

Generalizing Device Interfaces

Hugo Sadok

CMU-CS-26-132

September 2026

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:

Justine Sherry, Chair

David G. Andersen

James C. Hoe

Arvind Krishnamurthy (UW)

Aurojit Panda (NYU)

*Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.*

Copyright © 2026 Hugo Sadok

This research was sponsored by: the National Science Foundation under award numbers CNS1700521, CNS2212390 and CNS1850384; the Office of Naval Research under award number N000142412059; Vanguard Charitable under award number A020010; the Semiconductor Research Corporation under award number 2018JU2779; and Intel Corporation under award number A026118.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: Host interconnects, Hardware accelerators, SmartNICs, Peer-to-peer communication, Computer networking, Operating systems

*To my mother, for encouraging me to be creative;
my father, for encouraging me to work hard;
and my wife, for so many reasons.*

Abstract

Accelerators and I/O devices perform much of today’s computation. Accelerators (e.g., for machine learning, cryptography, and video processing) are now commonplace, and even long-established I/O devices such as network interface cards (NICs) and disks have gained programmability and compute of their own. Yet devices today communicate using *specialized interfaces* better suited to an era when each device performed a single, fixed function under the CPU’s control. Specialized interfaces limit devices in two ways. First, interfaces are specialized to a particular data format, unable to adapt to new capabilities as devices gain them. Second, interfaces are specialized to a particular pair of endpoints: the device and the CPU. As a result, when applications would benefit from devices interacting directly, they must rely on vendor-specific solutions that exist for only a handful of device pairs.

A *general-purpose* interface could address both limits: it could keep pace as devices gain capabilities, and let any device communicate with any other without the need for vendor-specific solutions. What stands in the way is the long-held assumption that generality costs performance. This dissertation disproves that assumption. Our thesis is that *devices can adopt a general-purpose interface without compromising performance*—in fact, we find that generality can even improve performance by reducing the CPU’s involvement in communication. We pursue generality along two axes: “what” devices communicate and with “whom.”

To generalize “what” devices communicate, we introduce Ensō, a streaming abstraction between devices and the CPU. Instead of specializing to the data’s format and size, Ensō delivers arbitrary data over a stream. This allows devices to extend their capabilities without shoehorning data into legacy formats—a NIC can use the same interface to deliver raw packets, application-level messages, or bytestreams, depending on the offloads it provides. We show that streaming not only makes the interface format-agnostic but also eliminates the overheads that fixed-size buffers impose, letting applications achieve up to 6× the throughput of a specialized NIC.

To generalize with “whom” devices can communicate, we introduce DCP, a general-purpose peer-to-peer Device Communication Protocol. Although interconnects such as PCIe have long supported peer-to-peer communication, device interfaces are designed for talking to the CPU. This is due to differences in capabilities: the CPU is better at exposing memory and centralizing coordination, and devices are better at accessing memory asynchronously. DCP instead provides a symmetric interface that works the same way regardless of devices’ capabilities and allows devices to perform distributed coordination, without the CPU. As a result, DCP cuts latency by 83% compared to CPU-mediated communication, while offering similar performance to specialized interfaces when devices do talk to the CPU.

Together, Ensō and DCP show that devices need not remain workers behind restricted interfaces, but can instead act as peers—enabling a future in which a computer looks less like a centralized hub-and-spoke of CPU and accelerators and more like a distributed system of composable peers.

Acknowledgments

A dissertation describes a success, a direct sequence of steps from problem to solution. But research rarely works this way. More often, it is a sequence of setbacks: directions that do not pan out, design flaws no one anticipated, and, in the case of systems research, unimaginable bugs. I was very lucky to have many people help me through both setbacks and successes. Here I briefly, and insufficiently, acknowledge their help.

Justine Sherry, my advisor, had a deep impact not only on this dissertation but, more broadly, on how I conduct research and see the world. Justine was who I wanted to work with the most when I applied to CMU—and she almost gave me a heart attack when she emailed me (in Portuguese) a few days after I applied. But I could not have imagined then how lucky I was to get that email. I started my PhD with a taste for making systems faster, but Justine encouraged me to work instead on projects that enable systems to do things that were not possible before. And, in doing so, she made me love research even more than I already did. Justine also taught me to spend my time on things I believe in, and that includes writing papers I would love to read and giving talks I would love to watch. Having Justine as an advisor was so wonderful that I secretly planned to remain a PhD student forever. Unfortunately, this plan ended last Christmas, when Justine gave me a copy of Umberto Eco’s *How to Write a Thesis* [47], and I got the message that it was finally time for me to graduate.¹ While I must graduate, I hope I can continue to rely on Justine for advice wherever I end up.

Another wonderful thing Justine did was inviting Aurojit Panda to join my research early in my PhD. Panda ended up becoming my unofficial co-advisor—and I am beyond grateful he did. Panda is truly extraordinary, to the point that I have come to suspect that he is actually a team of clones. I have two pieces of evidence to support this hypothesis. The first is the breadth and depth of his knowledge. At some point, I stopped asking Panda if he had read a paper I was about to comment on and started just assuming that he had—I was rarely wrong. The second piece of evidence is how generous he is with his time. Panda read most of what I wrote during my PhD, and his comments were precise every time. Panda was also always available to discuss ideas, and those discussions would often become arguments that allowed me to solidify my ideas. He is probably the person I argued with the most during my PhD, and I hope he had as much fun in those arguments as I did.

Justine and Panda also stood by me through many setbacks: the many introductions we threw away, the designs I discarded after working on them for months, and the countless times I changed research direction. This dissertation exists because of their unconditional support.

Thank you to my closest collaborators, Nirav Atre and Zhipeng Zhao. Collaborating with

¹In the book, Eco claims that thanking one’s advisor is in bad taste because, if they helped, they simply did their job [47, § 5.5]. This rule does not apply to Justine, since she did much more than her job.

Nirav was one of the best parts of my PhD. Nirav and I wrote most of our papers together [13, 14, 168, 169, 171, 202], and I have fond memories of our many discussions over whiteboards, lunches, dinners, and (too many) coffees and desserts. It is an honor to be able to call Nirav a friend, and I hope we can keep working on fun problems together. As for Zhipeng, this dissertation would have looked very different without his help. He taught me how to use FPGAs for high-speed packet processing and was really generous with his time at the beginning of my PhD. I will never forget our many meetings to discuss both Pigasus [202] and Norman [168]. Zhipeng's generosity is something I aspire to replicate.

The members of my thesis committee, James Hoe, Arvind Krishnamurthy, and David Andersen, stayed with me through many changes in research direction and helped turn both my proposal and defense into fun debates. They were also mentors to me. James met with me over many coffees to give me advice, Arvind made key contributions to how I framed this dissertation, and Dave's comments made my writing and talks better. James also contributed to the work in Chapter 3, and Arvind to the work in Chapter 4. I was also lucky to work with many other collaborators during my PhD. They improved this dissertation through their contributions, mentorship, and feedback, including Daniel Berger, Margarida Ferreira, Srikanth Kandula, Grace Liu, Ilias Marinos, Bryan Parno, Francisco Pereira, Vyas Sekar, and Ren Wang. Thank you, Daniel, for staying on a call with me the entire day the first time we submitted Ensō [169]. Thank you, Margarida, for your friendship and for teaching me about SMT. Thank you, Ilias and Srikanth, for your mentorship during and after my internship at Microsoft Research. Thank you, Grace and Bryan, for the many fun meetings about AuditBox [109]. Thank you, Francisco, for your friendship, for the fun projects, and for giving me comments on many papers and talks. Thank you, Vyas, for the high-SNR feedback on the projects we worked on together, and on many others. Thank you, Ren, for your mentorship during my time at Intel and for giving me an industry perspective on device interfaces.

Another perk of having Justine as an advisor was joining her research group, which allowed me to meet many wonderful students, including Adithya Philip, Christopher Canel, Ranysha Ware, Isabel Suizo, Yiran Lei, Pratiksha Thaker, Antonis Manousis, Miguel Ferreira, Hamna Shafqat, Zili Meng, Valerie Choung, Kaajal Gupta, Kshitij Rana, Erica Chiang, Fabian Mkocheke, and Rukshani Athapathu. They gave me feedback on talks and drafts, as well as friendship and mentorship, and together we celebrated many paper submissions, acceptances, and rejections. I also shared reading groups, practice talks, and the traditional post-NSDI-submission parties with the other networking students at CMU, especially Anup Agarwal, Sagar Bharadwaj, and Shawn Chen; and the non-networking students who involuntarily became congestion control experts, Ankush Jain, Ioannis Anagnostides, and Aditi Kabra. Thank you for making my time at CMU so great.

I am also grateful to the CMU faculty members who sat through many versions of my talks and gave me incredibly useful feedback: Srinivasan Seshan, Peter Steenkiste, Akshitha Sriraman,

Riccardo Paccagnella, Feras Saad, and Zhihao Jia. Many faculty members outside CMU also took the time to give me feedback and advice: Aditya Akella, Venkat Arun, Derek Chiou, Manya Ghobadi, Antoine Kaufmann, Daehyeok Kim, Eddie Kohler, Philip Levis, Vincent Liu, Akshay Narayan, Deepti Raghavan, Mark Silberstein, Ryan Stutsman, and Minlan Yu.

One aspect of CMU that is mostly hidden from the outside is how outstanding the staff is. Students see almost no red tape and everything just works. Thank you, Deborah Cavlovich, Catherine Copetas, Sara Kuntz, Karen Lindenfelser, Angy Malloy, Emily Spencer, Matthew Stewart, and Charlotte Yano, for your patience and kindness, even as I followed the academic tradition of submitting everything at (or after) the deadline.

Outside of research, my office/gym mates helped me stay strong, mentally and physically: Long Pham, Justin Whitehouse, Yi Zhou, Justin Raizes, Yonghao Zhuang, and Siddharth Prasad. Thank you for making our office the best office in Gates (with a good supply of snacks from every continent, coffee, and matcha) and for helping me keep a consistent gym routine despite the chaos of the PhD. Long, in particular, has been a great friend, listening to my many rants and politely calling out my nonsense. I was also lucky to be part of a wonderful PhD cohort. I especially thank Dorian Chan, Sara McAllister, Steven Jecmen, Jatin Arora, Peter Manohar, Magdalen Dobson Manohar, and Brian Zhang for their friendship and support. During COVID, Dorian, Steven, Peter, Magdalen, Brian, and I formed a bubble so we could hang out and take care of each other—although mostly they took care of me. Dorian and Sara also kept our BBQ tradition alive, which took us to almost every BBQ place in Pittsburgh. At the same time, Jatin and I were on a mission to try every beer in Pittsburgh. Thank you all for allowing me to forget about research for a little while.

I was only able to come to CMU because of my undergraduate and master's advisors at UFRJ, Miguel Campista and Luís Costa. I started doing research with Miguel and Luís in my first year of undergrad, after taking a class with Miguel, and I never stopped. They let me explore a wide range of topics, allowing me to find what I truly love. Thank you, Miguel and Luís, for introducing me to computer networking research and for supporting my ambition to come to the US for a PhD.

I thank my family: my parents, Marcelo and Márcia Sadok; my siblings, Bruno and Luna Sadok; and my grandmother, Carmen Siqueira. Their endless support made all of this possible.

Finally, I thank my wife, Mara Sadok, whose love, kindness, and patience got me through the final years of my PhD. I love you.

Contents

1	Introduction	1
1.1	Limitations of Specialized Device Interfaces	3
1.1.1	Specialization to Data Formats	3
1.1.2	Specialization to Endpoints	4
1.2	Benefits of a General-Purpose Device Interface	5
1.3	Challenges in Generalizing Device Interfaces	6
1.4	Summary of Results	7
1.4.1	Ensō: Generalizing “What” Devices Communicate	7
1.4.2	DCP: Generalizing “Whom” Devices Communicate With	8
1.5	Dissertation Outline	10
2	Background	11
2.1	How Devices Communicate	11
2.1.1	Host Interconnect	12
2.1.2	Coordination Layer	13
2.2	Device Interfaces Specialize Coordination	14
2.2.1	Packetized NIC Interface and Specialization to Data Formats	14
2.2.2	Accelerator Interface and Specialization to Endpoints	16
2.3	Existing Techniques Keep Specialized Coordination	17
2.3.1	RDMA	18
2.3.2	GPU-Initiated Communication	19
3	Generalizing Across Data Formats	21
3.1	Background and Motivation	24
3.1.1	Packetized NIC Interface	24
3.1.2	Issues with a Packetized Interface	26
3.2	Ensō Overview	27
3.2.1	High-Level Operation	28
3.2.2	Ensō Pipe’s Flexibility	28

3.2.3	Performance Advantages of an Ensō Pipe	29
3.2.4	Challenges	29
3.2.5	Target Applications	30
3.3	Efficient Notifications	30
3.3.1	Efficient Ensō Pipe Coordination	30
3.3.2	Pacing Notifications	32
3.4	Rejected Designs for Pointer Updates	34
3.4.1	Rejected Design #1: MMIO Synchronization	35
3.4.2	Rejected Design #2: Shared Memory Synchronization	35
3.4.3	Rejected Design #3: Inline Synchronization	35
3.5	Ensō Implementation	36
3.5.1	Software Implementation	36
3.5.2	Hardware Implementation	37
3.6	Using Ensō	39
3.7	Evaluation	40
3.7.1	Setup and Methodology	40
3.7.2	EnsōGen Packet Generator	40
3.7.3	Microbenchmarks	41
3.7.4	Application Benchmarks	47
3.8	Related Work	52
3.8.1	Direct Application Access	52
3.8.2	Alternative NIC Interfaces	52
3.8.3	Application-Specific Hardware Optimizations	53
3.8.4	Application-Specific Software Optimizations	53
3.9	Conclusion	54
4	Generalizing Across Endpoints	55
4.1	Background and Motivation	58
4.1.1	Target Use-Cases	58
4.1.2	Overheads of CPU-Mediated Communication	59
4.1.3	Limitations of Existing Peer-to-Peer Approaches	61
4.2	DCP Vision	62
4.3	Design	64
4.3.1	Memory-Mapped Queue (MMQ)	64
4.3.2	Data Transmission and Ownership	66
4.3.3	Credit-Based Flow Control	67
4.4	Using DCP	68
4.4.1	CPU Bypass	68
4.4.2	Accelerator Chain	68

4.4.3	Interposed Accelerator	69
4.5	DCP Implementation	69
4.6	Evaluation	70
4.6.1	Evaluated Devices	70
4.6.2	Setup and Methodology	71
4.6.3	CPU Bypass	72
4.6.4	Accelerator Chain	73
4.6.5	Interposed Accelerator	74
4.6.6	Device-to-CPU Communication	75
4.6.7	GPU-to-GPU Communication over NVLink	76
4.6.8	Cost	78
4.7	Related Work	78
4.8	Conclusion	79
5	Future Work	81
5.1	Streaming Network Stacks	81
5.2	Better Message-Passing Primitives	82
6	Conclusion	85
6.1	Lessons Learned	86
6.2	Adoption	87
6.2.1	Adopting Ensō	87
6.2.2	Adopting DCP	88
6.3	Looking Ahead	88
	Bibliography	91

List of Figures

2.1	Example of a host address space with the CPU and three devices. All devices and host memory are assigned a fraction of the address space. PCIe (or another interconnect) forwards reads and writes to host memory or the corresponding device based on the address.	12
2.2	Layers implemented by the endpoints and by the intermediate node in (a) a TCP/IP network and (b) PCIe. The TCP/IP network exposes a send/recv API to applications, while the PCIe network exposes a load/store API to endpoints. Endpoints then use the load/store API to implement a coordination layer on top.	13
2.3	Example of a packetized NIC interface. NICs and other devices communicate over a shared-memory abstraction where both the device and the CPU access each other's memories. The interface dictates how devices coordinate the memory accesses.	15
2.4	Example of an RDMA write operation. Software on Host 1's CPU uses the RDMA NIC's specialized interface to initiate the RDMA write. The RDMA NIC on Host 1 then reads the data using DMA reads and transfers the data to the RDMA NIC on Host 2. Upon receiving the data, the RDMA NIC on Host 2 uses DMA writes to place the data in Host 2's memory. Note how RDMA allows one host to access another's address space through arbitrary reads and writes. This effectively pushes coordination to the application. It is the application's responsibility to decide where to place the data and when it is safe to access it. .	18
3.1	Data structures used to receive packets in a packetized NIC interface. Each packet is placed in a separate buffer that can be arranged arbitrarily in memory.	25
3.2	PCIe bandwidth and cache misses for an application forwarding small packets with a packetized NIC interface (E810).	27
3.3	Steps to <i>receive</i> batches of messages in two Ensō Pipes.	28
3.4	Notification buffer being used to notify the arrival of three chunks of data in an RX Ensō Pipe. Notifications contain the latest Tail _{NIC} for a given Ensō Pipe as well as a flag signal.	31

3.5	Reactive notification mechanism for an Ensō Pipe. It only sends notifications when new data arrives and $\text{status} = 0$, or if software updates Head_{SW} and it is different from Tail_{NIC}	33
3.6	NIC RX Datapath.	37
3.7	NIC TX Datapath.	38
3.8	Raw packet rate. Ensō is bottlenecked by Ethernet while the E810 does not scale beyond two cores. The dashed line represents the 100 Gb Ethernet limit.	42
3.9	Throughput when forwarding packets between two queues. Ensō outperforms zero-copy E810 even when it needs to copy packets.	43
3.10	PCIe bandwidth utilization for read (RD) and write (WR) transactions. The bars represent the overall PCIe bandwidth utilization, while the lines represent the goodput, i.e., the amount of PCIe bandwidth used to transmit packet data. Ensō uses little PCIe bandwidth for metadata, causing it to achieve a higher goodput while consuming less overall PCIe bandwidth.	44
3.11	RTT for different loads when using a notification per packet or reactive notifications without notification prefetching.	46
3.12	RTT for different loads for the E810 as well as Ensō with and without notification prefetching.	47
3.13	Packet rate for different numbers of Ensō Pipes and core counts. The dashed line represents the 100 Gb Ethernet limit.	47
3.14	Sensitivity analysis with different numbers of cores and packet sizes. Ensō is bottlenecked by 100 Gb Ethernet in all scenarios.	48
3.15	Packet rate for the Maglev load balancer under two types of workloads. The dashed line is the 100 Gb Ethernet limit.	49
3.16	MICA throughput with 8B keys and values.	50
3.17	Mean RTT for MICA as a function of offered load.	51
3.18	Log monitor throughput for different target applications.	52
4.1	Use-cases DCP aims to support.	58
4.2	CPU-mediated communication.	60
4.3	DCP does not replace device drivers. Instead, drivers are used by the CPU in the control plane to configure devices and instantiate MMQs. DCP, the protocol that runs between MMQs, minimally standardizes the mechanisms for dataplane operations, i.e., transferring data between devices.	62
4.4	Data transfer modes in DCP.	65
4.5	DCP implementation of the Figure 4.1 dataflows.	68
4.6	Throughput and latency for the CPU-Bypass topology.	72
4.7	Throughput and latency for the Accelerator-Chain topology.	73
4.8	Throughput and latency for the Interposed-Accelerator topology.	74

4.9	Throughput and latency for transfers between the CPU and a single device. . . .	75
-----	---	----

List of Tables

3.1	Effect of chaotic memory accesses on throughput and cache misses. Results are the average of 10 runs, each lasting 10 s.	43
4.1	Metadata overhead per transfer for each interface. For the baseline interfaces, we omit per-batch overheads, such as doorbell register writes and completion notifications.	74
4.2	GPU-GPU latency over NVLink for DCP and Nvidia’s NVSHMEM, with increasing transfer sizes.	76
4.3	Peak GPU-GPU throughput over NVLink, along with the number of SMs and the transfer size that each mechanism requires to reach its peak.	77
4.4	FPGA resource utilization of DCP’s hardware implementation on an AU55C, as a fraction of the FPGA’s resources.	78

Chapter 1

Introduction

Historically, the primary purpose of host interconnects was to connect the CPU to “peripheral devices” [63, 72, 149]. The underlying assumption behind this design is that the CPU is the primary computing device, and peripheral devices simply serve the CPU, allowing it to interface with the outside world. Network Interface Cards (NICs) connect the CPU to the network, graphics cards (GPUs) connect the CPU to displays, and sound cards connect the CPU to speakers and microphones.

Over the last 30 years, the CPU has ceded its monopoly on computation to a growing collection of increasingly capable, and less “peripheral,” devices [62, 182]. Accelerators and compute-capable I/O devices are essential to many of the most important applications today, from AI systems [93] and multi-tenant clouds [5, 54, 191, 201] to efficient video processing on mobile phones [190]. Accelerators train and serve machine learning models [2, 59, 93], encrypt and compress data [86], transcode videos [8, 163, 165], and serialize remote procedure calls (RPCs) [99]. At the same time, long-established I/O devices, such as NICs and disks, have gained compute of their own. Modern NICs offer functionality ranging from simple operations, such as checksum and queue demultiplexing [44, 78], through complex transport protocol implementations [10, 36, 52, 176, 179], to arbitrary computation [85, 132, 135, 151]. Storage devices are moving in the same direction with the standardization of computational storage [142].

While devices have taken over an increasing fraction of the computation inside the host, the *interfaces* they use to communicate have not changed accordingly. Devices communicate over shared memory: the interconnect (e.g., PCIe [150], CXL [37], NVLink [32]) provides the shared-memory abstraction, but it is up to each device to determine how to coordinate access to this memory. The way a device implements coordination is what we call the device’s interface. Although the device alone defines this interface, the interface binds both endpoints: software (or a peer device) must follow the device’s conventions to communicate with it. And every device today defines its own *specialized* interface. Each NIC, for instance, defines a coordination scheme tied to the data format it expects, and the scheme differs not only from those of accelerators but

even from those of NICs from other vendors. Specialized interfaces worked well when devices were simple, each handling a single kind of data and communicating only with the CPU. But they now prevent compute-capable devices from realizing their full potential.

Specialization limits devices on two fronts. Because interfaces are specialized to a particular data format, they constrain “what” devices can communicate: the interface’s format fixes a division of labor between the device and the CPU, limiting the kinds of computation that devices can efficiently express. For instance, a NIC that implements a transport protocol must still split the resulting bytestreams back into packets before sending the processed data to the CPU (§ 1.1.1). And because interfaces are specialized to a particular pair of endpoints, they constrain “whom” devices can communicate with: two devices cannot exchange data directly but must instead go through the CPU. For instance, a NIC cannot deliver incoming data straight to the decryption accelerator that will process it, even when the CPU is not needed in the processing (§ 1.1.2).

A *general-purpose* device interface would remove both limitations. An interface that is not specialized to a particular data format would let a device communicate at whatever level of abstraction its computation operates on—the same NIC could deliver raw packets when the CPU runs the network stack, or complete messages and bytestreams when the NIC itself is responsible for transport. An interface that is not specialized to a particular pair of endpoints would let any two devices communicate directly, without per-pair engineering—a NIC could deliver incoming data straight to the accelerator that consumes it.

Given the clear benefits of general-purpose interfaces, why specialize? The main obstacle in the way of general-purpose device interfaces is the long-held assumption that generality comes at the cost of performance. This is particularly concerning for devices since performance is the main reason we turned to accelerators and to computation in I/O devices in the first place [68].

This dissertation challenges that assumption by introducing general-purpose device interfaces that not only match the performance of specialized interfaces but can outperform them. Empirically, we find that generalizing “what” devices communicate can increase application throughput by up to 6×, by better reflecting today’s division of labor between devices and the CPU. And generalizing “whom” devices communicate with can cut device-to-device communication latency by 83% (from 8.1 μ s to 1.4 μ s), by avoiding CPU mediation.

THESIS STATEMENT

A general-purpose device interface can improve performance and interoperability among hardware accelerators, I/O devices, and CPUs with different capabilities and data formats.

In the rest of this chapter, we expand on this argument. We first discuss how specialized interfaces limit devices (§ 1.1), then detail the benefits of a general-purpose device interface (§ 1.2) and the challenges in designing one (§ 1.3). We conclude with a summary of our results (§ 1.4) and an outline of the remaining chapters (§ 1.5).

1.1 Limitations of Specialized Device Interfaces

In this section, we discuss how specialization to data formats (“what”) and to endpoints (“whom”) limits modern, compute-capable devices.

1.1.1 Specialization to Data Formats

The first limitation of specialized interfaces concerns “what” devices communicate. Existing device interfaces are specialized to a particular format and granularity of data: most NICs expose interfaces built around packets, and most storage devices expose interfaces built around blocks. An interface specialized for a fixed data format reflects a fixed division of labor, delineating where the device’s processing ends and the CPU’s begins. But when devices are capable of performing computation, this boundary is no longer fixed.

Consider NICs as an example. Existing NICs exchange data using a *packetized interface*, which relies on sequences of fixed-size packet buffers, each accompanied by metadata. This design reflects the assumption that the NIC operates at the lower layers of the network stack (L2/L3) while the CPU handles the layers above, including demultiplexing and transport. Indeed, the use of per-packet metadata helps software demultiplex and reassemble flows efficiently.

While this division of labor closely reflects the NIC capabilities of the 1990s,¹ NICs have since evolved dramatically. Virtually all high-performance NICs today can demultiplex incoming data among multiple queues, and many implement transport protocols—or even arbitrary computation in the case of SmartNICs [85, 132, 135, 151]. Not only is the computation flexible, but modern NICs can also apply different computations to different sets of flows. A NIC’s interface must therefore accommodate data at multiple layers at once. But because the interface remains specialized for packets, NICs must split higher-level data back into packet buffers. In addition, each packet buffer still carries metadata to aid demultiplexing that the NIC already performed. This creates extra work exactly where the NIC’s computation was meant to remove it.

The overheads induced by the use of per-packet buffers and metadata are significant. As we will see in more detail in Chapter 3, per-packet metadata can consume as much as 39% of the NIC’s PCIe bandwidth when handling small packets. In addition, since every packet occupies a separate buffer, consecutive packets end up scattered across memory, making it hard for the CPU to prefetch data. This results in an L2 cache miss ratio of up to 55%.

While we focus on the problem of specialization to data formats on NICs, NICs are not unique. More broadly, the more flexible the computation a device performs, the more likely it is to outgrow an interface specialized to a single data format. Storage devices, for instance,

¹Partridge’s description of a state-of-the-art NIC interface in his 1993 book [148] is nearly identical to the one that most modern NICs adopt. Notably, Partridge also rejected the idea of NICs performing computation, on the grounds that it was hard to beat the performance per watt of host CPUs at the time.

face the same pressure: as SmartSSDs [172] gain the ability to process data on the drive, their outputs no longer match the fixed-size blocks their interfaces are built around.

1.1.2 Specialization to Endpoints

The second limitation of specialized interfaces concerns “whom” devices communicate with. Existing device interfaces are specialized to a particular pair of endpoints: the device itself at one end and the CPU at the other. But because modern devices can perform independent computation, they increasingly benefit from communicating directly with other devices—a use case their interfaces were not specialized for.

For instance, one might want to route data coming from a NIC directly to a decryption accelerator, or a camera feed directly to a hardware video encoder. But because interfaces are specialized to interact with the CPU, communication between two devices needs CPU mediation. This mediation imposes extra interconnect round trips, wastes CPU cycles, and puts the CPU back on the critical path—the bottleneck that specialized hardware was meant to remove in the first place. As we will see in more detail in Chapter 4, these overheads are substantial. In our testbed, mediation alone more than doubles the latency of a device-to-device transfer, and routing data through host memory cuts the available bandwidth in half for every device-to-device hop.

Specialization makes CPU mediation hard to avoid. Because each interface is unique, communicating using existing interfaces requires one endpoint of every connection to adapt to the interface of the other. This is precisely what the CPU does today by using a custom driver for every device. But doing so requires general-purpose programmability (to run a different driver for each interface) and exposed memory (to place the buffers and queues that typically sit in host memory). Most devices offer neither. An alternative is to introduce new *hardcoded* interfaces specialized for specific pairs of devices, avoiding the need for the device to offer general-purpose computation. Unfortunately, neither approach scales. Each implementation covers only a single pair of devices, so supporting n devices requires $\Theta(n^2)$ implementations.² In practice, only a handful of device pairs can justify this cost.

We see this effect in practice. The need for GPUs to communicate with NICs without CPU mediation has led to vendor-specific solutions such as GPUNetIO [66, 134]. The existence of these solutions is evidence of the demand for direct communication between devices. However, GPUNetIO also exemplifies both problems above. GPUNetIO leverages the GPU’s general-purpose programmability and exposed memory to take over the CPU’s role on the NIC’s existing specialized interface. This approach works for GPUs precisely because they are among the few devices that offer general-purpose compute and exposed memory. But fixed-function devices, such as encoders and compression engines with little to no addressable memory, cannot follow

²Using the CPU as a mediator keeps the cost at $\Theta(n)$ implementations, since every interface can specialize to the CPU as the other endpoint.

the same strategy. GPUNetIO also exemplifies the scaling problem. It covers a single pair of devices from the same vendor (Nvidia GPUs paired with Nvidia NICs). Supporting other devices requires additional device-specific implementations.

* * *

Specialized interfaces muzzle modern accelerators and I/O devices by preventing them from efficiently expressing their functionality and outputs, and by limiting their ability to interoperate and collaborate. In the next section, we discuss how a general-purpose interface can free devices from such constraints.

1.2 Benefits of a General-Purpose Device Interface

In this dissertation, we make a case for general-purpose device interfaces. By general-purpose, we mean interfaces that make no assumption about the data they carry or about the endpoint devices that use the interface. By not specializing to data formats, the interface no longer prescribes a division of labor between the device and the CPU. And by not specializing to a particular pair of endpoints, the interface no longer determines whom a device can communicate with.

Generalizing across data formats. Because the interface makes no assumption about the format or the granularity of what it carries, it is up to the endpoints to decide how to interpret and segment the bytes sent through the interface. For instance, a NIC that implements transport can use the same interface to deliver bytestreams directly to applications or raw packets to a network stack in software. The device’s capabilities and configuration determine where the device’s processing ends and the CPU’s begins, not the interface.

Generalizing across endpoints. Because the interface makes no assumption about the device at the other end, any two devices that implement it can communicate directly. Neither has to adapt to the other’s interface, avoiding the need for CPU mediation. For instance, a NIC can send incoming data straight to a decryption accelerator, without having to go through the CPU. This avoids spending CPU cycles or waiting on the CPU to issue the transfer to the device. The application logic determines what the next hop should be, not the interface.

Beyond the immediate benefits we described above, a general-purpose device interface also brings additional benefits as it gains widespread adoption:

- *Compose accelerators.* When any device can communicate directly with any other, accelerators can be designed to be composed. This makes finer-grained specialization viable: functionality that is typically bundled in a single accelerator can instead be split across specialized accelerators and leveraged by other devices.

- *Deduplicate engineering effort.* Hardware designers and driver developers can reuse the same mechanisms across different devices, instead of redesigning and revalidating them for each new interface.
- *Reuse tools across devices.* When devices use the same interface, tools built for one device can also be used for another. For instance, we developed a packet-capture tool to aid debugging and have used it to diagnose issues in several devices that use the same interface.
- *Simplify reasoning about device interactions.* When every device communicates through the same interface, the number of possible interactions among devices shrinks. This makes bugs and misconfigurations easier to identify and simplifies modeling and analysis (e.g., proving the absence of deadlocks).
- *Focus research effort on a common target.* When many devices share one interface, improvements to the interface can benefit all devices that implement it. For instance, since most applications on the Internet use TCP, decades of work on improving TCP performance (e.g., congestion control) have benefited many applications.

To realize these benefits, we need interfaces that are simple enough for any device to implement, yet fast enough to be competitive with specialized interfaces. Designing these interfaces is the main topic of this dissertation.

1.3 Challenges in Generalizing Device Interfaces

The main challenge in designing a general-purpose device interface is performance. The same assumptions that limit specialized interfaces also contribute to their efficiency. In this section, we examine where this efficiency comes from and the challenges in keeping it without specialization.

Recall that devices communicate over shared memory. Their interfaces typically work by having one endpoint write data to a region of memory that the other endpoint reads. Communicating over shared memory requires *coordination*. The endpoints must agree on how memory is allocated (where new data may be written and when memory can be reused), and they must determine when it is safe to access data (the producer must know when it is safe to write to the memory; and the consumer must know when it is safe to read the memory). Coordination adds overhead. While specialized interfaces can leverage specificities of the data format or of the endpoint devices to reduce this overhead, a general-purpose interface cannot rely on these specificities.

Coordinating without a fixed data format. Interfaces specialized to a particular data format, like packetized interfaces in NICs and block-based interfaces in storage devices, can leverage the known granularity of the data they handle. NICs use the fact that packet sizes cannot exceed the network’s MTU; and storage devices use the fact that blocks have fixed sizes. Because no unit of data exceeds the known maximum size, software can preallocate a pool of fixed-size

buffers and hand those to the device ahead of time. This allows memory to be allocated *before* the output data is ready, reducing coordination overhead on the critical path. In addition to allowing preallocation, making the interface aware of the data’s granularity also helps determine when it is safe to access the data (e.g., on NIC interfaces the producer can tell the consumer to access the data whenever a complete packet is ready). Once we remove the assumption about data format, the interface can no longer preallocate fixed-size buffers, and it is no longer clear when the producer should notify the consumer of incoming data.

Coordinating without the CPU. Interfaces specialized to communicate with a particular device and the CPU can leverage the asymmetry between the two endpoints. CPUs are better suited to *expose* memory, since they have access to large amounts of host DRAM, while devices are better suited to *access* this memory, since they are capable of accessing memory asynchronously through DMA. Specialized interfaces divide the coordination work accordingly. They typically keep the data in host DRAM and have the CPU act as a centralized coordinator, responsible for allocating memory and orchestrating transfers (telling devices when the inputs are ready and where to place the outputs). For instance, in the packetized interface, the CPU exposes the packet buffers and the queues used for coordination in host memory, and the NIC uses “DMA writes” to send data to the CPU and “DMA reads” to receive data. Once we remove the assumption that the endpoints are fixed, the interface can no longer be asymmetric. It cannot assume that either endpoint exposes memory to hold the data, and it cannot rely on a particular endpoint to be the coordinator. Coordination can no longer be centralized at the CPU. It must be distributed among the endpoints.

1.4 Summary of Results

We address the challenges of providing coordination without specialization by introducing two systems. Each system generalizes one of the two sources of specialization: Ensō generalizes “what” devices communicate, by enabling coordination without fixed data formats, while DCP generalizes “whom” they communicate with, by enabling coordination without the CPU. Beyond demonstrating that the coordination challenges can be solved, the two systems empirically support the thesis statement by showing that general-purpose interfaces can *improve* performance relative to specialized interfaces.

1.4.1 Ensō: Generalizing “What” Devices Communicate

To coordinate without a fixed data format, we design and implement Ensō (Chapter 3). Ensō replaces the fixed-size buffers of existing interfaces with a streaming abstraction: the device and the CPU exchange data by appending it to an Ensō Pipe, a shared data ring buffer that imposes

no format or size on its contents. Devices can then use the stream to transmit whatever data suits their functionality (e.g., raw packets, application-level messages, bytestreams).

Streaming addresses the performance limitations of the packetized interface without relying on a particular data format. Two of Ensō’s design choices enable this:

Sequential placement. With a stream, we write new data in the same buffer right after the end of the previous data. This avoids the need to break incoming data across fixed-size buffers. Moreover, since subsequent data is placed contiguously in memory, it is easy for the CPU to predict what the next access will be. In our benchmarks, sequential accesses alone reduce L2 cache misses by 99.5% compared to scattered accesses.

Reactive notifications. A stream also lets the endpoints coordinate when it is safe to access data without relying on the data format. Ensō uses notifications to report how far the stream has advanced, so a single notification can cover an arbitrary amount of data. Ensō paces these notifications *reactively*: the NIC withholds the next notification until software has consumed the previous one. This mechanism slows down notifications when software is slow to process data and speeds them up when it keeps up, naturally adapting to the pace of both endpoints. And because notifications no longer need to accompany every packet, Ensō reduces PCIe metadata traffic by 96.9%, eliminating the PCIe bottleneck experienced by the packetized interface when using small transfers.

