Streaming Abstractions for Intra-Host Communication

Thesis Proposal

Hugo de Freitas Siqueira Sadok Menna Barreto
sadok@cmu.edu

March 2024

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.

Abstract

Hosts have historically been designed around the CPU, with peripheral devices playing an auxiliary role. Today, however, the rise of specialized computing architectures has shifted the role of peripheral devices. Accelerators and SmartNICs now perform a significant fraction of computation and can work independently from the CPU. Yet, existing abstractions for intra-host communication still assume CPU control. This abstraction mismatch forces the CPU to be on the datapath; compromising performance and scalability. These problems stem from three fundamental issues with existing abstractions. First, existing abstractions need ad hoc and complex routing logic between devices, requiring the CPU to be involved in the routing of data, even when the data does not need to be processed on the CPU. Second, existing abstractions make data accesses unpredictable, making it hard to mask access latencies. Finally, existing abstractions impose fixed and application-specific data formats, requiring the CPU to reformat the data in order to glue different devices and applications.

The main claim in this work is that streaming abstractions allow intra-host communication to be more scalable and performant. Regarding routing, streaming allows the CPU to be removed from the datapath, only being used to make coarse-grained decisions that can then be implemented in the dataplane outside the CPU. Regarding access latencies, streaming makes data accesses predictable; this can be leveraged by CPUs and accelerators, allowing them to fetch the next input ahead of time. Regarding data format, streaming does not impose data boundaries, allowing the same interface to be used for different kinds of functionalities with minimal glue logic.

We show the benefits of streaming for intra-host communication with Ensō, a streaming interface designed for communication between NICs and the CPU. We then discuss our proposal for bringing the benefits of streaming to other devices through a programmable host interconnect.

Contents

1 Introduction

Historically, the primary purpose of host interconnects was to connect the CPU to I/O devices [10, 35, 36]. The underlying assumption behind this design is that CPUs are the primary computing device, and peripheral devices simply allow the CPU 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. The interface exposed by these devices also reflects this assumption. Existing interfaces assume CPU control, making the CPU responsible for orchestrating data that come in and out of the peripheral device.

Over the last 20 years, however, peripheral devices have become more complex, taking over an increasing fraction of the computation originally performed at the CPU. This trend manifests itself in two ways: Existing I/O devices are now capable of much richer computation and many accelerators have been proposed that deliver better performance per watt than traditional CPUs by targeting a narrower set of applications [1, 34, 48, 51, 54, 80, 81, 88].

I/O Devices with richer computation: I/O devices such as NICs and GPUs now support a wide range of offloads. In the case of NICs, offloads range from simple operations, such as checksum [22, 41], to complex transport protocol implementations [4, 17, 28, 87] or even support for arbitrary computation—in the case of SmartNICs [67, 71] and DPUs [45]. GPUs have followed a similar path, offering increasingly more complex offloads as well as the ability to perform arbitrary computation [75].

Accelerators: Different from I/O devices, where data sent from the CPU is directed outside. Accelerators do some computation on the data and then direct the output back to the CPU. Accelerators also rely on specialized architecture in order to improve performance for a particular class of applications. In fact, while GPUs were originally conceived as an I/O device, their ability to do highly parallel computation more efficiently than the CPU has caused them to be increasingly used as an accelerator, where the result of the computation is sent not to an external display but back to the CPU itself.

With peripheral devices becoming more capable, the assumption of CPU control is increasingly inadequate. Some peripheral devices are now able to process data at an order of magnitude higher bandwidth than existing CPUs [51]. As a result, imposing CPU control causes the CPU to become a bottleneck when the goal is simply to feed accelerators with data coming from the network. Moreover, as we will see in more detail in §2.2, having the CPU in control imposes extra round trips over the host interconnect and prevents peripheral devices from communicating efficiently among each other.

