications use—NetDriver allows creating one or more `channels` between a given pair of servers, independent of the number of applications running on these servers, and the number of sockets/cores used by these applications. Further, it allows flexible fine-grained multiplexing and demultiplexing of data from/to virtual socket to/from `channels`. This enables several interesting operating points. For instance, if one `channel` has high CPU load, the subsequent data from a virtual socket can be dynamically scheduled to different `channels`. Such dynamic multiplexing can enable utilizing multiple cores to scale network layer processing, while also enabling utilization of multiple network paths similar to MPTCP.

Integrating new transport designs. Given that NetDriver is an abstraction of a multi-queue device, integrating a network transport is now equivalent to writing a new device driver. This essentially makes it easier to integrate and experiment with new protocols. Protocol developers do not need to worry about implementing cumbersome APIs related to socket interfaces (*e.g.*, epoll) and things like data copy processing, instead focusing only on implementing their own network protocol logic plus simple APIs of the `channel` abstraction as shown in Table 3.2.

Piggybacking on transport-level flow control. To avoid Rx buffer overflow of virtual sockets, NetDriver naturally piggybacks on the flow control of the underlying transport protocol(s) through backpressure, without having to introduce a new flow control protocol. When a virtual socket's Rx buffer becomes full, VNS stops receiving data from `channels`, leading to accumulation of data in the `channel`'s Rx buffer, eventually triggering the flow control mechanism of the underlying transport. VNS resumes receiving data when the virtual socket's Rx buffer is available again as the application reads the data.

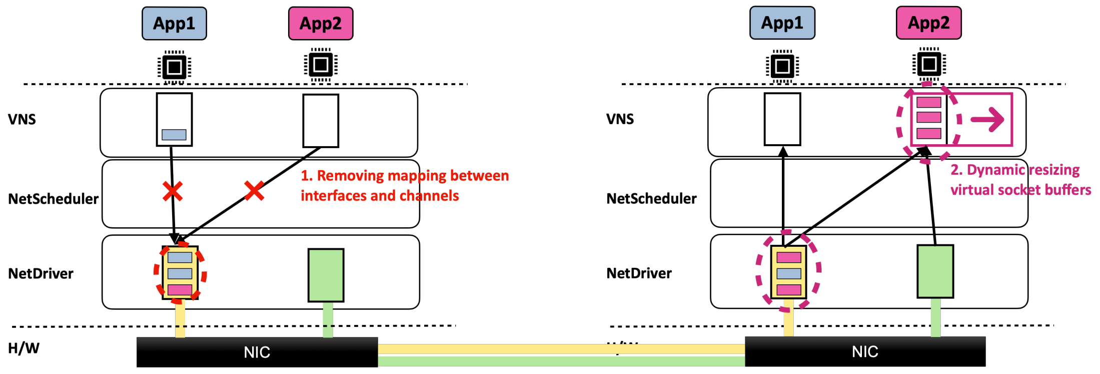

App1
App2
VNS
NetScheduler
NetDriver
H/W
NIC
1. Removing mapping between interfaces and channels
App1
App2
VNS
NetScheduler
NetDriver
NIC
2. Dynamic resizing virtual socket buffers

Figure 3.5: Resolving HoL blocking. HoL blocking can occur when multiple apps share the same channel. In this example, App1 and App2 use the yellow channel to send data. App2 does not promptly read data from its socket buffer on the receiver side, causing HoL blocking of App1's packets. To resolve this, 1) on the sender side, NetScheduler removes the mapping between sockets and channels when it detects a queue buildup, preventing more packets from being pushed to the channel; 2) on the receiver side, NetScheduler, with full information on buffer occupancy, identifies App2 as the cause and resizes its buffer, allowing packets from the yellow channel to be sent to sockets, thus resolving the HoL blocking. More discussion in §3.3.2.

Resolving HoL blocking. Since it is possible for a single `channel` to be shared by multiple virtual sockets, we need to handle corner-case scenarios where one virtual socket causes head-of-line (HoL) blocking for the others, leading to the performance degradation. This can happen if an application does not read data from a virtual socket for an extended period of time (e.g., because it is malfunctioning, or busy with other work).

To resolve head-of-line (HoL) blocking, NetScheduler stops sending packets when it observes a queue buildup on a shared channel. On the receiver side, NetChannel dynamically scales virtual socket buffers using information from NetScheduler, which has full visibility of both virtual socket and channel buffer occupancy. This allows NetScheduler to trigger the resizing of the socket buffer causing the HoL blocking, similar to how today's network stacks dynamically scale buffers. After expanding the virtual socket buffer, packets in the channel can be forwarded to the sockets, addressing HoL blocking.

For example, as shown in Fig 3.5, two applications share the yellow channel, and App2 does not read data from its virtual socket promptly, causing HoL blocking of App1's packets. In this case, NetChannel expands App2's buffer to accommodate more packets from the channel's buffer. One important point is that the increase in App2's socket buffer size is bounded, typically at a maximum of 2x the channel buffer size, assuming all packets in the channel on both sender and receiver sides belong to App2. After resizing, packets in the channel are transmitted to the receiver's virtual socket buffer, resolving HoL blocking. Once NetScheduler observes the queue is empty on the sender side, it can resume transmission by assigning the applications to separate channels, preventing future HoL blocking.

3.3.3 NetScheduler Layer

NetScheduler performs three main tasks: (1) fine-grained multiplexing and scheduling of application data to `channels` to achieve various performance objectives, (2) scaling number of `channels` between a pair of hosts dynamically, and (3) scheduling of data copy requests across per-core data copy worker threads at fine-grained timescales. Given its location in the kernel, NetScheduler has visibility into various kinds of metrics such as the occupancy level of queues, number of virtual sockets, CPU core utilization, application priority, and so forth. We note that our goal is not to design scheduling policies, but rather, to provide mechanisms to enable different policies. One can implement any scheduling policy within our NetScheduler framework. Here, we discuss some simple example policies that enable new operating points. The specific policies used in our current implementation are discussed in §3.4.

Dynamic scheduling of application data to channels. At any given point, there can be one or more active `channels` between a pair of hosts. Upon receiving data from an application, NetScheduler determines the target `channel` to send the data on at per-`skb` granularity using a configurable scheduling policy (*e.g.*, round-robin, shortest-queue-first, etc.) to achieve fine-grained load balancing across `channels`. As we show in §3.5, this enables scaling network layer processing across cores.

Dynamic scaling and placement of channels. NetScheduler scales the number of `channels` to a given remote host dynamically by monitoring scheduling metrics at coarse-grained timescales. An example policy is to increase the number of `channels` when the average CPU utilization across channel workers is persistently high. Further, NetScheduler also controls the mapping of `channels` to cores. This can be exploited to achieve performance isolation by separating `channels` for L-app and T-app processing and mapping these `channels` to different cores. As we show in §3.5, this enables performance isolation when L-apps and T-apps are co-located.

Dynamic scheduling of data copy requests. NetScheduler schedules data copy requests generated by virtual sockets over multiple per-core worker threads at per-request granularity. It uses the cores in the same NUMA node as the application core to avoid cross-NUMA data copy overheads. As we show in §3.5, this makes it possible to selectively parallelize data copy across cores for T-apps.

3.4 NetChannel Implementation

We have implemented NetChannel in Linux kernel v5.6. Throughout the implementation, our goal was to re-use existing kernel network stack infrastructure as much as possible. To that end, most of our current implementation re-uses existing code in the kernel. In this section, we discuss some interesting details of our NetChannel implementation.