Implementation and evaluation. We implement Ensō on an FPGA-based 100 Gbps SmartNIC and port four applications—a load balancer, a network telemetry application, a key-value store, and a log monitor—to measure its end-to-end impact. The main results are as follows:

- Ensō outperforms the packetized interface even when transferring packets. With a single CPU core, Ensō saturates a 100 Gbps link with minimum-size packets (achieving 148.8 Mpps). This is 2.5× the rate achieved by a state-of-the-art packetized NIC [78].
- Applications benefit from the performance improvements. Compared to the packetized interface, our port of a load balancer [48] achieves up to 6× higher throughput, and our port of the MICA key-value store [107] achieves up to 43% lower latency.
- Surprisingly, Ensō can outperform the packetized interface even when software needs to demultiplex packets (where the packetized interface’s use of individual packet buffers avoids a copy). A simple packet forwarding application still achieves 28% higher throughput than its zero-copy equivalent using the packetized interface.

1.4.2 DCP: Generalizing “Whom” Devices Communicate With

To coordinate without the CPU, we design and implement DCP, a general-purpose Device Communication Protocol (Chapter 4). DCP is built around a new queue primitive called the *Memory-Mapped Queue* (MMQ). An MMQ is a region of the address space that, when written to,

enqueues a command at the device it belongs to.

MMQs let any pair of devices coordinate directly, regardless of their capabilities. Two of DCP’s design choices enable this:

Symmetric interface. MMQs decouple what senders see from how receivers implement it. From the sender’s perspective, every MMQ looks the same: a ring buffer of fixed-size slots into which the sender writes commands. But from the receiver’s perspective, an MMQ can be implemented in two different ways: devices with exposed memory and programmability, such as CPUs and GPUs, can implement MMQs using exposed addressable memory, while fixed-function devices can implement MMQs using hardware queues and fixed logic. The hardware queue implementation requires neither exposed memory nor programmability and consumes few hardware resources (around 4% of an AU55C FPGA [7]).

Distributed coordination. DCP then builds upon the MMQ primitive and establishes a small set of commands to replace CPU coordination with distributed coordination:

- *Pass-by-value* transfers inline data directly in the command, allowing direct device-to-device communication even when neither endpoint exposes memory.
- *Pass-by-reference* transfers carry a pointer to memory anywhere in the host address space. The data can reside in the sender, the receiver, CPU-attached DRAM, or CXL memory. This allows transfers to make the best use of the memory available in the system.
- *Buffer requests* let a sender ask the receiver to allocate a buffer that the sender then writes into. This supports pass-by-reference transfers in which the receiver, not the sender, owns the memory.

In addition to these commands, DCP employs credit-based flow control, letting receivers control the rate at which senders can enqueue commands. This provides backpressure without a centralized coordinator.

Implementation and evaluation. We implement DCP in software (targeting both CPUs and GPUs) and in hardware (synthesizing and running on FPGAs), evaluating it on three device topologies that would typically require CPU mediation today. The main results are as follows:

- Direct communication through DCP more than doubles goodput compared to CPU mediation (119 Gbps, 97% of the interconnect’s theoretical maximum, versus 51 Gbps) and cuts latency by 83% (from 8.1 μ s to 1.4 μ s).
- Even when communicating with the CPU, DCP achieves goodput and latency comparable to those of specialized NIC and accelerator interfaces—the use case they were designed for. DCP matches or exceeds their goodput, and its latency matches the NIC’s with pass by reference and stays within about 2 μ s of the accelerator’s.
- For GPU-to-GPU communication over NVLink [32], DCP achieves 20% higher throughput than Nvidia’s specialized interface (NVSHMEM [137]) while using a quarter of the GPU

resources (SMs). DCP’s latency is within $1.3\ \mu\text{s}$ of the specialized interface’s for transfers up to 4 KB and up to $9.9\ \mu\text{s}$ *lower* for larger transfers.

1.5 Dissertation Outline

This dissertation is organized as follows. In Chapter 2, we introduce background on how devices communicate and explain how existing techniques cannot solve the issues caused by specialized interfaces. In Chapter 3, we present Ensō, our design for generalizing “what” devices communicate. Then, in Chapter 4, we present DCP, our design for generalizing “whom” devices communicate with. Chapter 5 discusses immediate directions for future work that build on these two systems. Finally, Chapter 6 concludes with broader lessons learned, the path to adoption, and a reflection on the role of hardware interfaces in an evolving architecture landscape.

The content in Chapter 3 is adapted from [169], and the content in Chapter 4 is adapted from [171].

Chapter 2

Background

In Chapter 1, we argued that device interfaces are specialized to data formats and endpoints. In this chapter, we show that this specialization comes from the way devices implement *coordination*. Devices and the CPU communicate over a host interconnect, but unlike a typical TCP/IP network, the host interconnect exposes a shared-memory abstraction instead of message passing. Shared memory lets any endpoint read and write the memory of any other endpoint, but it requires the endpoints to decide *where* to place data, *when* it is safe to access it, and *what* function to apply to it. Each device makes these decisions differently when defining its interface, assuming a specific data format and a specific pair of endpoints.

We first describe how devices communicate over the host interconnect and why this requires a coordination layer (§ 2.1). We then use a NIC and an accelerator to show how device interfaces specialize coordination to data formats and to endpoints (§ 2.2). Finally, we show that RDMA and GPU-initiated communication, the most prominent techniques for working around specialized interfaces, are not widely applicable since they still rely on specialized coordination (§ 2.3).

2.1 How Devices Communicate

In this section, we describe how devices typically communicate inside the host. We contrast existing host interconnects with a typical TCP/IP network, highlighting how they expose a different abstraction to the endpoints: shared memory instead of message passing (§ 2.1.1). We then show how this difference requires endpoints in a host interconnect to implement a coordination layer (§ 2.1.2).

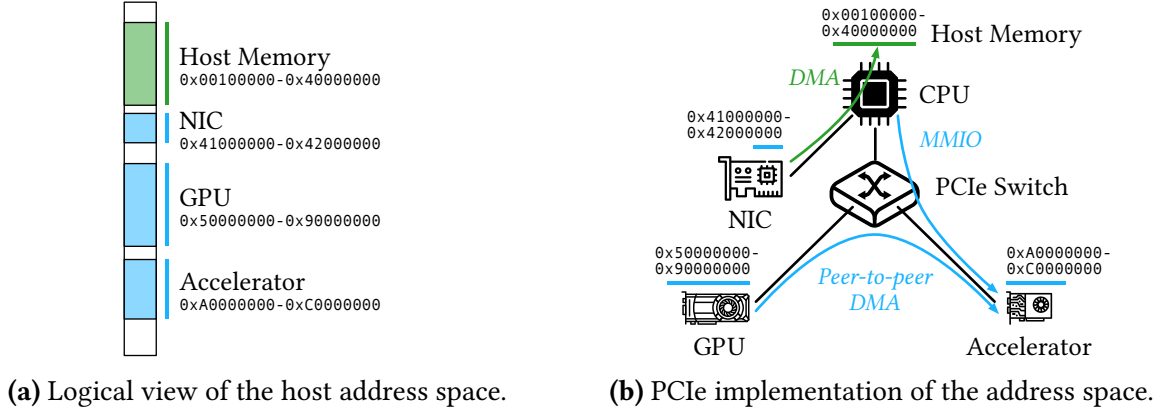


Figure 2.1: Example of a host address space with the CPU and three devices. All devices and host memory are assigned a fraction of the address space. PCIe (or another interconnect) forwards reads and writes to host memory or the corresponding device based on the address.

2.1.1 Host Interconnect

CPUs and devices today communicate over a network inside the host called the *host interconnect* (e.g., PCIe [150], CXL [37], and NVLink [32]). Like a traditional packet-switched network based on TCP/IP, host interconnects provide connectivity among all endpoints inside the host. One important difference between TCP/IP networks and host interconnects is the abstraction they expose to the upper layer. Applications over TCP/IP communicate using a message-passing abstraction (e.g., sockets) that allows data to be transferred using send and receive calls. But endpoints in a modern host interconnect communicate using a shared-memory abstraction, where endpoints have access to a single address space and share data by reading from and writing to it.

Figure 2.1 shows an example of how the host and different devices have their memory mapped to different regions of the same address space. PCIe, or another host interconnect, forwards reads and writes to the right endpoint depending on the memory address they need to access. Software running on the CPU can access the memory of any device by issuing regular load and store instructions, which we refer to as Memory-Mapped I/O (MMIO). Devices can also access any memory in the address space, independently of the software that runs on the CPU, which we refer to as Direct Memory Access (DMA). Because the interconnect is capable of forwarding reads and writes based solely on the memory address, it does not differentiate between MMIO accesses and DMA accesses. Moreover, this means that devices are capable of using DMA to access memory from other devices without involving the CPU (peer-to-peer DMA).

Other interconnects, such as CXL and NVLink, expose the same shared-memory abstraction

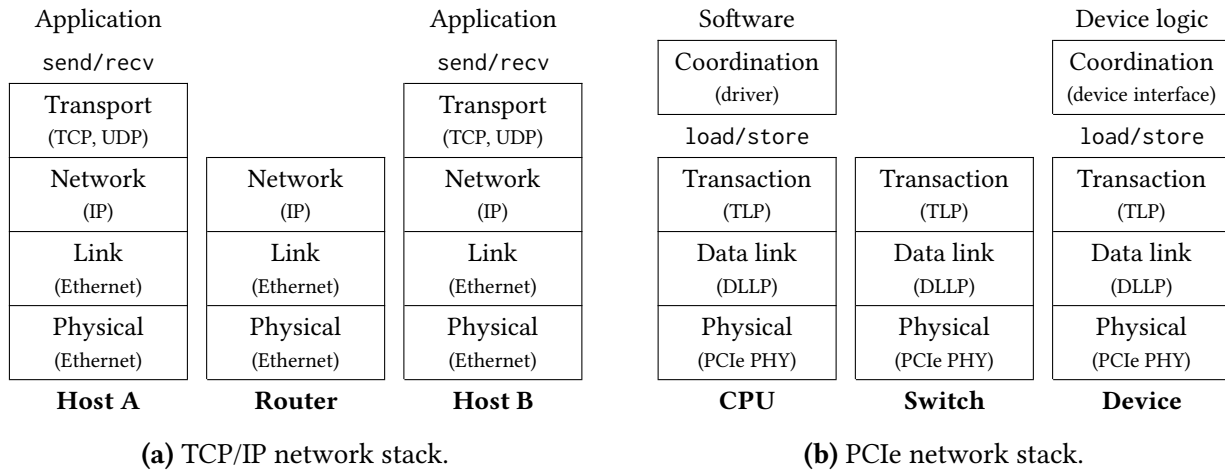


Figure 2.2: Layers implemented by the endpoints and by the intermediate node in (a) a TCP/IP network and (b) PCIe. The TCP/IP network exposes a send/recv API to applications, while the PCIe network exposes a load/store API to endpoints. Endpoints then use the load/store API to implement a coordination layer on top.

as PCIe. NVLink is a proprietary interconnect from Nvidia and is typically used to connect Nvidia GPUs at a significantly higher bandwidth than PCIe.¹ CXL builds on PCIe and adds cache coherence, allowing endpoints to safely cache data from memory addresses that belong to other endpoints, while keeping the shared-memory abstraction.

2.1.2 Coordination Layer

A fundamental consequence of using a shared-memory abstraction is that *coordination* must be defined *explicitly* at an upper layer. Devices and the CPU must carefully synchronize their accesses to the shared memory to ensure correct operation. Implementing coordination means that endpoints must agree on the answers to the following three questions:

- **Where in memory to place the data?** The endpoints must agree on which addresses the sender may write to and the receiver may read from.
- **When can the memory be accessed?** The sender must know when it is safe to write the data, and the receiver must know when it is safe to read the data.
- **What function does the receiver apply to the data?** Upon receiving the data, the receiver must know which operations to apply to it.

¹The best NVLink devices available on the market today (NVLink 5) support up to 900 GB/s per direction [130], while the best available PCIe devices (PCIe 6) support up to 121 GB/s [150].

Contrast this with message passing, where endpoints cannot customize coordination, since it happens *implicitly* at the layers below. For instance, in a TCP/IP network, senders and receivers do not need to agree on *where* to place the data in the receiver’s memory; coordination over *when* it is safe to transmit data is part of the protocol (i.e., flow control in TCP); and we define *what* function to run by associating a program with a socket.

To illustrate the difference, compare the network stacks of a typical TCP/IP network and of PCIe (Figure 2.2). The TCP/IP network stack exposes a message-passing abstraction to the upper layer with an API based on `send/recv`. In contrast, PCIe exposes a shared-memory abstraction with an API based on `load/store`. As a result, PCIe endpoints need to implement a *coordination layer* on top in order to explicitly answer the three questions above. One feature that distinguishes the coordination layer from all the other layers in the example is that it is not implemented using a standard protocol. Instead, every unique pair of endpoints implements its own coordination layer.

To implement coordination, devices define an *interface*. A device interface answers the three coordination questions by describing how the other endpoint should use loads and stores to communicate with the device. Software then implements its own coordination layer by using a custom driver that follows the coordination mechanisms specified by the interface. We describe device interfaces in more detail in the next section.

2.2 Device Interfaces Specialize Coordination

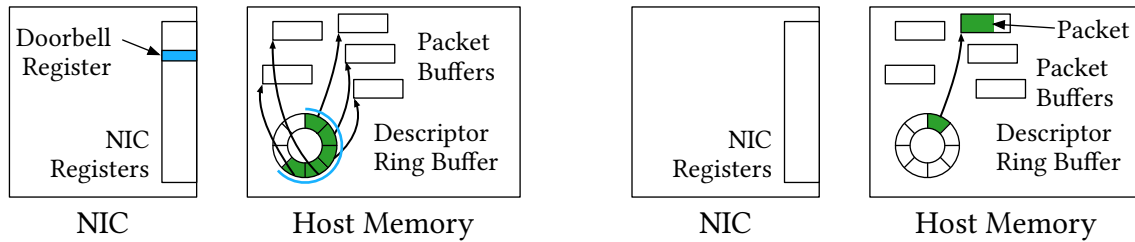
We now use the example of two device interfaces to demonstrate how they rely on specialization to data formats and endpoints to answer the three coordination questions from § 2.1.2. We use a NIC interface to illustrate specialization to data formats (§ 2.2.1) and the QAT accelerator interface to illustrate specialization to endpoints (§ 2.2.2).

2.2.1 Packetized NIC Interface and Specialization to Data Formats

Most NICs today expose a *packetized interface* [156], which specializes to exchange packets with software on the CPU.

Interface

As illustrated in Figure 2.3, the interface requires software to allocate two structures in host memory: a pool of *packet buffers*, each large enough to hold an MTU-sized packet, and a *descriptor* ring buffer, with descriptors pointing to the buffers. In addition to the structures in host memory, the NIC exposes *doorbell* registers as part of its own region in the address space.



(a) Software posts empty buffers to the NIC by writing descriptors with the buffer addresses to the descriptor ring buffer and writing how far it has filled the ring buffer to the NIC's doorbell register (using MMIO). The NIC can then read the descriptors from host memory (using DMA).

(b) When a packet arrives from the network, the NIC writes it to host memory (using DMA) and notifies software by writing a new descriptor to the descriptor ring buffer (using DMA). Software detects the new descriptor by polling for a change in the descriptor's flag bit.

Figure 2.3: Example of a packetized NIC interface. NICs and other devices communicate over a shared-memory abstraction where both the device and the CPU access each other's memories. The interface dictates how devices coordinate the memory accesses.

Whenever software writes new descriptors to the descriptor ring buffer, it also writes to the doorbell register, notifying the NIC that there are new descriptors to be consumed.

Where to place data. Software and the NIC use the descriptor ring buffer to exchange packet buffer addresses. Before receiving data, software posts addresses of empty buffers to the NIC and, upon receiving data, the NIC posts addresses of filled buffers to software.

When to access memory. Besides allowing the software and the NIC to exchange buffer addresses, descriptors also tell the endpoints when it is safe to access the memory. When software posts empty buffers to the NIC, it tells the NIC it is safe to use those buffers to write incoming data; and when the NIC posts filled buffers to software, it tells software it is safe to consume the data in these buffers.

What function to apply. The interface also helps determine which operations the NIC should apply to each packet. When transmitting packets, software can specify which offloads (e.g., checksum) the NIC should run for the corresponding packet by setting flags in the descriptor.

Specialization

Note how the interface ties coordination to the data format by assuming that it exclusively carries packets. To determine *where* to place the data, software preallocates buffers sized so that they can fit a maximum-sized packet. To determine *when* to access memory, the interface relies on packet boundaries, sending a notification whenever a complete packet is available. Finally,

to determine *what* function to apply, the interface also relies on packet granularity, determining offloads on a packet-by-packet basis.

Specializing for packets worked when NICs performed little computation. But modern NICs are capable of demultiplexing packets among multiple queues [44, 78], implement transport protocols [10, 36, 52, 176, 179], and even run arbitrary computation [85, 132, 135, 151]. Because the interface still exchanges data one packet at a time, it introduces significant overheads for these NICs. As we will see in more detail in Chapter 3, NICs that demultiplex packets still place each packet in a separate buffer, even when software processes the packets in a queue sequentially, which leads to poor cache performance. Moreover, NICs that could deliver data as higher-level constructs, such as application-level messages or bytestreams, must break the data back into packet-sized chunks, each with its own descriptor, consuming PCIe bandwidth and CPU cycles to process the extra metadata.

While we illustrate specialization to data formats using a NIC interface, other device interfaces are specialized in the same way: storage devices transfer fixed-size blocks [143], display controllers consume fixed-size frames [76], and audio devices exchange fixed-size sample buffers [73]. In each case, software allocates buffers sized for the expected data format, and the device notifies software using the data format as a boundary.

2.2.2 Accelerator Interface and Specialization to Endpoints

Accelerators typically expose a *command queue interface*. We use QAT [86, 87], Intel’s cryptography and compression accelerator, as an example, but other accelerators use a similar model.

Interface

Like the packetized NIC, QAT coordinates using ring buffers in host memory and doorbell registers, but it answers the three questions slightly differently. To determine *where* to place the data, software does not post buffers ahead of time. Instead, each request that software writes to a *request* ring buffer includes the addresses of the input and output buffers. To determine *when* to access memory, QAT uses a separate *response* ring buffer. Software notifies QAT of new requests through the doorbell, and QAT writes a response once the output is ready. To determine *what* function to apply, each request points to a *content descriptor* that software sets up in host memory for each session, holding the algorithm and the key to use.

Specialization

Note how the QAT interface specializes to communicate with the CPU. To determine *where* to place the data, the interface assumes that the other endpoint exposes memory for the required

data structures, and that it tells the device where to read the inputs and write the outputs. To determine *when* to access memory, the interface assumes that the other endpoint tells the device when it is safe to read the inputs and write the outputs, by writing requests to memory and writing to the device’s doorbell. Finally, to determine *what* function to apply, the interface assumes that the other endpoint tells the device which operation to perform, by setting up content descriptors and building each request in QAT’s format. The CPU satisfies these assumptions because it exposes large amounts of memory and is capable of running general-purpose software, but most devices do not.

The interface is also specialized to QAT. Its use of content descriptors to specify algorithms and keys, for instance, would not be useful for an ML accelerator. More generally, every device defines its own interface, and software on the CPU needs a separate driver for each one. Even two NICs that implement the same packetized design use different descriptor formats, ring semantics, and registers [78, 117].

Some device classes have converged on cross-vendor standards, such as NVMe for storage [141] and HDA for audio [73], which let devices from many vendors share a single interface. But these standards are still specialized, since each applies to a single class of devices and still assumes the CPU at the other end. As a result, letting two devices communicate directly requires a separate implementation for each pair of devices.

Specializing for the CPU worked when devices communicated only with the CPU. But as devices take on more of the computation, they increasingly benefit from communicating directly with each other. For instance, we might consider connecting a NIC to the QAT accelerator, allowing incoming data to be decrypted. Because both the QAT and the NIC interfaces expect the CPU at the other end, the CPU must mediate every transfer. As we will see in more detail in Chapter 4, CPU mediation adds interconnect round trips and forces data through host memory. Allowing devices to communicate directly instead reduces device-to-device latency from 8.1 μ s to 1.4 μ s and increases goodput from 51 Gbps to 119 Gbps.

2.3 Existing Techniques Keep Specialized Coordination

The need for devices to communicate using different data formats and with different endpoints has produced a number of techniques that try to work around the limitations of today’s specialized interfaces. We examine the most prominent of them, RDMA (§ 2.3.1) and GPU-initiated communication (§ 2.3.2), and show that they still rely on specialized coordination, which only allows them to work in narrow settings.

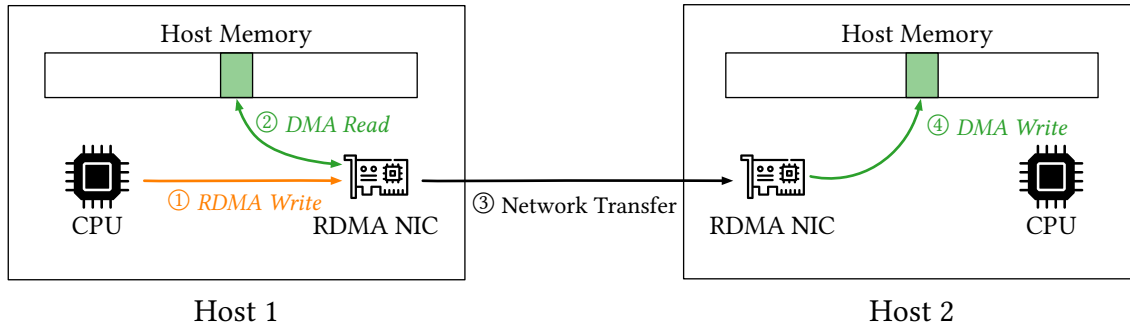


Figure 2.4: Example of an RDMA write operation. Software on Host 1’s CPU uses the RDMA NIC’s specialized interface to initiate the RDMA write. The RDMA NIC on Host 1 then reads the data using DMA reads and transfers the data to the RDMA NIC on Host 2. Upon receiving the data, the RDMA NIC on Host 2 uses DMA writes to place the data in Host 2’s memory. Note how RDMA allows one host to access another’s address space through arbitrary reads and writes. This effectively pushes coordination to the application. It is the application’s responsibility to decide where to place the data and when it is safe to access it.

2.3.1 RDMA

Remote Direct Memory Access (RDMA) [164] is a technique that allows a host to read and write another host’s address space directly. Figure 2.4 illustrates how RDMA is implemented using a pair of RDMA NICs: software on the CPU submits read and write transactions through the RDMA NIC’s own specialized interface, the NIC forwards them to a peer RDMA NIC on the remote host, and that NIC applies them to its local address space using DMA, returning a response in the case of reads.

We now describe how one might consider using RDMA to work around the two forms of specialization we discussed in § 2.2:

Data formats. One can use RDMA to carry data of any format. A write can send an arbitrary number of bytes to an address at the remote host that the sender picks, so the sender can place data of any size and layout in the remote host’s memory. The remote host registers the memory region in advance, but it does not need to preallocate buffers sized for a particular unit of data.

But if RDMA can carry any data format, why is it not a solution? For the same reason PCIe by itself is not. RDMA extends the shared-memory abstraction described in § 2.1 across hosts, but it does not implement a coordination layer. As a result, RDMA pushes coordination to the application: systems built using RDMA implement their own specialized coordination layer in

software, at both ends [45, 112, 120, 121, 196].² Because each system defines its own coordination layer, the two ends must also be built together: only peers with compatible coordination layers can communicate. In practice, this couples the two endpoints, ruling out many important networked applications (e.g., web servers and load balancers) and requiring endpoints to trust each other (otherwise they risk breaking the established coordination mechanism). Finally, RDMA is a NIC feature. Other devices still communicate through their own interfaces, and RDMA does not help them avoid specialization to data formats.

Endpoints. RDMA does not change the endpoints that existing device interfaces are specialized for, but it can be used to move data from one device’s memory directly to the memory of another device located in a different host. Nvidia’s GPUDirect RDMA [133] works this way: an RDMA NIC reads data out of a GPU’s memory and transfers it to a remote RDMA NIC, which writes it into the memory of a GPU on the remote host. This avoids a copy through host memory, but it does not remove the CPU from the transfer. Software on the initiating host must still submit every read and write transaction to its local RDMA NIC, and software at both ends must still implement a coordination layer, deciding where data may be placed and when it is safe to access it. The technique is also narrow. Because RDMA is implemented by NICs, it only moves data between hosts, and two devices in the same host must still communicate through the CPU. It also requires both devices to expose memory. GPUs do, but many devices expose no addressable memory.

* * *

RDMA is not a device interface. It bridges interconnects of different hosts, carrying reads and writes between two address spaces without supplying coordination, just as PCIe does within a single host. The device interfaces, including the one an RDMA NIC exposes, remain specialized. RDMA therefore helps only when both endpoints are programmable, and only when they are designed together, since each pair of endpoints must agree on the coordination mechanism.

2.3.2 GPU-Initiated Communication

Another way to allow devices to communicate directly is to have them initiate the transfer without the CPU being involved. In the RDMA mechanisms of § 2.3.1, software running on the CPU still needs to submit every transfer. A growing number of solutions have GPUs implement

²RDMA also provides two-sided operations, which expose a message-passing abstraction and, as such, do not require the application to implement coordination. However, the interface that RDMA NICs expose for them resembles the packetized NIC interface we describe in § 2.2.1. It uses fixed-size buffers posted in advance and couples completions with messages. The overheads of this interface are such that some systems opt for one-sided operations, even when they need message-passing semantics [45].

drivers, taking over the coordination layer that the CPU typically handles today. GPUs are among the few devices that can replace the CPU side of the coordination because they expose memory and are capable of general-purpose computation. For instance, to communicate directly with a NIC, GPUs can use their exposed memory to hold a NIC’s ring and packet buffers. They can also rely on their ability to run general-purpose computation to execute arbitrary drivers.

Solutions such as NVSHMEM [137] provide a GPU-to-GPU interface, letting a CUDA kernel write data into the memory of another GPU without CPU mediation. Other solutions such as GPUNetIO [66, 134] provide a GPU-NIC interface, letting CUDA kernels drive the NIC’s queues directly. Removing the CPU from the transfer improves performance substantially: for small messages, GPU-initiated writes achieve up to 9.5× the throughput obtained with CPU mediation [114].

While these solutions can dramatically improve performance, they have two limitations. First, they work only when one of the endpoints can replace the CPU. NVSHMEM and GPUNetIO rely on the GPU’s exposed memory and general-purpose programmability to move the driver that implements the coordination layer from the CPU to the GPU. Most devices offer neither. Second, they remain specialized to a specific pair of endpoints. NVSHMEM is specialized for communication between pairs of Nvidia GPUs, and GPUNetIO for communication between an Nvidia GPU and an Nvidia NIC. Each unique pair of devices therefore needs its own implementation, requiring $\Theta(n^2)$ implementations to support n different devices. This scales worse than specializing all interfaces to communicate with the CPU, which requires only $\Theta(n)$ implementations, one for each device. Consequently, direct peer-to-peer communication remains restricted to the handful of device pairs in which at least one device can replace the CPU and whose vendors can justify the implementation cost.

* * *

What existing techniques lack is a general-purpose coordination layer. A coordination layer that depends on neither the data format nor the endpoints would let devices use the data format that best fits their computation, and communicate directly with other devices without the overheads of CPU mediation. We tackle these two sources of specialization in the next two chapters: Ensō (Chapter 3) introduces a device interface that removes the dependence on the data format, and DCP (Chapter 4) introduces a protocol that removes the dependence on the endpoints.

Chapter 3

Generalizing Across Data Formats

This chapter addresses the first front of specialization for existing device interfaces: “what” devices communicate. As we discussed in Chapter 1, interfaces specialized to a data format rely on the format’s known granularity to implement coordination, which creates problems when data deviates from the format. This is increasingly an issue for devices that perform computation. The more flexible a device’s computation, the more freedom the device has to produce and consume data in different formats.

NICs are a clear example of this problem: as link speeds outpaced CPU scaling, NICs absorbed increasing amounts of computation. Network performance dictates application performance for many of today’s distributed and cloud computing applications [98]. While growing application demands have led to a rapid increase in link speeds from 100 Mbps [61] in 2003 to 100 Gbps in 2020 [201] and 200 Gbps in 2022 [118], a slowdown in CPU scaling has meant that applications often cannot fully utilize these links. Consequently, recent changes to NICs and networked software have focused on reducing the number of CPU cycles required for communication: NIC offloads allow the NIC to perform common tasks (e.g., segmentation) previously implemented in software; and more efficient network I/O libraries and interfaces, including DPDK and XDP, allow applications to reduce processing in the network stack. Despite these changes, utilizing 100 Gbps or 400 Gbps links is still challenging. We demonstrate that this is because of inefficiencies in how software communicates with the NIC. While NICs and the software that communicates with them have themselves changed significantly in the last decade, the NIC-to-software interface has remained unchanged for decades.¹

Most NICs currently provide an interface specialized for packets, where all communication between software and the NIC requires sending (and receiving) a sequence of fixed-size buffers. We refer to these buffers as *packet buffers* in this dissertation. Packet buffer size is dictated by

¹Osiris [46], published in 1994, describes an interface that is nearly identical to the one adopted by many modern NICs.

software, and is usually chosen to be large enough to fit MTU-sized packets, e.g., Linux uses 1536-byte packet buffers (`sk_buffs`) and DPDK [43] uses 2 KB packet buffers (`mbufs`) by default. We use the term *packetized NIC interface* to refer to any NIC-to-software interface that uses packet buffers for communication. We observe that two changes in how NICs are used today have led to an impedance mismatch with packetized interfaces.

First, many NIC offloads such as TCP Segmentation Offload (TSO) [44, 78], Large Receive Offload (LRO) [36], serialization offloads [89, 159, 198], and transport offloads [10, 36, 52, 176] take inputs (and produce outputs) that can span multiple packets and vary in size. In using these offloads with a packetized interface, software must needlessly split (and recombine) data into multiple packet buffers when communicating with the NIC.

Second, software logic for sending (and receiving) packets uses batches of multiple packets to reduce I/O overheads. In the common case, NICs and software process packets in a batch sequentially. However, packetized interfaces cannot ensure that packets in a batch are in contiguous and sequential memory locations, reducing the effectiveness of several CPU and I/O optimizations.

This mismatch between how modern NICs are used and what the packetized interface provides causes three problems that affect application performance:

Packetized abstraction. While imposing fixed-size buffers works reasonably well when software *always* needs to exchange MTU-sized packets, it becomes clumsy when used with higher-level abstractions such as application-level messages (e.g., RPCs), bytestreams, or even simpler offloads such as LRO. When using this interface, the NIC (or software) must split messages that are larger than the packet buffer into multiple packet buffers. Applications then need to deal with input that is split across multiple packet buffers. Doing so either requires that they first copy data to a separate buffer, or that the application logic itself be designed to deal with packetized data. Indeed, implementing any offload or abstraction that deals with more than a single packet’s worth of data (e.g., transport protocols, such as TCP, that provide a bytestream abstraction) in a NIC that implements the packetized interface requires copying data from packet buffers to a stream. This additional copy can add significant overhead, negating some of the benefits of such offloads [161, 200].

Poor cache interaction. Because the packetized interface forces incoming and outgoing data to be scattered across memory, it limits the effectiveness of prefetchers and other CPU optimizations that require predicting the next memory address that software will access—a phenomenon that we refer to as *chaotic memory access*. As we show in § 3.7, chaotic memory accesses can significantly degrade application performance, particularly for applications that deal with small requests such as object caches [21, 116] and key-value stores [12, 107].

Metadata overhead. Since the packetized interface relies on per-packet metadata, it spends a significant portion of the PCIe bandwidth transferring metadata—as much as 39% of the available bandwidth when using small messages. This causes applications that deal with small requests

to be bottlenecked by PCIe, which prevents them from scaling beyond a certain number of cores. The use of per-packet metadata also contributes to an increase in the number of memory accesses required for software to send and receive data, further reducing the cycles available for the application. We observed scalability issues due to this PCIe bottleneck in our implementation of Google’s Maglev Load Balancer [48].

In this chapter, we describe Ensō, a new interface for NIC-application communication that breaks from the lower-level concept of packets. Instead, Ensō provides a streaming abstraction that the NIC and applications can use to communicate arbitrary-sized chunks of data. Doing so not only frees the NIC and application to use arbitrary data formats that are more suitable for the functionality implemented by each one but also avoids the performance issues present in the packetized interface. Because Ensō makes no assumption about the data format itself, it can be repurposed depending on the application and the offloads enabled on the NIC. For instance, if the NIC is only responsible for multiplexing/demultiplexing, it can use Ensō to deliver raw packets; if the NIC is also aware of application-level messages, it can use Ensō to deliver entire messages and RPCs to the application; and if the NIC implements a transport protocol, such as TCP, it can use Ensō to communicate with the application using bytestreams.

To provide a streaming abstraction, Ensō replaces ring buffers containing *descriptors*, used by the current NIC interface, with a ring buffer containing *data*. The NIC and the software communicate by appending data to these ring buffers. Ensō treats buffers as *opaque* data, and does not impose any requirements on their content, structure, or size, thus allowing them to be used to transfer arbitrary data, whose size can be as large as the ring buffer itself. Ensō also significantly reduces PCIe bandwidth overhead due to metadata, because it is able to aggregate notifications for multiple chunks of data written to the same buffer. Finally, it enables better use of the CPU prefetcher to mask memory latency, thus further improving application performance.

Although the insight behind this design is simple, it is challenging to implement in practice. For example, CPU-NIC synchronization can easily lead to poor cache performance: any approach where the NIC and CPU poll for changes at a particular memory location will lead to frequent cache invalidation. Ensō avoids this obstacle by relying on explicit notifications for CPU-NIC synchronization. Unfortunately, explicit notifications require additional metadata to be sent over the CPU-NIC interconnect, which can negate any benefits for interconnect bandwidth utilization. Ensō mitigates this overhead by sending notifications *reactively*. We discuss our synchronization strategy in detail, as well as other challenges to the Ensō design in § 3.3.

To understand its performance, we fully implement Ensō using an FPGA-based SmartNIC. We describe our hardware and software implementations in § 3.5 and how Ensō can be used depending on the functionality offered by the NIC in § 3.6. In § 3.7 we present our evaluation of Ensō, including its use in four applications: the Maglev load balancer [48], a network telemetry application based on NitroSketch [111], the MICA key-value store [107], and a log monitor inspired by AWS CloudWatch Logs [15]. We also implemented a software packet generator

that we use in most of the experiments.² We observe speedups of up to $6\times$ relative to a DPDK implementation for Maglev, and up to $1.47\times$ for MICA with no hardware offloads.

Finally, while Ensō is optimized for applications that process data in order, we show that Ensō also outperforms the existing packetized interface when used by applications that process packets out of order (e.g., virtual switches), despite requiring an additional memory copy (§ 3.7.3).

3.1 Background and Motivation

The way software (either the kernel or applications using a kernel-bypass API) and the NIC exchange data is defined by the interface that the NIC hardware exposes. Today, most NICs expose a *packetized* NIC interface. This includes NICs from several companies including Amazon [4], Broadcom [27], Intel [78], Marvell [115], and others. Indeed, prior work [156] found that of the 44 NIC drivers included in DPDK, 40 use this interface. Due to its ubiquity, the packetized NIC interface has dictated the API provided by nearly all high-performance network libraries, including `io_uring` [38], DPDK [43] and `netmap` [166]. In this section, we describe the packetized NIC interface and highlight some of the issues that it brings to high-performance applications.