This work argues that *streaming abstractions* are better suited for interconnecting computing devices within a host. Using streaming abstractions leads to several advantages over the existing CPU-controlled interfaces we use today:

Dataplane routing: Existing CPU-controlled interfaces assume a shared memory abstraction, which requires the CPU and the device to coordinate access to the memory. Because of the assumption of CPU control, typically the CPU is responsible for managing the shared memory, deciding where to place every piece of data that the device outputs. With streaming, all pieces

of data belonging to the same stream can be routed in the same way. As a result, the decision of where to send each piece of data can be executed in the dataplane—demultiplexing in I/O devices (§3) or routing in the host interconnect (§4). This avoids coordination overheads, as well as the need for the CPU to be involved in every transfer.

Data push: Another issue with the shared memory model is that communication is often done by exchanging pointers to arbitrary memory locations. While exchanging pointers between devices can be used to avoid explicit data copies, it also leads to unpredictable memory accesses—as the devices cannot know where the data is until they access the pointer. This unpredictability prevents optimizations such as prefetching, exposing memory access latencies. Streaming instead allows one device to *push* data to another, masking memory access latencies. In the case of data being pushed to the CPU, streaming allows data from the same stream to be placed sequentially in host memory, allowing the CPU to prefetch subsequent data ahead of time (§3). When sending data to devices, streaming allows data to be pushed directly to device memory (§4), avoiding multiple round trips over the host interconnect.

Bytestream abstraction: The need to coordinate shared memory also requires assumptions about data formats. When the size of the data is not known *a priori*, devices need to choose between two different strategies: They can either wait for the data to arrive before requesting space in the shared memory, which inflates latency; or they can preallocate fixed-sized buffers. Fixed-sized buffers work reasonably well for lower-level functionality that rely on fixed-sized units, e.g., packets in the case of NICs, or blocks in the case of disks. But because peripheral devices increasingly operate at a higher level, data sizes can exceed the low-level blocks. As such, fixed-sized buffers often lead to data fragmentation when high-level data exceed the buffer size. Fragmentation adds overhead, as it requires us to recombine the different pieces of data before processing. Fragmentation also amplifies memory access latency overheads, as we need to do multiple pointer chases to access the same data. Streaming instead allows us to expose a bytestream abstraction that gives the illusion of an unbounded buffer. This avoids fragmentation without the need to know the data sizes *a priori*. A bytestream abstraction also provides generality as it can serve as a unifying abstraction in which devices can overlay different kinds of data. Having a unified abstraction simplifies interoperability between different devices.

In summary, this work supports the following thesis:

Thesis Statement: *Streaming abstractions improve interoperability and efficiency for intra-host communication by providing a unifying abstraction, routing data on the dataplane, and making data access patterns predictable.*

1.1 Outline

We start by describing how existing device interfaces typically work and the problems they cause by assuming CPU control §2. We then describe Ensō, a streaming interface between an I/O device (NIC) and the CPU in §3. Then, in §4, we discuss our proposed work to enable streaming abstractions among peripheral devices on the server through a programmable host interconnect. Finally, in §5 we detail the expected timeline until the PhD defense.

2 Background and Motivation

In what follows we describe how existing accelerator interfaces work (§2.1) and how the assumption of CPU control imposes unnecessary overheads (§2.2).

2.1 Existing Accelerator Interface

While peripheral devices perform an increasing amount of functionality and can often process data at a higher bandwidth than the CPU itself, the interface that they expose is still designed with the same assumption that we had when peripheral devices were simply a way for the CPU to interface with the outside world. As a result, their interfaces push control to the CPU. Making the CPU responsible for coordinating all data that come in and out of the device.

We illustrate a typical accelerator interface in Figure 1. It shows an example of a CPU contacting an Intel QAT Card [48] used to offload encryption. The CPU sends commands to the QAT card using a command queue that is stored on host memory. The command queue is implemented as a ring buffer with the tail pointer controlled by the CPU and the head pointer controlled by the QAT card. Each descriptor enqueued by the CPU includes a command (e.g., encryption algorithm to use), as well as pointers to the input and output buffers.