Application interfaces. At VNS, our goal is to support unmodified application interfaces. We thus implement the virtual socket interface by adding a new flag— `IPPROTO_VIRTUAL_SOCK` in the standard socket interface to create virtual sockets. Applications can specify NetChannel-related attributes via `setsockopt()` — *e.g.*, the `SO_APP_TYPE` attribute determines the application class (*e.g.*, latency-sensitive, throughput-bound, etc.). The RPC interfaces are similar to those in prior work [80].

Virtual socket connections. The virtual socket interface uses the following procedure to set up connections (similar to existing socket interface): clients initiate `connect` system calls and the corresponding *listen* sockets on the remote host accept the connection requests and return a new socket per request (`accept` system call). Underneath, virtual sockets perform a handshake using `NCSYN` and `NCSYN_ACK` control packets to set up a connection. Note that since the underlying `channels` in NetDriver already provide reliability, virtual sockets only require a 2-way handshake, unlike TCP's 3-way handshake.

NetChannel headers. Since one virtual socket can use multiple `channels` and/or multiple virtual sockets can share the same `channel`, NetChannel needs to uniquely identify packets to their corresponding virtual sockets. To do so, NetChannel wraps an additional header atop of the packet payload. The NetChannel header consists of (1) a pair of virtual socket source and destination ports, to uniquely identify virtual socket-level connections; (2) virtual socket sequence number, to perform packet reordering when multiple `channels` are used; and (3) packet type, to distinguish data packets from control packets (*e.g.*, `NCSYN` and `NCSYN_ACK`). Use of NetChannel header allows virtual sockets to work in conjunction with underlying `channels` without *any* modifications in the `channel`'s header.

Reducing page allocation overheads for DMA. To reduce page allocation overheads during DMA (caused by `get_page_from_freelist()` calls), our implementation sets up a dedicated page pool for each receive queue of NIC. While a large page pool size may help reduce the page allocation overhead, it may also increase L3 cache miss rate due to DCA effects [17]. We use 256 as the default page pool size. We found that it is sufficient to achieve reasonably low page allocation overhead while still maintaining a low DCA cache miss rate. Even for a NIC with 256 receive queues, the memory overhead of maintaining these page pools is $256 \times 256 \times 4$KB = 256MB, which is negligible relative to the DRAM sizes of modern servers.

Scheduling policy. Our current NetScheduler implementation adopts a simple round-robin scheduling policy for scheduling of (1) application data to channels and (2) data copy requests to workers. For (1), we use only `channels` of the same type as the application (i.e., L-app or T-app `channels`). To avoid overloading already busy `channels`/workers, we exclude those which have queue occupancy higher than a certain threshold. Through simple sensitivity analysis,

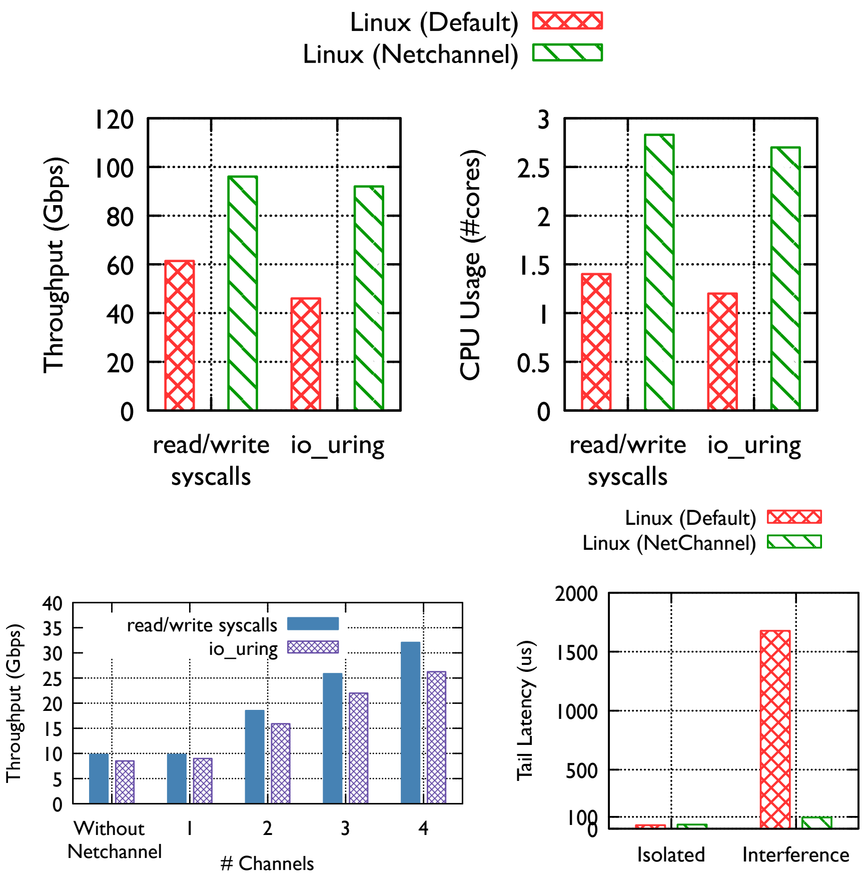


Figure 3.6: NetChannel enables Linux to achieve new operation points (left-right and top-bottom, a-d). (a, b) For a single long flow, it can saturate 100Gbps (a), by utilizing multiple cores for data copy processing (b). (c) It enables near-linear scaling of short-message throughput with an increasing number of `channel`s. (d) It is able to provide performance isolation even when an L-app is co-located with 8 T-apps over 8 cores. More discussion in §3.5.2.

we found $2MB$ and $640KB$ to be good thresholds for (1) and (2) respectively, and use these by default for our evaluation (§3.5).

3.5 NetChannel Evaluation

In this section, we demonstrate that NetChannel is able to achieve new operating points that were previously unachievable by the Linux network stack, in particular, saturating a 200Gbps link using a single socket, increasing short message throughput almost linearly with cores, and achieving μs-scale tail latency for L-apps even when they are co-located with T-apps. We describe our evaluation setup in §3.5.1. We use the same experimental scenarios in §3.2 and provide insights on how NetChannel alleviates the previously discussed limitations of today's Linux network stack (§3.5.2), followed by an investigation of the overheads of our current NetChannel prototype (§3.5.3). Next, we demonstrate NetChannel's effectiveness with real-world applications (§3.5.4), and finally show that it can scale to Terabit Ethernet (§3.5.5).

Before diving in, we make three important notes. First, NetChannel supports unmodified applications (§3.3), and one can run any application on top of it. In order to focus on the network stack, we use lightweight applications which perform minimal compute similar to prior works [16, 17], and additionally demonstrate NetChannel's effectiveness with two real world applications (Redis and SPDK). Second, our goal is not to show that NetChannel beats state-of-the-art network stack performance in absolute terms, but rather to demonstrate the benefits enabled by NetChannel architecture and understand its overheads. In order to do so, we naturally compare our prototype with the baseline system that it is implemented on top of (Linux). And we will discuss in §4, NetChannel's ideas could very well be implemented on top of userspace stacks, hence making it complementary to these systems. Third, while parallelizing parts of the packet processing pipeline across multiple cores, NetChannel naturally introduces some (although relatively minimal) CPU overheads. Since the processing speed of a single CPU core has long since been saturated, utilizing multiple cores is essential. Hence, paying a small cost in per-core overheads to enable this is worthwhile.