3.1.1 Packetized NIC Interface

A core design choice in the packetized NIC interface is to place every packet in a dedicated packet buffer. The NIC and the software communicate by exchanging packet *descriptors*. Descriptors hold metadata, including packet size, what processing the NIC should perform (e.g., update the checksum or segment the packet), a flag bit, and a pointer to a separate *packet buffer* which holds the actual packet data. Most packet processing software preallocates a fixed number of buffers for packets; new packets (either generated by an application or incoming from the network) are assigned to the next available buffer in the pool, which may not reside in memory anywhere near the preceding or following packet. Because software does not know the size of incoming packets beforehand, buffers are often sized so that they can accommodate MTU-sized packets (e.g., 1536B in Linux and 2kB in DPDK).

Figure 3.1 shows an example of a packetized NIC interface being used to receive four packets from a particular hardware queue on the NIC. The NIC queue is associated with a set of NIC registers that can be used to control a receive (RX) descriptor ring buffer and a transmit (TX) descriptor ring buffer. Before being able to receive packets, the software informs the NIC of the addresses of multiple available buffers in its pool by enqueueing descriptors pointing to each one in the RX descriptor ring buffer. The NIC can then use DMA to write the incoming packet

²Developing this software packet generator was a necessary first step in evaluating Ensō because no existing software packet generators could scale to the link rates we needed to stress test Ensō.

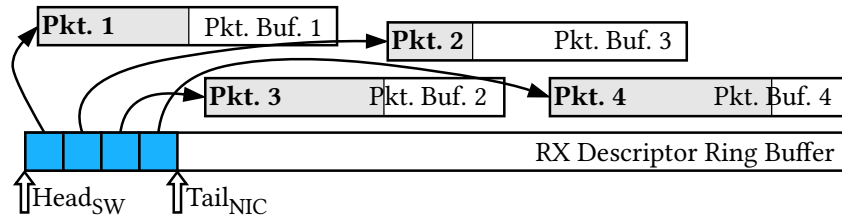


Figure 3.1: Data structures used to receive packets in a packetized NIC interface. Each packet is placed in a separate buffer that can be arranged arbitrarily in memory.

data into the next available packet buffer and enqueue updated descriptors containing metadata such as the packet size. Importantly, the NIC also sets a “flag” bit in the descriptor to signal to the software that packets have arrived for this buffer. Observing a notification bit for the descriptor under the head pointer, the software can then increment the head pointer.

A similar process takes place for transmission: the sending software assembles a set of descriptors for packet buffers that are ready to be transmitted and copies the descriptors—but not the packets themselves—into the TX ring buffer; the flag bit in the descriptor is now used to signal that the NIC has transmitted (rather than received) a packet.

One of the major benefits of dedicating buffers for each packet is that multiplexing/demultiplexing can be done efficiently in software. If the software transmitting packets is the kernel, this might mean associating each descriptor/packet pair with an appropriate socket; if the software in use is a software switch [67, 154] this might mean steering the right packet to an appropriate virtual machine. Either way, the cleverness of the packetized NIC interface in using dedicated packet buffers shines here: rather than copying individual packets in the process of sorting through inbound packets, the switching logic can deliver packet pointers to the appropriate endpoints. These packets can then be processed and freed in arbitrary order.

The usage model for a modern high-performance software stack, however, looks very different. Instead of *one* software entity (e.g., kernel, software switch) mediating access to the NIC, there may be many threads or processes with *direct NIC access* (i.e., kernel bypass). High-performance NIC ASICs expose *multiple hardware queues* (as many as thousands [78]) so that each thread or process can transmit and receive data directly to the NIC without coordination between them. The NIC then takes on all of the responsibilities of demultiplexing, using, e.g., RSS [192], Intel’s Flow Director [78], or (for a very rich switching model) Microsoft’s AccelNet [54]. In this setting, the multiplexing/demultiplexing capabilities of the packetized NIC interface offer no additional value.

3.1.2 Issues with a Packetized Interface

While many high-performance applications today gain little from a packetized interface, they still need to pay for the overheads accompanying it. Shoehorning data communication between the NIC and applications into fixed-sized chunks leads to inefficient use of CPU caches and PCIe bandwidth for small requests, as well as additional data copies due to fragmentation for applications that rely on large messages or bytestreams.

In this section, we conduct microbenchmarks that isolate these issues, and in § 3.7, we also show the impact that these issues have on real applications.

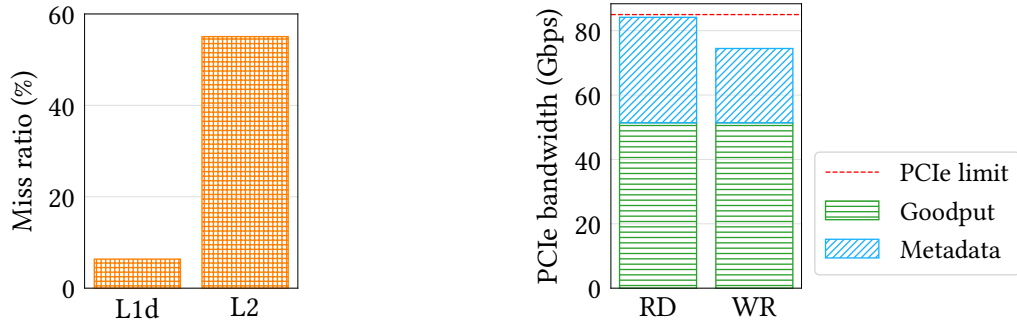
Chaotic Memory Access. We experiment with a simple DPDK-based ping/pong program (a description of our testbed is in § 3.7) which receives a packet, increments a byte in the packet payload, and re-transmits it. For this program, we observed maximum throughput of 40 Gbps using a 100 Gb NIC (Intel E810) and a single 3.1 GHz CPU core. When we conduct a top-down analysis [88], we see that the application is backend-bound, primarily due to L1 and L2 cache misses. Figure 3.2a shows around 6% miss ratio for the L1d and a 55% miss ratio for the L2 cache. This high cache miss ratio is a direct consequence of using per-packet buffers in the packetized NIC interface. First, because *packet buffers themselves are scattered in memory, reads and writes to packet data evade any potential benefit from shared cache lines or prefetching*.³

Applications like key-value stores [12, 107] or packet processors [42] exhibit very high *spatio-temporal* locality in their data access: they are designed to run to completion (i.e., they continue working on a packet or batch until the work for that item is completed, leading to repeated accesses to the same data), and they operate over incoming packets or batches in the order in which they arrive (i.e., the current item being processed serves as an excellent predictor of the next one). However, this structure is not realized in the memory layout of packetized buffers, and hence to any cache optimizations, reads and writes appear unpredictable. Second, because *every packet is paired with a descriptor, the total amount of memory required to store all of the data required for I/O increases, exacerbating last-level cache contention simply because more data needs to be accessed*.

Indeed, prior work [113, 186] has repeatedly demonstrated that the size of the working set for packet processing applications often outgrows the amount of cache space dedicated to DDIO [74], negating the benefits of this hardware optimization to bring I/O data directly into the cache. As we discuss in detail in § 3.7.3, using a different NIC interface that facilitates sequential memory accesses can drop the miss ratio from 6% to 0.2% for the L1d cache, and from 55% to 9% for the L2 cache.

Metadata Bandwidth Overhead. We observe that the packetized NIC interface requires the CPU and the NIC to exchange *both* descriptors and packet buffers. This leads to the second

³We note here that the aforementioned performance penalty arises in spite of the fact that DPDK performs mbuf-level software prefetching.



(a) Miss ratio for L1d and L2 caches. We observe 55% miss ratio for the L2 cache.

(b) PCIe bandwidth utilization. Up to 39% of the read bandwidth is consumed with metadata.

Figure 3.2: PCIe bandwidth and cache misses for an application forwarding small packets with a packetized NIC interface (E810).

problem with the packetized interface: *up to 39% of the CPU to NIC interconnect bandwidth is spent transferring descriptors* (Figure 3.2b). While NIC-CPU interconnect line rates are typically higher than network line rates, the gap between them is relatively narrow. This is particularly problematic for small transfers as the PCIe theoretical limit drops to only 85 Gbps with 64-byte transfers [126]. We also expect this gap to remain small in the future as a state-of-the-art next generation server with a 400 Gbps Ethernet connection and 512 Gbps of PCIe 5.0 bandwidth would still bottleneck with 39% of bandwidth wasted on metadata. This observation complements recent studies that also point to the PCIe as a source of congestion for transport protocols [3].

* * *

By pairing every packet with a separate descriptor, the packetized NIC interface was well designed for a previous generation of high-throughput networked applications which needed to implement multiplexing in software. However, for today’s high-performance applications, it introduces unnecessary performance overheads.

3.2 Ensō Overview

Ensō is a new streaming interface for NIC-application communication. Ensō’s design has three primary goals: (1) *flexibility*, allowing it to be used for different classes of offloads operating at different network layers and with different data sizes; (2) *low software overhead*, reducing the number of cycles that applications need to spend on communication; and (3) *hardware simplicity*, enabling practical implementations on commodity NICs.

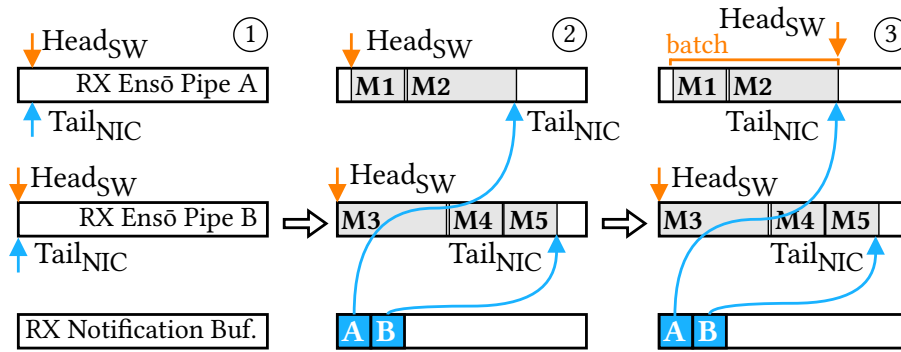


Figure 3.3: Steps to *receive* batches of messages in two Ensō Pipes.

Ensō is designed around the *Ensō Pipe*, a new buffer abstraction that allows applications and the NIC to exchange arbitrary chunks of data as if reading and writing to an unbounded memory buffer. Different from the ring buffers employed by the packetized interface (which hold descriptors to scattered packet buffers), an Ensō Pipe is implemented as a *data* ring buffer that contains the actual packet data.

3.2.1 High-Level Operation

In Figure 3.3 we show how an application, with two Ensō Pipes, receives messages. Initially, the Ensō Pipes are empty, and the $Head_{SW}$ and $Tail_{NIC}$ point to the same location in the buffer ①. When the NIC receives messages, it uses DMA to enqueue them in contiguous memory owned by the Ensō Pipes ②. In the figure, the NIC enqueues two messages in Ensō Pipe A’s memory, and three in Ensō Pipe B’s memory. The NIC informs the software about this by also enqueueing two notifications (one for each Ensō Pipe) in the notification buffer. The software uses these notifications to advance $Tail_{NIC}$ and process the messages. Once the messages have been processed, the software writes to a Memory-Mapped I/O (MMIO) register (advancing $Head_{SW}$) to notify the NIC—allowing the memory to be reused by later messages ③. Sending messages is symmetric, except for the last step: the NIC notifies the software that messages have been transmitted by overwriting the notification that the CPU used to inform the NIC that a message was available to be transmitted.

3.2.2 Ensō Pipe’s Flexibility

Although Figure 3.3 shows the steps to send *messages*, because Ensō Pipes are opaque, they can be used to transmit arbitrary chunks of data. These can be raw packets, messages composed of multiple MTU-sized packets, or even an unbounded bytestream. The format of the data is dictated

by the application and the offloads running on the NIC. Moreover, Ensō Pipes’ opaqueness means that they can be mapped to any pinned memory within the application’s memory space. Thus, by mapping both the RX and TX Ensō Pipes to the same region, network functions and other forwarding applications can avoid copying packets. In our evaluation (§ 3.7) we use this approach when implementing Maglev and a Network Telemetry application.

3.2.3 Performance Advantages of an Ensō Pipe

The fact that data can be placed back-to-back inside an Ensō Pipe addresses both of the performance challenges we listed previously: First, Ensō Pipes allow applications to read and write I/O data *sequentially*, thus avoiding chaotic memory accesses. Second, as shown in Figure 3.3, inlining data in an Ensō Pipe removes the need for per-packet descriptors, thus reducing the amount of metadata exchanged over the PCIe bus, and reducing cycles spent managing (i.e., allocating and freeing) packet buffers.

3.2.4 Challenges

Although implementing a ring buffer for data transfer is, on its own, a simple idea, coordinating the notifications between the CPU and the NIC to update head and tail pointers turns out to be challenging.

Efficient coordination. The packetized interface coordinates incoming and outgoing packets by “piggybacking” notifications in the descriptor queue itself. Each descriptor includes a “flag bit” that can be used to signal when the descriptor is valid. Software polls the next descriptor’s flag bit to check if a new packet arrived. We cannot use the same strategy for Ensō Pipes as they do not assume a format for the data in the buffer, and hence cannot embed control signals in it.

In § 3.3.1, we discuss how naïve approaches to notification can stress worst-case performance of MMIO and DMA. In particular, concurrent accesses to the same memory address can create cache contention between the CPU and the NIC. Ensō uses dedicated *notification buffers* to synchronize updates to head and tail pointers; when combined with batching and multiqueue processing, the notification buffer approach reduces the threat of cache contention.

Notification pacing. Ensō Pipes are designed so that notifications for multiple packets can be combined, reducing the amount of metadata transferred between the CPU and the NIC. However, it is still important to decide *when* to send notifications: when sent too frequently they waste PCIe bandwidth and add software overheads, but if sent too infrequently the core might be idle waiting for notification, thus reducing throughput. Ensō includes two mechanisms, *reactive notifications* and *notification prefetching* (§ 3.3.2), that control when notifications are sent. These mechanisms are naturally adaptive, i.e., they minimize the number of notifications sent without limiting throughput, and can be implemented without adding hardware complexity.

Low hardware complexity and state. Because the design of Ensō involves both hardware and software, we must be careful to not pay for software simplicity with hardware complexity. Ensō favors coordination mechanisms that require little NIC state. We aim for a design that is simple and easily parallelized. We present Ensō’s hardware design in § 3.5.

3.2.5 Target Applications

Ensō implements a streaming interface that is optimized for cases where software processes received data in order. Our evaluation (§ 3.7) shows that this covers a wide range of network-intensive applications.

One might expect that the resulting design is ill-suited for applications that need to multiplex and demultiplex packets (e.g., virtual switches like Open vSwitch [154] and BESS [67]), as such applications require additional copies with Ensō.⁴ However, perhaps surprisingly, Ensō outperforms the packetized interface *even when it requires such additional copies*. As we show in § 3.7.3, when comparing the performance of an application that uses Ensō and copies each packet, to a similar DPDK-based application that does not copy packets, using a CAIDA trace [30] (average packet size of 462 B), we find that Ensō’s throughput is still 28% higher than DPDK’s (92.6 Gbps vs. 72.6 Gbps). We discuss how Ensō can be used depending on the applications and the functionality offered by the NIC in § 3.6.

3.3 Efficient Notifications

The key challenges in Ensō arise from efficiently coordinating Ensō Pipes between the CPU and the NIC. In this section, we describe how Ensō efficiently coordinates pipes using notifications (§ 3.3.1) and how it paces such notifications (§ 3.3.2).

3.3.1 Efficient Ensō Pipe Coordination

Recall from Figure 3.3 that the software and the NIC coordinate access to RX Ensō Pipes using Head_{SW} and Tail_{NIC} and to TX Ensō Pipes using Head_{NIC} and Tail_{SW} . How should software and the NIC communicate pointer updates?

In the descriptor ring buffers employed by the packetized NIC interface, software communicates pointer updates to the NIC using MMIO writes, and the NIC communicates pointer updates via inline signaling in the descriptor buffer itself [50, 78], avoiding the overheads of MMIO reads. Because the descriptor’s format is defined by the NIC, the NIC can dedicate a “flag signal” in

⁴Note that this overhead only affects applications that multiplex/demultiplex packets, and does not apply to software, e.g., TCP stacks, that processes packet data but might need to reorder packets. This is because reordering packet data (rather than whole packets) requires a memory copy when using either interface.

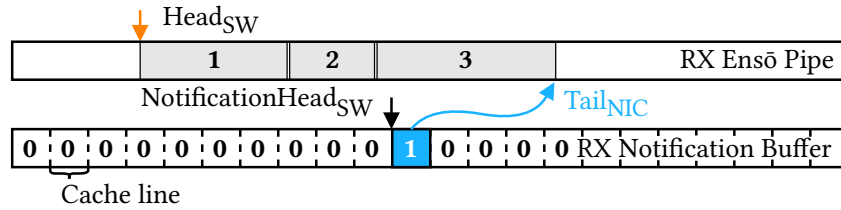


Figure 3.4: Notification buffer being used to notify the arrival of three chunks of data in an RX Ensō Pipe. Notifications contain the latest Tail_{NIC} for a given Ensō Pipe as well as a flag signal.

every descriptor to signal that the descriptor is valid. Software can then poll the next descriptor until its flag becomes valid. This way, there is no need for the NIC to explicitly tell software of pointer updates. Unfortunately, we cannot use this approach for Ensō. While Ensō Pipes can still use MMIO writes for pointer updates from software, we cannot embed inline signaling in the Ensō Pipe itself since we do not impose any structure on the data.

We considered several design options to communicate pointer updates. We focus on one illuminating rejected design here: sharing an address in main memory between the NIC and software. With each Ensō Pipe, we might have a dedicated address in memory where the NIC writes the latest Tail_{NIC} . The software can then poll this address to determine the latest value. Unfortunately, this approach leads to contention because every time the NIC writes to memory, the cache entry on the CPU is invalidated. If software continues to poll the same cache line, the resulting contention reduces throughput by orders of magnitude: we measured a throughput below 5 Gbps when using small transfers with this approach. We discuss other rejected approaches in § 3.4.

Notification Buffer

Ensō uses a *notification buffer* to communicate pointer updates. Although the structure of the notification buffer on its own does not solve the cache contention challenge, when combined with *batched notifications* and when it is used to aggregate notification updates to/from *multiple* Ensō Pipes, this approach prevents the CPU from busy waiting on the shared cache line and hence avoids contention-induced slowdowns.

Figure 3.4 shows an RX Ensō Pipe with its corresponding notification buffer. It shows how a single notification indicates the presence of multiple sequential chunks of data at the same time; we discuss how notifications are “batched” or coalesced in § 3.3.2. Notifications contain the latest Tail_{NIC} for a given RX Ensō Pipe as well as a flag signal that software can use to check if the next notification is ready to be consumed. As is done typically with descriptor ring buffers, software advances the $\text{NotificationHead}_{\text{SW}}$ using an MMIO write after consuming a notification. Like an RX Ensō Pipe, a TX Ensō Pipe also uses a separate notification buffer to synchronize

pointer updates. But software enqueues new notifications when it wants to transmit a chunk of data and the NIC overwrites the transmission notification with a completion notification once it is done transmitting the corresponding batch. Completions flip a flag signal, so that software can check if the following cache line corresponds to a completion or a pending TX notification.

Multiplexing and Scaling

We now briefly discuss how to multiplex and scale Ensō across different dimensions.

Within a single thread. To let a single thread efficiently access multiple Ensō Pipes, we associate multiple Ensō Pipes with the same notification buffer. To accomplish this, notifications include an *Ensō Pipe ID* alongside the Tail_{NIC} and the flag signal that we discussed before. As a result, software can probe a *single* notification buffer to retrieve updates from *multiple* Ensō Pipes. This avoids the known scalability issues from needing to poll multiple queues [119, 173].

Among multiple threads. To let multiple threads send and receive data independently, Ensō supports multiple notification buffers. Each thread can use a *dedicated* notification buffer, avoiding costly synchronization primitives. When setting up a new Ensō Pipe, software tells the NIC which notification buffer is associated with it. Therefore, the NIC knows to which notification buffer to send a notification.

Among multiple applications. In addition to using independent notification buffers, Ensō ensures that applications only have access to their own subset of Ensō Pipes and notification buffers. Each queue’s MMIO pointer register pair is kept in its own dedicated page-aligned block of memory [46]. This lets the kernel map the pointer registers at a per-queue granularity to the address space of the application that requested it.

Notifications: Contention and Overhead

Allowing multiple Ensō Pipes to share the same notification queue (§ 3.3.1), and having notifications arrive only for larger batches of data (§ 3.3.2) naturally prevents contention by keeping the NIC *ahead* of the CPU in updating the notification buffer, and also reduces the PCIe overhead of communicating these notifications. As the CPU reads in data for one Ensō Pipe, the NIC is writing new entries for subsequent Ensō Pipes. Because the CPU is processing larger batches of data, it is busier for longer before it needs to check the notification buffer. Hence, as line rates go up, the two are unlikely to be accessing the same cache line simultaneously.

3.3.2 Pacing Notifications

As mentioned above, Ensō batches notifications aiming to reduce metadata bandwidth consumption and to keep CPUs busy processing data, rather than waiting for notifications. Using the

```

func onPktArrival()
    if status = 0 then
        |   notify (TailNIC)
        |   status ← 1

func onRxUpdate(HeadSW)
    if HeadSW = TailNIC then
        |   status ← 0
    else
        |   notify (TailNIC)

```

Figure 3.5: Reactive notification mechanism for an Ensō Pipe. It only sends notifications when new data arrives and *status* = 0, or if software updates *Head*_{SW} and it is different from Tail_{NIC}.

wrong batch size, however, is problematic: a system that uses batch sizes that are too small would unnecessarily transmit extra metadata, and a system that uses batch sizes that are too large might unnecessarily inflate latency. Ideally, the NIC could keep track of how frequently software is consuming notifications and try to send a notification right before software needs the next one, with all the data that have arrived since the last software read. Ensō approximates this ideal approach but without the impossibility of perfectly predicting when the next read is coming.

Instead of trying to predict when to send the next notification, Ensō lets software dictate the pace of notifications by sending notifications *reactively*. It leverages the fact that the NIC is already aware of when the software is consuming data, as the NIC is notified whenever software updates *Head*_{SW} through MMIO writes. Therefore, the NIC can send notifications in response to these updates. Specifically, we allow the actual NIC tail pointer (Tail_{NIC}) to diverge temporarily from what is observed from software via the notification buffer. Ensō suppresses updates to Tail_{NIC} until the NIC sees an update to *Head*_{SW}. Our implementation uses a single status bit for every Ensō Pipe, initialized to ‘0’, which indicates if the buffer has data.

Operation. Figure 3.5 summarizes the reactive notification mechanism. Whenever data arrive at an RX Ensō Pipe (onPktArrival), we check the status bit, only sending a notification if *status* = 0 (indicating that the buffer is empty). We also potentially send notifications whenever software updates *Head*_{SW} (onRxUpdate). If the new *Head*_{SW} is the same as Tail_{NIC}, it means that the buffer is now empty and we can reset the status back to ‘0’ without sending a new notification. Otherwise, it indicates that software is unaware of some of the latest data in the buffer, which triggers a new notification.

Discussion. Reactive notifications cause the notification rate to naturally adapt to the rate at which software is consuming data, as well as how fast incoming data is arriving. When software is slow to consume data from a particular Ensō Pipe, bytes accumulate and the NIC sends fewer notifications for that Ensō Pipe. When software is fast to consume data, it advances the *Head*_{SW} pointer more often, causing the NIC to send frequent notifications.

Reactive notifications ensure that every piece of data is notified, but as we will see in § 3.7.3,

they can impose a small latency overhead. This overhead occurs if packet arrivals are known to the NIC but have not yet been communicated in the notification buffer. In this case, when Head_{SW} reaches Tail_{NIC} , software has to wait for a PCIe RTT before it is notified of the waiting packets. While an extra PCIe RTT is unlikely to be an issue for Internet-facing applications, it might be an issue for some latency-sensitive applications [19].

Notification prefetching. To improve latency for latency-sensitive applications, Ensō also implements a notification prefetching mechanism. Notification prefetching allows software to explicitly request a new notification to the NIC. Applications can use notification prefetching either explicitly, by calling a function to prefetch notifications for a given Ensō Pipe, or transparently, by compiling the Ensō library in low-latency mode. When compiled in low-latency mode, the library always prefetches notifications for the next Ensō Pipe before returning data for the current one. We evaluate the impact of using notification prefetching in § 3.7.3.

3.4 Rejected Designs for Pointer Updates

Before concluding the design, we describe some of the designs we considered for communicating pointer updates from the NIC to software. While these designs might seem to suit our needs at first, we ultimately discarded them since they perform poorly due to architectural details of PCIe or the CPU. We first describe these constraints and then explain why they ultimately led us to discard three different pointer update designs.

The performance constraints we found are primarily based on architectural details of PCIe-based communicating between the processor and the NIC either via Direct Memory Access (DMA) to host memory or Memory-Mapped I/O (MMIO) to device registers [83]. They can be summarized in the following guidelines:

1. **Avoid partial DMA writes.** Writes that are less than a cache line (64 bytes) or writes that are not cache line aligned reduce performance as the CPU needs to fetch the remaining cache line before it can commit the write. Writing multiple cache lines further amortizes PCIe overheads [126].
2. **Avoid repeated DMA accesses to the same cache line.** Repeatedly accessing (read or write) the *same* cache line causes serialization which prevents effective use of CPU pipeline slots [83]. Subsequent DMA accesses should go to *different* cache lines.
3. **Avoid cache line contention.** DMA writes to a cache line present in a core’s L1d cache require a slow ownership transfer [194]. The interface should avoid having software and the NIC frequently requesting ownership to the same cache line.
4. **Avoid MMIO reads.** Software-issued MMIO reads cannot be served from the cache. The CPU core will stall until the read request is sent and the response is received from the

NIC (two PCIe transfers). Additionally, PCIe serializes MMIO reads, further reducing performance.

The above guidelines led us to discard three different pointer update designs, that looked simple at first but that failed to comply with some of those.

3.4.1 Rejected Design #1: MMIO Synchronization

The simplest design would be for the NIC to update the pointer values to its internal memory and make software periodically issue an MMIO read to fetch the latest value. Unfortunately, software-issued MMIO reads cannot be served from the cache, causing the CPU core to stall until the read request is sent and the response is received (two PCIe transfers). Additionally, PCIe serializes MMIO reads, further reducing performance.

3.4.2 Rejected Design #2: Shared Memory Synchronization

A second simple alternative would be to dedicate an address in host memory to hold the pointer value. The NIC can issue a DMA write whenever it needs to update the pointer value. Software can then periodically poll the same address to figure out if the NIC advanced the pointer. The issue with this design is that it makes the software and the NIC contend for the same cache line, which requires a slow ownership transfer whenever the NIC or the CPU access the cache line. To verify this, we implemented this design and obtained a throughput of less than 5 Gbps when enqueueing 64-byte packets using the same setup as described in § 3.7.

3.4.3 Rejected Design #3: Inline Synchronization

The last discarded design is motivated by traditional NIC descriptor ring buffers which contain constant-sized descriptors with metadata in a format defined by the NIC. These designs designate a bit of the descriptor as a “flag bit” [50, 78]. Initially, all the descriptor slots in the buffer have their flag bit zeroed. Whenever the NIC DMA writes a new descriptor to host memory, it overwrites the old ‘0’ flag with a ‘1’ flag. To figure out if a new descriptor is ready to be consumed, software simply needs to check if the next slot’s flag is set. After consuming a descriptor, software sets the flag back to ‘0.’ Because many descriptors are likely to be present in the buffer, this reduces the chance that software and the NIC will contend for the same cache line.

Since Ensō Pipes are opaque, implementing the same strategy used in the descriptor ring buffer is impossible. Even if we zeroed the entire buffer after the data is consumed, we do not know what data to expect in the buffer—the next incoming data might also be zero. Therefore we tested an alternative design: we picked a 128-bit random cookie, to make the chance of collision

with incoming data negligible and placed it at the beginning of every cache line of the Ensō Pipe. Software now only needs to check if the next 128 bits match the cookie. Unfortunately, filling the buffer with cookies whenever the data is consumed imposes considerable overhead. For this reason, this design worked well for small data transfers but poorly when using large chunks of data. This design also prevents software from detecting unaligned writes.

3.5 Ensō Implementation

Our Ensō implementation consists of three pieces: a userspace library, a kernel module, and a NIC hardware that implements Ensō. We implement the library and kernel module on Linux in about 9k lines of C and C++17. Our hardware implementation uses about 10k lines of SystemVerilog, excluding tests and auto-generated IPs. Our hardware implementation targets an FPGA SmartNIC but the same design could also be implemented in an ASIC.

3.5.1 Software Implementation

Applications use a library call to request notification buffers and Ensō Pipes. Typically, applications will request a notification buffer for each thread but may use multiple Ensō Pipes per thread, depending on their needs. The library sends requests for new Ensō Pipes and notification buffers to the kernel, which checks permissions, allocates them on the NIC, and then maps them into application space. Ensō Pipes are typically allocated in pairs of RX and TX Ensō Pipe but may also be allocated as unified RX/TX Ensō Pipes. Unified RX/TX Ensō Pipes map the RX and TX buffer to the same memory region, which is useful for applications that modify data in place and send them back, e.g., network functions. In contrast, separate Ensō Pipes map the RX and TX buffers to the different memory regions and are useful for typical request-response applications.

To ensure a consistent abstraction of unbounded Ensō Pipes we must also deal with corner cases that arise when data wrap around the buffer limit. To prevent breaking received data that wrap around, we map the same Ensō Pipe’s physical address twice in the application’s virtual memory address space. This means that, to the application, the buffer appears to be twice its actual size, with the second half always mirroring the first one. The application can then consume up to the full size of the buffer of data at once, regardless of where in the buffer the current `HeadSW` is. To transmit data that wrap around the buffer limit, the library checks if the transfer goes around the Ensō Pipe boundary and automatically partitions it into several transfers, each of which is smaller than the overall buffer size.

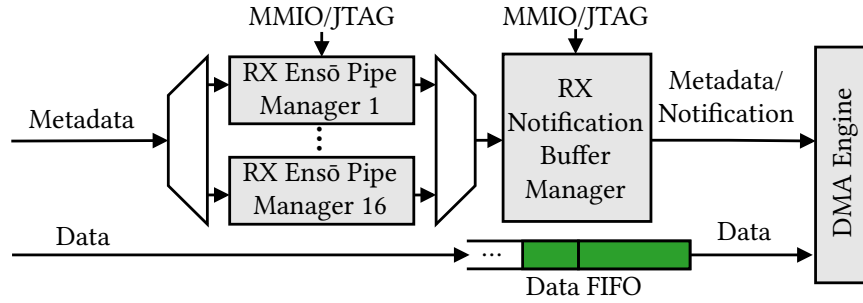


Figure 3.6: NIC RX Datapath.

3.5.2 Hardware Implementation

We now describe key RX and TX hardware modules.

RX Datapath

Figure 3.6 illustrates the RX datapath. It receives as input data and metadata, which includes the Ensō Pipe ID and the size of the corresponding data that is being enqueued. Metadata is handled separately from the data, which allows for smaller queues between modules. The RX datapath is composed of the following modules:

RX Ensō Pipe Managers. The Ensō Pipe managers are responsible for keeping Ensō Pipe state such as the buffer’s physical address, Head_{SW} , Tail_{NIC} , notification buffer ID, and notification status bit. When metadata arrive for Ensō Pipe i , the manager checks for sufficient space in i and advances i ’s Tail_{NIC} . The manager also determines whether to trigger a notification according to the reactive notification strategy (§ 3.3.2). To trigger a notification, the manager includes the notification buffer ID and sets a bit in the metadata that is sent to the Notification Buffer Manager.

We use multiple Ensō Pipe Managers for two reasons. First, it enables flow-level parallelism. Each manager requires two cycles to process each metadata, achieving a request rate of 125 Mpps at a 250 MHz clock. To achieve 100 Gbps line rate (148.8 Mpps) we thus need at least two managers. Second, multiple managers enable scaling to high Ensō Pipe counts. We split the state for different Ensō Pipes among different managers, allowing the logic to be closer to the memory that it needs to access. We configure the number of Ensō Pipe Managers at synthesis time with a default of 16, which allows the design to meet timing for up to 16k Ensō Pipes.

RX Notification Buffer Manager. This module issues notifications when needed. It spends one cycle for every request and an extra cycle for those that require a notification. If we can suppress notifications for at least 20% of requests, we only need a single notification manager at a 250 MHz clock. A single notification manager is also sufficient since we use fewer notification

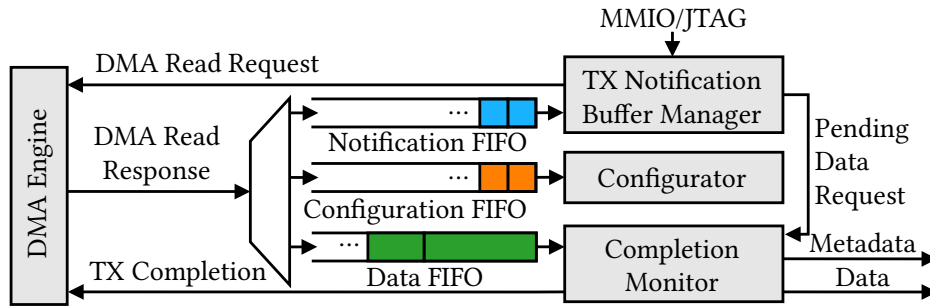


Figure 3.7: NIC TX Datapath.

buffers than Ensō Pipes, e.g., one notification buffer per CPU core. Our implementation defaults to 128 notification buffers but also meets timing with 1024.

DMA Engine. This module uses DMA to write data and notifications to the correct address in host memory based on the metadata computed by the upstream modules.

TX Datapath and Configuration Path

The TX datapath (Figure 3.7) is composed of:

TX Notification Buffer Manager. This module keeps state for all TX notification buffers. When software enqueues a new TX notification and advances $\text{NotificationTail}_{\text{SW}}$ using an MMIO write, the manager requests a DMA read to fetch the notification from memory. The read request is sent to the DMA Engine, which enqueues the DMA read response to the Notification FIFO. The manager can then consume the notifications, allowing it to request DMA reads for the actual data inside an Ensō Pipe. It also sends information about each data request to the Completion Monitor module.

Completion Monitor. When the data requested by the TX Notification Buffer Manager arrive, the DMA Engine enqueues them to the Data FIFO. The Completion Monitor consumes the data and keeps track of the number of bytes pending in each request using information that it received from the TX Notification Buffer Manager. When the request completes, the Completion Monitor sends a TX completion notification to the DMA Engine that writes it to host memory. It can then output the data and the metadata, containing the Ensō Pipe ID and size, to downstream modules on the NIC.

Configurator. Configuration notifications are enqueued to the Configuration FIFO instead of the Notification FIFO. These are consumed by the Configurator, which directs the configuration to the appropriate module on the NIC. For instance, when the kernel sets a new Ensō Pipe, it inserts an entry in the NIC Flow Table to direct a set of packets to it.

3.6 Using Ensō

Ensō provides a zero-copy streaming abstraction that the NIC and applications can use to exchange data. Thus far, we have shown how this abstraction can be efficiently *implemented*. We now discuss how Ensō should be *used*.

How one uses Ensō depends on how features are split between hardware and software. We consider three settings: (1) Traditional NICs which implement simple offloads, such as checksum and RSS [192], and rely on software implementation for the rest. These can use Ensō Pipes to deliver raw packets to/from a network stack implemented in software. (2) NICs that implement transport offloads [22, 36, 176, 178] in hardware. These can deliver application-level messages or reassembled bytestreams through the Ensō Pipes. And finally, (3) NICs that implement application logic [85, 100], which can use Ensō Pipes to exchange application data. We elaborate on each of these settings below.

Traditional NICs. For traditional NICs that perform simple offloads such as checksum and segmentation, using Ensō is not significantly different than using a packetized interface. In both cases, the network stack is implemented in software and only needs to be changed to use Ensō Pipes. Ensō is designed to support several Ensō Pipes, and for most applications this change does not induce any additional software overheads. Furthermore, high-performance packet processing applications that process packets in order, as is the case for most network functions [156] and applications that use UDP, can consume and transmit raw packets through an Ensō Pipe without copies.