After enqueueing the descriptors, the CPU advances the tail pointer using an MMIO write to the device ①. The device then uses a DMA read to read the descriptor ② and a second DMA read to read the input buffer ③. After completing the operation, the device writes the output of the computation to the output buffer, using a DMA write ④, and overwrites the original descriptor with a completion notification, with another DMA write ⑤. Note how this interface gives full control over both input and output data to the CPU. It also gives the CPU control over the accelerator functionality, making it responsible for feeding the accelerator with commands that determine what it should process next.

The above description applies to a typical accelerator interface—where the descriptor specifies both an input and an output. But the interface exposed by existing NICs also assume CPU control and works similarly. In §3.1.1, we describe existing NIC interfaces in more detail, highlighting how the assumption of CPU control leads to analogous issues.

2.2 Mismatch Between Devices' Interface and Usage

Although modern peripheral devices are increasingly more capable of operating independently from the CPU, they still expose an interface designed for CPU control. This mismatch artificially

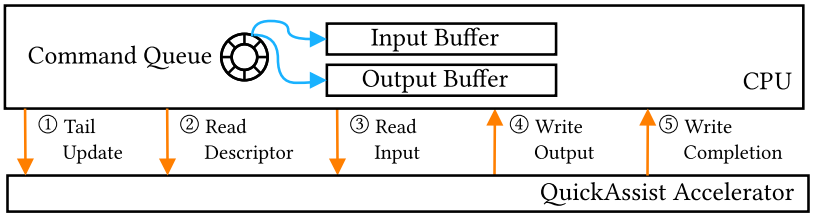


Figure 1: Steps to send an operation to an accelerator using a typical interface. Existing interfaces assume CPU control, making the CPU decide which commands to run and where to place every output.

increases the importance of the CPU when pushing data in and out of devices. Ultimately, this reliance on the CPU leads to performance and usability issues. In what follows, we highlight three main features of existing device interfaces that lead to this mismatch as well as the issues that they cause:

CPU-based routing: Existing interfaces assume peripheral devices are not in control of where to direct data. These data can be the output of a computation, in the case of accelerators; or data that arrived in the host, in the case of NICs. Instead, the other endpoint, typically the CPU, is responsible for determining where the device will direct the data. Requiring that the CPU decide the fate of every piece of data leads to two main issues:

Coordination overhead: Because routing decisions (i.e., where to direct the data) are made separately for every piece of data, this adds computation overhead to the CPU. Moreover, in order for the device to know where to direct the data, the CPU must send routing decisions over the interconnect, which wastes interconnect bandwidth with metadata. As we will see in §3.1.1, in the case of NICs, having an interface that is designed to facilitate routing on the CPU leads to up 39% of the PCIe bandwidth to be used with metadata alone. In contrast, when the NIC itself is in charge of making routing decisions, this overhead drops to around 1%.

Inefficient device chaining: Since the CPU determines the destination for the device's output, multiple devices cannot be chained together without CPU mediation. As a result, we need to use the CPU to orchestrate transfers between devices, even when data never need to go to the CPU.

Communication using shared memory addresses: As exemplified in §2.1, existing NIC and accelerator interfaces often expose a combination of command queues and data buffers to coordinate access to a shared memory. The CPU enqueues descriptors to the command queue that point to data buffers in arbitrary memory locations. Because memory locations are arbitrary, this leads to what we call *chaotic memory accesses*. Unpredictable accesses prevent devices from benefiting from optimizations such as prefetching, used to fetch the next data ahead of time. For instance, in the case of NICs communicating with the CPU, we see that unpredictable accesses cause up to 55% miss ratio in the CPU L2 cache. A similar problem manifests itself in accelerators, where the device can only fetch the data after receiving a descriptor, having to wait an extra PCIe RTT before it can access the data. Giving up the shared memory model in favor of more predictable accesses allow us to mask access latency and improve performance.