3.5.1 Evaluation Setup

Hardware setup. Our experimental testbed consists of two servers connected directly via a 100Gbps link. Each server has a 4 NUMA nodes with 8 cores per NUMA node (Intel Xeon Gold 6234 3.3GHz CPU), 32KB/1MB/25MB L1/L2/L3 caches, 384GB DRAM and a 100Gbps NVIDIA Mellanox ConnectX-5 NIC. Both servers run Ubuntu 20.04 (Linux kernel v5.6). By default, we enable TSO, GRO, Jumbo Frames (9000B), aRFS, and Dynamically-Tuned Interrupt Moderation (DIM) [81] while disabling hyperthreading and IOMMU, since doing so maximizes performance. Direct Cache Access (Intel DDIO) is enabled and configured to use the maximum possible number of L3 cache ways for all experiments.

Evaluated workloads. Similar to §3.2, T-apps generate long-lived flows (i.e., stream traffic) and L-apps generate ping-pong style 4KB RPC requests/responses. Both of these applications perform minimal application-level processing, ensuring that the network stack is the bottleneck. We consider both scenarios of standard read/write system calls, and `io_uring` [67]. In all experiments, we only cores in the NUMA node where the NIC is attached. We also evaluate NetChannel with two real-world applications, Redis [6], and SPDK [82].

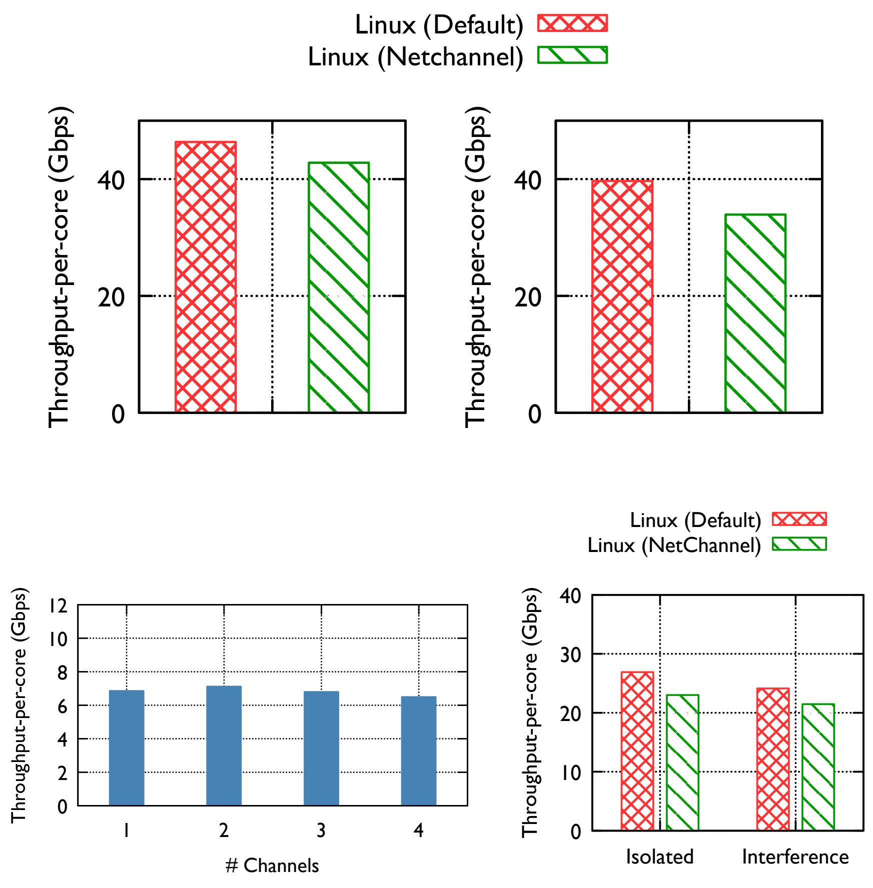


Figure 3.7: Understanding NetChannel overheads (left-right and top-bottom, a-d). NetChannel incurs minimal throughput-per-core overheads while emulating today's Linux pipeline (a), and while scaling data copy processing across cores (b). (c) throughput-per-core does not change independent of the number of `channels`. (d) it is able to isolate L-app latency with minimal throughput-per-core degradation. More discussion in §3.5.3.

Performance metrics. We measure performance in terms of throughput for T-apps and P99.9 tail latency for L-apps. In order to quantify CPU efficiency and understand overheads, we use throughput-per-core which is measured as the total throughput / CPU utilization (we take the maximum of the client-side and server-side CPU utilization when computing CPU utilization).

3.5.2 New operating points

We now demonstrate how NetChannel enables three new operating points using the experimental scenarios from §3.2.2.

Scalability for long flows. In the extreme case of a T-app using a single TCP socket, Linux fails to saturate the 100Gbps link due to its static packet processing pipeline (§3.2.2), despite the availability of CPU cores. As shown in Fig. 3.6(a), while using standard read/write system calls, NetChannel enables Linux to saturate the 100Gbps link by making use of multiple cores (Fig. 3.6(b)). This is because NetChannel allows independent scaling of data copy processing (which is the bottleneck in this scenario) at VNS, so that NetScheduler can use two data copy worker threads on two different cores while maintaining one `channel` at NetDriver layer. In the `io_uring` case as well, we find that NetChannel enables Linux to nearly saturate 100Gbps by utilizing multiple cores (Fig. 3.6(a, b)).

Scalability for short messages. The second scenario in §3.2.2 considers a short message scenario with 4KB RPCs where network layer processing overheads are more dominant. To push these overheads to an extreme, we disable throughput optimization techniques including TSO/GRO and Jumbo Frames (both with and without NetChannel for a fair comparison). NetChannel enables Linux to dynamically scale network layer processing by allowing data from a single virtual socket to be multiplexed across multiple `channels` (§3.3). To demonstrate this, we measure throughput in this scenario while increasing the number of `channels` (each running on a separate core). With standard read/write system calls, we find that throughput increases near-linearly with the addition of `channels` (Fig. 3.6(c)). We discuss overheads in §3.5.3. With `io_uring` as well, we see that throughput increases with the addition of `channels` (Fig. 3.6(c)). For the same number of channels, we observe slightly lower throughput than with read/write syscalls due to `io_uring` having extra overheads on the application core.

Enabling performance isolation. In the third scenario that captures performance interference, we run 8 T-apps and one L-app over 8 cores. We focus on the case of standard read/write system calls. Since `io_uring` does not improve per-application performance, as observed in previous experiments, we omit it in the interest of brevity. As discussed in §3.2.2, with today's Linux network stack, the L-app suffers from high tail latency inflation due to network layer processing interference between T-apps and the L-app. NetChannel enables isolating network layer processing for L-apps from T-apps, even if they share the same core, by decoupling virtual sockets from `channels`— `channels` can be flexibly mapped to different cores. In this experiment, NetScheduler uses up to 4 `channels` for T-apps and a single `channel` for L-app as the L-app generates low load. These `channels` are assigned to different cores in order to separate network layer processing for the T-apps from that for the L-app. As shown in Fig. 3.6(d), we see that with NetChannel, Linux is able to achieve 17.5× lower tail latency for L-app, hence demonstrating that NetChannel can indeed enable performance isolation. We also repeated this experiment with 8 L-apps instead of 1, and confirmed that benefits remain — NetChannel enables Linux to achieve 9.5× lower tail latency in this case.

3.5.3 Understanding NetChannel Overheads

We now investigate the overheads incurred by NetChannel in the process of enabling new operating points.

Overheads of emulating the Linux network stack. To better understand the overheads that NetChannel introduces, we emulate a single packet processing pipeline of today's Linux stack using NetChannel. To this end, we use a single application thread, and a single `channel` thread, while placing everything on the same core. For a fair comparison, Linux uses a single application thread, and we enable aRFS for both systems to ensure that the Rx packet processing is also run on the same core. In Fig. 3.7(a), we observe that NetChannel shows a minimal ~ 7% reduction in throughput-per-core (the server-side core is the bottleneck for both systems).