However, applications such as virtual switches [67, 154], that multiplex and demultiplex packets (but do not perform reassembly or other functions that require re-ordering packets, where both interfaces require copies), need to perform an additional copy when forwarding packets to their destination. As we show in § 3.7.3, Ensō’s performance advantages can outweigh the cost of copies even for such applications.

NICs with transport offload. NICs that implement message-based transports (e.g., SCTP, Homa [122]) or streaming-based transports (e.g., TCP) may also choose to use Ensō Pipes to deliver messages or reassembled bytestreams directly to the application without copies.

NICs with application logic. NICs that implement application-layer protocols or include part of the application logic may use Ensō Pipes to exchange application-level messages with applications. For instance, a NIC that is aware of both TCP and HTTP may deliver incoming HTTP requests back to back to a web server, effectively converting the application to a run-to-completion model.

3.7 Evaluation

We now evaluate our design decisions using microbenchmarks, and then use four real-world applications to show how Ensō improves end-to-end performance.

3.7.1 Setup and Methodology

Device Under Test (DUT). We synthesize and run the Ensō NIC on an Intel Stratix 10 MX FPGA NIC [77] with 100 Gb Ethernet and a PCIe 3.0 x16 interface. Most of the NIC design runs at 250 MHz. Our baseline uses an Intel E810 NIC [80] with 100 Gb Ethernet and a PCIe 4.0 x16 interface, and uses DPDK to minimize software overheads. All our experiments are run on a server with an Intel Core i9-9960X CPU [82] with 16 cores running at 3.1 GHz base frequency, 22 MB of LLC, and PCIe 3.0. We disable dynamic frequency scaling, hyper-threading, power management features (C-states and P-states), and isolate CPU cores from the scheduler.

Packet generator. The packet generator machine is equipped with an Intel Core i7-7820X CPU [81] with 8 cores running at 3.6 GHz base frequency, 11 MB of LLC, and PCIe 3.0. It includes another Stratix 10 MX FPGA connected back to back to the E810 and the FPGA on the DUT machine.

We found that existing high-performance packet generators such as DPDK Pktgen [197] and Moongen [49] are unable to keep up with Ensō’s packet rate because their performance is limited by the packetized NIC interface. We thus implement EnsōGen, a packet generator based on Ensō. EnsōGen generates packets from a pcap file, and can send and receive arbitrary-sized packets at 100 Gbps line rate using a *single* CPU core. We describe EnsōGen in more detail in § 3.7.2. We use EnsōGen in all experiments except for MICA, where we send requests from a MICA client (§ 3.7.4).

Methodology. We measure zero-loss throughput as defined in RFC 2544 [24, 123] with a precision of 0.1 Gbps. We report median throughput and error bars for one standard deviation from ten repetitions. We measure latency by implementing hardware timestamping on the FPGA, which achieves 5 ns precision for packet RTTs. EnsōGen keeps a histogram with the RTT of every received packet, which we use to compute median and 99th percentile latencies. PCIe bandwidth measurements use PCM [41] and we obtain other CPU counters using perf [152]. To evaluate MICA, we use the same methodology as the original paper [107] for consistency.

3.7.2 EnsōGen Packet Generator

EnsōGen is a software packet generator built on top of the Ensō NIC interface that achieves 100 Gbps with a single core and arbitrary packet sizes. Here we briefly describe how it operates and how we ensure that it is correct.

Operation. At startup EnsōGen reads a user-supplied pcap file and allocates enough Ensō Pipes to be able to fit all its packets. At run time, EnsōGen simply needs to round-robin among the preallocated Ensō Pipes enqueueing a *single* notification in order to transmit the *entire* 2 MB buffer content. This makes it trivial for EnsōGen to saturate the link with very little CPU overhead. Since transmission is cheap, EnsōGen spends most of its CPU cycles receiving packets. It parses every incoming packet to track the number of bytes and packets received.

Simple offloads. We implemented hardware support for timestamping and rate limiting, which helps EnsōGen achieve cycle-accurate precision while saving CPU cycles. These features are also commonly offered in existing NICs [78, 132] and are leveraged by some software packet generators [49]. When hardware timestamping is enabled, EnsōGen keeps a histogram in host memory with the RTT of every received packet with 5 ns granularity (the same precision as the hardware timestamp, which operates at 200 MHz). To spread the load equally among the RX Ensō Pipes regardless of the workload, EnsōGen also configures the hardware to direct packets to pipes in a round-robin fashion [167].

Correctness. We verified EnsōGen’s performance and rate-limiting capabilities using another in-house packet generator fully implemented on an FPGA, as well as using software counters, to ensure that the rate limited throughput always matches the specification.

3.7.3 Microbenchmarks

We start by using microbenchmarks to evaluate Ensō’s performance and the design decisions we made.

Packet Rate

We start by measuring how fast Ensō can process packets. We compare the performance of an Ensō-based echo server to that of a DPDK-based echo server. On receiving a packet, both versions increment a value in each packet’s payload and then send the packet back out through the same interface. We increment the payload value to ensure that all packets are brought into the processing core’s L1d cache. For the Ensō echo server, we use an RX/TX Ensō Pipe, which lets it echo packets without copies.

Figure 3.8 compares the packet rate for Ensō and DPDK for different numbers of cores. Even with a single core, Ensō is bottlenecked by Ethernet, achieving 148.8 Mpps. In contrast, the E810 with DPDK achieves 59 Mpps with a single core and does not scale beyond two cores, where it peaks at 88 Mpps. Beyond two cores, the experiments with the E810 are bottlenecked by PCIe bandwidth, which is insufficient for transferring packet data and descriptor metadata (§ 3.7.3). As a result, the number of packets dropped by the E810 NIC increases as we increase the number of cores, and the zero-loss throughput *decreases* beyond two cores.

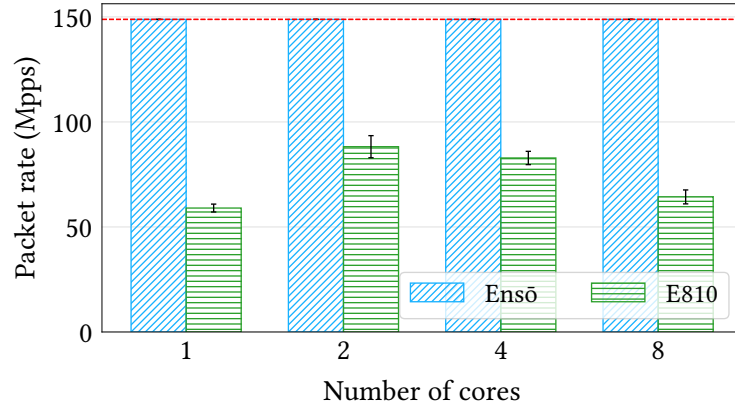


Figure 3.8: Raw packet rate. Ensō is bottlenecked by Ethernet while the E810 does not scale beyond two cores. The dashed line represents the 100 Gb Ethernet limit.

Packet Forwarding with Copies

As we discussed in § 3.2, Ensō’s streaming interface targets applications that process received data in order. With Ensō, applications that multiplex and demultiplex data, such as virtual switches [67, 154], need to copy packets to forward them to their destination. However, as we will see next, Ensō outperforms the E810 even in this scenario.

To quantify the overhead of multiplexing and demultiplexing data, we implement a simple packet forwarding application that swaps MAC addresses and copies incoming packets to a different TX Ensō Pipe. We compare this application against an equivalent zero-copy DPDK implementation. Figure 3.9 shows the throughput when forwarding packets of different sizes using a single CPU core. Packet copies add overhead to Ensō, which can no longer forward 64-byte packets at 100 Gbps (148.8 Mpps). However, Ensō’s throughput with 64-byte packets (91.7 Gbps) is still more than 2× that achieved by the E810 with DPDK *without* copies (39.6 Gbps). For other packet sizes, Ensō achieves line rate. We also evaluate the throughput for the same application when sending packets from a CAIDA trace [30]. For this trace Ensō’s throughput is 92.6 Gbps, compared to 72.6 Gbps for the E810 with DPDK without copies.

This result came as a surprise to us as our goal with Ensō was never to target multiplexing/demultiplexing in software. It also puts into question the usefulness of a packetized interface, as its overheads can be greater than those imposed by packet copies.

Next, we use more detailed microbenchmarks that help explain where Ensō’s performance improvement comes from.

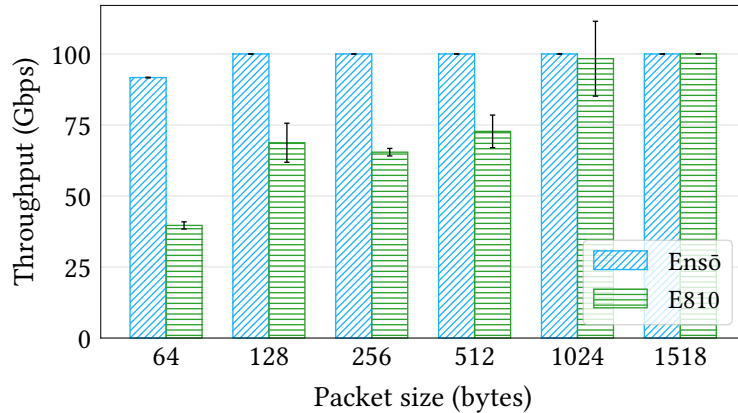


Figure 3.9: Throughput when forwarding packets between two queues. Ensō outperforms zero-copy E810 even when it needs to copy packets.

	Ensō	Chaotic Ensō	E810
Throughput	100.0 Gbps	38.0 Gbps	41.1 Gbps
Rate	148.8 Mpps	56.5 Mpps	61.1 Mpps
L1d miss/total	22.8M/11,597M (0.2%)	557M/3,949M (14%)	1,357M/21,339M (6%)
L2 miss/total	2.1M/22.8M (9%)	382M/558M (68%)	747M/1,357M (55%)
LLC miss/total	541/2.1M (0.03%)	348k/382M (0.09%)	9,086/747M (0.001%)

Table 3.1: Effect of chaotic memory accesses on throughput and cache misses. Results are the average of 10 runs, each lasting 10 s.

Effect of Chaotic Memory Accesses on Cache

We now show that placing messages sequentially in an Ensō Pipe is important for Ensō’s performance. As we discussed previously, not using sequential buffers results in chaotic memory accesses, which reduces the effectiveness of hardware prefetchers (e.g., streaming prefetcher [83]). To evaluate this claim, we built a modified version of Ensō, which we call Chaotic Ensō, that changes the gaps in memory between subsequent messages. We compute the gap deterministically based on the message’s current position in an Ensō Pipe, ensuring that we add no additional software overhead when using Chaotic Ensō.

We benchmarked Ensō, Chaotic Ensō, and the E810 NIC using a program that receives and increments packets (but does not send them back). We measured zero-loss throughput, and cache miss rates at this throughput.

We show the results in Table 3.1. Observe that Chaotic Ensō achieves a lower throughput of 56.5 Mpps than even the E810. The number of cache misses reveals why: despite processing

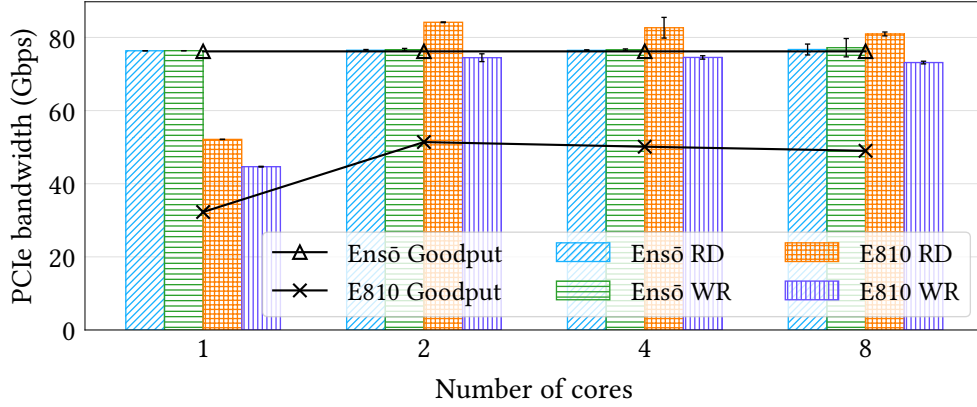


Figure 3.10: PCIe bandwidth utilization for read (RD) and write (WR) transactions. The bars represent the overall PCIe bandwidth utilization, while the lines represent the goodput, i.e., the amount of PCIe bandwidth used to transmit packet data. Ensō uses little PCIe bandwidth for metadata, causing it to achieve a higher goodput while consuming less overall PCIe bandwidth.

fewer packets per second (and hence having fewer cache accesses), Chaotic Ensō has an order-of-magnitude (557 million vs. 22.7 million) more L1d cache misses than Ensō. This, in turn, leads to an order of magnitude (558 million vs. 22.8 million) more accesses to L2 cache, thus increasing packet processing overheads. We also observe that the E810 has more cache accesses than either Ensō or chaotic Ensō, this is because it uses a descriptor per packet and thus requires the application to read more data. Finally, we observed that LLC misses were rare in all three configurations.

PCIe Bandwidth

Next, we evaluate the importance of reducing packet metadata by eliminating descriptors. We do so by measuring PCIe bandwidth when using the echo server described in § 3.7.3 with 64-byte packets. In what follows, PCIe writes refer to DMA transfers from NIC to host memory, while PCIe reads refer to DMA transfers from the host memory to the NIC.

Unlike other microbenchmarks, we do not limit ourselves to zero-loss throughput for this experiment, and instead send packets at line rate. This ensures that software overheads do not limit observed PCIe bandwidth, since drops due to queuing do not reduce throughput. We measure PCIe bandwidth and the rate at which the packet generator receives packets. We use this to calculate the fraction of PCIe bandwidth used for actual packet data (goodput).⁵

⁵The maximum goodput achievable with 64-byte packets and 100 Gb Ethernet is 76.19 Gbps, since Ethernet adds 20 bytes of overhead per packet.

We report the results in Figure 3.10, where we show both PCIe goodput and total PCIe bandwidth for read (RD) and write (WR) directions. We draw four conclusions from it:

1. With one core, the E810 is CPU-bound, and the NIC drops incoming packets because of a lack of buffers in host memory. This reduces goodput and PCIe bandwidth utilization.
2. Beyond two cores, the E810 becomes PCIe bound, and consumes up to 84.1 Gbps of PCIe read bandwidth. This is close to 85 Gbps, the theoretical limit for PCIe Gen3 x16 with 64-byte transfers [126].
3. Even though the E810 has a lower goodput, it consumes more PCIe bandwidth due to metadata. This overhead accounts for up to 39% of the PCIe read bandwidth. In contrast Ensō’s metadata overhead remains below 1.2%.
4. Although barely noticeable in the plot, Ensō’s PCIe write bandwidth utilization increases as we increase the number of cores: going up from 76.4 Gbps (one core) to 77.2 Gbps (eight cores). This is because the NIC sends reactive notifications more frequently when software consumes packets faster.

While newer PCIe generations, including PCIe Gen 4, have higher capacity and will no longer be a bottleneck for 100 Gbps traffic, they will continue to be a problem for NICs that have multiple 100 Gbps interfaces and for future 400 and 500 Gbps NICs. Even with PCIe Gen 5 and a 400 Gbps NIC, the ratio of PCIe to Ethernet bandwidth remains the same as in our setting with PCIe Gen 3.

Reactive Notifications and Latency

As we discussed in § 3.3.2, Ensō is able to reduce metadata overhead by sending notifications reactively. We measure the impact reactive notifications have on throughput and latency, by comparing Ensō’s performance (reactive) to that of a variant of Ensō (per-packet) that sends a notification for each packet. We again reuse the echo server from previous microbenchmarks for this.

Figure 3.11 shows the RTT (50th and 99th percentiles) as we increase load for both cases. While reactive notifications can sustain up to 100 Gbps of offered load, a design using per-packet notifications can only sustain 50 Gbps. However, reactive notifications also add latency with increased load.

We use notification prefetching to minimize latency under high loads.⁶ When using notification prefetching, the software explicitly sends the NIC a request for notifications pertaining to the next Ensō Pipe, while consuming data from the current Ensō Pipe. This effectively doubles

⁶By default Ensō does not prefetch notifications. Latency-sensitive applications may enable notification prefetching at compile time.

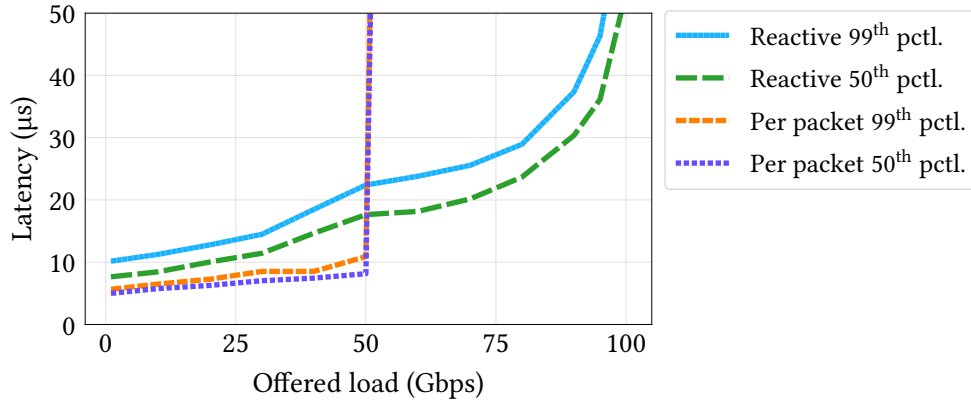


Figure 3.11: RTT for different loads when using a notification per packet or reactive notifications without notification prefetching.

the number of notifications that the NIC sends to software at a high rate but ensures that the software does not need to wait for a PCIe RTT before processing the next Ensō Pipe.

Figure 3.12 shows the RTT with an increasing load for Ensō with and without notification prefetching and for an E810 NIC with DPDK. We observe that notification prefetching significantly reduces Ensō’s latency, and allows us to achieve latency comparable to the E810, while still sustaining 100 Gbps.

Sensitivity Analysis

Finally, we use microbenchmarks to evaluate Ensō’s sensitivity to different configuration parameters:

Impact of the number of Ensō Pipes. We measure the impact of increasing the number of Ensō Pipes by varying the number of active Ensō Pipes, and using a workload where each incoming packet goes to a different Ensō Pipe. We partition Ensō Pipes evenly across all cores. Our results in Figure 3.13 show that (a) we need at least two Ensō Pipes to achieve line rate, since this allows us to mask notification latency; and (b) that throughput drops when a core has more than 32 Ensō Pipes, or eight cores have more than 512. Note that this is a pessimal workload, and realistic workloads are likely to perform better even with many queues [60].

Impact of packet sizes and cores. In Figure 3.14 we measure the impact of varying packet size and number of cores, and find that Ensō can always sustain full line rate, regardless of packet size and core count.

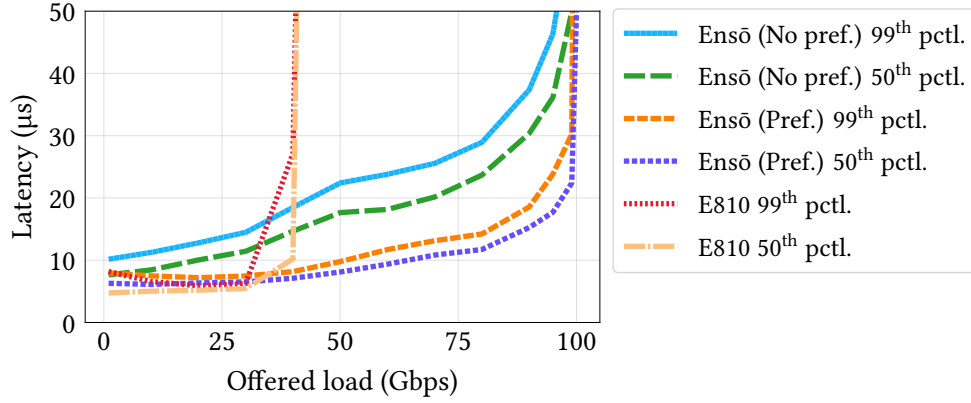


Figure 3.12: RTT for different loads for the E810 as well as Ensō with and without notification prefetching.

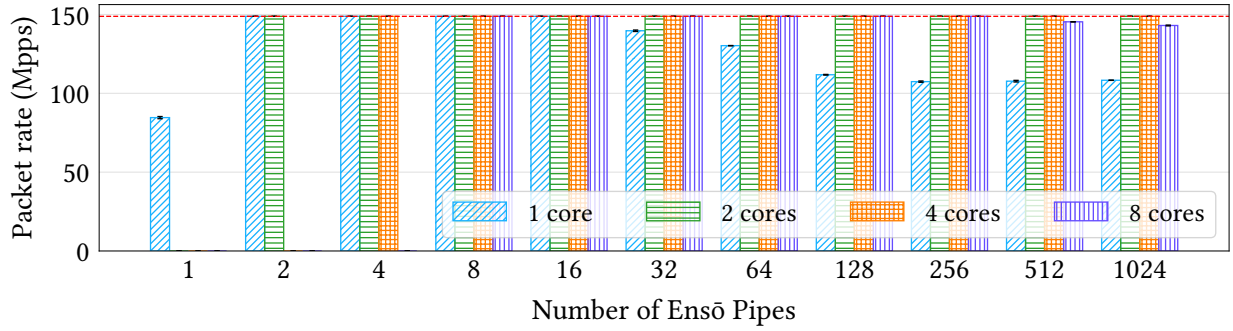


Figure 3.13: Packet rate for different numbers of Ensō Pipes and core counts. The dashed line represents the 100 Gb Ethernet limit.

3.7.4 Application Benchmarks

We now evaluate how Ensō impacts the performance of real applications. We ported four different applications to use both DPDK and Ensō. These applications represent three classes of network-intensive applications (raw packets, message-based, and streaming) that we expect to be used with Ensō: Google’s Maglev Load Balancer [48], a network-telemetry application based on NitroSketch [111], MICA Key Value Store [107], and a log monitor inspired by AWS CloudWatch Logs [15]. To enable a fair comparison, we use the same processing logic for both DPDK and Ensō-based implementations, changing only the wrapper code used to send and receive packets. Moreover, we only enable simple traditional offloads on the NIC, e.g., RSS, Flow Director, and checksum, for both Ensō and DPDK. We expect Ensō to perform even better with more offloads on the NIC (§ 3.6).

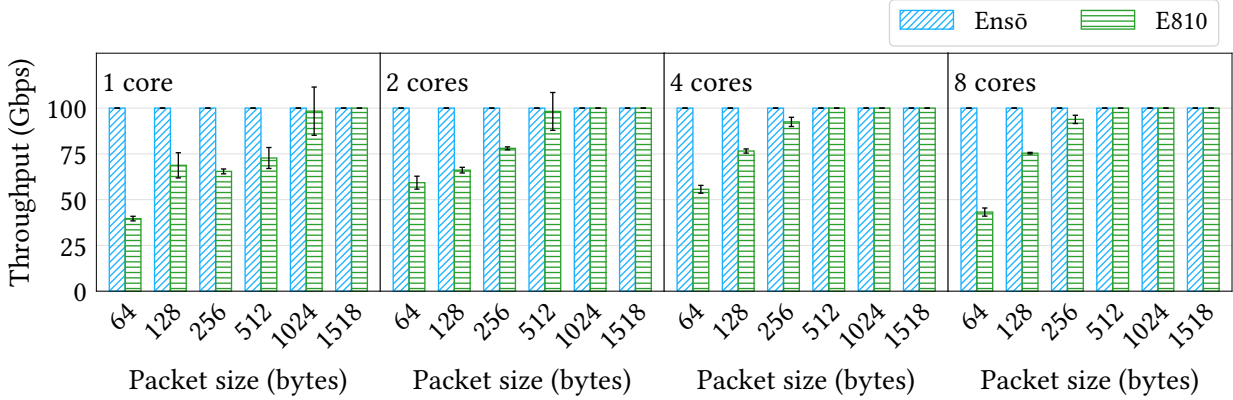


Figure 3.14: Sensitivity analysis with different numbers of cores and packet sizes. Ensō is bottlenecked by 100 Gb Ethernet in all scenarios.

Maglev Load Balancer

We implemented Google’s Maglev load balancer [48] as follows. We replicate the consistent-hashing algorithm proposed in the Maglev paper, caching recent flows in a flow table, as also suggested in the paper. The load balancer ultimately determines a backend server for every incoming packet, rewriting the packet’s destination IP to that of the chosen backend server. To steer packets among different cores, we use a hash of the 5-tuple (RSS) in both systems. We evaluate our implementation using two extreme types of workloads: one with only 16 flows, which means that packets always hit the flow cache; and another with a SYN flood, which means that packets always miss the flow cache. In both cases, we use small 64-byte packets as Maglev is motivated by the need to load balance small requests [48, § 3.2]. For Ensō, we use unified RX/TX Ensō Pipes to avoid copying the packet when forwarding it back.

Figure 3.15 shows the packet rate with both the E810 NIC using DPDK and Ensō as we scale the number of cores. With a single core and the cached workload, Ensō achieves a packet rate of 138 Mpps, approximately 6× the 23 Mpps achieved by the E810. With the SYN flood workload, Ensō achieves 79 Mpps, approximately 5× the 16 Mpps achieved by the E810.⁷ With four cores, Ensō becomes bottlenecked by Ethernet for both workloads; and with eight cores, the DPDK implementation becomes bottlenecked by PCIe (§ 3.7.3).

⁷We note that DPDK’s packet rate of 16 Mpps with a single core is in fact a good packet rate for DPDK. For instance, NetBricks’ Maglev implementation achieves 9.2 Mpps with a single core [146]. We attribute the improvement in our DPDK numbers to NetBricks’ unoptimized implementation and our use of a newer CPU with better single-thread performance.

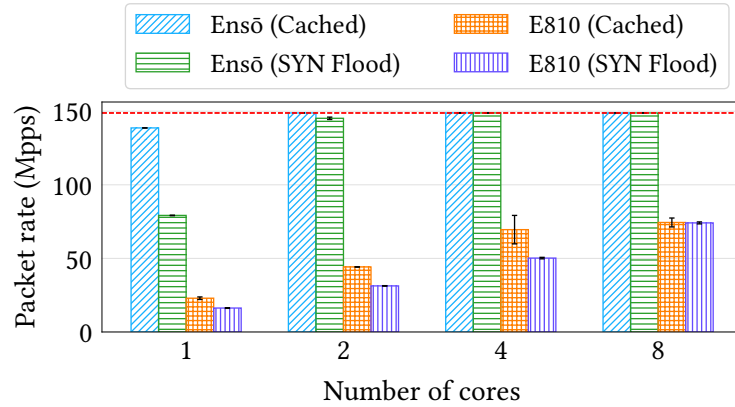


Figure 3.15: Packet rate for the Maglev load balancer under two types of workloads. The dashed line is the 100 Gb Ethernet limit.

Network Telemetry

Sketching algorithms are popular primitives for many network telemetry tasks (e.g., heavy-hitter detection, flow count estimation) because of their small memory footprint and theoretical accuracy guarantees. NitroSketch [111] is a sketching framework that enables *software sketches* to achieve high performance on commodity servers without sacrificing accuracy. To evaluate this class of applications, we implemented a Count-Min Sketch (CMS) using the NitroSketch framework and Ensō. As in Maglev, we use unified RX/TX Ensō Pipes.

We benchmarked our implementation using two workloads: 64B packets (emulating the stress-test performed in [111]), and a busy period sampled from the 2016 CAIDA Equinix 10G dataset [30], with an average packet size of 462B. With a single core, Ensō sustains a zero-loss throughput of 81.5 Gbps (121 Mpps), approximately 3.5× that achieved by the E810 (22.8 Gbps or 33.9 Mpps) on 64B packets. On the CAIDA trace, Ensō achieves 94.9 Gbps, a 21% improvement over the E810 (78.3 Gbps); the application remains compute-bound in this setting, but Ensō shrinks the fraction of time spent performing network I/O, which improves performance.

MICA Key-Value Store

MICA [107] is a state-of-the-art key-value (KV) store which is also a popular benchmark in the literature [71, 96, 97, 102, 157, 183]. Different from Maglev and the Network Telemetry application, that operate on raw packets and forward them back with modifications, MICA represents a typical message-based application whose responses must be constructed separately from the incoming request. Also different from these applications, latency is typically more critical for key-value stores [19]. MICA also entails significantly more work per packet (in terms of both compute and memory accesses) and is, therefore, less likely to become network-bound.

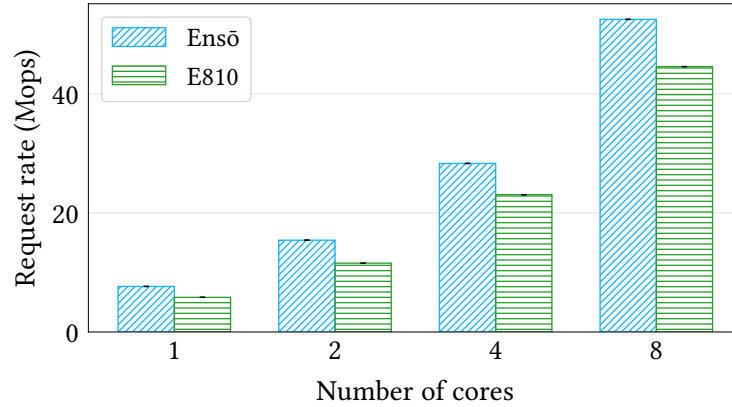


Figure 3.16: MICA throughput with 8B keys and values.

For the following experiments, we set the size for both keys and values to 8B (corresponding to the “tiny” configuration in [107]). We report results for operations skewed 50% towards GET requests and with a uniform distribution of key popularity, but these generalize to other configurations as well. We use the same throughput metric as described in [107] (tolerating <1% loss at the NIC), and the same methodology for measuring end-to-end latency (the client tags each request with an 8B timestamp, then computes latency based on the arrival time of the corresponding response).

Figure 3.16 shows the steady-state throughput in millions of operations per second (Mops) achieved by MICA for both E810 with DPDK and Ensō for different numbers of cores. With a single core, Ensō achieves 7.65 Mops, a 31% improvement over the E810.⁸ While this might seem modest at first (compared to the 6× speedup on Maglev), note that MICA is significantly more compute- and memory-intensive than Maglev. Thus, while DPDK adds considerable CPU overhead, it corresponds to a smaller fraction of the overall compute time. With 2 and 4 cores, we see throughput speedups of 33% and 23%, respectively. At 8 cores, the bottleneck moves to a different part of the system (i.e., memory). We report numbers for the “tiny” configuration since it represents a significant fraction of requests found in real workloads [12, 116], while also being the most challenging workload for MICA. We also tested other configurations with larger keys and values, obtaining up to 29% improvement for the “small” workload (16B keys and 64B values) and up to 12% for the “large” workload (128B keys and 1024B values).

We also evaluate latency, plotting the average request latency as a function of the offered load when using a server with a single core (Figure 3.17a) or four cores (Figure 3.17b). For both configurations, we find that Ensō outperforms the E810 in terms of both throughput and latency.

⁸For consistency, all the MICA experiments use the original, unmodified MICA client implemented with DPDK. We observe even better performance when using a MICA client ported to Ensō (up to 47% improvement in throughput).

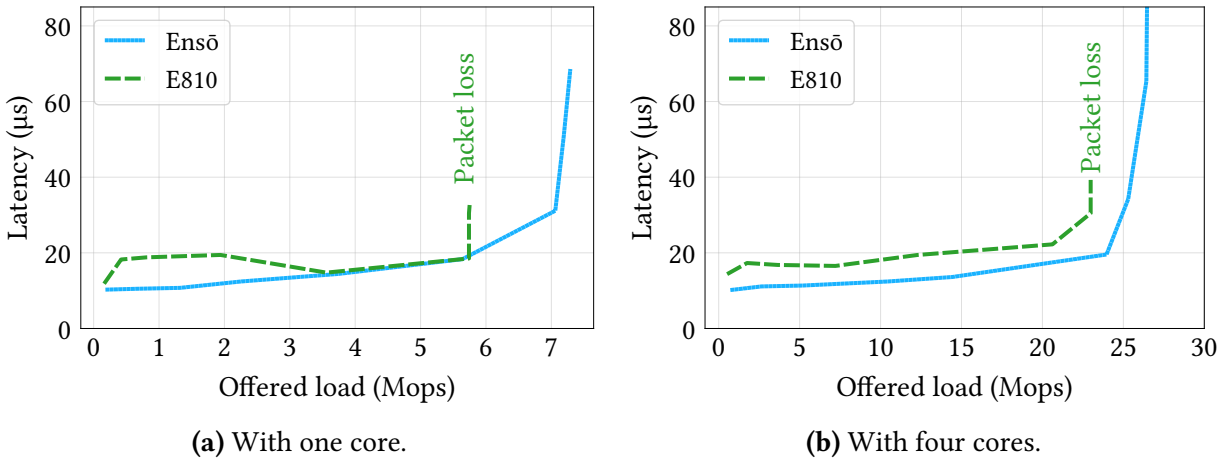


Figure 3.17: Mean RTT for MICA as a function of offered load.

With a single core, Ensō reduces latency by up to $8\mu\text{s}$ (43% reduction) before the queues start to build up and, with four cores, Ensō reduces latency by up to $6\mu\text{s}$ (36% reduction).

Log Monitor

To understand Ensō’s effect on streaming applications, we implemented a log monitor. The log monitor is inspired by AWS CloudWatch Logs [15], which lets users centralize logs from different hosts and craft queries to look for specific patterns and raise alarms. Like, AWS CloudWatch Logs Insights [16], the log monitor also supports Hyperscan [195] to search for multiple regular expressions. We use Hyperscan in streaming mode for both Ensō and DPDK implementations. To feed the system, we use MTU-sized packets, carrying system logs extracted from long-running Linux hosts. We also configured the log monitor to look for regular expressions extracted from Fail2ban [90]. We run experiments targeting each of the ten most popular applications supported by Fail2ban according to the Debian package statistics [40].

Figure 3.18 shows the throughput we achieve when targeting each of the ten applications. Performance is dictated primarily by the number and complexity of the regular expressions that are required by each target. Ensō’s throughput is higher across all targets but the gap is more noticeable for those with simpler or fewer regular expressions, with almost double the throughput when targeting postfix, selinux, or sieve. The reasons for Ensō’s improvement in performance are twofold: First, Hyperscan performs better when larger chunks of data are handed to it at once. With Ensō, we can invoke Hyperscan with large chunks of contiguous logs delivered from the NIC but with DPDK we need to invoke Hyperscan for every DPDK mbuf. Second, as demonstrated in § 3.7.3, Ensō’s memory access patterns are sequential, making better use of the CPU prefetcher.

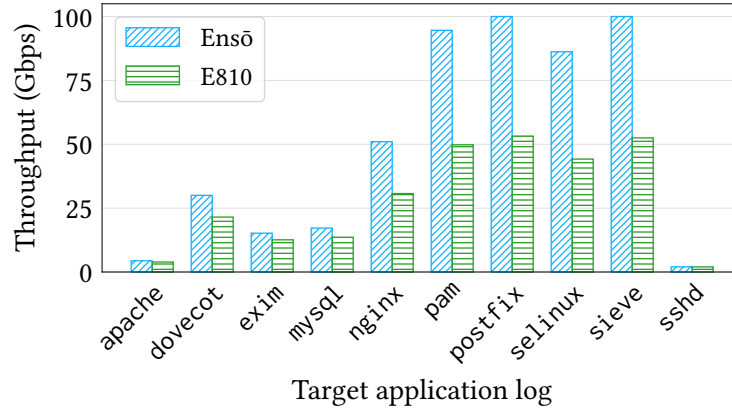


Figure 3.18: Log monitor throughput for different target applications.