Incompatible data formats: The assumption of CPU control also places constraints on the data format output by the devices. Because the location where the device will write the output of a computation is controlled by the CPU, the CPU needs to predetermine the size of the output buffer. This is a problem when we do not know the size of the data beforehand. If the data end up being larger than the buffer, the device needs to split the data among multiple buffers, leading to fragmentation. Fragmentation amplifies the problem of chaotic memory accesses and adds overhead at the CPU needs to spend cycles recombining the data.

All of these issues arise from the existing interfaces exposed by peripheral devices. As we will see in the next sections, using streaming abstractions helps us overcome the above issues.

3 Ensō: A Streaming Interface for NIC-Application Communication

Network performance dictates application performance for many of today's distributed and cloud computing applications [53]. While growing application demands have led to a rapid increase in link speeds from 100 Mbps links [33] in 2003 to 100 Gbps in 2020 [97] and 200 Gbps in 2022 [64], 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. We begin with the observation that despite these changes, utilizing 100 Gbps or 400 Gbps links remains challenging. We demonstrate that this is because of inefficiencies in how software communicates with the NIC. While NICs and the software that communicate 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 where all communication between software and the NIC requires sending (and receiving) a sequence of fixed-size buffers, which we call *packet buffers* in this proposal. Packet buffer size is dictated by 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 [21] uses 2kB 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 Offloading (TSO) [22, 41], Large Receive Offloading (LRO) [17], serialization offloads [49, 79, 94], and transport offloads [4, 17, 28, 87] 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 IO 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

[1] Osiris [24], published in 1994, describes an interface that is nearly identical to the one adopted by many modern NICs.

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 [82, 96].

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 §3.3, chaotic memory accesses can significantly degrade application performance, particularly those that deal with small requests such as object caches [9, 63] and key-value stores [5, 57].

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 PCIe bottleneck in our implementation of Google's Maglev Load Balancer [25].

In this section, 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 moves away from 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 prac-

tice. 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*.

To understand its performance, we fully implement Ensō using an FPGA-based SmartNIC. In §3.3 we present our evaluation of Ensō, including its use in four applications: the Maglev load balancer [25], a network telemetry application based on NitroSketch [60], the MICA key-value store [57], and a log monitor inspired by AWS CloudWatch Logs [7]. We also implemented a software packet generator that we use in most of the experiments.[2] We observe speedups of up to 6× relative to a DPDK implementation for Maglev, and up to 1.47× 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.

Ensō is fully open source, with our hardware and software implementations available at `https://enso.cs.cmu.edu/`.

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 [3], Broadcom [13], Intel [41], Marvell [62], and others. Indeed, prior work [76] 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 [18], DPDK [21] and netmap [83]. 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 pre-allocate 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).

[2]Developing 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!

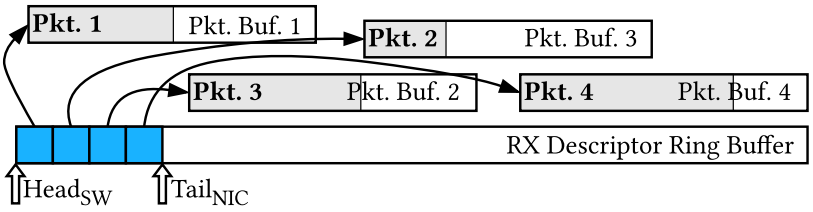


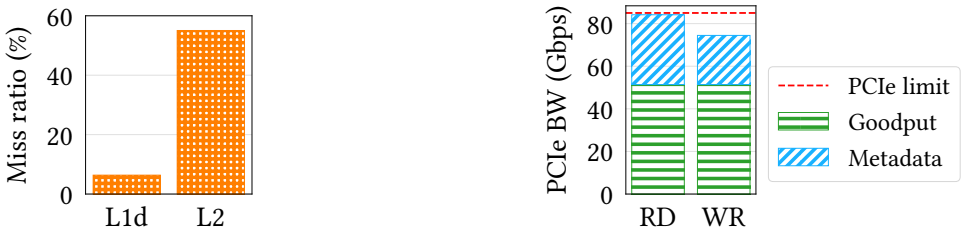
Figure 2: Data structures used to receive packets in a packetized NIC interface. Each packet is placed in a separate buffer that can be arranged arbitrarily in memory.