Overheads of scaling data copy processing. In order to understand the overheads of scaling data copy processing, we compare NetChannel and Linux using scenarios where they are both able to saturate the 100Gbps link, and compare the total CPU utilization. For NetChannel, we use 2 data copy threads and 1 `channel` thread running on separate cores (similar to Fig. 3.6(a)). For Linux, we run 3 long-lived TCP connections over 3 cores to fully saturate the 100Gbps link (this is the minimum number needed to saturate 100Gbps). We find that NetChannel incurs a minimal 12% reduction in throughput-per-core as shown in Fig. 3.7(b). The main reason for this is because the application buffers are not warm in the L1 cache of the cores where the data copy worker threads run, leading to higher L1 cache misses during data copy.

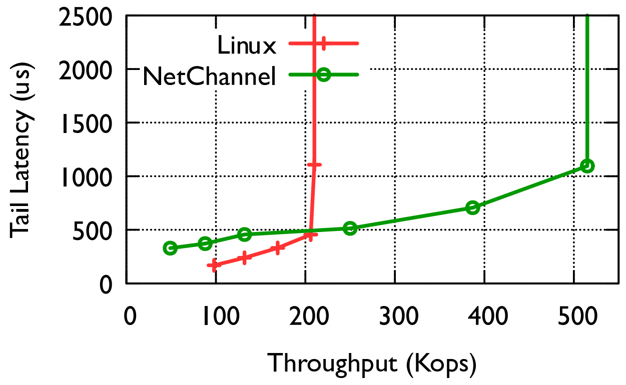


Figure 3.8: NetChannel enables Linux to achieve 2.4× higher throughput for Redis. More discussion in §3.5.4.

Overheads of scaling network layer processing. To understand overheads incurred by NetChannel while scaling network layer processing, we measure the throughput-per-core from Fig. 3.6(c), as the number of `channels` increases from 1 to 4. As shown in Fig. 3.7(c), throughput-per-core remains the same independent of the number of channels. Compared to Linux (default), there is a relatively small degradation in throughput-per-core. We found that the reason for this overhead is that NetScheduler needs to wake up `channel` threads more frequently as each `channel` thread goes to sleep after processing each short message (4KB). Such overheads could be reduced if we perform batching at the NetScheduler layer.

Overheads of achieving performance isolation. Now we consider the performance isolation scenario in Fig. 3.6(d), where 8 T-apps and 1 L-app are running over 8 cores, to understand the overheads of achieving performance isolation. Fig. 3.7(d) shows the throughput-per-core of T-apps in this experiment. We see that NetChannel incurs only a minimal throughput-per-core reduction (12%) in the Interference case, while achieving more than an order-of-magnitude reduction in tail latency (Fig. 3.6(d)).

3.5.4 Real-world applications with NetChannel

Redis with NetChannel. We now evaluate NetChannel with Redis [6], a well-known in-memory key-value database. We use the standard YCSB workload [83], with 95%/5% read/write ratio. Each RPC request is 4KB in size. We use 8 threads on the client side to fully utilize all cores on a single NUMA node. Since the workload generates small-sized messages, we use the same configuration used in Fig. 3.6(c).

Fig. 3.8 shows the latency-throughput curve with and without NetChannel. We see that NetChannel enables Linux to achieve 2.4× higher throughput for Redis. This is because NetChannel enables scaling network layer processing across cores through multiple `channels`. (In this experiment, NetChannel increases the number of `channels` to 4.). In terms of tail latency, NetChannel incurs slightly higher scheduling and reordering latency as the virtual socket of the Redis server is mapped to multiple `channels`. While this latency overhead is more visible at low load, it is ~160μs.

SPDK-based remote storage stack with NetChannel. There has been significant recent work on designing remote storage stacks in the disaggregation context [22, 69, 84, 85]. To evaluate this scenario, we use SPDK [82], a widely-deployed userspace storage stack. In particular, we use SPDK's NVMe-over-TCP stack [84] that uses the Linux TCP/IP stack by default to access a remote storage device over the network. We use an experimental setup similar to prior work [69] — in our two-machine testbed, the SPDK client (we use the standard SPDK perf benchmark tool [86]) runs on one of the machines and issues I/O requests to a remote in-memory storage device (RAM block device) exposed by the SPDK server running on the other machine. Both the SPDK client and server application threads run on a single CPU core each, and use a single TCP connection for data transfer. We use a sequential 100% read workload with large I/Os (2MB) in order to maximize throughput.

Fig. 3.9 shows the total throughput that SPDK achieves with and without NetChannel. With NetChannel, we use a single channel and vary the number of data copy threads. Even with a single data copy thread, using NetChannel already leads to a 1.19× increase in SPDK throughput as network layer processing is offloaded onto a separate core. Increasing the number of data copy threads leads to significant improvements in throughput as data copy is parallelized across multiple cores. With 3 data copy threads, NetChannel enables SPDK to achieve 2.06× higher throughput and saturate the 100Gbps link with a single TCP connection. Relative to the Fig. 3.6(a) experiment, SPDK requires one extra data copy thread to saturate the link bandwidth. This is due to the SPDK client using larger buffers and additional application-level delays for processing responses before data copy. Both of these factors contribute to higher L3 cache miss rate as analyzed in prior work [17], and hence result in reduced data copy efficiency.

3.5.5 NetChannel with Terabit Ethernet

We now demonstrate that NetChannel scales to link speeds beyond 100Gbps, i.e., Terabit Ethernet [87]. For this we use a different testbed which has two servers directly connected by a 200Gbps link. Each server has 2 NUMA nodes (Intel Xeon Gold 6354 3.0GHz CPUs) and a Mellanox ConnectX-6 NIC. Each NUMA-node has 18 cores and 39MB of L3 cache. We re-ran the Fig. 3.6(a) ex-

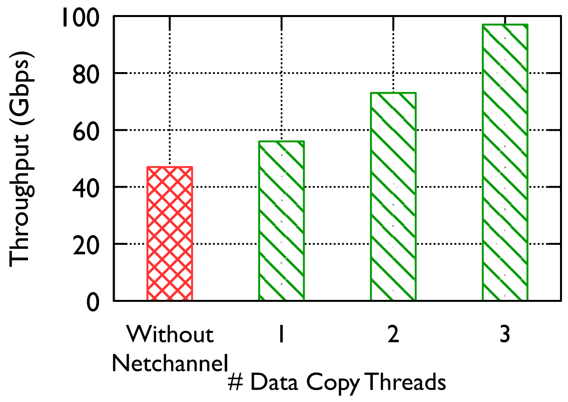


Figure 3.9: NetChannel enables Linux to achieve 2.06× higher throughput for SPDK. More discussion in §3.5.4.

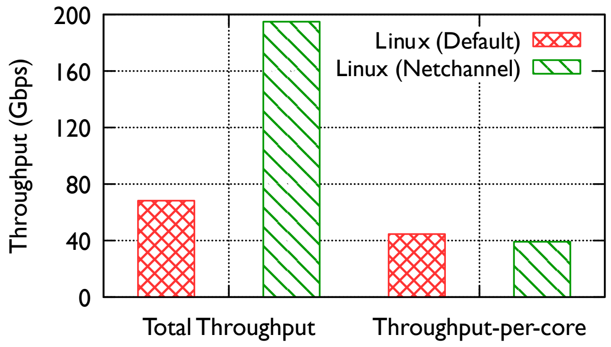


Figure 3.10: NetChannel enables Linux to saturate 200Gbps link bandwidth using a single socket. More discussion in §3.5.5.

periment of a single T-app with a single socket on this new setup. As shown in Fig. 3.10, NetChannel enables Linux to saturate the 200Gbps link bandwidth using a single application thread. To do so, it uses 2 `channels` and 3 data copy threads. Unlike in the previous setup (Fig. 3.6(a)), a single `channel` is no longer sufficient to saturate the link, as a single core is not able to perform all of the network layer processing at the required rate. We find that the throughput-per-core achieved by Linux both without and with NetChannel (Fig. 3.10) increases on the new testbed (by 12% and 15% respectively) due to improved data copy efficiency as a result of larger L3 cache size.