3.8 Related Work

We now discuss work that relates to Ensō along three dimensions: Allowing applications to directly access the NIC (§ 3.8.1), alternative NIC interfaces (§ 3.8.2), and proposals—in both hardware (§ 3.8.3) and software (§ 3.8.4)—that address some of the performance issues that Ensō tackles but that are specialized to some application.

3.8.1 Direct Application Access

While giving applications direct access to the NIC has been a common theme of research for more than three decades [20, 46, 51, 69, 101, 153, 166, 168, 185, 199, 200], most work accepts the NIC interface as a given and instead look at how to optimize the software interface exposed to applications. A notable exception is Application Device Channels [46], which gives control of the NIC to the kernel while giving applications independent access to different queues. We take inspiration from it in the way that we allow multiple applications to share the same NIC.

3.8.2 Alternative NIC Interfaces

There are also proposals that try to address some of the performance and abstraction issues that we highlighted for the packetized interface.

In terms of performance, Nvidia MLX 5 NICs [44] provide a feature named Multi-Packet Receive Queue (MPRQ) that can potentially reduce PCIe RD bandwidth utilization with metadata by allowing software to post multiple packet buffers at once. However, this is not enough to completely avoid PCIe bottlenecks as the NIC still needs to notify the arrival of every packet, consuming PCIe WR bandwidth. Another proposed change to the NIC interface is Batched

RxList [158]. This design aggregates multiple packets in the same buffer as a way to allow descriptor ring buffers to be shared more efficiently by multiple threads, which in turn could help them avoid the leaky DMA problem [186].

In terms of abstraction, U-Net [193] and, more recently, NICA [52] allow the NIC to exchange application-level messages directly. U-Net proposes a communication abstraction that resembles part of what is now libibverbs (RDMA) [164] and NICA uses a similar mechanism named “custom rings.” However, similar to the packetized interface, both U-Net and NICA use descriptors and scattered buffers and, as such, inherit its performance limitations.

3.8.3 Application-Specific Hardware Optimizations

Prior work has optimized the NIC for specific applications. FlexNIC [100] quantifies the benefits that custom NIC interfaces could have to different applications. NIQ [55] implements a mechanism to reduce latency for minimum-sized packets by using MMIO writes to transmit these packets. It also favors MMIO reads over DMA writes for notifying incoming packets. NIQ’s reliance on MMIO means that it is mostly useful for applications that are willing to vastly sacrifice throughput and CPU cycles to improve latency. nmNFV [157] stores packet payloads on NIC memory, sending only the packet headers inlined inside descriptors, which is useful for network functions that only need to modify the header. This is orthogonal to Ensō’s interface changes and could also be used in conjunction with it.

3.8.4 Application-Specific Software Optimizations

Some proposals avoid part of the overheads of existing NICs with application-specific optimizations in software. TinyNF [156] is a userspace driver optimized for network functions (NFs). It relies on the fact that NFs typically retransmit the same packet after processing. It keeps the set of buffers in the RX and TX descriptor rings fixed, reducing buffer management overheads. eRPC [97] is an RPC framework that employs many RPC-specific optimizations. For instance, it reduces transmission overheads by ignoring completion notifications from the NIC, instead relying on RPC responses as a proxy for completions. FaRM [45] is a distributed memory implementation. It uses one-sided RDMA to implement a message ring buffer data structure that has some similarities to an Ensō Pipe. However, different from an Ensō Pipe, FaRM’s message buffer is not opaque (enforcing a specific message scheme), must be exclusive to every sender, and lacks a separate notification queue (requiring the receiver to fill the buffer with zeros and to probe every buffer for new messages).

3.9 Conclusion

Ensō provides a new streaming abstraction for communication between software and NICs. It is better suited to modern NICs with offloads and improves application throughput by up to 6× by being more cache- and prefetch-friendly and by reducing the amount of metadata transferred over the host interconnect. While this chapter focused on using Ensō for NIC-to-software communication, we believe that a similar approach might also apply to other I/O devices and accelerators.

In the context of this dissertation, Ensō establishes the first half of the thesis: a device interface need not be specialized to a data format to be fast—removing the format assumption is precisely what enabled the performance gains we measured. But Ensō still inherits the second source of specialization. It connects a device to the CPU and only to the CPU, and its coordination leans on that fact: Ensō Pipes reside in host memory, and the CPU remains one end of every transfer. In the next chapter, we remove this last assumption and generalize “whom” devices communicate with.

Chapter 4

Generalizing Across Endpoints

This chapter addresses the second front of specialization: “whom” devices communicate with. The interfaces we have seen so far, including Ensō in Chapter 3, are asymmetric. They place data in host memory and rely on the CPU as the centralized coordinator. As we discussed in Chapter 1, this asymmetry forces device-to-device communication through the CPU.

Increasingly, modern applications leverage functionality sharded across accelerators and I/O devices within the server ecosystem: PCIe-attached compute substrates such as GPUs, NPUs, and FPGAs; task-specific accelerators like encryption/decryption engines (e.g., Intel QAT); and high-performance I/O devices such as SmartNICs. Leveraging this heterogeneity and the idiosyncratic strengths of these devices has enabled applications to move past immediate compute bottlenecks in their datapaths.

Nonetheless, *combining* the benefits of multiple devices remains bottlenecked by the legacy of CPU centrality. When, for example, a SmartNIC has data that must be decrypted by a custom encryption accelerator, or raw image data must pass from a camera to a hardware video encoder, the devices must still rely on the CPU for *coordination*. Per the traditional server model, responsibility for all intra-host data transfer is delegated to the CPU: devices (e.g., NICs or GPUs) connect to the CPU root complex via an intra-host network supported by an *interconnect protocol* such as PCIe or CXL; the CPU then programs each device with information about *memory allocation* (i.e., where data should be moved to and from, with either the source or destination pointing to an address in host memory), and *data movement* (i.e., when the transfer should occur).

As we will show in § 4.1, invoking the CPU to orchestrate data transfers can entail significant performance overheads in terms of latency, throughput, and compute cycles. In effect, it puts the CPU back on the critical path for every device, a problem reminiscent of the one that special-purpose hardware was meant to address in the first place.

The persistence of CPU-as-intermediary is surprising in light of the fact that interconnect protocols (e.g., PCIe, CXL) have supported P2P communication since their inception. *The problem,*

we argue, is that interconnect protocols alone make insufficient provisions for coordination between devices regarding memory management or data movement.

In recent years, we have witnessed a proliferation of bespoke, closed-source solutions (e.g., GPUNetIO [66, 134], GPUDirectRDMA [133]) ostensibly striving for the same set of goals in bypassing the CPU in device-to-device transfers: reducing transfer latency and PCIe congestion, and avoiding wasted CPU cycles. Most of these protocols operate as a separate layer above the existing interconnect protocol, bridging the gap between what protocols like PCIe provide and what is needed to carry out P2P transfers end to end. However, in their quest for performance, these end-to-end protocols are highly specialized to operate between *specific pairs* of devices (GPU-to-GPU, GPU-to-NIC, GPU-to-Disk, Disk-to-NIC). Not only does this artificially constrain the set of devices that can coexist in the same server, but it also raises the barrier to entry for new devices and accelerators.

In this chapter, we ask: *Is there a general approach for coordinating memory allocation and data movement atop the existing interconnect that scales with both the diversity and number of devices, while still achieving close-to-optimal performance enabled by the physical fabric?*

The key challenge in designing a general coordination scheme is that accelerators and I/O devices express significant *heterogeneity in memory and compute capabilities*, and therefore have mismatched expectations about the *where* and *when* of communication. Developing a unifying approach means catering to this full spectrum of device capabilities.

Where should data be stored before and after transfer? Many simple devices expose no BAR-accessible memory and instead receive pointers to host memory; they read in data, operate over it, and write the result back out to host memory. Hence, the idea of “receiving” data from another sender is an entirely alien concept. At the other extreme, GPUs expose gigabytes of memory and can receive data directly written into memory via RDMA. This mismatch complicates protocol design as the simpler devices are more closely tied to a shared memory model, while the more sophisticated devices might be amenable to sockets or streams.

When should a sender transmit? Because most protocols are designed with the expectation of a CPU-to-device communication pattern, the sender is expected to know when the receiving device is busy versus idle. Devices explicitly send “completion” notifications once they are done performing work, informing a sender (typically the CPU) that they are ready to receive more data. In a general coordination scheme—as seen in collective computation clusters of GPUs and ML accelerators [93, 110]—arbitrary devices may communicate with each other, even simultaneously. So how can a sender know that the next device is available to receive data, or if it is busy with work passed to it from another device?

We propose a general-purpose peer-to-peer Device Communication Protocol (DCP) centered on a minimal and flexible interface called the Memory-Mapped Queue (MMQ). An MMQ is a BAR-accessible memory region that, when written to, enqueues a request to transfer data to or from a particular device. On the surface, MMQs might resemble the classic “doorbell” interface

used by many accelerators. With doorbells, the CPU initiates an MMIO write to signal to the device that data and instructions are available at a provided memory address; the device then reacts to the doorbell by reading in the data and instructions to perform. With an MMQ, sending devices use writes to directly enqueue requests for processing at the receiving device, and the receiver, in turn, reacts. However, there are several key differences from the doorbell model.

First, instructions or code are bound to the MMQ when the MMQ is instantiated, not passed with the request itself. This abstraction means that the receiving device invokes the same functionality over all data passed via the same MMQ; the sending device only ever needs to know which MMQ to send data to, but does not need to be aware of what instructions or code are required at the receiver. As a consequence, devices do not need to be generally programmable and install “drivers” in order to communicate with other devices. The MMQ design then minimally standardizes what is needed to carry out data transfer. To perform data transfer, devices need to coordinate the *where* and the *when* as discussed above.

Coordinating the *where* is challenging because the MMQ needs to enable transfers when both sender and receiver have BAR-addressable memory, only one has such memory, or even when neither supports memory. For small transfers that fit into a single interconnect frame, MMQs support a *pass-by-value* transfer in which data is pushed directly into the MMQ itself. For larger transfers, MMQs support a flexible *pass-by-reference* transfer where data can be transferred via a pointer to memory in the sender’s address space, in the receiver’s address space, host address space, or any other PCIe-attached memory. The pass-by-reference model enables direct device-to-device transfer when *at least one* of the sender and receiver has BAR-addressable memory, and enables transfer via a “bounce buffer” when neither device does.

Coordinating the *when* is a new challenge relative to the doorbell model because the doorbell model does not need to account for multiple senders transmitting to the same receiver, as is common in collective computation tasks [70]. The MMQ enables multiplexing while rate-limiting. Multiple senders may enqueue data at the MMQ for processing without preempting each other; requests are served first-in-first-out. To avoid overflowing the MMQ, senders may only insert into the MMQ when they have been granted a credit by the receiver; receivers with deep buffers can grant more credits and scale out more extensively than receivers with shallow buffers, but the mechanism remains the same regardless of device capacity.¹ This solves the “when” problem.

While generality is a useful guiding principle, it is only meaningful when it entails *low resource overhead* and *high performance*. Despite the conceptual complexity of supporting abstractions that are flexible enough to support device heterogeneity, a fully featured DCP implementation on a commodity AMD AU55C FPGA [7] consumes only 99k LUTs (7.6% of

¹Note that these credits are orthogonal to underlying flow control in the interconnect protocol. Interconnect protocols rate limit transfers to avoid overloading network bandwidth; DCP’s rate limiting avoids overloading MMQ buffer space.

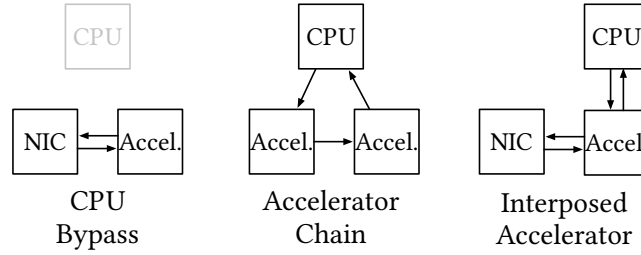


Figure 4.1: Use-cases DCP aims to support.

total) and 153k registers (5.9% of total). This represents an extra 4% of the overall utilization compared to a simple specialized interface with minimal buffer. In our microbenchmarks, DCP achieves up to $2.3\times$ higher goodput for device-to-device communication and $1.4\mu\text{s}$ latency (17% of the baseline) *even with just two devices*. Even for GPU-to-GPU communication over NVLink, where a single vendor controls the entire stack, DCP achieves 20% higher peak throughput than Nvidia’s specialized NVSHMEM interface [137] while using a quarter of the SMs.

4.1 Background and Motivation

We start by describing the use cases we target in this chapter, and then analyze the overheads of implementing them when CPUs mediate transfers. Finally, we discuss why existing device-to-device communication approaches that were designed to avoid CPU mediation do not suffice.

4.1.1 Target Use-Cases

Figure 4.1 shows some application dataflows between accelerators that we aim to support, and we describe these below. But we hasten to note that the workflows we describe here are among the simplest we support. This simplicity is in service of clarity, and our design supports more complex flows, including one with more accelerators or richer communication topologies. Later, in § 4.4 we detail how these application dataflows can be implemented using DCP.

CPU bypass. This pattern resembles what is used by today’s machine learning training and inference jobs: an accelerator (e.g., a sparse tensor accelerator [59]) exchanges network data by directly communicating with the NIC rather than going through the CPU. Of course, the same pattern is useful in many other contexts, including storage servers where the disk controller might directly talk to the network [140] or video streaming clients, where the network input might be sent directly to the decoder. GPUDirect and similar proprietary solutions can support CPU bypass use-cases where one of the devices is a GPU. However, as we discuss in § 4.1.3, these solutions do not generalize beyond the GPU use-case.

Accelerator chain. This dataflow pattern models a case where the CPU might use two accelerators to process data, e.g., a video analytics application might need to decompress inputs (using a QAT [86]-like accelerator), then decode them (using a video decoder) before processing the raw frames. As we show below, going to the CPU after each step can add substantial overhead, and DCP allows applications to avoid the CPU and thus these overheads.

Interposed accelerator. The final dataflow pattern models a case where an additional accelerator interposes on I/O initiated from the CPU, e.g., an RPC accelerator [99] that interposes on network traffic, thus eliminating serialization and deserialization overheads. The goal of such interposition is to reduce CPU cycles (the so-called “datacenter tax” [182]), but as we discuss below, current CPU-mediated approaches require cycles and thus reduce (or in the worst case, obviate) these benefits. Direct device-to-device communication enabled by DCP eliminates the processing overheads, thus maximizing the gains provided by such an accelerator.

4.1.2 Overheads of CPU-Mediated Communication

Currently, proprietary technologies, e.g., GPUDirect, can be used to implement the communication patterns described above for compatible devices: for example, an Nvidia GPU can directly communicate with an Nvidia NIC using GPUDirect RDMA [133]. But these approaches are proprietary and cannot be used by other devices. More importantly, as we explain in § 4.1.3, these approaches assume that at least one of the communicating devices (usually the GPU) has similar computation capabilities and memory capacity as the CPU, and thus they cannot be extended to apply to general devices.

Consequently, the most promising approach to implementing these use cases today would be to use CPU mediation (Figure 4.2). In this approach, communication between any pair of devices require that the CPU first ① *allocate memory* in host DRAM to buffer the data (a bounce buffer). After this buffer has been allocated, the CPU must then orchestrate *data movement* by ② initiating a DMA write from the sender to the buffer, e.g., by enqueueing a descriptor; ③ polling until the DMA write has completed; ④ initiating a transfer to the receiving device (either using MMIO write or by triggering a DMA read); and finally ⑤ polling until the transfer has completed, following which we can free the bounce-buffer.

CPU mediation allows for simpler devices because they can be designed assuming that all data transfer is initiated and coordinated by the CPU. However, it also has performance and resource overheads. Below, we characterize this design’s impact on communication latency and throughput, and on the additional CPU cycles required to implement this.

Our characterization is based on a microbenchmark where the communication chain is CPU → FPGA A → FPGA B → CPU. As can be observed, the main memory is both the source and sink for data. The FPGA datapath is modeled on Intel QAT; each device has a CPU-populated descriptor ring buffer (of 64-byte descriptors) encoding two addresses in host memory: an input

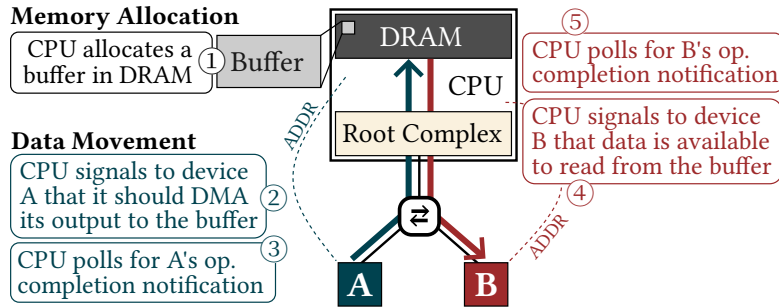


Figure 4.2: CPU-mediated communication.

buffer and an output buffer. The output buffer also serves as a completion notification that the CPU polls on.

Impact on latency. CPU mediation doubles the path length of any device-to-device communication, requiring a trip to and from the CPU. Concretely, consider the path between FPGA A and FPGA B. In the CPU-mediated case, this requires FPGA A to first send data to the CPU (and main memory), and FPGA B to fetch this data from the CPU, incurring two one-way PCIe transfers, in the *best possible scenario*. In our testbed, we estimate that this takes 3 μ s. By contrast, a more direct communication between FPGA A and FPGA B requires only one PCIe transfer, and we estimate that this would take 1.5 μ s (based on the unloaded round-trip time measured between the CPU and FPGA). Worse, this latency overhead increases with the length of the accelerator chain.

Impact on throughput. CPU-mediated transfers increase traffic on arterial PCIe links, leading to congestion and reduced throughput. Returning to Figure 4.2, using a DRAM-based bounce buffer means that the Root Complex $\leftrightarrow$ PCIe Switch link must handle *all* traffic in and out of both FPGAs, resulting in 2 $\times$ oversubscription. Consequently, with PCIe 3.0, inter-device transfer rates would be throttled to *at most* 64 Gbps, despite the fact that each link provides 128 Gbps bidirectional bandwidth. This effect aligns with prior work highlighting PCIe as a bottleneck under heavy data movement workloads [3] as well as what we observed in Chapter 3. In our microbenchmarks, the effective throughput is just 51 Gbps due to additional coordination overheads (descriptors, PCIe read requests). Once again, we will see in § 4.6.3 that enabling devices to coordinate transfers autonomously unlocks peer-to-peer (P2P) DMA capabilities of the underlying fabric, alleviating congestion in the upstream links, and improving the end-to-end throughput to 119 Gbps (within 97% of the theoretical maximum).

Wasted CPU cycles. Finally, CPU mediation requires that the CPU allocate buffers, poll for completion, and initiate DMAs. Each of these actions requires CPU cycles. These cycles are necessary even though, for device-to-device traffic, e.g., communication between FPGA A and B in our example, the CPU performs no useful computation. Furthermore, because the sequence

of steps in Figure 4.2 must be replicated for every *pair* of devices the CPU coordinates, longer device chains only exacerbate the problem, requiring increasingly more CPU cores in order to keep up with the load.

4.1.3 Limitations of Existing Peer-to-Peer Approaches

The analysis above shows that CPU mediation limits the benefits of chaining multiple accelerators. We are not the first to notice this, and peer-to-peer communication has been a focus for a while; we discuss existing approaches here.

Proprietary GPU-centric mechanisms. The most commonly used approach today relies on proprietary protocols, e.g., GPUDirect [133] and GPUNetIO [66, 134]. These protocols have focused on interconnecting GPUs to I/O devices (e.g., the NIC or storage) because of the importance of ML jobs and the I/O pattern exhibited by ML training and inference.

The design of these protocols is tightly coupled with the assumption that they are used for communication with a GPU: they rely on the programmability and memory capacity of GPUs, and thus merely substitute CPUs for GPUs. Put differently, they can be considered to change CPU-mediated communication to GPU-mediated communication. For example, GPUDirect leverages an RDMA-capable NIC to move data directly between GPUs on different servers, but implicitly assumes that the data sinks at both endpoints (GPUs) have high memory capacity. In the same vein, GPUNetIO allows CUDA kernels running on the GPU to orchestrate NIC RX and TX; this effectively adapts the CPU-as-controller model to a GPU-centric world, but therein assumes sufficient memory and general-purpose compute at the GPU.

Consequently, these protocols cannot be generalized and applied to communication between devices with limited compute capabilities, e.g., hardware video encoders or decoders, or those that have little or no BAR-addressable memory, e.g., compression engines.

Thus, these specialized protocols are misaligned with our goal of allowing applications to implement functionality by chaining several accelerators, each with varying capabilities.

PCIe peer-to-peer provides *mechanism*, but not *coordination*. Since 2002, PCIe has supported peer-to-peer communication. Unlike proprietary protocols, it is easier to adopt and does not assume the presence of a GPU (nor a CPU). Instead, they allow devices to exchange data directly with other devices by reading and writing to their BAR-accessible memory regions, with routing handled by intermediate PCIe switches along the interconnect tree.

However, while the PCIe peer-to-peer mechanism is necessary for inter-device communication of the form we described above, it is not sufficient. This is because it does not provide coordination mechanisms to address two problems:

- *Memory allocation.* First, PCIe transactions, whether read or write, need to specify memory addresses from (or to) which data should be transferred. Consequently, in many cases,

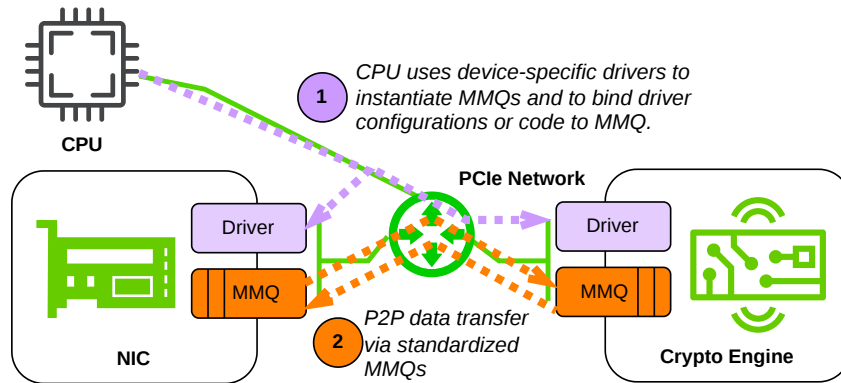


Figure 4.3: DCP does not replace device drivers. Instead, drivers are used by the CPU in the control plane to configure devices and instantiate MMQs. DCP, the protocol that runs between MMQs, minimally standardizes the mechanisms for dataplane operations, i.e., transferring data between devices.

devices need to be able to allocate and free buffers when necessary, but PCIe does not provide any standard mechanism for this.

- *Data movement.* Second, devices need to signal when data movement should be initiated, and when it has completed. For example, in Figure 4.2, Device A needs to be able to signal to Device B that its write has completed, and thus Device B can safely read the buffer. Similar to memory allocation, PCIe currently provides no standard mechanism for this purpose.

Thus, while PCIe provides a foundation, it does not offer the higher-level mechanisms required to allow applications to chain together arbitrary accelerators.

4.2 DCP Vision

We envision a future where PCIe (or CXL, or a future interconnect) allows devices to arbitrarily exchange data via a standardized interface. A standardized interface would enable the wider adoption of *known* optimizations (e.g., enabling NICs to directly transmit data to accelerators [53, 66, 187]; allowing clusters of accelerators to perform collective computation [70, 136]) as well as open doors to a future where several accelerators work together to perform a task as a “sea of accelerators” [62] that have been stitched together into an application specific pipeline.

Enabling a standard is challenging due to the variability in device programmability, memory, and capacity for throughput. It is necessary to be conservative in the design of a standard interface such that it is feasible for as many devices as possible to implement, while still striving

to achieve competitive performance goals. In short, it is important to be careful about what the protocol *does* and what it *does not do*.

Our vision for a standardized interface is the *Device Communication Protocol*, or DCP. DCP standardizes the mechanics of memory allocation, data transfer and signaling, and flow control between devices with heterogeneous capabilities in terms of programmability, memory access, and capacity/throughput. The core interface to implement DCP is called a *Memory Mapped Queue*, or MMQ. The MMQ backs a BAR-addressable memory region that, when written to, enqueues a DCP command for the hosting device. Senders enqueue DCP commands at receivers to (a) request allocation of memory at the receiving device, for devices that host banks of BAR-addressable memory; (b) initiate or terminate data transfer between devices; and (c) request permission (credits) to invoke more commands.

This list of commands is deliberately short. As alluded to above, important to DCP’s design is what DCP does *not* do. In order to target generality, DCP does *not* attempt to standardize anything to do with device-specific configuration. Devices present varying forms of configuration (e.g., NICs might be configured with packet filters [75] or hash functions for receive-side scaling [192]; cryptographic engines might be configured with commands to invoke a particular algorithm with specific parameters [86]) and programmability (e.g., FPGAs require RTL code and GPUs require CUDA code). We cannot expect fixed-function devices—such as NICs, cryptographic accelerators, or codec engines—to support programmable drivers to enable them to send receiver-specific commands to each device they might communicate with. Furthermore, this sophistication is *not necessary* to enable a wide range of useful applications because we can instead rely on a programmable controller—likely at the CPU—to enable device-specific configurations and code while leaving the sending device to be only concerned with the standardized mechanisms of data transfer. We illustrate this with an example.

DCP in Action. In Figure 4.3, we consider a use case that illustrates the separation of responsibilities between device-specific functionality (enabled via drivers and programmed by the CPU) and DCP-implemented operations (performed peer-to-peer and concerned only with data transfer). In this example, a software router [42] serves as the endpoint for a site-to-site encrypted IP tunnel between offices of the same enterprise. To save CPU cycles for frequent cryptographic operations, the enterprise installs a cryptographic engine on the server. The tunneling software at the CPU uses a *driver* on each device to establish a flow of data between the NIC and the cryptographic engine directly. In step 1, the CPU establishes two MMQs, one at the NIC and one at the crypto engine. The CPU instructs the NIC to select all traffic over the port used for tunneling (using the NIC’s own configuration language) and to copy those packets to the Crypto Engine’s memory, and to coordinate this memory exchange using the MMQ at the address exposed by the Crypto Engine. The CPU instructs the Crypto Engine to decrypt the traffic arriving over the new MMQ using a particular algorithm and set of parameters (e.g., AES-GCM-256) and for all resulting data to be written back to local memory, and then to send a

request to the NIC to read this memory at the MMQ exposed by the NIC. The MMQ at the NIC is configured to send the resulting packets back to the local network.²

* * *

DCP standardizes a very narrow band of functionality: the mechanics of data transfer. In the example above, the NIC does not need to know about cryptographic algorithms, and the cryptographic engine does not need to know about network routing.

In the following section, we turn to the core components of what DCP *does* do.

4.3 Design

We now present DCP’s design. We discuss the MMQ (§ 4.3.1) and how senders enqueue commands to receivers via MMQs that have been instantiated by software; we discuss DCP’s flexible approach to data transfer and ownership (§ 4.3.2) that adapts to scenarios where either sender, receiver, or both lack expansive addressable memory (as observable in the tunneling example in the previous section); and finally we overview DCP’s credit-based flow control (§ 4.3.3), which prevents lower-capacity receivers from being overwhelmed by data from external devices.

4.3.1 Memory-Mapped Queue (MMQ)

All devices that support DCP must implement at least one *memory-mapped queue*, or MMQ. An MMQ allows senders to enqueue *commands* by writing to a specific address region. As we will discuss in § 4.3.2 and § 4.3.3, DCP has five command types: pass-by-value transfers, pass-by-reference transfers, buffer requests, credit requests, and completion notifications. Each command must fit into a single message (i.e., TLP or frame for the underlying interconnect) of some k bytes. Senders may only send commands if they have *credits* to do so; some number of credits j are preallocated at configuration time by the CPU. From a sender’s perspective, the MMQ looks like a simple ring buffer of $k \times j$ bytes at some address a with a head and tail pointer; a sender iterates through writes to addresses $a, a + k, a + 2k, \dots$ so long as credits are available; every time a credit is returned to the sender the head pointer implicitly advances.

While this abstraction is faithful to shared memory, the true implementation may not rely on random-access memory at all; indeed, we envision that most devices will implement MMQs with dedicated hardware FIFO queues. At the same time, the abstraction is simple to adopt for legacy devices that already have exposed memory and which are programmable (e.g., the CPU, GPUs).

²If this looks a lot like Software Defined Networking [31], that is because it is indeed, very much inspired by Software Defined Networking.

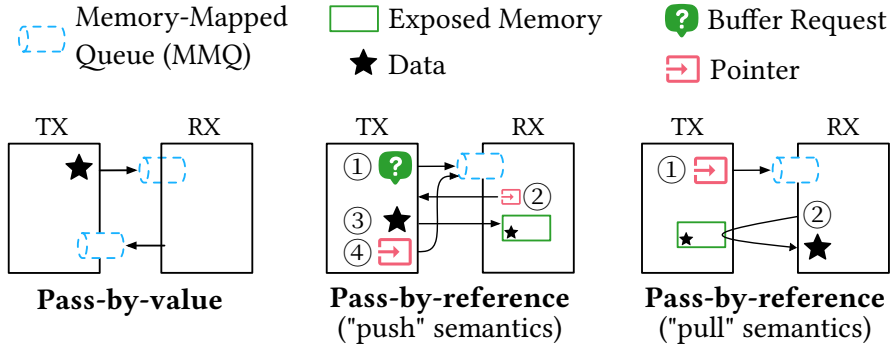


Figure 4.4: Data transfer modes in DCP.

Hardware queue implementation. In this approach, receive logic intercepts writes to the MMQ address space as frames arrive on the interconnect interface (e.g., PCIe MWr TLP) and then directly enqueues the packet payload into a hardware FIFO queue regardless of offset from the base address a . Hardware MMQs support multiple senders writing to the same queue: the arriving writes each fit in a single frame and hence can be handled atomically by the receiver, without the need for complex synchronization mechanisms. Devices may implement one or more MMQs in hardware (demultiplexing depending on which address is written to); each device has a fixed number of queues, of a fixed size.

Memory-based implementation. Because the MMQ abstraction replicates the abstraction of shared memory, devices with exposed memory and programmability can trivially support the abstraction without hardware changes. In this implementation, writes to the MMQ consist of regular memory writes. The receiver (device or CPU) keeps track of the head pointer, indicating the next command to process. It polls the head until the sender writes the command, at which point it processes the command and advances the head. Commands always start with a command type field, which allows the receiver to quickly determine the command type and how to process it. The receiver can also use this field to disambiguate between commands and empty slots by ensuring that empty slots always start with zero. Unlike the fixed hardware design, memory-based MMQs can scale out dynamically: CPUs and GPUs can create new MMQs when instructed to, so as long as memory is available.

However, the memory-based approach has significant disadvantages. First, memory-based MMQs can only support one sender because DCP does not support mechanisms to synchronize writes to the MMQ head; this requirement is enforced by the controller at the CPU when setting up the queues. Second, it requires the receiver to poll to detect new commands, eating cycles. Finally, it requires support for BAR-addressable memory and more complex pointer tracking, which is undesirable for limited devices.

4.3.2 Data Transmission and Ownership

Our core goal in designing DCP is to enable *direct* peer-to-peer communication without involving the CPU for coordination (e.g., invoking transfers) nor intermediation (writes and reads to host memory to enable transfer). DCP enables this for *all devices with an MMQ* for transfers carried out using a *pass-by value* command; DCP enables this for transfers such that *either* the sender or the receiver has addressable memory using *pass-by-reference* commands. In general (and as we will demonstrate in § 4.6), pass-by-value transfers offer the lowest latency, but pass-by-reference transfers are more flexible, not being susceptible to frame size limits.

Pass-by-value. Pass-by-value transfers embed the transmitted data directly into the payload for a command pushed into the MMQ. Individual transfers are capped in size by the maximum atomic write size supported by the path between the sender and receiver. This is primarily determined by two factors: the maximum frame size supported by the interconnect, which guarantees that the entire command is *delivered* atomically, and the atomic unit of the endpoints, which guarantees that the entire command is *processed* atomically. For instance, modern x86 CPUs typically allow TLPs with up to 256 B payloads, but only guarantee atomicity for writes of up to 64 B [34], so the maximum size that a pass-by-value transfer can support between an x86 CPU and a DCP device is 64 B. For peer-to-peer transfers between devices that go through a PCIe switch, the maximum size is determined by the maximum payload size (MPS) configuration of the switch and devices, which can be much larger (1024 B in the testbed we use in § 4.6). As made apparent in Figure 4.4, pass-by-value transfers offer the lowest latency as transfers complete with one-way delay between sender and receiver.

Pass-by-reference. Pass by reference can transfer more data via a single command as the sender pushes a pointer and length to the receiver’s MMQ; the receiver then reads data from the address up to the specified length. The specified address may point to data anywhere in the host address space (e.g., sender memory, receiver memory, interconnect-attached memory, or host memory), which allows the sender and receiver to make use of memory as available. Returning to Figure 4.4 (pull semantics), a sender can pass a reference to their own memory, and then the receiver can read it in, much like CPUs pass data to devices today.

However, some use-cases will lead to receiver-managed buffers. For example, a typical NIC does not manage buffers—it accepts data from the network and then writes it to a receiver that is expected to process the packet. To enable receiver-managed buffers, DCP offers a *buffer request* command by which senders can request that the receiver allocate a buffer in memory for the sender to write into, as illustrated in the “push” example of Figure 4.4. A sender first (1) sends a buffer request to the receiver’s MMQ; (2) the receiver responds with a pointer to allocated memory; (3) the sender writes into the receiver’s buffer; and (4) the sender transmits the same pointer back to the receiver for the receiver to read from. This process implicitly tracks *ownership* of the buffer: when the receiver passes the pointer to the sender, the sender now

owns the address space for writing; once the pointer is returned, the ownership is restored to the receiver who begins to read in data.³

Other pass-by-reference use cases. DCP also enables two devices that do not have any memory to use preallocated memory at the CPU, or CXL-attached memory, as a “bounce buffer” for pass-by-reference transfers; however, we expect that pass-by-value transfers will typically be sufficient.

4.3.3 Credit-Based Flow Control

DCP incorporates a receiver-driven, credit-based flow control mechanism to limit the amount of MMQ capacity a sender may consume at any given time. These credits serve two purposes. First, they rate-limit the aggregate RX rate to prevent overflow (for hardware MMQs) and overwrites (for memory-based MMQs), thereby preserving reliability and avoiding the complexities associated with lossy delivery (e.g., message reordering, retransmissions). Second, they allow fine-grained multiplexing at the granularity of senders (different senders may be allocated credits at different rates), MMQs (memory or compute resources can be partitioned differently among the various MMQs), or both.

Credit requests. Senders are initialized with a given number of credits; every command pushed to the MMQ consumes credits depending on its size. Once senders have consumed half of their initial credits or more, they may explicitly send a *credit request* to restore their credits as needed (but no more than the initial allocation).

Credit requests offer three main benefits compared to automatic credit return (in which the receiver returns credit to the sender as commands are consumed). First, they force senders to register their intent to send data; this avoids scenarios where inactive senders (without data to transmit) indefinitely hold on to credits, which can lead to under-utilization of the MMQs. Second, this design does not require the receiver to keep track of the senders transmitting to its MMQs; instead, the receiver only needs to keep track of the total amount of credits it has available to distribute. Finally, credit requests facilitate batching of credit allocations, allowing senders to request multiple credits at once, therefore reducing the overhead of credit synchronization.⁴

For highly multiplexed hardware MMQs, an initial credit allocation of 0 credits is treated as a special case. Senders with an initial allocation of 0 credits may request an arbitrary number of credits on demand, with the receiver determining how many credits it is able to provide

³Despite the potential complexity of buffer allocation at the receiver side, we find that a simple bump allocator works well in our hardware implementation with little resource overhead.