Figure 2 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 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 [37, 74] 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 [41]) 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 [91], Intel's Flow Director [41], or (for a very rich switching model) Microsoft's AccelNet [29]. In this setting, the multiplexing/demultiplexing capabilities of the packetized NIC interface offer no additional value.



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

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 §3.3, 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 §3.3) 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 [47], we see that the application is backend-bound, primarily due to L1 and L2 cache misses. Figure 3a 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 [5, 57] or packet processors [20] 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 [61, 90] has repeatedly demonstrated that the size of the working set for packet processing applications often outgrows the amount of cache space dedicated to DDIO [40], negating the benefits of this hardware optimization to bring I/O data directly into the cache. Using a different NIC interface that facilitates sequential memory accesses can drop

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

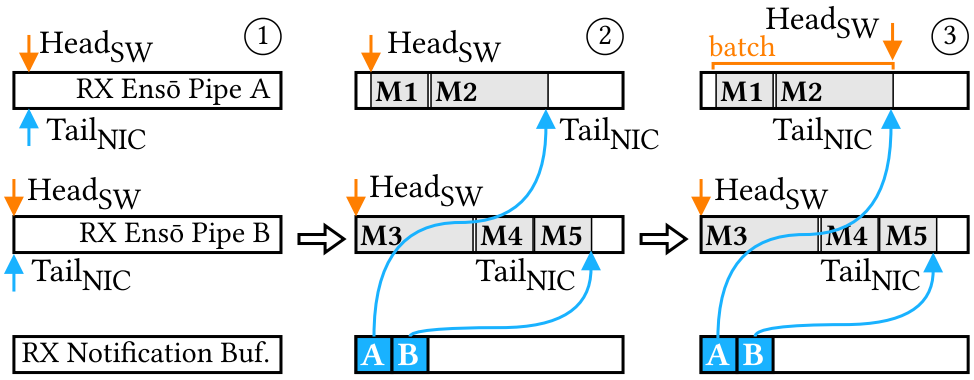


Figure 4: Steps to *receive* batches of messages in two Ensō Pipes.

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 problem with the packetized interface: *up to 39% of the CPU to NIC interconnect bandwidth is spent transferring descriptors* (Figure 3b). 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 [66]. 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 [2].

In summary: 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.

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.

High-level operation: In Figure 4 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 enqueuing 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.

Ensō Pipe's flexibility: Although Figure 4 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 (§3.3) we use this approach when implementing Maglev and a Network Telemetry application.

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 4, 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.

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.

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*, 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.

Target applications: Ensō implements a streaming interface that is optimized for cases where software processes received data in order. Our evaluation (§3.3) 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 [74] and BESS [37]), as such applications require additional copies with Ensō.[4] However, perhaps surprisingly, Ensō outperforms the packetized interface *even when it requires such additional copies*. 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 [15] (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).

3.3 Evaluation Summary

We now give a summary of our experiments. Refer to the full paper [85] for all the microbenchmarks and applications.

3.3.1 Setup and Methodology

Device Under Test (DUT): We synthesize and run the Ensō NIC on an Intel Stratix 10 MX FPGA NIC [46] 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 [42] 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 [44] 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 [43] 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 to the E810 and the FPGA on the DUT machine.

We found that existing high-performance packet generators such as DPDK Pktgen [93] and Moongen [26] 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

[4]Note 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.