3.6 Related Work

We discuss work that is most closely related to NetChannel's goals.

Linux network stack improvements. Linux now has support for TCP zero copy send [23] and receive [24]. However, as discussed in prior work [17], these mechanisms are far from offering a silver bullet. Zero-copy receive (which bears similarities to the older zero-copy mechanisms in Solaris [88]) in particular requires special hardware support (header-data split) in the NIC [24], and has several other limitations [89], which limits widespread adoption. Nevertheless, even the small subset of applications which use these zero-copy mechanisms can still benefit from network layer processing scalability, performance isolation, and other benefits enabled by NetChannel. NetChannel is complementary to many existing Linux kernel optimization efforts especially for small messages, through new interfaces [26, 27], system call optimizations [38, 39], and optimized socket implementations [25].

Recent work [17] has reported an in-depth analysis of overheads in the Linux network stack. NetChannel's design is motivated by observations and insights from this work. i10 [22] and blk-switch [69] are recent enhancements to the Linux storage stack. They make use of the unmodified Linux network stack for remote storage access, and could reap the benefits of NetChannel if run on top of it.

Userspace network stacks. There has been a significant amount of recent work on designing userspace network stacks [11–13, 15, 16, 20, 35, 64, 90–94], many of which are built on top of low-level frameworks such as DPDK and netmap [95].

We will discuss in §4 that NetChannel's architecture has the potential to provide benefits to userspace network stacks as well.

Hardware network stacks. There has also been a lot of recent work on both partially and fully offloading host network stacks to hardware [30,43,96,97]. At a conceptual level, FlexTOE [97] is the closest to NetChannel. Similar to NetChannel, it enables parallelization of the packet processing pipeline. However, it focuses on parallelizing a specific transport protocol implementation (TCP), unlike NetChannel which considers the end-to-end packet processing pipeline from the application to the NIC, and enables parallelization in a transport-agnostic manner. In §4, we will discuss that NetChannel's architectural ideas can be applied to hardware-offloaded network stacks as well.

Alternative solutions beyond the network stacks. There are also solutions beyond the network stack aimed at improving application performance, such as the RPC libraries [63,98]. While it is possible to achieve some of NetChannel's benefits using libraries and/or schedulers outside the network stack (e.g., gRPC can multiplex RPCs across different underlying connections), there are two limitations of such an approach. First, decoupling and independently scaling different parts of the packet processing pipeline (e.g., data copy and network layer processing) requires support from the network stack, thus necessitating modifications similar to NetChannel. Second, in multi-tenant deployments, these libraries do not have global visibility of all applications, system-level metrics like CPU uti- lization, and in particular, network-layer metrics like congestion information to effectively make multiplexing decisions. Thus, the network stack is the right place to realize the NetChannel architecture and its benefits

3.7 Summary

We have demonstrated that today's host network stacks are unable to fully exploit the capabilities of modern hardware due to their dedicated, tightly integrated, and static packet processing pipelines. Our core contribution is NetChannel, a new disaggregated host network stack architecture that rearchitects the stack into three loosely-coupled layers. We have implemented NetChannel in the Linux kernel, and evaluated it to demonstrate that it enables new operating points that were previously unachievable, including saturation of a Terabit ethernet link with a single application core, independent scaling of network layer processing, and performance isolation between latency-sensitive and throughput-bound applications.

CHAPTER 4

CONCLUSION AND FUTURE WORK

Datacenters now support Terabit Ethernet. However, due to the slowdown of Moore's Law and the end of Dennard scaling, the growth in server processing capacity (core speeds $\times$ core count) has been stagnant. These two trends have led to new, practically important and technically challenging, questions at the intersection of operating systems and computer networking: while network hardware can support μs-scale latency and hundreds of gigabits of bandwidth, designing end-host network stacks that can leverage these capabilities efficiently has become a key open research problem.

The first contribution of this dissertation is to build an in-depth understanding the core challenges hindering existing host network stacks from fully leveraging modern network hardware. Our study indicates: 1) With the rapid increase in network link bandwidth, data movement overheads (e.g., moving data from memory to CPUs) become the bottleneck for scaling single-core performance; and with existing network stacks, we need multiple cores to exploit the capabilities of Terabit network hardware. 2) However, using multiple cores further degrades the CPU efficiency. This is because host resources such as CPU caches and access link bandwidth are contended, leading to performance degradation.

The second contribution of this dissertation is to introduce NetChannel —a new network stack architecture that enables host network stacks to leverage network hardware without requiring application modifications. NetChannel disaggregates network stacks into multiple loosely-coupled layers, allowing each layer to scale and schedule across multiple cores independently. Using an end-to-end NetChannel realization within the Linux network stack, we demonstrate that NetChannel enables new operating points—(1) enabling a single application thread to saturate multi-hundred gigabit access link bandwidth; (2) enabling near-linear scalability for small message processing with an increasing number of cores, independent of the number of application threads; and, (3) enabling isolation of latency-sensitive applications, allowing them to maintain μs-scale tail latency even when competing with throughput-bound applications operating at near-line rate.

4.1 Future Work

This dissertation serves as an initial step toward the goal of efficiently exploiting the capabilities of Terabit network hardware. This thesis leaves open several interesting directions of future research, and we will describe each direction in detail below.

4.1.1 Improving CPU efficiency

First, to improve the CPU efficiency of network packet processing, as observed in Chapter 2, it is crucial to reduce not only packet processing overheads but also data movement overheads.

4.1.1.1 Reducing data movement overheads

Designing efficient and dynamic direct cache access (DCA) pipelines. Existing optimizations allow NICs to transfer data to the L3 cache. However, as shown in Chapter 2, the increasing bandwidth-delay product outpaces the growth of cache size due to the rapid increase in bandwidth. This results in packets being evicted to memory, thereby degrading performance. To reduce cache contention, we aim to explore the benefits of not just an efficient, but also a more dynamic DMA pipeline. The current DMA datapaths are static: all DMAed data are treated equally, entering the cache regardless of whether this leads to a performance improvement. For instance, certain applications do not need high network performance, and some may not promptly read data from their network stack buffer. In such cases, it may not be beneficial to push packets to the cache. In the future, we aim to develop an intelligent scheme for dynamically transferring packets to CPU caches or memory. This may include: 1) designing an application interface that allows applications to specify their performance requirements (e.g., whether packets need to be directed to the CPU cache via DCA), such as reusing/modifying the socket priority system calls [99]; 2) tracking performance counter values (e.g., L3 cache miss rates) and OS-level information (e.g., receiver socket buffer occupancy). Combining these values can help decide whether packets of specific applications should go to memory or cache. Once the decision has been made, we can utilize existing NICs' flow tables (e.g., those used in aRFS), where the key is the flow identification number (such as a five-tuple), and the value provides a hint to indicate whether the corresponding packets should be forwarded to memory or cache. This hint can be piggybacked in the corresponding PCIe transactions, allowing

CPUs to make DMA/DCA decisions [56].