⁴In principle, it is possible to batch automatically-returned credits; however, this would entail additional implementation complexity (e.g., a controller implementing Nagle’s algorithm for each active sender) with hard-to-tune performance parameters (e.g., timeouts for coalescing credit return messages).

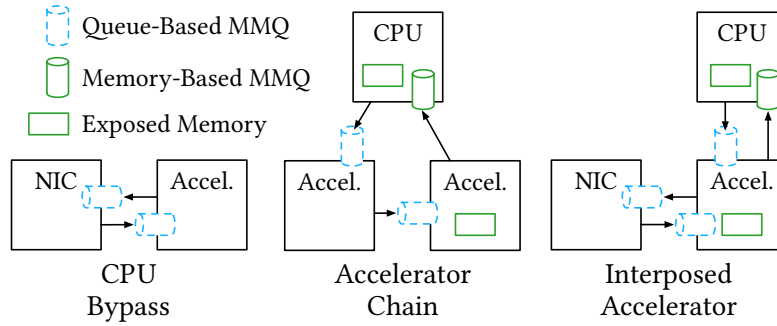


Figure 4.5: DCP implementation of the Figure 4.1 dataflows.

in response. These credit requests may be declined if the MMQ is full, requiring senders to re-attempt a credit request after a timeout. Hence, this model introduces higher latency—requests themselves may be declined, leading to uncertainty—but supports a higher degree of multiplexing.

4.4 Using DCP

Having described the key primitives underpinning DCP, in this section, we revisit our motivating use-cases (§ 4.1.1) and discuss how DCP can be used to implement these workflows. Figure 4.5 expands the dataflows of Figure 4.1 to highlight how the DCP primitives are used to implement them.

4.4.1 CPU Bypass

Queue-Based MMQs enable a simple yet performant dataplane architecture for simple devices (e.g., NICs, Intel QAT) that works as follows. First, at initialization time, the controller configures an MMQ on both devices and informs each device of its peer’s MMQ address and size. Then, when the NIC receives data over the network at run-time, it can directly enqueue the data to the accelerator’s MMQ using pass by value, bypassing the CPU and host memory entirely.

4.4.2 Accelerator Chain

The accelerator chain depicted in Figure 4.5 highlights two important capabilities of DCP.

First, *the flexibility to allocate buffers in a remote address-space provides support for devices that expose no (or very limited amounts of) memory of their own*. In this example, the accelerator on the left can receive data from the CPU as usual: either via direct transfer (pass-by-value) or by pulling data from a buffer in host memory (pass-by-reference). However, since the device

has no exposed memory of its own, it cannot rely on the subsequent device to “pull” the data the same way. Instead, it uses an alternative pass-by-reference mechanism, which we call “push” semantics. The accelerator would first issue a buffer request to allocate memory in the subsequent device’s exposed memory region; once this request is granted, we say that it *owns* the buffer. It then performs a DMA write to perform the data transfer, and enqueues a pointer to the buffer (pass-by-reference) to *push* the data to the next accelerator in the chain; at this point, buffer ownership is transferred back to the original device. Once the subsequent accelerator dequeues the pointer and finishes consuming the referenced data, it can safely deallocate the corresponding memory.⁵

A second salient point here is that *the MMQ abstraction allows devices to use precisely the same mechanics to interface with software-managed devices (CPUs, GPUs) as they would with other, hardware-based devices*. As before, the Memory-Based MMQ on the CPU allows the accelerator on the right to transfer data to the CPU either directly (using pass-by-value) or by enqueueing a pointer to host memory (pass-by-reference).

4.4.3 Interposed Accelerator

Finally, the rightmost example in Figure 4.5 depicts a scenario where *the amount of memory available aboard both communicating devices is either highly constrained or oversubscribed* (e.g., the accelerator must multiplex its exposed memory between RX/TX to both the CPU and the NIC). In this case, both the accelerator and NIC might choose to waypoint their data transfers via host memory.

* * *

DCP’s primitives enable a range of memory allocation schemes and data movement mechanisms to choose from, allowing communicating device pairs to dynamically scale up to the most performant option available to them (dictated by their mutual resource constraints).

4.5 DCP Implementation

We implemented the hardware component of DCP in approximately 13k lines of SystemVerilog, the controller and CPU-side datapath in approximately 7k lines of C++20 (excluding tests and application logic), and the GPU software datapath in about 2k lines of CUDA (also excluding tests and application).

⁵In keeping with a minimalistic design philosophy, DCP makes no prescriptions about *how* the second device allocates/deallocates buffers within its own memory region.

End-to-end simulation. To test DCP under a variety of settings and aid the development of new devices that use DCP we fully integrated the design with SimBricks [104]. We extended Simbricks to include complex PCIe trees with PCIe switches, allowing end-to-end simulations using peer-to-peer across devices and QEMU. We use simulations to verify correctness under a variety of settings, but exclusively report real-hardware performance in the evaluation.

Hardware portability. To allow the design to be portable to a wide range of FPGAs, we built upon Forencich’s PCIe core abstraction layer [56]. And while our implementation targets FPGAs, DCP’s design can also be incorporated into ASICs.

4.6 Evaluation

We now evaluate DCP, comparing its performance to that of custom device interfaces, as well as showcasing the different device workflows that DCP enables. We use the three different dataflows illustrated in Figure 4.5 to understand DCP’s performance in a variety of settings (§ 4.6.3, § 4.6.4, § 4.6.5). In addition, we zoom in on the device-CPU path to understand how DCP compares with custom interfaces in a more traditional setting where devices communicate with the CPU (§ 4.6.6). Finally, we evaluate DCP for GPU-GPU communication over NVLink (§ 4.6.7) and report the cost of implementing DCP in hardware (§ 4.6.8).

4.6.1 Evaluated Devices

Our experimental topologies include two devices: a NIC and an accelerator. We connect the devices under different logical topologies, as illustrated in Figure 4.5.

Baseline NIC (E810). We use an Intel E810 NIC [78] as a baseline NIC. The E810 implements a typical NIC interface with ring buffers of descriptors in host memory that it uses to exchange packet buffers with the CPU. Most existing commodity NICs implement a similar interface [156].

Baseline Accelerator (FPGA-Based). To evaluate a prototypical accelerator interface, we implement a command queue interface with support for scatter-gather DMA on an FPGA. The CPU enqueues commands to the accelerator by writing them to a ring buffer in host memory, followed by a write to a doorbell register at the device. When the accelerator receives the doorbell write, it can issue DMA reads to fetch multiple commands at a time. Commands specify the addresses of the input and output buffers, as well as the input size. The accelerator uses DMA reads to fetch the input and DMA writes to write the output back to host memory. To isolate the performance overheads of the interface alone, this accelerator performs no computation. This interface is similar to that of existing accelerators (e.g., Intel QAT [79, 86]).

DCP Devices. For DCP, we mock up a NIC and accelerator on the FPGA, now using the DCP interface. The controller configures their MMQs depending on the topology, as illustrated in

each experiment, and DCP is configured such that each sender writes to memory at the receiving device. We did not implement an Ethernet connection for the NIC. Instead, we generate traffic from inside the NIC and keep track of packets received back in order to measure RTT and throughput. For the accelerator, we once again perform no computation in order to isolate the overheads of DCP alone.

4.6.2 Setup and Methodology

Setup. Our device-under-test server has two Intel Xeon Gold 6226R CPUs [84], with 16 cores each. We use four PCIe devices in our evaluation: two AMD AU50 FPGAs [6], one AMD AU55C FPGA [7], and one Intel E810 NIC [78]. These devices are connected to the CPUs through two Broadcom PEX 8747 PCIe Gen 3 switches [25]. Each switch has 16 lanes upstream and 32 lanes downstream (16 for each device). On NUMA node 0, the PCIe switch connects to one AMD AU50 FPGA and one Intel E810 NIC. On NUMA node 1, the PCIe switch connects to two FPGAs (AMD AU50 and AMD AU55C).

For the experiments that use the NIC, we also use a second server to generate network traffic. This server has an Intel Core i7-7820X CPU [81] and an Altera Stratix 10 MX FPGA [77] connected directly to the E810 NIC on the DUT server. We generate traffic and measure RTT with cycle-accurate precision using EnsōGen (§ 3.7.2).

For the GPU experiments, we use an HPE Cray XD670 node [28] with two Intel Xeon Platinum 8470 CPUs, 2,048 GB of DDR5-4800 memory, and eight Nvidia H100 GPUs [32] with 80 GB of memory each. Our experiments use a pair of these GPUs connected through fourth-generation NVLink, which provides 450 GB/s of theoretical peak bandwidth between the two GPUs (in each direction) [32, 129].

Methodology. We conduct all DCP experiments using the two FPGAs on NUMA node 1, with each FPGA acting as an accelerator or NIC, depending on the configuration. For the baseline experiments, we use NUMA node 0 when using the NIC and NUMA node 1 when using two accelerators. This keeps the hardware setup as similar as possible for all configurations without requiring physical changes and avoiding the overheads of cross-NUMA traffic [181]. We report median values across 10 runs for all experiments and show the standard deviation as error bars.

Measuring Latency: For the experiments using only accelerators, we report latency as the RTT starting and finishing at the CPU. For the experiments using a NIC, we report latency as the RTT starting and finishing at the NIC. For DCP, we measure the RTT from the NIC by implementing timestamping at the NIC hardware. For the baseline, we are not able to efficiently calculate the RTT at the NIC. While the E810 provides a timestamp mechanism, it has high overhead. Given this limitation, we measure the latency for the baseline experiments in two different ways: from the external packet generator (which overestimates the RTT by around 1 μ s) and by separately measuring the RTT between the CPU and the NIC and the RTT between the CPU

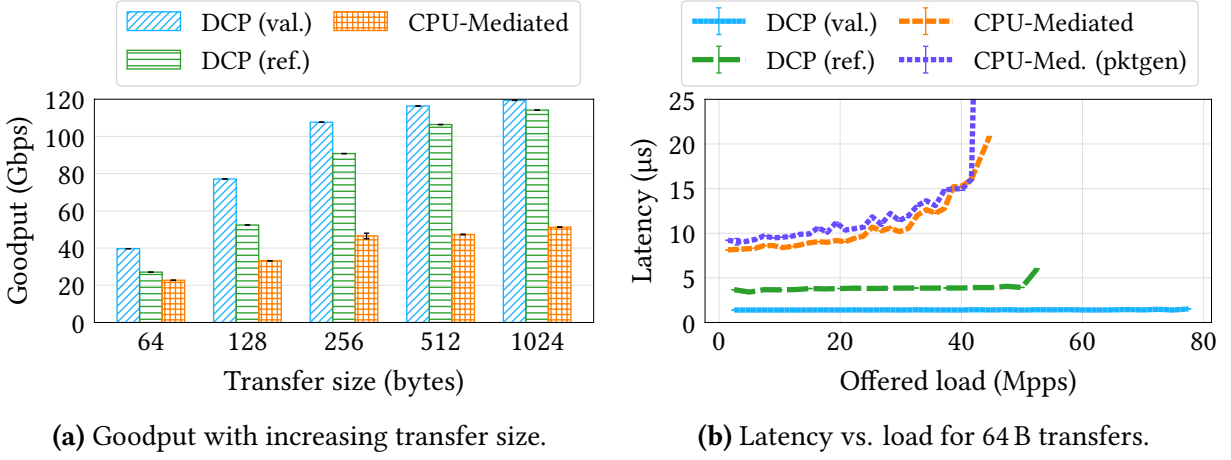


Figure 4.6: Throughput and latency for the CPU-Bypass topology.

and the accelerator and adding them together.

4.6.3 CPU Bypass

We start by evaluating the CPU-bypass topology, where a NIC and an accelerator communicate with each other. This lets us understand the performance of DCP when used for device-to-device communication. As a baseline, we use the CPU to mediate the communication between the E810 NIC and the baseline accelerator.

Figure 4.6 shows the goodput (4.6a) and latency (4.6b) for the CPU-bypass topology. DCP more than doubles the goodput compared to the baseline and drops latency from 8.1μ s to 1.4μ s, when using pass by value.

Goodput. For small transfers (64 B), DCP is bottlenecked by the FPGA’s PCIe core, achieving 40 Gbps using pass by value and 27 Gbps using pass by reference. In contrast, the CPU-mediated baseline is CPU-bound and achieves only 22.7 Gbps. For larger transfers, the bottleneck shifts to the PCIe link. In this regime, DCP achieves 119 Gbps using pass by value (close to the theoretical maximum) and 114 Gbps using pass by reference, while the baseline is bottlenecked on the link between the CPU and the PCIe switch, achieving 51 Gbps. Finally, note that DCP with pass by value consistently outperforms DCP with pass by reference. This is because pass by reference uses receiver-side buffer allocation in this configuration, which requires the sender to issue buffer requests to the receiver before writing data.

Latency. As expected, removing CPU mediation significantly reduces latency. DCP with pass by value achieves a latency of 1.4μ s (17% of the baseline), while DCP with pass by reference achieves a latency of 3.4μ s (42% of the baseline). Note that pass by reference’s latency is around

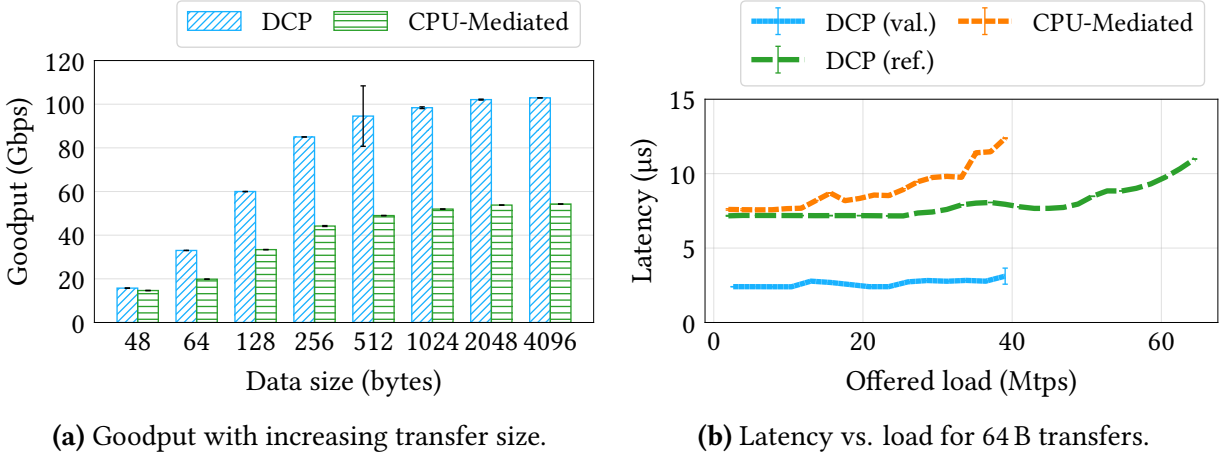


Figure 4.7: Throughput and latency for the Accelerator-Chain topology.

3 $\times$ that of pass by value since it adds an extra RTT for buffer requests in each direction.

4.6.4 Accelerator Chain

We now look at the performance of DCP in the accelerator chain topology, where the CPU sends the input data to a chain of two accelerators and receives the output back from the second accelerator. As a baseline, we use the CPU to mediate the communication, requiring the data to return to the CPU in between the two accelerators. Figure 4.7 shows the goodput (4.7a) and latency (4.7b).

Goodput. Both DCP and the baseline become bottlenecked by the CPU for small transfers and by the PCIe bandwidth for larger transfers. However, unlike the baseline, DCP does not need an extra CPU hop in between the two accelerators, which allows it to sustain the full link throughput for larger transfer sizes. In contrast, the baseline makes the data traverse the link between the CPU and the PCIe switch twice, reducing the goodput to half of what is achieved by DCP.

Latency. As we discuss in more detail in § 4.6.6, compared to the baseline accelerator, DCP imposes higher latency when communicating with the CPU using pass by reference. This results in the overall latency being comparable between the two configurations in the accelerator chain topology. DCP with pass by value does not face this issue and has an RTT of only 2.4 μ s, compared to 7.2 μ s for pass by reference and 7.5 μ s for the baseline, but is limited to transfers of up to 48 B.

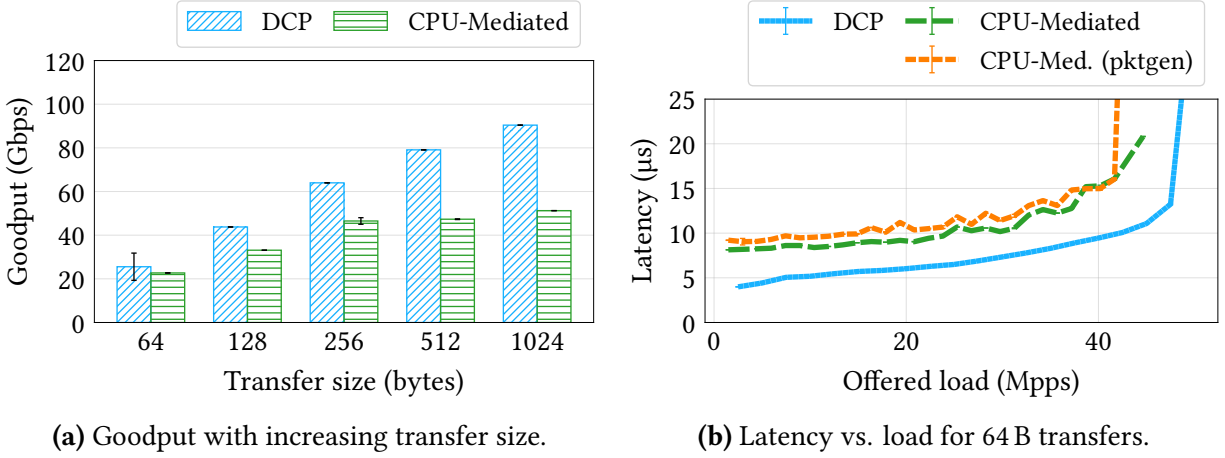


Figure 4.8: Throughput and latency for the Interposed-Accelerator topology.

Interface	CPU→Dev Overhead	Dev→CPU Overhead
DCP (val.)	16 B	16 B
DCP (ref.)	21.3 B	21.3 B
Accelerator	64 B	0 B
E810 NIC	32 B	16 B

Table 4.1: Metadata overhead per transfer for each interface. For the baseline interfaces, we omit per-batch overheads, such as doorbell register writes and completion notifications.

4.6.5 Interposed Accelerator

Finally, we evaluate the interposed accelerator topology, where a NIC receives traffic, forwards it to an accelerator, and the accelerator sends the result to the CPU. As with the previous topologies, the baseline requires the CPU to mediate the NIC-to-accelerator communication. Figure 4.8 shows the goodput (4.8a) and latency (4.8b). The results are similar to those of the accelerator chain topology, except that we are not able to use pass by value when communicating with the CPU, since packets cannot be smaller than 64 B.

There are two main differences compared to the accelerator chain. First, the baseline exhibits *higher* latency since it uses an E810 NIC, which has higher latency than the FPGA-based accelerators (§ 4.6.3). Second, DCP achieves *lower* latency since the software is simpler than that used in the accelerator chain, only echoing data back instead of having to both generate data and receive it back.

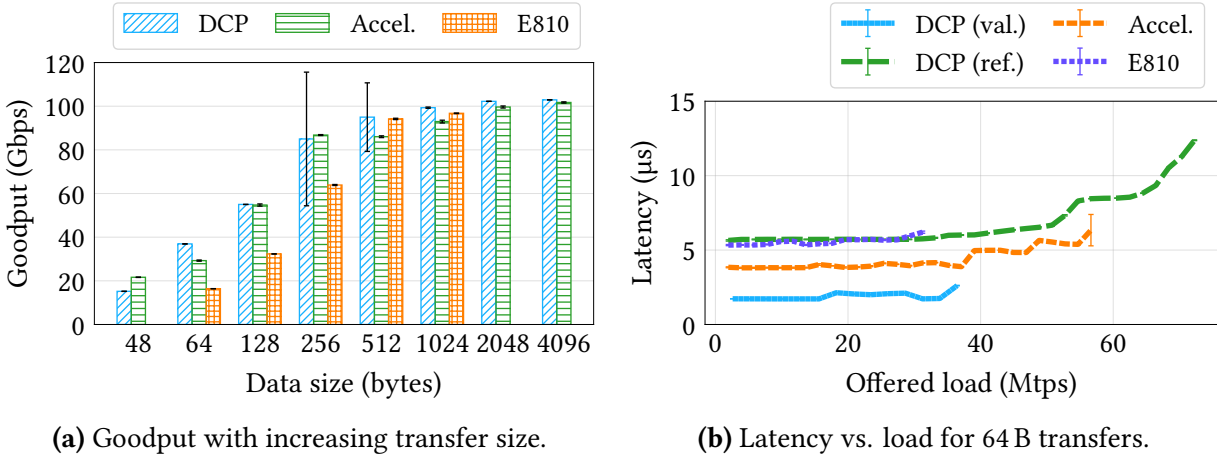


Figure 4.9: Throughput and latency for transfers between the CPU and a single device.

4.6.6 Device-to-CPU Communication

We now zoom in on the device-CPU communication path to understand how DCP compares with custom interfaces in a more traditional setting where devices communicate with the CPU. We compare the performance (goodput and latency) of DCP devices to the baseline NIC and accelerator. In this experiment, we generate data at the CPU and use the devices to forward it back. For the baseline NIC, we leverage its internal switch to loopback packets, which allows us to evaluate the cost of the NIC interface without the overhead of Ethernet. For the baseline accelerator, we configure it to perform no operation and write the input data back to host memory.

Goodput. DCP achieves similar goodput to that of custom interfaces when communicating with the CPU. Figure 4.9a depicts the goodput as we increase the transfer size. For transfers up to 512 B, all interfaces are CPU-bound. In this regime, DCP and the baseline accelerator achieve similar goodput, while the baseline NIC achieves around 25% lower goodput. For larger transfers, the bottleneck shifts to the PCIe Root Complex’s bandwidth. In this regime, the goodput is dictated by each interface’s bandwidth overhead due to metadata—with the interfaces with lower metadata overhead achieving higher goodput. Table 4.1 shows the metadata overhead per transfer for each interface. DCP has lower overhead compared to the baseline accelerator and NIC, which explains its slightly higher throughput for larger transfers.

Note that our message here is not that DCP has *fundamentally* lower overhead than custom interfaces—we could, for instance, further optimize the baseline accelerator to use smaller commands, allowing it to match DCP’s goodput. Instead, our claim is that DCP achieves comparable goodput to that of custom interfaces while providing generality.

Transfer Size	DCP	NVSHMEM	NVSHMEM (TMA)
64 B	4.8 μ s	3.5 μ s	5.7 μ s
256 B	4.8 μ s	3.5 μ s	5.7 μ s
1 KB	4.8 μ s	3.5 μ s	5.7 μ s
4 KB	4.8 μ s	3.7 μ s	5.8 μ s
16 KB	5.0 μ s	6.0 μ s	6.5 μ s
64 KB	6.8 μ s	16.7 μ s	8.5 μ s

Table 4.2: GPU-GPU latency over NVLink for DCP and Nvidia’s NVSHMEM, with increasing transfer sizes.

Latency. Figure 4.9b shows the latency as we increase the load for 64 B transfers. Pass by value allows DCP to achieve 1.7 μ s latency for small transfers (≤ 48 B) when communicating with the CPU. With pass by value, the CPU enqueues data using an MMIO write to the device’s MMQ and the device enqueues data at the CPU using DMA writes to the memory-based MMQ. As such, pass by value allows the CPU to enqueue an input and receive an output with 1 RTT total.

With pass by reference, the CPU owns the buffers in both directions. This means that it uses sender-owned buffers for sending inputs to the device, and receiver-owned buffers for receiving outputs. As such, transfers take 1.5 RTT from the CPU to the device (pass by reference + device-initiated DMA read) and another 1.5 RTT from the device to the CPU (buffer request + device-initiated DMA write). This matches the baseline accelerator and the E810 NIC, since both require 3 RTT total to send an input and receive an output. However, DCP’s buffer requests add buffer allocation to the critical path, which explains why DCP with pass by reference has around 1.9 μ s higher RTT compared to the baseline accelerator. Still, pass by reference can achieve similar latency to that of the E810 NIC, even though the E810 preallocates buffers.

4.6.7 GPU-to-GPU Communication over NVLink

So far, all the results we reported have used PCIe as the interconnect. We now focus on NVLink [32], the interconnect most commonly used for high-performance GPU-to-GPU transfers. This lets us both test DCP on a much higher-capacity link and demonstrate its software implementation on a different hardware target (GPUs instead of CPUs).

We implement DCP in CUDA. The implementation runs on the GPU’s Streaming Multiprocessors (SMs) and also leverages the GPU’s Tensor Memory Accelerator (TMA) to perform pass-by-reference using SM-initiated asynchronous transfers. Nvidia GPUs only support up to 16 B atomic writes [139], leaving no usable space for payload in the command. As such, we only consider pass-by-reference commands in the evaluation. We compare DCP against Nvidia’s specialized mechanisms: NVSHMEM [137], which only uses the GPU SMs to implement the

	DCP	NVSHMEM	NVSHMEM (TMA)	Copy Engine
Peak throughput	374 GB/s	313 GB/s	274 GB/s	390 GB/s
SMs used	16	64	64	0
Transfer size	16 KB	4 MB	4 MB	64 MB

Table 4.3: Peak GPU-GPU throughput over NVLink, along with the number of SMs and the transfer size that each mechanism requires to reach its peak.

communication; NVSHMEM (with TMA) [138], which, like DCP, uses TMA; and the GPU’s Copy Engine [131], which must be controlled from the CPU.

Latency. We start by looking at the effect transfer sizes have on latency for the different interfaces. We measure latency for DCP and NVSHMEM using non-posted transfers. We cannot use the same methodology for the Copy Engine transfers, since they are CPU-initiated and there is no equivalent non-posted mechanism. We did measure the time it takes for the CPU to dispatch a command to the Copy Engine and receive a confirmation (at least $7.3\ \mu\text{s}$ across transfer sizes), but this confirmation comes from the sender, not the receiver.

Table 4.2 shows the latency as we increase the transfer size. NVSHMEM without TMA achieves the lowest latency with small transfers (up to 4 KB) since it avoids the overhead of using TMA. However, by not using TMA, it suffers a higher latency when using larger transfers. For transfers up to 4 KB, DCP’s latency is within $1.3\ \mu\text{s}$ of NVSHMEM’s. For transfers of 16 KB and larger, DCP has the lowest latency of all mechanisms.

Throughput. We are interested in the peak throughput each interface achieves. Moreover, different interfaces derive different benefits from more hardware resources or larger transfer sizes. As such, we also report the number of SMs and the minimum transfer size required for each interface to reach its peak. Table 4.3 shows the results. DCP achieves 20% higher peak throughput than NVSHMEM (374 GB/s vs. 313 GB/s) while using a quarter of the SMs (16 vs. 64) and reaching its peak with $256\times$ smaller transfers (16 KB vs. 4 MB). The Copy Engine achieves a slightly higher peak (390 GB/s) without consuming any SMs, since it relies on dedicated hardware, but it requires 64 MB transfers to reach it.

* * *

These experiments show that, even with maximum specialization (i.e., a single hardware vendor controlling the system end to end), DCP still achieves performance comparable to the vendor’s own specialized mechanisms.

Component	LUTs	FFs	BRAM
Inner shell	2.3%	2.4%	0.9%
Outer shell	1.8%	1.4%	2.4%
PCIe core	3.3%	1.8%	1.1%

Table 4.4: FPGA resource utilization of DCP’s hardware implementation on an AU55C, as a fraction of the FPGA’s resources.

4.6.8 Cost

Before concluding the evaluation, we report the cost in terms of FPGA logic [170] needed to implement DCP. Table 4.4 shows the resource utilization of each component of our hardware implementation on an AU55C FPGA [7]. At a high level, the architecture is composed of two shells. The outer shell implements a producer and consumer of MMQ commands; the inner shell implements memory allocation and responses to credit and buffer requests. Devices that require custom allocation or credit handling can implement their own inner shell. All components run at 250 MHz. The PCIe core is also needed for the baseline, making the resource increase around 4% of the FPGA for a fully-featured DCP implementation.

4.7 Related Work

GPU communication. The popularity of machine learning workloads has spurred a proliferation of techniques for direct GPU communication. Many rely on vendor-specific solutions, such as Nvidia’s GPUDirect [66, 133, 134] and AMD’s ROCm [9], which target specific device pairs (e.g., Nvidia GPU to NVMe SSD). Because they depend heavily on the GPU’s programmability and exposed memory, they cannot be widely adopted.

Accelerator-NIC communication. Beyond GPUs, Lynx [187] allows accelerators to communicate directly with a NIC. Besides being specific to NIC-accelerator communication, its design assumes that accelerators expose memory and that both the accelerators and the NIC offer general-purpose programmability. FlexDriver [53] targets a similar use case but instead hardcodes a NIC driver at the accelerator, limiting compatibility. Other works implement network transport directly on the NIC [179, 180], which enables direct accelerator communication without CPU involvement. From a device-communication standpoint, these solutions are similar to traditional RDMA NICs, requiring the device to expose memory and support general programmability to avoid the CPU.

Direct communication among programmable devices. Existing works that envision general-purpose device-to-device communication assume that devices provide general pro-

programmability [11, 127, 177, 188]. DCP does not make this assumption. It supports devices with diverse capabilities, even those lacking general programmability. Furthermore, by standardizing communication, DCP avoids the need for every device to implement a custom driver for every other device; DCP-enabled devices can communicate with one another without customization.

CXL. Some works rely on CXL [37] or other cache-coherence techniques to share data across devices [35, 103] or between devices and the CPU [92, 124, 175]. However, cache coherence is orthogonal to DCP and does not solve the coordination problem. For instance, two CXL Type 2 devices might have cache-coherent access to each other’s memory, but lack an agreed-upon mechanism to coordinate that access. DCP solves this: devices can use its pass-by-reference commands to share data and coordinate access without a trip to host memory. Similarly, CXL Memory devices (Type 3) can be used with DCP to exchange data seamlessly.

Using MMIO writes for data transfer. Our use of write-combined MMIO to transfer data from the CPU to the device is inspired by prior work showing its potential for high throughput when paired with a receiver-side reorder buffer [105, 106].

4.8 Conclusion

Today, the grand vision of servers operating as a “sea of accelerators” is hampered by the lack of a general-purpose mechanism for device-to-device communication. We designed a general-purpose Device Communication Protocol (DCP) that makes progress towards this vision by enabling devices to autonomously initiate and coordinate data transfers among themselves. It does so by standardizing a very narrow band of device functionality: the mechanics of memory allocation and data movement. In effect, DCP obviates CPU-based orchestration (and the performance overheads it entails) without baking in any assumptions about devices’ memory resources and logical capabilities. As we look to a future with increasingly heterogeneous, hyper-specialized devices, we believe that DCP’s design serves as a starting point for a highly scalable and performant network of accelerators.

Chapter 5

Future Work

In this chapter, we discuss two immediate directions for future work building on Ensō and DCP. The first is to redesign network stacks around Ensō’s streaming interface, moving policy enforcement and process scheduling from packet granularity to stream granularity (§ 5.1). The second is to give software on CPUs and GPUs efficient message-passing primitives, by implementing DCP in hardware on these platforms (§ 5.2).

5.1 Streaming Network Stacks

The long use of packetized interfaces in NICs has influenced the design of network stacks. Network stacks today often operate at packet granularity, facing significant per-packet overheads [29]. Mechanisms such as TCP Segmentation Offload (TSO) and Generic Receive Offload (GRO) [1] exist precisely to reduce these overheads, but they are workarounds for an interface built for individual packets. Ensō instead naturally groups data into large transfers. Even raw packets arrive back to back in a single contiguous span. This creates two main opportunities:

Network policies with streams. The first opportunity is to restore kernel interposition in software without reintroducing significant performance overheads. Traditionally, the kernel network stack interposes between applications and the network, which allows it to enforce policies and improve manageability [168]. Unfortunately, because the kernel stack imposes significant overhead on every packet sent and received, many high-performance applications give up interposition entirely, relying on kernel bypass to improve performance [94, 97, 107, 144, 145, 153, 160, 200]. With a streaming network stack, we could reintroduce kernel interposition by enforcing policies at *stream* granularity instead of packet granularity, greatly reducing the overhead. Preliminary work shows that such an approach works well for a kernel rate limiter, capable of sustaining 100 Gbps line rate on a single core [162]. However, determining which policies can be expressed over a stream, and which fundamentally require packets, remains an

open question.

Process scheduling with streams. The second opportunity is to improve process scheduling for latency-sensitive applications. Latency-sensitive applications must wake up quickly when new data arrives [160]. One approach is to pin applications to cores and have them poll for data. But while polling ensures fast wake-up times, it wastes CPU cycles when applications are idle. Some proposals address this issue by introducing a centralized scheduler that monitors incoming data and allocates cores to applications on demand [57, 58, 144], but they face a trade-off between reactivity and scalability. The scheduler can interpose on every packet, which lets it react as soon as data arrives but turns the scheduler into a choke point. Alternatively, packets can be delivered directly to applications, but this creates a delay between when data arrives and when the scheduler notices it. Streaming can avoid this trade-off by decoupling data from notifications. Data flows directly to application threads, avoiding the choke point, while notifications flow to the centralized scheduler, letting it react as soon as data arrives. Preliminary work suggests that this approach can improve throughput compared to packet-based designs [65].

5.2 Better Message-Passing Primitives

Architectures converged on message passing as the mechanism and shared memory as the interface. In their 1998 book, Culler et al. [39] made the argument that parallel architectures that were once seen as competing had converged. One of the reasons for this claim is that architectural features such as shared memory and message passing can be constructed from each other. Software can provide a message-passing abstraction on top of a shared-memory hardware interface and a shared-memory abstraction on top of a message-passing hardware interface. But regardless of the interface that the hardware exposes to software, the implementation typically relies on message passing. This remains true today, with all major host interconnect technologies based on message passing (e.g., PCIe [150], CXL [37], NVLink [32]). While the underlying host interconnects use message passing, the interface that hardware exposes to software on most CPUs and GPUs today has converged on shared memory, with no efficient message-passing primitives exposed to software.¹

In terms of functionality, a shared-memory hardware interface is sufficient, since software can construct message passing on top of it. But when the goal is point-to-point communication, this layering adds unnecessary overheads: software implements message passing over shared memory, which the hardware in turn implements using message passing! We can see this cost by comparing DCP’s queue-based and memory-based implementations (§ 4.6.6). The queue-based

¹Modern x86 provides MOVDIR64B [34] and ENQCMD [33], but neither is efficient. ENQCMD is non-posted: it awaits a device completion to report success, making each enqueue wait an RTT. MOVDIR64B is posted but returns no status, providing no flow control. In addition, both bound transfers to 64 B, much smaller than the TLP payload limits.

implementation leverages the low-level message-passing primitives that devices can access but software on CPUs and GPUs cannot. This results in lower latency and higher throughput.

An interesting research direction is therefore to expose efficient hardware message-passing primitives to software running on both CPUs and GPUs. The challenge is that an efficient primitive must offer non-blocking semantics and flow control. Without them, senders either stall waiting for receivers to drain or overwhelm receivers that cannot keep up. Fortunately, DCP addresses both: because its credit system lets a sender know in advance whether the receiver can accept a command, we can prevent senders from blocking and receivers from being overwhelmed. A solution, then, is to have CPUs and GPUs implement DCP in hardware, keeping track of credits and issuing credit requests automatically, allowing software to send and receive commands directly to and from devices' MMQs. Determining the right software interface, and the right ISA extensions to support it, remains an open question.

Chapter 6

Conclusion

In an era where devices are capable of increasingly complex computation, how should they communicate? In this dissertation, we made the case that accelerators and I/O devices can benefit from adopting general-purpose interfaces for communication. We showed that, contrary to conventional wisdom, general-purpose device interfaces can match or even outperform specialized interfaces. We did so by building two systems that address different sources of specialization in existing device interfaces:

Ensō (Chapter 3) generalizes “what” devices communicate by introducing an interface that works well for different data formats. Ensō replaces the packetized interface commonly used in NICs with a *streaming abstraction* where software and the NIC communicate by simply appending data to a contiguous buffer with no constraints on the format or granularity of the data. We showed that this interface is better suited for higher-level formats while addressing the performance issues of the existing packetized interface.