Designing CPU-efficient transport protocols. Making the DMA/DCA data path more dynamic and efficient only mitigates the cache contention. For example, it does not address cache contention when multiple applications need their packets DCAed to the cache. To resolve the contention, an efficient host resource orchestration mechanism is needed. In this context, it is worthwhile to reconsider the role of transport protocols. Transport design has traditionally focused on resolving bottlenecks on the network side (*e.g.* achieving high network throughput and reducing queuing delay in the switch buffers). As the bottleneck has shifted from the network to the host, it is worthwhile to explore the potential of designing protocols to orchestrate host resources by considering not just traditional metrics like Round-trip-time (RTT) and packet drops, but also host resource availability, such as available cache size for DCA. For example, to reduce host resource contention, while not sacrificing performance, protocols could limit the number of active flows within a short period, rather than allowing all flows to share host resources equally. Recent receiver-driven protocols [61] have the potential to enable such fine-grained orchestration of both sender and receiver resources.

4.1.1.2 Reducing packet processing overheads

Long flow and short message processing are bottlenecked by different parts of the processing pipeline and may need different optimization techniques to improve CPU efficiency. Based on Chapters 2 and 3, transport layer processing is the main bottleneck for processing short messages. For long flows, as discussed in Chapter 2, we find that the bottleneck is on the receiver side and processing has two main overheads: 1) data copy processing, and 2) NIC driver processing, with its associated memory management overheads typically related to allocating memory space for packet metadata (e.g., `skb`) and the payload in the NIC driver.

Saving CPU cycles for short message processing is a well-explored problem. One possible direction is to reduce the number of packets being processed on CPUs. For example, hardware offloading of transport layer processing eliminates the need for network packet processing by CPUs, as packets are processed in hardware [30].

On the other hand, saving CPU cycles for long flow processing is an important and interesting direction to explore in the future. Simply offloading the transport layer to hardware will not provide CPU efficiency; even with hardware offloading, data copying still occurs to transfer data from the kernel buffer to the application buffer, and memory allocation/deallocation overheads still exist for managing payload in the kernel buffer. Here, several interesting directions are worth exploring:

Achieving zero-copy processing with HW/SW co-design. Data copying overheads can be reduced by improving data movement efficiency. However, to completely resolve this bottleneck, exploring the design of achieving zero-copy processing could be beneficial. Existing solutions for achieving zero-copy either focus on the sender side [23, 84, 100], require changes in application-layer modifications [24], or involve reimplementing the entire network and transport protocols in user space [101]. However, the receiver side is the main bottleneck, with data copy being the primary bottleneck, as discussed in Chapter 2. Designing zero-copy mechanisms on the receiver side that do not require application modifications or reimplementation of network protocols is an interesting direction to explore. Achieving this with HW/SW co-design is one potential way. As the first step, our most recent work [102] demonstrates that using the clean-slate approach—designing new NIC hardware—can achieve zero-copy processing with minimal application modifications. In addition, zero-copy processing can also eliminate the memory allocation overheads for payloads in the netdevice subsystem, as there is no need to allocate separate memory space for packet payloads.

Reducing NIC driver processing overheads with CPU-efficient protocols One main contributor to NIC driver processing overheads is constructing packets from the driver's descriptors and allocating memory space for packet metadata. Fundamentally, reducing the number of packets being processed can reduce this overhead. While offloading the transport layer to hardware can achieve such a reduction, batching can also achieve similar effects without offloading the entire transport layer. Unlike short messages that do not generate enough packets to benefit from batching, long flow processing can effectively utilize batching to reduce overhead. However, existing batching optimization techniques, such as LRO/GRO, become ineffective with an increasing number of flows due to packet interleaving among flows, as shown in Chapter 2. To resolve this issue, existing approaches [103] deliberately delay packets on the receiver side, allowing for packet batching inside the network stack. However, setting an appropriate threshold or timeout for triggering packet reception is challenging due to the bursty nature of traffic patterns inside datacenters. In fact, besides improving data movement efficiency, CPU-efficient transport protocols also have the potential to reduce NIC driver processing overheads with batching. The protocols can proactively control the traffic pattern through a receiver-driven scheme [47,61,104], allowing the receiver to proactively control the number of packets that will be received in the near future. This enables the batching scheme to set an appropriate threshold or timeout, thereby improving the effectiveness of batching.

4.1.2 Multi-core resource management

Given today's stagnant CPU speed and the rapid increase in network bandwidth, even after improving CPU efficiency, multiple cores are needed to fully exploit network hardware capabilities. Multi-core resource management remains crucial to meet application performance requirements. In Chapter 3, we show that rearchitecting host network stacks with NetChannel provides benefits even with simple policies. In the future, extending NetChannel to work in more realistic deployment scenarios would be valuable. This will allow us to fully realize the potential of NetChannel in diverse and complex environments.

Designing versatile NetScheduler policies. The NetScheduler layer in NetChannel's architecture provides the mechanisms to implement different scheduling and placement policies. We have demonstrated in §3.5 that even with simple policies, NetChannel can achieve significant benefits. In fact, in practice, application performance may degrade due to various reasons (*e.g.* network congestion and CPU contention). Based on network and host performance metrics (*e.g.* RTT, number of retransmitted packets, queuing occupancy of channels, CPU usage and so on), the scheduler needs to identify the root cause of degradation and react differently at runtime (*e.g.* creating a new channel that will go through a different path over the network or simply moving the channel to an idle CPU core). To fully utilize the NetChannel architecture, one promising direction for further research is designing versatile NetScheduler policies that can adapt to different scenarios, and meet application performance requirements in real-time.

Extending NetChannel to virtualized environments Additionally, extending NetChannel to virtualized environments, including VMs and containers, is another exciting avenue for exploration. Virtualized environments are increasingly prevalent in modern datacenter infrastructures. Besides the benefits demonstrated in Chapter 3, integrating NetChannel with these environments also offers efficient resource utilization for cloud providers to manage the network demands of different VMs/containers. Recent technology allows VMs to hot plug more CPU resources at runtime, eliminating the need for users to overprovision resources when allocating VMs [105]. Integrating NetChannel with this recent technology enables efficient scheduling of host resources to satisfy the network demands of VMs at runtime.

Applying NetChannel architecture to other network stacks. In Chapter 3, we have realized the NetChannel architecture within the Linux network stack. However, NetChannel's architecture and design ideas can be applied to host network stacks in general—even those placed in userspace and/or hardware. Microkernel-style userspace stacks [11,15,20] would be ideal candidates for implementing NetChannel's ideas. For example, while TAS [16] decouples the packet processing pipeline from application cores, it can benefit from additionally disaggregating different parts of the packet processing pipeline similar to NetChannel. NetChannel's ideas can also be applied to hardware network stacks. For example, in SoC-based smartNICs [106], NetChannel's design could enable transport-agnostic parallelization of processing across cores, which is even more important in this context since these devices typically contain a large number of wimpy cores [97, 106]. Further, by disaggregating the host network stack, NetChannel could enable easier integration of partial hardware offloads that offload different parts of the packet processing pipeline (e.g., I/OAT [43] for data copy, and Tonic [30] fo the transport layer) into Linux or other software network stacks.

4.1.3 Understanding host network stack latency

This dissertation discusses factors hindering network stacks from achieving high performance in terms of throughput and CPU efficiency. However, achieving low latency is also a critical factor for applications. Several recent studies have shown that the Linux network stack suffers from millisecond-scale tail latency [11–16]. This deficiency has led the networking, systems, and computer architecture communities to explore clean-slate solutions, including userspace stacks [11–13, 15, 16, 20, 35, 64, 91, 92] and specialized host network hardware [14, 30, 107]. One interesting direction for future exploration is to perform detailed profiling to understand the root causes of high tail latency in traditional host network stacks. Doing so can help us build an in-depth understanding of the root causes behind high latency in the Linux network stack, potentially benefiting the design of future network stacks, operating systems, and host hardware for low-latency networking.