DCP (Chapter 4) generalizes “whom” devices can communicate with by introducing a communication protocol that accommodates endpoints with different constraints and capabilities without requiring devices to adapt to one another. DCP introduces a new queue primitive called Memory-Mapped Queue (MMQ) that decouples what senders see from how receivers implement it. It then uses this unifying abstraction to implement distributed coordination among the endpoints. We showed that DCP is lightweight (requiring neither general-purpose programmability nor exposed memory), can be implemented by a wide range of devices (e.g., CPUs, GPUs, NICs, and specialized accelerators), and drastically improves throughput and latency by removing CPU mediation.

In the rest of this chapter, we comment on some of the broader lessons we learned from building these systems (§ 6.1) and reflect on the path to adopting general-purpose interfaces (§ 6.2). We then conclude the dissertation by reflecting on the role hardware interfaces should play in the evolving computer architecture landscape (§ 6.3).

6.1 Lessons Learned

Building Ensō and DCP taught us lessons that extend beyond the two systems. We now discuss a few of them.

Servers are still deeply CPU-centric. The assumption that the CPU is the primary compute platform and devices simply aid the CPU extends beyond device interfaces, permeating many other aspects of server design. PCIe is built as a tree with the CPU at the root, and most CPUs today do not forward peer-to-peer PCIe transfers [108]. Consequently, devices must sit behind a PCIe switch to be able to issue peer-to-peer transactions, and the server becomes partitioned: devices behind different switches cannot communicate directly, since the path between switches traverses the CPU, and devices in different NUMA nodes cannot reach one another. We also faced non-fundamental obstacles when running DCP on a real server. For instance, we were unable to cascade PCIe switches because of artificial limitations imposed by the server vendor. Fortunately, hardware vendors seem to be moving in the right direction: the latest generation of PCIe switches supports substantially more devices and even allows multiple CPUs to connect to the same switch [26]. This has the potential to level the playing field between devices and the CPU.

The performance of device interfaces is hard to predict and diagnose. The performance of a device-CPU interface depends on subtle details of how the CPU manages its cache. Designs that look reasonable on paper, and even perform well in simulation, can still fail in practice. For instance, when designing Ensō, we had to discard many designs that seemed reasonable at first but performed poorly when running on the real hardware (§ 3.4). Coherence is a frequent source of performance problems. Because the CPU and the device access the same cache lines, every write from the device must first acquire exclusive ownership of the line, invalidating any copy held by the CPU. When the CPU then reads the line, it takes a copy back into its cache, and the device’s next write must acquire ownership again. If the CPU polls too aggressively, almost every write from the device faces this issue.

Diagnosing the source of these issues can also be hard. For instance, cache-line contention creates backpressure on PCIe, which eventually propagates to the device’s internal hardware pipeline. As a result, a problem that originates in the CPU’s cache may become visible only as a stall inside the device. Finding the root cause often involves examining hundreds of hardware counters. But because these effects are tied to details of the CPU architecture, they are typically not reproduced in simulation, so the only way to reliably evaluate the performance of a candidate design is to build it and measure it on real hardware.

The value of end-to-end simulation. Although performance must ultimately be measured on real hardware, simulation is essential during development, since hardware is much easier to debug in simulation than after synthesis. End-to-end simulation extends this benefit from the device to the entire system by running the actual software against the simulated hardware.

This lets us develop the hardware and software at the same time while avoiding long synthesis cycles, and allows us to check the correctness of their interactions rather than of the device by itself. We learned this the hard way with Ensō, which we built without end-to-end simulation. When testing Ensō’s hardware, we wrote testbenches that emulated what the software side of the interface would do, so the tests could exercise only the interactions we anticipated. When building DCP, we had learned our lesson and instead adapted SimBricks [104] to simulate the driver running on the CPU, a hierarchy of PCIe switches, and multiple devices, all in a single end-to-end simulation. This reproduced interactions we would not have thought to test and let us catch bugs early, while they were still easy to observe and fix. It also let us reproduce bugs that we only found later, on the real hardware, such as deadlocks involving multiple devices that would have been very hard to diagnose without simulation. In general, the more of the system we can include in the simulation, the further these benefits extend.

6.2 Adoption

While we have shown that adopting better device interfaces can solve many problems, a natural question to ask is: *What about backward compatibility?* A major reason for keeping hardware interfaces unchanged is compatibility with legacy software and hardware. We discuss this concern in the context of each of the systems we introduced.

6.2.1 Adopting Ensō

Fully benefiting from Ensō requires breaking existing software APIs. While drivers could adapt Ensō to fit existing APIs, doing so risks losing some of its performance benefits. For instance, DPDK [43] sends and receives data by passing around arrays of pointers to packet buffers. This closely reflects the expectations of packetized NIC interfaces, since the driver can construct descriptors from these pointers on transmission and pointers from descriptors on reception. Unfortunately, keeping the existing DPDK semantics with Ensō would require copying each packet to a dedicated packet buffer, negating many of Ensō’s performance advantages. Indeed, in Chapter 3 we had to modify the DPDK-based applications we used in our experiments to use the Ensō API instead. Other software that wants to benefit from Ensō will need to do the same.

We believe this cost is acceptable for two reasons. First, developers of high-performance applications are accustomed to adapting their code to the hardware, as evidenced by the many applications that abandoned traditional sockets in favor of DPDK [18, 49, 67, 91, 101, 111, 122, 146, 154, 184, 200], RDMA [17, 45, 64, 96, 136, 200], or AF_XDP [69, 189]. These are also the applications that benefit the most from Ensō. Second, many applications rely on a high-level interface, such as an RPC library. Porting such libraries to Ensō benefits every application built on top of them, without changes to the applications themselves.

6.2.2 Adopting DCP

Adopting DCP is no harder than adopting any other new device interface. Every device today typically ships with a custom interface and a matching driver. In this sense, adopting DCP should be *easier* than the status quo, since both the hardware implementation and the software can be reused across devices. Moreover, unlike Ensō, DCP does not force changes to software APIs when communicating with the CPU. DCP is flexible enough to mimic the patterns of existing interfaces by changing how allocation works. For instance, by using a memory pool allocator, DCP can mimic a packetized interface, making it a good fit for DPDK. Alternatively, by using a bump allocator, DCP can mimic an Ensō-style interface.

A more significant concern for DCP adoption is not software APIs but the fact that DCP’s benefits depend on *multiple* devices implementing the same protocol. In particular, the first device to adopt DCP might see no benefit since it would join an ecosystem where every other device uses a specialized interface. Fortunately, DCP can be implemented in software on *existing* programmable devices, such as CPUs and GPUs. A new device that implements DCP in hardware can therefore communicate with these devices from day one. Moreover, legacy devices that cannot implement DCP can continue to rely on CPU mediation to communicate with both DCP-capable and legacy devices, which is no different from what happens today. From there, we expect a compounding effect, with DCP’s benefits growing with adoption.

* * *

Regardless of the specific device interface, the barrier to adopting new hardware is often the cost of changing software. As machine learning models become increasingly capable of writing and porting code, that cost is falling [125, 128], and we expect the barrier to adopting new hardware interfaces to fall with it. We return to this point in the next section.

6.3 Looking Ahead

Before concluding, we reflect on the role hardware interfaces should play as computer architectures become more specialized. This specialization reverses decades of convergence. By the 1990s, computer architectures had largely converged on a single programming model, with sequential programs executing over a coherent shared memory [39]. Successful architectural innovations were the ones that preserved the programming model (e.g., cache coherence, out-of-order execution, prefetching, and branch prediction), and thus avoided breaking compatibility with existing software. And architectures that instead tried to change the model repeatedly failed to gain widespread adoption [95, 147, 174].

More recently, however, architectures have begun to diverge again. Computation is increas-

ingly moving away from the CPU and into specialized architectures, each with a programming model that better reflects its structure [68]. GPUs are programmed through data-parallel kernels [155], ML accelerators through graphs of tensor operators [93], and programmable network devices through match-action pipelines [23].

Yet these new architectures do not operate as standalone devices. They assume that the CPU remains at the center of the system, responsible for gluing the different devices together. This happens in two ways: first, the CPU mediates *communication*, adapting each device’s specialized interface so that data can flow from one device to the next; second, the CPU centralizes *control*, allowing programmers to manage devices from software running on the CPU.

In this dissertation, we showed that a general-purpose device interface removes the dependency on the CPU for *communication*. Devices with very different architectures can communicate directly, without adapting to one another and without routing data through the CPU. But we still depend on the traditional CPU for *control*.

Depending on the CPU for control, however, is less costly than depending on it for communication, because control can often be kept off the critical path. When a device lets the CPU bind each queue to a specific functionality ahead of time, the device knows what to do with the data the moment it arrives and needs the CPU only at setup time. DCP follows this model, fixing the functionality of each queue when the queue is instantiated, or letting the device itself determine the functionality. Only when a control decision can neither be encoded ahead of time nor made by the device itself does it fall to software on the CPU at runtime, bringing the CPU back onto the critical path.

Instead of removing control from the CPU, we should focus on reducing the cost of relying on it. Many of the costs of falling back to the CPU for control are not fundamental. CPUs today are optimized for long-running sequential execution, not for short computations that react to I/O. This mismatch is, at least in part, a consequence of the CPU’s existing programming model. Fortunately, the main barrier to changing this model, the cost of porting legacy software, is falling with advances in machine learning [125, 128]. If software becomes cheap to port to new programming models, architects can finally redesign the CPU to eliminate this mismatch. It is also an opportunity to revisit good ideas that failed for reasons that no longer hold and to propose new ones.

However we perform control in the future, we expect one lesson of this dissertation to hold: the more specialized and independent devices become, the greater the need for general-purpose communication among them.

Bibliography

- [1] Segmentation offloads. The Linux Kernel documentation, v7.1, 2026. URL <https://www.kernel.org/doc/html/v7.1/networking/segmentation-offloads.html>. § 5.1
- [2] Dennis Abts, Jonathan Ross, Jonathan Sparling, Mark Wong-VanHaren, Max Baker, Tom Hawkins, Andrew Bell, John Thompson, Temesghen Kahsai, Garrin Kimmell, Jennifer Hwang, Rebekah Leslie-Hurd, Michael Bye, E. R. Creswick, Matthew Boyd, Mahitha Venigalla, Evan Laforge, Jon Purdy, Purushotham Kamath, Dinesh Maheshwari, Michael Beidler, Geert Rosseel, Omar Ahmad, Gleb Gagarin, Richard Czekalski, Ashay Rane, Sahil Parmar, Jeff Werner, Jim Sproch, Adrian Macias, and Brian Kurtz. Think fast: A tensor streaming processor (TSP) for accelerating deep learning workloads. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ISCA '20, pages 145–158, Virtual Event, 2020. IEEE Press. ISBN 978-1-7281-4661-4. URL <https://doi.org/10.1109/ISCA45697.2020.00023>. § 1
- [3] Saksham Agarwal, Rachit Agarwal, Behnam Montazeri, Masoud Moshref, Khaled Elmelegy, Luigi Rizzo, Marc Asher de Kruijf, Gautam Kumar, Sylvia Ratnasamy, David Culler, and Amin Vahdat. Understanding host interconnect congestion. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*, HotNets '22, pages 198–204, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 978-1-4503-9899-2. URL <https://doi.org/10.1145/3563766.3564110>. § 3.1.2, § 4.1.2
- [4] Amazon. DPDK driver for elastic network adapter (ENA). <https://github.com/amzn/amzn-drivers/tree/master/userspace/dpdk>, 2026. § 3.1
- [5] Amazon Web Services. The security design of the AWS Nitro System. Aws whitepaper, Amazon Web Services, February 2024. URL <https://docs.aws.amazon.com/whitepapers/latest/security-design-of-aws-nitro-system/security-design-of-aws-nitro-system.html>. § 1
- [6] AMD. Alveo U50 data center accelerator card data sheet. Technical Report DS965, AMD, June 2023. <https://docs.amd.com/r/en-US/ds965-u50>. § 4.6.2

- [7] AMD. Alveo U55C data center accelerator card data sheet. Technical Report DS978, AMD, June 2023. <https://docs.amd.com/r/en-US/ds978-u55c>. § 1.4.2, § 4, § 4.6.2, § 4.6.8
- [8] AMD. Alveo MA35D media acceleration card data sheet. Technical Report DS1010, AMD, November 2023. <https://docs.amd.com/r/en-US/ds1010-ma35d>. § 1
- [9] AMD. ROCm software, 2026. URL <https://www.amd.com/en/products/software/rocm.html>. § 4.7
- [10] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling programmable transport protocols in high-speed NICs. In *17th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '20, pages 93–109, Santa Clara, CA, February 2020. USENIX Association. ISBN 978-1-939133-13-7. URL <https://www.usenix.org/conference/nsdi20/presentation/arashloo>. § 1, § 2.2.1, § 3
- [11] Nils Asmussen, Marcus Völz, Benedikt Nöthen, Hermann Härtig, and Gerhard Fettweis. M3: A hardware/operating-system co-design to tame heterogeneous manycores. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 189–203, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 978-1-4503-4091-5. URL <https://doi.org/10.1145/2872362.2872371>. § 4.7
- [12] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 978-1-4503-1097-0. URL <https://doi.org/10.1145/2254756.2254766>. § 3, § 3.1.2, § 3.7.4
- [13] Nirav Atre, Hugo Sadok, Erica Chiang, Weina Wang, and Justine Sherry. SurgeProtector: Mitigating temporal algorithmic complexity attacks using adversarial scheduling. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, pages 723–738, New York, NY, USA, August 2022. Association for Computing Machinery. URL <https://doi.org/10.1145/3544216.3544250>.
- [14] Nirav Atre, Hugo Sadok, and Justine Sherry. BBQ: A fast and scalable integer priority queue for hardware packet scheduling. In *21st USENIX Symposium on Networked Systems Design and Implementation*, NSDI '24, pages 455–475, Santa Clara, CA, April 2024. USENIX Association. ISBN 978-1-939133-39-7. URL <https://www.usenix.org/conference/nsdi24/presentation/atre>.
- [15] AWS. What is Amazon CloudWatch Logs?, 2022. <https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/WhatIsCloudWatchLogs.html>. § 3, § 3.7.4

- [16] AWS. CloudWatch Logs Insights query syntax, 2022. https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/CWL_QuerySyntax.html. § 3.7.4
- [17] Wei Bai, Shanim Sainul Abdeen, Ankit Agrawal, Krishan Kumar Attre, Paramvir Bahl, Ameya Bhagat, Gowri Bhaskara, Tanya Brokhman, Lei Cao, Ahmad Cheema, Rebecca Chow, Jeff Cohen, Mahmoud Elhaddad, Vivek Ette, Igal Figlin, Daniel Firestone, Mathew George, Ilya German, Lakhmeet Ghai, Eric Green, Albert Greenberg, Manish Gupta, Randy Haagens, Matthew Hendel, Ridwan Howlader, Neetha John, Julia Johnstone, Tom Jolly, Greg Kramer, David Kruse, Ankit Kumar, Erica Lan, Ivan Lee, Avi Levy, Marina Lipshteyn, Xin Liu, Chen Liu, Guohan Lu, Yuemin Lu, Xiakun Lu, Vadim Makhervaks, Ulad Malashanka, David A. Maltz, Ilias Marinos, Rohan Mehta, Sharda Murthi, Anup Namdhari, Aaron Ogus, Jitendra Padhye, Madhav Pandya, Douglas Phillips, Adrian Power, Suraj Puri, Shachar Raindel, Jordan Rhee, Anthony Russo, Maneesh Sah, Ali Sheriff, Chris Sparacino, Ashutosh Srivastava, Weixiang Sun, Nick Swanson, Fuhou Tian, Lukasz Tomczyk, Vamsi Vadlamuri, Alec Wolman, Ying Xie, Joyce Yom, Lihua Yuan, Yanzhao Zhang, and Brian Zill. Empowering Azure Storage with RDMA. In *20th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '23, pages 49–67, Boston, MA, April 2023. USENIX Association. ISBN 978-1-939133-33-5. URL <https://www.usenix.org/conference/nsdi23/presentation/bai>. § 6.2.1
- [18] David Barach, Leonardo Linguaglossa, Damjan Marion, Pierre Pfister, Salvatore Pontarelli, and Dario Rossi. High-Speed Software Data Plane via Vectorized Packet Processing. *IEEE Communications Magazine*, 56(12):97–103, December 2018. URL <https://doi.org/10.1109/MCOM.2018.1800069>. § 6.2.1
- [19] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Communications of the ACM*, 60(4):48–54, March 2017. ISSN 0001-0782. URL <https://doi.org/10.1145/3015146>. § 3.3.2, § 3.7.4
- [20] 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 *11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '14, pages 49–65, Broomfield, CO, October 2014. USENIX Association. ISBN 978-1-931971-16-4. URL <https://www.usenix.org/conference/osdi14/technical-sessions/presentation/belay>. § 3.8.1
- [21] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosz, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. The CacheLib caching engine: Design and experiences at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '20, pages 753–768. USENIX Association, November 2020. ISBN 978-1-939133-19-9. URL <https://www.usenix.org/conference/osdi20/presentation/berg>. § 3

- [22] Junehyuk Boo, Yujin Chung, Eunjin Baek, Seongmin Na, Changsu Kim, and Jangwoo Kim. F4T: A fast and flexible FPGA-based full-stack TCP acceleration framework. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 979-8-4007-0095-8. URL <https://doi.org/10.1145/3579371.3589090>. § 3.6
- [23] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 99–110, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 978-1-4503-2056-6. URL <https://doi.org/10.1145/2486001.2486011>. § 6.3
- [24] S. Bradner and J. McQuaid. Benchmarking methodology for network interconnect devices. RFC 2544, March 1999. URL <https://www.rfc-editor.org/info/rfc2544>. § 3.7.1
- [25] Broadcom. PEX 8747, PCI Express Gen 3 switch, 48 lanes, 5 ports, October 2010. <https://docs.broadcom.com/docs/12351854>. § 4.6.2
- [26] Broadcom. PEX 89000 series: Managed PCI Express 5.0 switches ranging from 144 to 24 lanes, October 2023. <https://docs.broadcom.com/doc/PEX89000-Managed-PCI-Express-5.0-Switches>. § 6.1
- [27] Broadcom. BNXT poll mode driver. <https://doc.dpdk.org/guides/nics/bnxt.html>, 2026. § 3.1
- [28] Shawn T. Brown, Paola Buitrago, Edward Hanna, Sergiu Sanielevici, Robin Scibek, and Nicholas A. Nystrom. Bridges-2: A platform for rapidly-evolving and data intensive research. In *Practice and Experience in Advanced Research Computing (PEARC)*, 2021. § 4.6.2
- [29] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. Understanding host network stack overheads. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, pages 65–77, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 978-1-4503-8383-7. URL <https://doi.org/10.1145/3452296.3472888>. § 5.1
- [30] CAIDA. Anonymized internet traces 2016. https://catalog.caida.org/dataset/passive_2016_pcap, 2016. § 3.2.5, § 3.7.3, § 3.7.4
- [31] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking control of the enterprise. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '07, pages 1–12, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 978-1-59593-713-1. URL <https://doi.org/10.1145/1282380.1282382>. § 4.2

- [32] Jack Choquette. NVIDIA Hopper H100 GPU: Scaling performance. *IEEE Micro*, 43(3):9–17, 2023. § 1, § 1.4.2, § 2.1.1, § 4.6.2, § 4.6.7, § 5.2
- [33] Félix Cloutier. ENQCMD — enqueue command, 2026. URL <https://www.felixcloutier.com/x86/enqcmd>. § 5.2
- [34] Félix Cloutier. MOVDIR64B — move 64 bytes as direct store, 2026. URL <https://www.felixcloutier.com/x86/movdir64b>. § 4.3.2, § 5.2
- [35] David Cock, Abishek Ramdas, Daniel Schwyn, Michael Giardino, Adam Turowski, Zhenhao He, Nora Hossle, Dario Korolija, Melissa Licciardello, Kristina Martsenko, Reto Achermann, Gustavo Alonso, and Timothy Roscoe. Enzian: An open, general, CPU/FPGA platform for systems software research. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '22*, pages 434–451, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 978-1-4503-9205-1. URL <https://doi.org/10.1145/3503222.3507742>. § 4.7
- [36] Chelsio Communications. Terminator 5 ASIC, 2021. <https://www.chelsio.com/terminator-5-asic/>. § 1, § 2.2.1, § 3, § 3.6
- [37] CXL Consortium. Compute express link (CXL) specification revision 4.0, version 1.0. Technical report, CXL Consortium, August 2025. URL https://computeexpresslink.org/wp-content/uploads/2026/02/CXL-Specification_rev4p0_ver1p0_2026February26_clean_evalcopy_v2.pdf. § 1, § 2.1.1, § 4.7, § 5.2
- [38] Jonathan Corbet. Ringing in a new asynchronous I/O API, 2019. <https://lwn.net/Articles/776703/>. § 3.1
- [39] David Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann, San Francisco, CA, USA, 1998. ISBN 1-55860-343-3. § 5.2, § 6.3
- [40] Debian. Debian popularity contest: Statistics by source packages (sum) sorted by fields, 2022. https://popcon.debian.org/source/by_inst. § 3.7.4
- [41] Roman Dementiev et al. Processor Counter Monitor (PCM), 2026. <https://github.com/opcm/pcm>. § 3.7.1
- [42] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. RouteBricks: Exploiting parallelism to scale software routers. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, SOSP '09*, pages 15–28, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 978-1-60558-752-3. URL <https://doi.org/10.1145/1629575.1629578>. § 3.1.2, § 4.2

- [43] DPDK. Data Plane Development Kit, 2026. <https://dpdk.org>. § 3, § 3.1, § 6.2.1
- [44] DPDK. NVIDIA MLX5 ethernet driver, 2026. <https://doc.dpdk.org/guides/nics/mlx5.html>. § 1, § 2.2.1, § 3, § 3.8.2
- [45] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '14, pages 401–414, Seattle, WA, April 2014. USENIX Association. ISBN 978-1-931971-09-6. URL <https://www.usenix.org/conference/nsdi14/technical-sessions/dragojević>. § 2.3.1, § 3.8.4, § 6.2.1
- [46] Peter Druschel, Larry L. Peterson, and Bruce S. Davie. Experiences with a high-speed network adaptor: A software perspective. In *Proceedings of the Conference on Communications Architectures, Protocols and Applications*, SIGCOMM '94, pages 2–13, New York, NY, USA, 1994. Association for Computing Machinery. ISBN 0-89791-682-4. URL <https://doi.org/10.1145/190314.190315>. § 3, § 3.3.1, § 3.8.1
- [47] Umberto Eco. *How to Write a Thesis*. MIT Press, 2015. ISBN 978-0-262-52713-2.
- [48] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *13th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '16, pages 523–535, Santa Clara, CA, March 2016. USENIX Association. ISBN 978-1-931971-29-4. URL <https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/eisenbud>. § 1.4.1, § 3, § 3.7.4
- [49] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. MoonGen: A scriptable high-speed packet generator. In *Proceedings of the 2015 Internet Measurement Conference*, IMC '15, pages 275–287, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 978-1-4503-3848-6. URL <https://doi.org/10.1145/2815675.2815692>. § 3.7.1, § 3.7.2, § 6.2.1
- [50] Paul Emmerich, Maximilian Pudelko, Simon Bauer, Stefan Huber, Thomas Zwickl, and Georg Carle. User space network drivers. In *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '19, pages 1–12, Cambridge, UK, 2019. IEEE. ISBN 978-1-7281-4387-3. URL <https://doi.org/10.1109/ANCS.2019.8901894>. § 3.3.1, § 3.4.3
- [51] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 251–266, New York, NY, USA, 1995. Association for Computing Machinery. ISBN 0-89791-715-4. URL <https://doi.org/10.1145/224056.224076>. § 3.8.1

- [52] Haggai Eran, Lior Zeno, Maroun Tork, Gabi Malka, and Mark Silberstein. NICA: An infrastructure for inline acceleration of network applications. In *2019 USENIX Annual Technical Conference, ATC '19*, pages 345–362, Renton, WA, July 2019. USENIX Association. ISBN 978-1-939133-03-8. URL <https://www.usenix.org/conference/atc19/presentation/eran>. § 1, § 2.2.1, § 3, § 3.8.2
- [53] Haggai Eran, Maxim Fudim, Gabi Malka, Gal Shalom, Noam Cohen, Amit Hermony, Dotan Levi, Liran Liss, and Mark Silberstein. FlexDriver: A network driver for your accelerator. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '22*, pages 1115–1129, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 978-1-4503-9205-1. URL <https://doi.org/10.1145/3503222.3507776>. § 4.2, § 4.7
- [54] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure accelerated networking: SmartNICs in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI '18*, pages 51–66, Renton, WA, April 2018. USENIX Association. ISBN 978-1-939133-01-4. URL <https://www.usenix.org/conference/nsdi18/presentation/firestone>. § 1, § 3.1.1
- [55] Mario Flajslik and Mendel Rosenblum. Network interface design for low latency request-response protocols. In *2013 USENIX Annual Technical Conference, ATC '13*, pages 333–346, San Jose, CA, June 2013. USENIX Association. ISBN 978-1-931971-01-0. URL <https://www.usenix.org/conference/atc13/technical-sessions/presentation/flajslik>. § 3.8.3
- [56] Alex Forencich. Verilog PCI Express components, 2026. URL <https://github.com/alexforencich/verilog-pcie>. § 4.5
- [57] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI '20*, pages 281–297. USENIX Association, November 2020. ISBN 978-1-939133-19-9. URL <https://www.usenix.org/conference/osdi20/presentation/fried>. § 5.1
- [58] Joshua Fried, Gohar Irfan Chaudhry, Enrique Saurez, Esha Choukse, Inigo Goiri, Sameh Elnikety, Rodrigo Fonseca, and Adam Belay. Making kernel bypass practical for the cloud with Junction. In *21st USENIX Symposium on Networked Systems Design and Implementation, NSDI '24*, pages 55–73, Santa Clara, CA, April 2024. USENIX Association. ISBN 978-

- 1-939133-39-7. URL <https://www.usenix.org/conference/nsdi24/presentation/fried>. § 5.1
- [59] Gerasimos Gerogiannis, Serif Yesil, Damitha Lenadora, Dingyuan Cao, Charith Mendis, and Josep Torrellas. SPADE: A flexible and scalable accelerator for SpMM and SDDMM. In *Proceedings of the 50th Annual International Symposium on Computer Architecture, ISCA '23*, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 979-8-4007-0095-8. URL <https://doi.org/10.1145/3579371.3589054>. § 1, § 4.1.1
- [60] 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 *19th USENIX Symposium on Networked Systems Design and Implementation, NSDI '22*, pages 807–827, Renton, WA, April 2022. USENIX Association. ISBN 978-1-939133-27-4. URL <https://www.usenix.org/conference/nsdi22/presentation/ghasemirahni>. § 3.7.3
- [61] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, SOSP '03*, pages 29–43, New York, NY, USA, 2003. Association for Computing Machinery. ISBN 1-58113-757-5. URL <https://doi.org/10.1145/945445.945450>. § 3
- [62] Abraham Gonzalez, Aasheesh Kolli, Samira Khan, Sihang Liu, Vidushi Dadu, Sagar Karandikar, Jichuan Chang, Krste Asanovic, and Parthasarathy Ranganathan. Profiling hyperscale big data processing. In *Proceedings of the 50th Annual International Symposium on Computer Architecture, ISCA '23*, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 979-8-4007-0095-8. URL <https://doi.org/10.1145/3579371.3589082>. § 1, § 4.2
- [63] PCI Special Interest Group. PCI Local Bus Specification (Revision 2.2). Technical report, PCI Special Interest Group, December 1998. § 1
- [64] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over commodity ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference, SIGCOMM '16*, pages 202–215, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 978-1-4503-4193-6. URL <https://doi.org/10.1145/2934872.2934908>. § 6.2.1
- [65] Kaajal Gupta. Hermes: Evaluating Ensō schedulers for high-performance networking. Senior thesis, Carnegie Mellon University, Pittsburgh, PA, April 2024. URL <http://reports-archive.adm.cs.cmu.edu/anon/srthesis/SeniorThesis24-GUPTA-Kaajal.pdf>. § 5.1
- [66] Khaled Hamidouche, John Bachan, Pak Markthub, Peter-Jan Gootzen, Elena Agostini, Sylvain Jaugey, Aamir Shafi, Georgios Theodorakis, and Manjunath Gorentla Venkata.

- GPU-initiated networking for NCCL, 2025. URL <https://arxiv.org/abs/2511.15076>. § 1.1.2, § 2.3.2, § 4, § 4.1.3, § 4.2, § 4.7
- [67] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. SoftNIC: A software NIC to augment hardware. Technical Report UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015. URL <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-155.html>. § 3.1.1, § 3.2.5, § 3.6, § 3.7.3, § 6.2.1
- [68] John L. Hennessy and David A. Patterson. A new golden age for computer architecture. *Communications of the ACM*, 62(2):48–60, January 2019. ISSN 0001-0782. URL <https://doi.org/10.1145/3282307>. § 1, § 6.3
- [69] 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 *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT ’18, pages 54–66, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 978-1-4503-6080-7. URL <https://doi.org/10.1145/3281411.3281443>. § 3.8.1, § 6.2.1
- [70] Changho Hwang, Peng Cheng, Roshan Dathathri, Abhinav Jangda, Saeed Maleki, Madan Musuvathi, Olli Saarikivi, Aashaka Shah, Ziyue Yang, Binyang Li, Caio Rocha, Qinghua Zhou, Mahdieh Ghazimirsaeed, Sreevatsa Anantharamu, and Jithin Jose. MSCCL++: Rethinking GPU communication abstractions for AI inference. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS ’26, pages 1201–1215, New York, NY, USA, 2026. Association for Computing Machinery. ISBN 979-8-4007-2359-9. URL <https://doi.org/10.1145/3779212.3790188>. § 4, § 4.2
- [71] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Changhoon Kim, and Nick McKeown. The nanoPU: A nanosecond network stack for datacenters. In *15th USENIX Symposium on Operating Systems Design and Implementation*, OSDI ’21, pages 239–256. USENIX Association, July 2021. ISBN 978-1-939133-22-9. URL <https://www.usenix.org/conference/osdi21/presentation/ibanez>. § 3.7.4
- [72] Intel. Intel ISA bus specification and application notes. Technical report, Intel Corporation, September 1989. URL https://archive.org/details/bitsavers_intelbusSpec89_3342148/. § 1
- [73] Intel. High definition audio specification, revision 1.0a. Technical report, Intel Corporation, June 2010. <https://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/high-definition-audio-specification.pdf>. § 2.2.1, § 2.2.2

- [74] Intel. Intel data direct I/O technology (Intel DDIO): A primer. Technical report, Intel, February 2012. § 3.1.2
- [75] Intel. How to set up intel ethernet flow director, 2017. <https://www.intel.com/content/www/us/en/developer/articles/training/setting-up-intel-ethernet-flow-director.html>. § 4.2
- [76] Intel. Intel Iris Xe and UHD Graphics open source programmer's reference manual, volume 12: Display engine. Technical report, Intel Corporation, December 2021. <https://cdrdv2-public.intel.com/705833/intel-gfx-prm-src-tgl-vol-12-display-engine.pdf>. § 2.2.1
- [77] Intel. Intel Stratix 10 MX FPGA family overview product table, February 2021. URL <https://www.intel.com/content/www/us/en/content-details/652458/intel-stratix-10-mx-fpga-family-overview-product-table.html>. § 3.7.1, § 4.6.2
- [78] Intel. Intel Ethernet Controller E810 - Revision 2.8. Technical Report 613875-009, Intel, September 2024. URL <https://www.intel.com/content/www/us/en/content-details/613875/intel-ethernet-controller-e810-datasheet.html>. § 1, § 1.4.1, § 2.2.1, § 2.2.2, § 3, § 3.1, § 3.1.1, § 3.3.1, § 3.4.3, § 3.7.2, § 4.6.1, § 4.6.2
- [79] Intel. Intel QuickAssist technology (Intel QAT) driver for Linux for hardware version 2.0, 2025. <https://www.intel.com/content/www/us/en/download/765501/intel-quickassist-technology-intel-qat-driver-for-linux-for-hardware-version-2-0.html>. § 4.6.1
- [80] Intel. Intel Ethernet Network Adapter E810-CQDA2, 2026. <https://ark.intel.com/content/www/us/en/ark/products/210969/intel-ethernet-network-adapter-e8102cqda2.html>. § 3.7.1
- [81] Intel. Intel Core i7-7820X X-series Processor, 2026. <https://ark.intel.com/content/www/us/en/ark/products/123767/intel-core-i77820x-xseries-processor-11m-cache-up-to-4-30-ghz.html>. § 3.7.1, § 4.6.2
- [82] Intel. Intel Core i9-9960X X-series Processor, 2026. <https://ark.intel.com/content/www/us/en/ark/products/189123/intel-core-i99960x-xseries-processor-22m-cache-up-to-4-50-ghz.html>. § 3.7.1
- [83] Intel. Intel 64 and IA-32 architectures optimization reference manual. Technical Report 248966-045, Intel, 2026. <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html>. § 3.4, § 3.7.3
- [84] Intel. Intel Xeon Gold 6226R Processor, 2026. <https://www.intel.com/content/www/us/en/products/sku/199347/intel-xeon-gold-6226r-processor-22m-cache-2-90-ghz/specifications.html>. § 4.6.2

- [85] Intel. Intel infrastructure processing unit (Intel IPU), 2026. URL <https://www.intel.com/content/www/us/en/products/details/network-io/ipu.html>. § 1, § 1.1.1, § 2.2.1, § 3.6
- [86] Intel. Intel QuickAssist Adapter 8960, 2026. <https://www.intel.com/content/www/us/en/products/sku/125199/intel-quickassist-adapter-8960/specifications.html>. § 1, § 2.2.2, § 4.1.1, § 4.2, § 4.6.1
- [87] Intel. Queues and queue pairs — Intel QuickAssist Technology documentation, 2026. https://intel.github.io/quickassist/PG/infrastructure_queue_pairs.html. § 2.2.2
- [88] Intel. Top-down microarchitecture analysis method. <https://www.intel.com/content/www/us/en/develop/documentation/vtune-cookbook/top/methodologies/top-down-microarchitecture-analysis-method.html>, 2026. § 3.1.2
- [89] Jaeyoung Jang, Sung Jun Jung, Sunmin Jeong, Jun Heo, Hoon Shin, Tae Jun Ham, and Jae W. Lee. A specialized architecture for object serialization with applications to big data analytics. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ISCA '20, pages 322–334, Virtual Event, 2020. IEEE Press. ISBN 978-1-7281-4661-4. URL <https://doi.org/10.1109/ISCA45697.2020.00036>. § 3
- [90] Cyril Jaquier et al. Fail2ban, 2026. <https://www.fail2ban.org/>. § 3.7.4
- [91] 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 *11th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '14, pages 489–502, Seattle, WA, April 2014. USENIX Association. ISBN 978-1-931971-09-6. URL <https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/jeong>. § 6.2.1
- [92] Houxiang Ji, Yifan Yuan, Yang Zhou, Ipoom Jeong, Ren Wang, Saksham Agarwal, and Nam Sung Kim. Re-architecting end-host networking with CXL: Coherence, memory, and offloading. In *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture*, MICRO '25, pages 1809–1823, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 979-8-4007-1573-0. URL <https://doi.org/10.1145/3725843.3756102>. § 4.7
- [93] Norman P. Jouppi, Cliff Young, Nishant Patil, and David Patterson. A domain-specific architecture for deep neural networks. *Communications of the ACM*, 61(9):50–59, August 2018. ISSN 0001-0782. URL <https://doi.org/10.1145/3154484>. § 1, § 4, § 6.3
- [94] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for μ second-scale tail latency. In *16th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '19, pages 345–360, Boston, MA, February 2019. USENIX Association. ISBN 978-1-931971-49-2. URL <https://www.usenix.org/conference/nsdi19/presentation/kaffes>. § 5.1