BIBLIOGRAPHY

[1] Bisection Bandwidth. https://www.sciencedirect.com/topics/computer-science/bisection-bandwidth#:~:text=The%20full%20bisection%20bandwidth%20allows,network%20has%20full%20bisection%20bandwidth.

[2] ChatGPT. https://chatgpt.com.

[3] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. In *USENIX HotCloud*, 2010.

[4] SnowFlake. https://www.snowflake.com/en/.

[5] MemCached. http://www.memcached.org.

[6] Redis. http://www.redis.io.

[7] Google. https://www.google.com.

[8] Facebook. https://www.facebook.com.

[9] Zoom. https://zoom.us.

[10] Mark Zhao, Niket Agarwal, Aarti Basant, Buğra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei Lu, Sundaram Narayanan, Jack Langman, Kevin Wilfong, Harsha Rastogi, Carole-Jean Wu, Christos Kozyrakis, and Parik Pol. Understanding Data Storage and Ingestion for Large-scale Deep Recommendation Model Training: Industrial Product. In *ACM/IEEE ISCA*, 2022.

[11] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *USENIX NSDI*, 2019.

[12] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *USENIX OSDI*, 2014.

[13] George Prekas, Marios Kogias, and Edouard Bugnion. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *ACM SOSP*, 2017.

[14] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. AccelTCP: Accelerating Network Applications with Stateful TCP Offloading . In *USENIX NSDI*, 2020.

[15] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating Interference at Microsecond Timescales. In *USENIX OSDI*, 2020.

[16] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. TAS: TCP Acceleration as an OS Service. In *ACM Eurosys*, 2019.

[17] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. Understanding Host Network Stack Overheads. In *ACM SIGCOMM*, 2021.

[18] Terabit Ethernet. https://en.wikipedia.org/wiki/Terabit_Ethernet#History.

[19] iPerf - The Ultimate Speed Test Tool for TCP, UDP and SCTP. https://iperf.fr/, 2021.

[20] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kokonov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: a Microkernel Approach to Host Networking. In *ACM SOSP*, 2019.

[21] Qizhe Cai, Midhul Vuppalapati, Jaehyun Hwang, Christos Kozyrakis, and Rachit Agarwal. Towards μs Tail Latency and Terabit Ethernet: Disaggregating the Host Network Stack. In *ACM SIGCOMM*, 2022.

[22] Jaehyun Hwang, Qizhe Cai, Ao Tang, and Rachit Agarwal. TCP $\approx$ RDMA: CPU-efficient Remote Storage Access with i10. In *USENIX NSDI*, 2020.

[23] Jonathan Corbet. Zero-copy networking. https://lwn.net/Articles/726917/, 2017.

[24] Jonathan Corbet. Zero-copy TCP Receive. https://lwn.net/Articles/752188/, 2018.

[25] Xiaofeng Lin, Yu Chen, Xiaodong Li, Junjie Mao, Jiaquan He, Wei Xu, and Yuanchun Shi. Scalable Kernel TCP Design and Implementation for Short-Lived Connections. In *ACM ASPLOS*, 2016.

[26] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. MegaPipe: A New Programming Interface for Scalable Network I/O. In *USENIX OSDI*, 2012.

[27] Kenichi Yasukata, Michio Honda, Douglas Santry, and Lars Eggert. StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs. In *USENIX ATC*, 2016.

[28] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, et al. A Cloud-scale Acceleration Architecture. In *IEEE/ACM MICRO*, 2016.

[29] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. Azure Accelerated Metworking: SmartNICs in the Public Cloud. In *USENIX NSDI*, 2018.

[30] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling Programmable Transport Protocols in High-Speed NICs. In *USENIX NSDI*, 2020.

[31] Intel. TCP/IP Offload Engine (TOE) for an SOC System. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/dc/_3_3-2005_taiwan_3rd_chengkungu-web.pdf, 2005.

[32] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion Control for Large-scale RDMA Deployments. *ACM SIGCOMM CCR*, 2015.

[33] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. Revisiting Network Support for RDMA. In *ACM SIGCOMM*, 2018.

[34] Yuliang Li, Rui Miao, Hongqiang Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. HPCC: High Precision Congestion Control. In *ACM SIGCOMM*, 2019.

[35] EunYoung Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *USENIX NSDI*, 2014.

[36] Ilias Marinos, Robert NM Watson, and Mark Handley. Network Stack Specialization for Performance. In *ACM SIGCOMM CCR*, 2014.

[37] Amazon. Amazon EC2 F1 Instances. https://aws.amazon.com/ec2/instance-types/f1/, 2021.

[38] Livio Soares and Michael Stumm. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *USENIX OSDI*, 2010.

[39] Vijay Vasudevan, David G. Andersen, and Michael Kaminsky. The Case for VOS: The Vector Operating System. In *USENIX HotOS*, 2011.

[40] Theophilus Benson, Aditya Akella, and David A Maltz. Network Traffic Characteristics of Data Centers In the Wild. In *ACM IMC*, 2010.

[41] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. The Nature of Data Center Traffic: Measurements & Analysis. In *ACM IMC*, 2009.

[42] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data Center TCP (DCTCP). In *ACM SIGCOMM*, 2010.

[43] Intel. Fast memcpy with SPDK and Intel I/OAT DMA Engine. https://software.intel.com/content/www/us/en/develop/articles/fast-memcpy-using-spdk-and-ioat-dma-engine.html, 2017.

[44] Intel. Intel® Data Direct I/O Technology. https://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/data-direct-i-o-technology-brief.pdf, 2012.

[45] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Changhoon Kim, and Nick McKeown. The nanoPU: A Nanosec-

ond Network Stack for Datacenters. In *USENIX OSDI*, pages 239–256, 2021.

[46] Mark Sutherland, Siddharth Gupta, Babak Falsafi, Virendra Marathe, Dionisios Pnevmatikatos, and Alexandros Daglis. The NEBULA RPC-optimized Architecture. In *ACM/IEEE ISCA*, 2020.

[47] Peter X Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. pHost: Distributed Near-optimal Datacenter Transport over Commodity Network Fabric. In *ACM CoNEXT*, 2015.

[48] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A Receiver-driven Low-latency Transport Protocol Using Network Priorities. In *ACM SIGCOMM*, 2018.

[49] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. Understanding Host Network Stack Overheads. https://github.com/Terabit-Ethernet/terabit-network-stack-profiling, 2021.

[50] The Linux Foundation. Linux Foundation DocuWiki: napi. https://wiki.linuxfoundation.org/networking/napi, 2016.

[51] HewlettPackard. Netperf. https://github.com/HewlettPackard/netperf, 2021.

[52] Sebastien Godard. Performance Monitoring Tools for Linux. https://github.com/sysstat/sysstat, 2021.

[53] Brendan Gregg. Linux perf Examples. http://www.brendangregg.com/perf.html, 2020.

[54] Jonathan Corbet. JLS2009: Generic Receive Offload. https://lwn.net/Articles/358910/, 2009.

[55] Amin Tootoonchian, Aurojit Panda, Chang Lan, Melvin Walls, Katerina Argyraki, Sylvia Ratnasamy, and Scott Shenker. ResQ: Enabling SLOs in Network Function Virtualization. In *USENIX NSDI*, 2018.