- [95] James A Kahle, Michael N Day, H Peter Hofstee, Charles R Johns, Theodore R Maeurer, and David Shippy. Introduction to the Cell multiprocessor. *IBM journal of Research and Development*, 49(4.5):589–604, 2005. § 6.3
- [96] Anuj Kalia, Michael Kaminsky, and David G. Andersen. FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '16, pages 185–201, Savannah, GA, November 2016. USENIX Association. ISBN 978-1-931971-33-1. URL <https://www.usenix.org/conference/osdi16/technical-sessions/presentation/kalia>. § 3.7.4, § 6.2.1
- [97] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '19, pages 1–16, Boston, MA, February 2019. USENIX Association. ISBN 978-1-931971-49-2. URL <https://www.usenix.org/conference/nsdi19/presentation/kalia>. § 3.7.4, § 3.8.4, § 5.1
- [98] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 158–169, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 978-1-4503-3402-0. URL <https://doi.org/10.1145/2749469.2750392>. § 3
- [99] Sagar Karandikar, Chris Leary, Chris Kennelly, Jerry Zhao, Dinesh Parimi, Borivoje Nikolic, Krste Asanovic, and Parthasarathy Ranganathan. A hardware accelerator for protocol buffers. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, pages 462–478, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 978-1-4503-8557-2. URL <https://doi.org/10.1145/3466752.3480051>. § 1, § 4.1.1
- [100] Antoine Kaufmann, Simon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. High performance packet processing with FlexNIC. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 67–81, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 978-1-4503-4091-5. URL <https://doi.org/10.1145/2872362.2872367>. § 3.6, § 3.8.3
- [101] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. TAS: TCP acceleration as an OS service. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 978-1-4503-6281-8. URL <https://doi.org/10.1145/3302424.3303985>. § 3.8.1, § 6.2.1

- [102] Nikita Lazarev, Shaojie Xiang, Neil Adit, Zhiru Zhang, and Christina Delimitrou. Dagger: Efficient and fast RPCs in cloud microservices with near-Memory reconfigurable NICs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, pages 36–51, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 978-1-4503-8317-2. URL <https://doi.org/10.1145/3445814.3446696>. § 3.7.4
- [103] Anatole Lefort, Julian Pritzi, Nicolò Carpentieri, David Schall, Simon Dittrich, Soham Chakraborty, Nicolai Oswald, and Pramod Bhatotia. vCXLGen: Automated synthesis and verification of CXL bridges for heterogeneous architectures. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '26, pages 2177–2196, New York, NY, USA, 2026. Association for Computing Machinery. ISBN 979-8-4007-2359-9. URL <https://doi.org/10.1145/3779212.3790245>. § 4.7
- [104] Hejing Li, Jialin Li, and Antoine Kaufmann. SimBricks: End-to-end network system evaluation with modular simulation. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, pages 380–396, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 978-1-4503-9420-8. URL <https://doi.org/10.1145/3544216.3544253>. § 4.5, § 6.1
- [105] Wei Siew Liew, Md Ashfaque Rahaman, James McMahon, Ryan Stutsman, and Vijay Nagarajan. Stop taking the scenic route: The shortest distance between the CPU and the NIC is MMIO. In *Proceedings of the 2025 Workshop on Hot Topics in Operating Systems*, HotOS '25, pages 144–150, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 979-8-4007-1475-7. URL <https://doi.org/10.1145/3713082.3730389>. § 4.7
- [106] Wei Siew Liew, Md Ashfaque Rahaman, Adarsh Patil, Ryan Stutsman, and Vijay Nagarajan. Efficient remote memory ordering for non-coherent systems. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '26, pages 647–661, New York, NY, USA, 2026. Association for Computing Machinery. ISBN 979-8-4007-2359-9. URL <https://doi.org/10.1145/3779212.3790156>. § 4.7
- [107] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '14, pages 429–444, Seattle, WA, April 2014. USENIX Association. ISBN 978-1-931971-09-6. URL <https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/lim>. § 1.4.1, § 3, § 3.1.2, § 3.7.1, § 3.7.4, § 5.1

- [108] Linux kernel contributors. `pci_p2pdma_whitelist[]`, `drivers/pci/p2pdma.c`, lines 504–528. Linux kernel v6.16.4, 2025. URL <https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/drivers/pci/p2pdma.c?h=v6.16.4#n504>. § 6.1
- [109] Guyue Liu, Hugo Sadok, Anne Kohlbrenner, Bryan Parno, Vyas Sekar, and Justine Sherry. Don't yank my chain: Auditable NF service chaining. In *18th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '21, pages 155–173, Boston, MA, April 2021. USENIX Association. ISBN 978-1-939133-21-2. URL <https://www.usenix.org/conference/nsdi21/presentation/liu-guyue>.
- [110] Hong Liu, Ryohei Urata, Kevin Yasumura, Xiang Zhou, Roy Bannon, Jill Berger, Pedram Dashti, Norm Jouppi, Cedric Lam, Sheng Li, Erji Mao, Daniel Nelson, George Papen, Mukarram Tariq, and Amin Vahdat. Lightwave fabrics: At-scale optical circuit switching for datacenter and machine learning systems. In *Proceedings of the ACM SIGCOMM 2023 Conference*, SIGCOMM '23, pages 499–515, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 979-8-4007-0236-5. URL <https://doi.org/10.1145/3603269.3604836>. § 4
- [111] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. NitroSketch: Robust and general sketch-based monitoring in software switches. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, pages 334–350, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 978-1-4503-5956-6. URL <https://doi.org/10.1145/3341302.3342076>. § 3, § 3.7.4, § 6.2.1
- [112] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. Octopus: an RDMA-enabled distributed persistent memory file system. In *2017 USENIX Annual Technical Conference*, ATC '17, pages 773–785, Santa Clara, CA, July 2017. USENIX Association. ISBN 978-1-931971-38-6. URL <https://www.usenix.org/conference/atc17/technical-sessions/presentation/lu>. § 2.3.1
- [113] Antonis Manousis, Rahul Anand Sharma, Vyas Sekar, and Justine Sherry. Contention-aware performance prediction for virtualized network functions. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, pages 270–282, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 978-1-4503-7955-7. URL <https://doi.org/10.1145/3387514.3405868>. § 3.1.2
- [114] Pak Markthub, Jim Dinan, Sreeram Potluri, and Seth Howell. Improving network performance of HPC systems using NVIDIA Magnum IO NVSHMEM and GPUDirect Async, 2022. URL <https://developer.nvidia.com/blog/improving-network-performance-of-hpc-systems-using-nvidia-magnum-io-nvshmem-and-gpudirect-async/>. § 2.3.2

- [115] Marvell. CNXK DMA device driver. <https://doc.dpdk.org/guides/dmadevs/cnxk.html>, 2026. § 3.1
- [116] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S. Berger, Nathan Beckmann, and Gregory R. Ganger. Kangaroo: Caching billions of tiny objects on flash. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, pages 243–262, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 978-1-4503-8709-5. URL <https://doi.org/10.1145/3477132.3483568>. § 3, § 3.7.4
- [117] Mellanox Technologies. Mellanox adapters programmer's reference manual (PRM). Technical report, Mellanox Technologies, 2020. <https://network.nvidia.com/files/doc-2020/ethernet-adapters-programming-manual.pdf>. § 2.2.2
- [118] Rui Miao, Lingjun Zhu, Shu Ma, Kun Qian, Shujun Zhuang, Bo Li, Shuguang Cheng, Jiaqi Gao, Yan Zhuang, Pengcheng Zhang, Rong Liu, Chao Shi, Binzhang Fu, Jiaji Zhu, Jiesheng Wu, Dennis Cai, and Hongqiang Harry Liu. From luna to solar: The evolutions of the compute-to-storage networks in Alibaba Cloud. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, pages 753–766, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 978-1-4503-9420-8. URL <https://doi.org/10.1145/3544216.3544238>. § 3
- [119] Amirhossein Mirhosseini, Hossein Golestani, and Thomas F. Wenisch. HyperPlane: A scalable low-latency notification accelerator for software data planes. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '20, pages 852–867, 2020. URL <https://doi.org/10.1109/MICRO50266.2020.00074>. § 3.3.1
- [120] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *2013 USENIX Annual Technical Conference*, ATC '13, pages 103–114, San Jose, CA, June 2013. USENIX Association. ISBN 978-1-931971-01-0. URL <https://www.usenix.org/conference/atc13/technical-sessions/presentation/mitchell>. § 2.3.1
- [121] Christopher Mitchell, Kate Montgomery, Lamont Nelson, Siddhartha Sen, and Jinyang Li. Balancing CPU and network in the cell distributed B-Tree store. In *2016 USENIX Annual Technical Conference*, ATC '16, pages 451–464, Denver, CO, June 2016. USENIX Association. ISBN 978-1-931971-30-0. URL <https://www.usenix.org/conference/atc16/technical-sessions/presentation/mitchell>. § 2.3.1
- [122] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 221–235, Budapest, Hungary, 2018. Association for Computing

- Machinery. ISBN 978-1-4503-5567-4. URL <https://doi.org/10.1145/3230543.3230564>. § 3.6, § 6.2.1
- [123] Al Morton. RFC Errata, Erratum ID 412, RFC 2544, November 2006. <https://www.rfc-editor.org/errata/eid422>. § 3.7.1
- [124] Shubhendu S. Mukherjee, Babak Falsafi, Mark D. Hill, and David A. Wood. Coherent network interfaces for fine-grain communication. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, ISCA '96, pages 247–258, New York, NY, USA, 1996. Association for Computing Machinery. ISBN 0-89791-786-3. URL <https://doi.org/10.1145/232973.232999>. § 4.7
- [125] Emerson Murphy-Hill, Jenna Butler, and Alexandra Savelieva. Adoption and impact of command-line ai coding agents: A study of microsoft’s early 2026 rollout of claude code and github copilot cli, 2026. URL <https://arxiv.org/abs/2607.01418>. § 6.2.2, § 6.3
- [126] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. Understanding PCIe performance for end host networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 327–341, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 978-1-4503-5567-4. URL <https://doi.org/10.1145/3230543.3230560>. § 3.1.2, § 3.4, § 3.7.3
- [127] Joel Nider and Alexandra (Sasha) Fedorova. The last CPU. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, pages 1–8, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 978-1-4503-8438-4. URL <https://doi.org/10.1145/3458336.3465291>. § 4.7
- [128] Stoyan Nikolov, Daniele Decasas, Anna Sjövall, Maxim Tabachnyk, Satish Chandra, Siddharth Taneja, and Celal Ziftci. How is google using ai for internal code migrations? In *2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 481–492. IEEE Press, 2025. doi: 10.1109/ICSE-SEIP66354.2025.00048. URL <https://doi.org/10.1109/ICSE-SEIP66354.2025.00048>. § 6.2.2, § 6.3
- [129] Nvidia. NVIDIA Hopper architecture in-depth, 2022. URL <https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/>. § 4.6.2
- [130] Nvidia. NVIDIA Blackwell architecture: Technical brief. Technical report, Nvidia, 2025. <https://resources.nvidia.com/en-us-blackwell-architecture>. § 2.1.1
- [131] Nvidia. Fusing communication and compute with new device api and copy engine collectives in NVIDIA NCCL 2.28, November 2025. URL <https://developer.nvidia.com/blog/fusing-communication-and-compute-with-new-device-api-and-copy-engine-collectives-in-nvidia-nccl-2-28/>. § 4.6.7

- [132] Nvidia. NVIDIA BlueField-3 DPU, 2026. <https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-3-dpu.pdf>. § 1, § 1.1.1, § 2.2.1, § 3.7.2
- [133] Nvidia. NVIDIA GPUDirect, 2026. URL <https://developer.nvidia.com/gpudirect>. § 2.3.1, § 4, § 4.1.2, § 4.1.3, § 4.7
- [134] Nvidia. DOCA GPUNetIO, 2026. URL <https://docs.nvidia.com/doca/sdk/doca-gpunetio/index.html>. § 1.1.2, § 2.3.2, § 4, § 4.1.3, § 4.7
- [135] Nvidia. NVIDIA Mellanox Innova-2 Flex open programmable smartnic, 2026. <https://www.nvidia.com/en-in/networking/ethernet/innova-2-flex/>. § 1, § 1.1.1, § 2.2.1
- [136] Nvidia. NVIDIA collective communications library (NCCL), 2026. URL <https://developer.nvidia.com/nccl>. § 4.2, § 6.2.1
- [137] Nvidia. NVSHMEM, 2026. URL <https://developer.nvidia.com/nvshmem>. § 1.4.2, § 2.3.2, § 4, § 4.6.7
- [138] Nvidia. Using TMA with NVSHMEM, 2026. URL <https://docs.nvidia.com/nvshmem/api/latest/tma.html>. § 4.6.7
- [139] Nvidia. Parallel thread execution ISA: Release 9.3. Technical report, NVIDIA Corporation, 2026. URL https://docs.nvidia.com/cuda/pdf/ptx_isa_9.3.pdf. § 4.6.7
- [140] NVM Express, Inc. NVMe over Fabrics: White Paper. https://nvmexpress.org/wp-content/uploads/NVMe_Over_Fabrics.pdf, 2016. § 4.1.1
- [141] NVM Express, Inc. NVM Express base specification, revision 2.1. Technical report, NVM Express, Inc., August 2024. <https://nvmexpress.org/wp-content/uploads/NVM-Express-Base-Specification-Revision-2.1-2024.08.05-Ratified.pdf>. § 2.2.2
- [142] NVM Express, Inc. NVM Express: Computational programs command set specification, revision 1.2. Technical report, NVM Express, Inc., July 2025. <https://nvmexpress.org/wp-content/uploads/NVM-Express-Computational-Programs-Command-Set-Specification-Revision-1.2-2025.08.01-Ratified.pdf>. § 1
- [143] NVM Express, Inc. NVM Express: NVM command set specification, revision 1.2. Technical report, NVM Express, Inc., July 2025. <https://nvmexpress.org/wp-content/uploads/NVM-Express-NVM-Command-Set-Specification-Revision-1.2-2025.08.01-Ratified.pdf>. § 2.2.1
- [144] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '19, pages 361–378, Boston, MA, February 2019. USENIX Association. ISBN 978-1-931971-49-2. URL <https://www.usenix.org/conference/nsdi19/presentation/ousterhout>. § 5.1

- [145] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. The RAMCloud storage system. *ACM Transactions on Computer Systems*, 33(3), August 2015. ISSN 0734-2071. URL <https://doi.org/10.1145/2806887>. § 5.1
- [146] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. NetBricks: Taking the V out of NFV. In *12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '16, pages 203–216, Savannah, GA, November 2016. USENIX Association. ISBN 978-1-931971-33-1. URL <https://www.usenix.org/conference/osdi16/technical-sessions/presentation/panda>. § 3.7.4, § 6.2.1
- [147] Gregory M. Papadopoulos and David E. Culler. Monsoon: An explicit token-store architecture. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA '90, pages 82–91, 1990. URL <https://doi.org/10.1145/325164.325117>. § 6.3
- [148] Craig Partridge. *Gigabit Networking*. Addison Wesley, Boston, MA, October 1993. ISBN 978-0201563337. § 1.1.1
- [149] PCI-SIG. PCI Express Base Specification Revision 1.0. Technical report, PCI-SIG, 2002. URL <https://pcisig.com/>. § 1
- [150] PCI-SIG. PCI Express Base Specification Revision 6.0, Version 1.0. Technical report, PCI-SIG, 2022. URL <https://pcisig.com/>. § 1, § 2.1.1, § 5.2
- [151] Pensando. Pensando DSC-100 distributed services card. Technical Report PPB19002. Rev 6, Pensando, 2021. § 1, § 1.1.1, § 2.2.1
- [152] Perf. perf: Linux profiling with performance counters, 2026. <https://perf.wiki.kernel.org>. § 3.7.1
- [153] 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 *11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '14, pages 1–16, Broomfield, CO, October 2014. USENIX Association. ISBN 978-1-931971-16-4. URL <https://www.usenix.org/conference/osdi14/technical-sessions/presentation/peter>. § 3.8.1, § 5.1
- [154] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martin Casado. The design and implementation of Open vSwitch. In *12th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '15, pages 117–130, Oakland, CA, May 2015. USENIX Association. ISBN 978-1-931971-21-8. URL <https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/pfaff>. § 3.1.1, § 3.2.5, § 3.6, § 3.7.3, § 6.2.1

- [155] Matt Pharr and Randima Fernando, editors. *GPU Gems 2: Programming Techniques For High-Performance Graphics And General-Purpose Computation*. Addison-Wesley Professional, hardcover edition, 2005. ISBN 978-0321335593. § 6.3
- [156] Solal Pirelli and George Candea. A simpler and faster NIC driver model for network functions. In *14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '20, pages 225–241. USENIX Association, November 2020. ISBN 978-1-939133-19-9. URL <https://www.usenix.org/conference/osdi20/presentation/pirelli>. § 2.2.1, § 3.1, § 3.6, § 3.8.4, § 4.6.1
- [157] Boris Pismenny, Liran Liss, Adam Morrison, and Dan Tsafir. The benefits of general-purpose on-NIC memory. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, pages 1130–1147, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 978-1-4503-9205-1. URL <https://doi.org/10.1145/3503222.3507711>. § 3.7.4, § 3.8.3
- [158] Boris Pismenny, Adam Morrison, and Dan Tsafir. ShRing: Networking with shared receive rings. In *17th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '23, pages 949–968, Boston, MA, July 2023. USENIX Association. ISBN 978-1-939133-34-2. URL <https://www.usenix.org/conference/osdi23/presentation/pismenny>. § 3.8.2
- [159] Arash Pourhabibi, Siddharth Gupta, Hussein Kassir, Mark Sutherland, Zilu Tian, Mario Paulo Drumond, Babak Falsafi, and Christoph Koch. Optimus prime: Accelerating data transformation in servers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, pages 1203–1216, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 978-1-4503-7102-5. URL <https://doi.org/10.1145/3373376.3378501>. § 3
- [160] George Prekas, Marios Kogias, and Edouard Bugnion. ZygOS: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 325–341, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 978-1-4503-5085-3. URL <https://doi.org/10.1145/3132747.3132780>. § 5.1
- [161] Deepti Raghavan, Philip Levis, Matei Zaharia, and Irene Zhang. Breakfast of champions: Towards zero-copy serialization with NIC scatter-gather. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, pages 199–205, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 978-1-4503-8438-4. URL <https://doi.org/10.1145/3458336.3465287>. § 3

- [162] Kshitij Rana. Building a high-performance in-kernel traffic shaper. Master’s thesis, Carnegie Mellon University, Pittsburgh, PA, 2024. ProQuest No. 3145403094. § 5.1
- [163] Parthasarathy Ranganathan, Daniel Stodolsky, Jeff Calow, Jeremy Dorfman, Marisabel Guevara, Clinton Wills Smullen IV, Aki Kuusela, Raghu Balasubramanian, Sandeep Bhatia, Prakash Chauhan, Anna Cheung, In Suk Chong, Niranjani Dasharathi, Jia Feng, Brian Fosco, Samuel Foss, Ben Gelb, Sara J. Gwin, Yoshiaki Hase, Da-ke He, C. Richard Ho, Roy W. Huffman Jr., Elisha Indupalli, Indira Jayaram, Poonacha Kongetira, Cho Mon Kyaw, Aaron Laursen, Yuan Li, Fong Lou, Kyle A. Lucke, JP Maaninen, Ramon Macias, Maire Mahony, David Alexander Munday, Srikanth Muroor, Narayana Penukonda, Eric Perkins-Argueta, Devin Persaud, Alex Ramirez, Ville-Mikko Rautio, Yolanda Ripley, Amir Salek, Sathish Sekar, Sergey N. Sokolov, Rob Springer, Don Stark, Mercedes Tan, Mark S. Wachsler, Andrew C. Walton, David A. Wickerad, Alvin Wijaya, and Hon Kwan Wu. Warehouse-scale video acceleration: Co-design and deployment in the wild. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS ’21, pages 600–615, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 978-1-4503-8317-2. URL <https://doi.org/10.1145/3445814.3446723>. § 1
- [164] Linux RDMA. RDMA core userspace libraries and daemons, 2026. <https://github.com/linux-rdma/rdma-core>. § 2.3.1, § 3.8.2
- [165] H. M. Reddy, Y. Chen, J. Lan, I. Katsavounidis, B. Anandharengan, H. G. Lalgudi, S. Alaparthi, G. Hua, H.-C. Chuang, P.-H. Wu, Z. Lei, A. Mastro, C. Petersen, G. Chaudhari, P. Prakash, S. Regunathan, S. Reddy, P. Venkatapuram, V. Rao, K. Noru, A. Bjorlin, M. Zeile, A. Lewis, A. Singh, A. Sunil, C.-C. Chen, C.-F. Lin, C. Chen, D. P. Sundar, D. Jayaraman, H. Ucar, H. Li, J. Singh, J. C. C. Liu, K. R. Rachamreddy, K. Sriadibhatla, K. Datla, L. V. D. Berg, L. Feng, P. Jampani, R. Moola, R. Mallya, S. Jha, S. Pan, S. Srinivasan, V. Vaduganathan, X. Zha, Z. Wang, A. K. Sengottuvel, B. Alluri, B. Oshin, C. Kanumetta, E. Sahin, J. M. Athaide, J. Wu, K. C. Kurapati, K. Manthathi, K. Thottempudi, R. R. Chennamsetti, K. R. Jagannath, S. Arvapalli, T. Kala, T. Wang, P. Chopda, K. Gandhi, A. Ramesh, R. Gupta, S. Fadnavis, A. Qassoud, C. Friedt, F. Li, H. Gao, J. Lee, M. Dixit, S. Ugaji, T. Karuturi, X. Xie, A. Narasimha, B. Jakka, B. Dodds, J. Yang, K. Skandakumaran, M. Modi, P. Modi, C. Stejerean, D. Ronca, H. Wang, N. Pham, L. Lu, H. Shen, J. Ning, K. Narayanan, L. Chen, N. Avidan, W. Arnold, F. Xu, G. Patil, V. Balan, and S. D. Grandhi. Efficient video processing at scale using MSVP. In Andrew G. Tescher and Touradj Ebrahimi, editors, *Applications of Digital Image Processing XLVI*, volume 12674, page 1267414. SPIE / International Society for Optics and Photonics, 2023. URL <https://doi.org/10.1117/12.2685875>. § 1
- [166] Luigi Rizzo. netmap: A novel framework for fast packet I/O. In *2012 USENIX Annual Technical Conference*, ATC ’12, pages 101–112, Boston, MA, 2012. USENIX Association.

- ISBN 978-931971-93-5. URL <https://www.usenix.org/conference/atc12/technical-sessions/presentation/rizzo>. § 3.1, § 3.8.1
- [167] Hugo Sadok, Miguel Elias M. Campista, and Luís Henrique M. K. Costa. A case for spraying packets in software middleboxes. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, HotNets '18, pages 127–133, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 978-1-4503-6120-0. URL <https://doi.org/10.1145/3286062.3286081>. § 3.7.2
- [168] Hugo Sadok, Zhipeng Zhao, Valerie Choung, Nirav Atre, Daniel S. Berger, James C. Hoe, Aurojit Panda, and Justine Sherry. We need kernel interposition over the network dataplane. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, pages 152–158, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 978-1-4503-8438-4. URL <https://doi.org/10.1145/3458336.3465281>. § 3.8.1, § 5.1
- [169] Hugo Sadok, Nirav Atre, Zhipeng Zhao, Daniel S. Berger, James C. Hoe, Aurojit Panda, Justine Sherry, and Ren Wang. Ensō: A streaming interface for NIC-application communication. In *17th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '23, pages 1005–1025, Boston, MA, July 2023. USENIX Association. ISBN 978-1-939133-34-2. URL <https://www.usenix.org/conference/osdi23/presentation/sadok>. § 1.5
- [170] Hugo Sadok, Aurojit Panda, and Justine Sherry. Of apples and oranges: Fair comparisons in heterogenous systems evaluation. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*, HotNets '23, pages 1–8, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 979-8-4007-0415-4. URL <https://doi.org/10.1145/3626111.3628186>. § 4.6.8
- [171] Hugo Sadok, Nirav Atre, Arvind Krishnamurthy, Aurojit Panda, and Justine Sherry. A general device communication protocol. Under submission, 2026. § 1.5
- [172] Samsung. Samsung electronics develops second-generation SmartSSD computational storage drive with upgraded processing functionality. Samsung Newsroom, July 2022. URL <https://news.samsung.com/global/samsung-electronics-develops-second-generation-smartssd-computational-storage-drive-with-upgraded-processing-functionality>. § 1.1.1
- [173] Alireza Sanaee, Farbod Shahinfar, Gianni Antichi, and Brent E. Stephens. Backdraft: A lossless virtual switch that prevents the slow receiver problem. In *19th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '22, pages 1375–1392, Renton, WA, April 2022. USENIX Association. ISBN 978-1-939133-27-4. URL <https://www.usenix.org/conference/nsdi22/presentation/sanaee>. § 3.3.1

- [174] Michael S. Schlansker and B. Ramakrishna Rau. EPIC: Explicitly parallel instruction computing. *IEEE Computer*, 33(2):37–45, 2000. doi: 10.1109/2.820037. § 6.3
- [175] Henry N. Schuh, Arvind Krishnamurthy, David Culler, Henry M. Levy, Luigi Rizzo, Samira Khan, and Brent E. Stephens. CC-NIC: A cache-coherent interface to the NIC. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '24, pages 52–68, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 979-8-4007-0372-0. URL <https://doi.org/10.1145/3617232.3624868>. § 4.7
- [176] Rajath Shashidhara, Tim Stamler, Antoine Kaufmann, and Simon Peter. FlexTOE: Flexible TCP offload with fine-grained parallelism. In *19th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '22, pages 87–102, Renton, WA, April 2022. USENIX Association. ISBN 978-1-939133-27-4. URL <https://www.usenix.org/conference/nsdi22/presentation/shashidhara>. § 1, § 2.2.1, § 3, § 3.6
- [177] Ran Shu, Peng Cheng, Guo Chen, Zhiyuan Guo, Lei Qu, Yongqiang Xiong, Derek Chiou, and Thomas Moscibroda. Direct Universal Access: Making data center resources available to FPGA. In *16th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '19, pages 127–140, Boston, MA, February 2019. USENIX Association. ISBN 978-1-931971-49-2. URL <https://www.usenix.org/conference/nsdi19/presentation/shu>. § 4.7
- [178] Arjun Singhvi, Aditya Akella, Dan Gibson, Thomas F. Wenisich, Monica Wong-Chan, Sean Clark, Milo M. K. Martin, Moray McLaren, Prashant Chandra, Rob Cauble, Hassan M. G. Wassel, Behnam Montazeri, Simon L. Sabato, Joel Scherpelz, and Amin Vahdat. 1RMA: Re-envisioning remote memory access for multi-tenant datacenters. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, pages 708–721, Virtual Event, USA, 2020. Association for Computing Machinery. ISBN 978-1-4503-7955-7. URL <https://doi.org/10.1145/3387514.3405897>. § 3.6
- [179] Arjun Singhvi, Nandita Dukkupati, Prashant Chandra, Hassan M. G. Wassel, Naveen Kr. Sharma, Anthony Rebello, Henry Schuh, Praveen Kumar, Behnam Montazeri, Neelesh Bansod, Sarin Thomas, Inho Cho, Hyojeong Lee Seibert, Baijun Wu, Rui Yang, Yuliang Li, Kai Huang, Qianwen Yin, Abhishek Agarwal, Srinivas Vaduvatha, Weihuang Wang, Masoud Moshref, Tao Ji, David Wetherall, and Amin Vahdat. Falcon: A reliable, low latency hardware transport. In *Proceedings of the ACM SIGCOMM 2025 Conference*, SIGCOMM '25, pages 248–263, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 979-8-4007-1524-2. URL <https://doi.org/10.1145/3718958.3754353>. § 1, § 2.2.1, § 4.7

- [180] Athinagoras Skiadopoulos, Zhiqiang Xie, Mark Zhao, Qizhe Cai, Saksham Agarwal, Jacob Adelman, 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 *18th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '24, pages 405–423, Santa Clara, CA, July 2024. USENIX Association. ISBN 978-1-939133-40-3. URL <https://www.usenix.org/conference/osdi24/presentation/skiadopoulos>. § 4.7
- [181] Igor Smolyar, Alex Markuze, Boris Pismenny, Haggai Eran, Gerd Zellweger, Austin Bolen, Liran Liss, Adam Morrison, and Dan Tsafrir. IOctopus: Outsmarting nonuniform DMA. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, pages 101–115, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 978-1-4503-7102-5. URL <https://doi.org/10.1145/3373376.3378509>. § 4.6.2
- [182] Akshitha Sriraman and Abhishek Dhanotia. Accelerometer: Understanding acceleration opportunities for data center overheads at hyperscale. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, pages 733–750, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 978-1-4503-7102-5. URL <https://doi.org/10.1145/3373376.3378450>. § 1, § 4.1.1
- [183] Mark Sutherland, Siddharth Gupta, Babak Falsafi, Virendra Marathe, Dionisios Pnevmatikatos, and Alexandres Daglis. The NeBuLa RPC-optimized architecture. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ISCA '20, pages 199–212, Virtual Event, 2020. IEEE Press. ISBN 978-1-7281-4661-4. URL <https://doi.org/10.1109/ISCA45697.2020.00027>. § 3.7.4
- [184] Tencent. F-Stack: A High Performance Userspace Network Stack Based on DPDK and FreeBSD. <https://github.com/F-Stack/f-stack>, 2026. § 6.2.1
- [185] Chandramohan A. Thekkath, Thu D. Nguyen, Evelyn Moy, and Edward D. Lazowska. Implementing network protocols at user level. In *Conference Proceedings on Communications Architectures, Protocols and Applications*, SIGCOMM '93, pages 64–73, New York, NY, USA, 1993. Association for Computing Machinery. ISBN 0-89791-619-0. URL <https://doi.org/10.1145/166237.166244>. § 3.8.1
- [186] Amin Tootoonchian, Aurojit Panda, Chang Lan, Melvin Walls, Katerina Argyraki, Sylvia Ratnasamy, and Scott Shenker. ResQ: Enabling SLOs in network function virtualization. In *15th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '18, pages 283–297, Renton, WA, April 2018. USENIX Association. ISBN 978-1-939133-01-4. URL <https://www.usenix.org/conference/nsdi18/presentation/tootoonchian>. § 3.1.2, § 3.8.2

- [187] Maroun Tork, Lina Maudlej, and Mark Silberstein. Lynx: A SmartNIC-driven accelerator-centric architecture for network servers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, pages 117–131, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 978-1-4503-7102-5. URL <https://doi.org/10.1145/3373376.3378528>. § 4.2, § 4.7
- [188] Animesh Trivedi and Marco Spaziani Brunella. CPU-free computing: A vision with a blueprint. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, HotOS '23, pages 1–14, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 979-8-4007-0195-5. URL <https://doi.org/10.1145/3593856.3595906>. § 4.7
- [189] William Tu, Yi-Hung Wei, Gianni Antichi, and Ben Pfaff. Revisiting the Open VSwitch dataplane ten years later. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, pages 245–257, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 978-1-4503-8383-7. URL <https://doi.org/10.1145/3452296.3472914>. § 6.2.1
- [190] Willy R. Vasquez, Stephen Checkoway, and Hovav Shacham. The most dangerous codec in the world: Finding and exploiting vulnerabilities in h.264 decoders. In *32nd USENIX Security Symposium*, USENIX Security '23, pages 6647–6664, Anaheim, CA, August 2023. USENIX Association. ISBN 978-1-939133-37-3. URL <https://www.usenix.org/conference/usenixsecurity23/presentation/vasquez>. § 1
- [191] Anshuman Verma, Tian Tan, Ahmed Abdelsalam, Milan Dasgupta, Jonathan Hunter, Zach Libby, Narayanan Ravichandran, Harish Srinivasan, Matt Reat, Nadeen Gebara, Vishal Gondaliya, Ezz Hamed, Rahul Garlapati, Lok Chand Koppaka, Abdullah Mughrabi, Dev Desai, Alexander Malyshev, Shwetha Bhat, Rohan Kandi, Megan Sng, Tushar Garg, Muluken Hailesellasie, Andrew Putnam, Derek Chiou, Osman Ertugay, Alireza Dabagh, Vivek Bhanu, and Daniel Firestone. Rules offload engine (ROE): Accelerating host SDN policy evaluation. In *Proceedings of the ACM SIGCOMM 2026 Conference*, SIGCOMM '26, pages 837–850, New York, NY, USA, 2026. Association for Computing Machinery. ISBN 979-8-4007-2467-1. URL <https://doi.org/10.1145/3789240.3829118>. § 1
- [192] Amy Viviano. Introduction to receive side scaling, 2026. <https://docs.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling>. § 3.1.1, § 3.6, § 4.2
- [193] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-Net: A user-level network interface for parallel and distributed computing. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 40–53, New York, NY, USA, 1995. Association for Computing Machinery. ISBN 0-89791-715-4. URL <https://doi.org/10.1145/224056.224061>. § 3.8.2

- [194] Minhu Wang, Mingwei Xu, and Jianping Wu. Understanding I/O direct cache access performance for end host networking. In *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, volume 6, New York, NY, USA, March 2022. Association for Computing Machinery. URL <https://doi.org/10.1145/3508042>. § 3.4
- [195] Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park, Geoff Langdale, Jiayu Hu, and Heqing Zhu. Hyperscan: A fast multi-pattern regex matcher for modern CPUs. In *16th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '19, pages 631–648, Boston, MA, February 2019. USENIX Association. ISBN 978-1-931971-49-2. URL <https://www.usenix.org/conference/nsdi19/presentation/wang-xiang>. § 3.7.4
- [196] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 87–104, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450338349. doi: 10.1145/2815400.2815419. URL <https://doi.org/10.1145/2815400.2815419>. § 2.3.1
- [197] Keith Wiles et al. The Pktgen application, 2026. <https://pktgen-dpdk.readthedocs.io/>. § 3.7.1
- [198] Adam Wolnikowski, Stephen Ibanez, Jonathan Stone, Changhoon Kim, Rajit Manohar, and Robert Soulé. Zerializer: Towards zero-copy serialization. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, pages 206–212, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 978-1-4503-8438-4. URL <https://doi.org/10.1145/3458336.3465283>. § 3
- [199] Kenichi Yasukata, Michio Honda, Douglas Santry, and Lars Eggert. StackMap: Low-latency networking with the OS stack and dedicated NICs. In *2016 USENIX Annual Technical Conference*, ATC '16, pages 43–56, Denver, CO, June 2016. USENIX Association. ISBN 978-1-931971-30-0. URL <https://www.usenix.org/conference/atc16/technical-sessions/presentation/yasukata>. § 3.8.1
- [200] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, 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 *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, pages 195–211, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 978-1-4503-8709-5. URL <https://doi.org/10.1145/3477132.3483569>. § 3, § 3.8.1, § 5.1, § 6.2.1
- [201] Xiantao Zhang, Xiao Zheng, Zhi Wang, Hang Yang, Yibin Shen, and Xin Long. High-density multi-tenant bare-metal cloud. In *Proceedings of the Twenty-Fifth International*

- Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, pages 483–495, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 978-1-4503-7102-5. URL <https://doi.org/10.1145/3373376.3378507>. § 1, § 3
- [202] Zhipeng Zhao, Hugo Sadok, Nirav Atre, James C. Hoe, Vyas Sekar, and Justine Sherry. Achieving 100Gbps intrusion prevention on a single server. In *14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '20, pages 1083–1100. USENIX Association, November 2020. ISBN 978-1-939133-19-9. URL <https://www.usenix.org/conference/osdi20/presentation/zhao-zhipeng>.