[56] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. Reexamining Direct Cache Access to Optimize I/O Intensive Applications for Multi-hundred-gigabit Networks. In *USENIX ATC*, 2020.

[57] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR: Congestion-Based Congestion Control. In *ACM Queue*, 2016.

[58] Neal Cardwell Yuchung Cheng. Making Linux TCP Fast. "https://netdevconf.info/1.2/papers/bbr-netdev-1.2.new.new.pdf".

[59] Stephen M Rumble, Diego Ongaro, Ryan Stutsman, Mendel Rosenblum, and John K Ousterhout. It's Time for Low Latency. In *USENIX HotOS*, 2011.

[60] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency. In *USENIX NSDI*, 2019.

[61] Qizhe Cai, Mina Tahmasbi Arashloo, and Rachit Agarwal. dcPIM: Near-Optimal Proactive Datacenter Transport. In *ACM SIGCOMM*, 2022.

[62] John Ousterhout. A Linux Kernel Implementation of the Homa Transport Protocol. In *USENIX ATC*, 2021.

[63] Google. gRPC: A High Performance, Open Source Universal RPC Framework. https://grpc.io/, 2022.

[64] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs Can Be General and Fast. In *USENIX NSDI*, 2019.

[65] Linux. Socket. https://man7.org/linux/man-pages/man2/socket.2.html, 2021.

[66] Linux. epoll: I/O Event Notification Facility. https://man7.org/linux/man-pages/man7/epoll.7.html, 2021.

[67] Kernel. Efficient IO with io_uring. https://kernel.dk/io_uring.pdf, 2019.

[68] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. Linux Block IO: Introducing Multi-Queue SSD Access on Multi-Core Systems. In *ACM SYSTOR*, 2013.

[69] Jaehyun Hwang, Midhul Vuppalapati, Simon Peter, and Rachit Agarwal. Rearchitecting Linux Storage Stack for Latency and High Throughput. In *USENIX OSDI*, 2021.

[70] Gregory Detal and Sebastien Barre. MultiPath TCP - Linux Kernel Implementation. https://multipath-tcp.org/, 2022.

[71] Linux. MPTCP Upstream Implementation. https://github.com/multipath-tcp/mptcp_net-next/wiki, 2022.

[72] Jonathan Corbet. Zero-copy Network Transmission with io_uring. https://lwn.net/Articles/879724/, 2021.

[73] io_uring Based Networking in Prod Experience. https://tinyurl.com/iouring-reddit, 2021.

[74] Alex Hultman. io_uring is Slower Than Epoll. https://github.com/axboe/liburing/issues/189, 2020.

[75] Traffic Control. https://man7.org/linux/man-pages/man8/tc.8.html, 2001.

[76] Linux. Qdisc: Pfifo Fast Scheduling Policy. https://man7.org/linux/man-pages/man8/tc-pfifo_fast.8.html, 2002.

[77] Linux. Linux Kernel CFS Scheduler. https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html, 2022.

[78] Jonathan Corbet. TCP Small Queues. https://lwn.net/Articles/507065/, 2012.

[79] Qizhe Cai, Midhul Vuppalapati, Jaehyun Hwang, Christos Kozyrakis, and Rachit Agarwal. Towards μs Tail Latency and Terabit Ethernet: Disaggregating the Host Network Stack. https://github.com/Terabit-Ethernet/NetChannel, 2022.

[80] Collin Lee and Yilong Li. Homa DPDK Implementation. https://github.com/PlatformLab/Homa, 2021.

[81] Mellanox Technologies. Dynamically-Tuned Interrupt Moderation (DIM). https://community.mellanox.com/s/article/dynamically-tuned-interrupt-moderation--dim-x, 2019.

[82] Intel. Storage Performance Development Kit. https://spdk.io/, 2022.

[83] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *ACM SoCC*, 2010.

[84] Patrick Dehkord. NVMe over TCP Storage with SPDK. https://ci.spdk.io/download/events/2019-summit/(Solareflare)+NVMe+over+TCP+Storage+with+SPDK.pdf, 2019.

[85] Peter Xiang Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network Requirements for Resource Disaggregation. In *USENIX OSDI*, 2016.

[86] Intel. https://github.com/spdk/spdk/tree/master/examples/nvme/perf, 2022.

[87] Terabit Ethernet. https://en.wikipedia.org/wiki/Terabit_Ethernet, 2022.

[88] H. K. Jerry Chu. Zero-Copy TCP in Solaris. In *USENIX ATC*, 1996.

[89] Eric Dumazet. The Path To TCP 4K MTU and RX Zero-Copy. https://legacy.netdevconf.info/0x14/pub/slides/62/ImplementingTCPRXzerocopy.pdf, 2012.

[90] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M. Voelker, and Amin Vahdat. Chronos: Predictable Low Latency for Data Center Applications. In *ACM SoCC*, 2012.

[91] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System is the Control Plane. In *USENIX OSDI*, 2014.

[92] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynykr, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. The Demikernel Datapath OS Architecture for Microsecond-scale Datacenter Systems. In *ACM SOSP*, 2021.

[93] Ilias Marinos, Robert N.M. Watson, Mark Handley, and Randall R. Stewart. Disk|Crypt|Net: Rethinking the Stack for High-Performance Video Streaming. In *ACM SIGCOMM*, 2017.

[94] Ilias Marinos, Robert N.M. Watson, and Mark Handley. Network Stack Specialization for Performance. In *ACM SIGCOMM*, 2014.

[95] Luigi Rizzo. netmap: A Novel Framework for Fast Packet I/O. In *USENIX ATC*, 2012.

[96] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. hXDP: Efficient Software Packet Processing on FPGA NICs. In *USENIX OSDI*, 2020.

[97] Rajath Shashidhara, Tim Stamler, Antoine Kaufmann, and Simon Peter. FlexTOE: Flexible TCP Offload with Fine-Grained Parallelism. In *USENIX NSDI*, 2022.

[98] Apache. Apache Thrift. https://thrift.apache.org, 2024.

[99] Linux. socket(7) — Linux manual page. https://man7.org/linux/man-pages/man7/socket.7.html, 2024.

[100] Intel. SPDK NVMe-oF TCP Performance Report. https://ci.spdk.io/download/performance-reports/SPDK_tcp_perf_report_2010.pdf, 2020.

[101] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel. In *ACM CoNEXT*, 2018.

[102] Athinagoras Skiadopoulos, Zhiqiang Xie, Mark Zhao, Qizhe Cai, Saksham Agarwal, Jacob Adelmann, David Ahern, Carlo Contavalli, Michael Goldflam, Vitaly Mayatskikh, Raghu Raja, Daniel Walton, Rachit Agarwal, Shrijeet Mukherjee, and Christos Kozyrakis. High-throughput and Flexible Host Networking for Accelerated Computing. In *USENIX OSDI*, 2024.

[103] Hamid Ghasemirahni, Tom Barbette, Georgios P. Katsikas, Alireza Farshin, Amir Roozbeh, Massimo Girondi, Marco Chiesa, Gerald Q. Maguire Jr., and Dejan Kostić. Packet Order Matters! Improving Application Performance by Deliberately Delaying Packets. In *USENIX NSDI*, 2022.

[104] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, An-

drew Moore, Gianni Antichi, and Marcin Wojcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *ACM SIGCOMM*, 2017.

[105] VMware. VMware vSphere Resources: HotAdd/HotPlug Explained. https : / / www . nakivo . com / blog / vmware-vsphere-resources-hotadd-hotplug-explained/.

[106] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading Distributed Applications onto SmartNICs using iPipe. In *ACM SIGCOMM*, 2019.

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