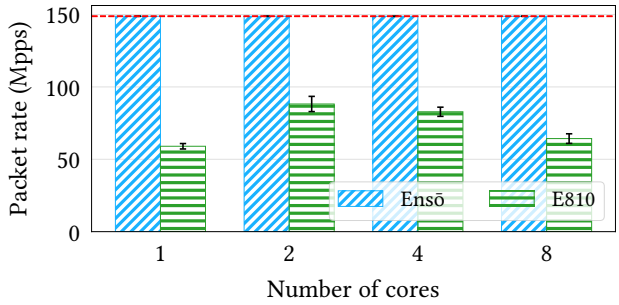


Figure 5: Raw packet rate. Ensō is bottlenecked by Ethernet while the E810 does not scale beyond two cores. The dashed line represents the 100 Gb Ethernet limit.

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 use EnsōGen in all experiments except for MICA, where we send requests from a MICA client.

Methodology: We measure zero-loss throughput as defined in RFC 2544 [12, 65] 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 [19] and we obtain other CPU counters using perf [72]. To evaluate MICA, we use the same methodology as the original paper [57] for consistency.

3.3.2 Microbenchmarks

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 5 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. 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.

Reactive Notifications and Latency: 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 6 shows the RTT (50th and 99th percentiles) as we increase load for both cases. While

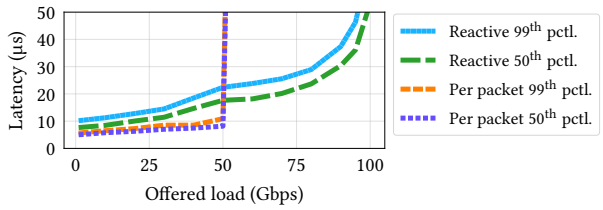


Figure 6: RTT for different loads when using a notification per packet or reactive notifications without notification prefetching.

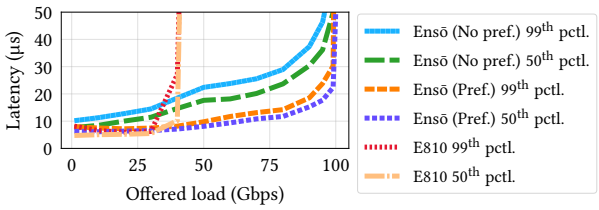


Figure 7: RTT for different loads for the E810 as well as Ensō with and without notification prefetching.

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.[5] 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 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 7 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.

3.3.3 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 [25], a network-telemetry application based on NitroSketch [60], MICA Key Value Store [57], and a log monitor inspired by AWS CloudWatch Logs [7]. 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.

Table 1 summarizes the performance we measure for all the applications that we evaluated when using a single CPU core. Note how Ensō improves application throughput by up to 6×. Besides throughput improvements, MICA also sees a reduction in latency of up to 43% when using Ensō.

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

Application	Ensō Throughput	E810 Throughput
Maglev Load Balancer	138 Mpps	23 Mpps
Network Telemetry with NitroSketch	121 Mpps	33.9 Mpps
MICA Key-Value Store	7.6 Mops	5.8 Mops
Log Monitor	2–100 Gbps	2–53.2 Gbps

Table 1: Throughput obtained when running different applications with a single CPU core when using Ensō or the E810 NIC with DPDK. Log monitor numbers are shown as ranges as they depend on the target application.

3.4 Related Work

Direct application access: While giving applications direct access to the NIC has been a common theme of research for more than three decades [8, 24, 27, 38, 56, 73, 83, 84, 89, 95, 96], 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 [24], 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.

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 [22] 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 [78]. 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 [90].

In terms of abstraction, U-Net [92] and, more recently, NICA [28] allow the NIC to exchange application-level messages directly. U-Net proposes a communication abstraction that resembles part of what is now `libibverbs` (RDMA) [58] 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.

Application-specific hardware optimizations: Prior work has optimized the NIC for specific applications. FlexNIC [55] quantifies the benefits that custom NIC interfaces could have to different applications. NIQ [30] 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 [77] 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 with it.

Application-specific software optimizations: Some proposals avoid part of the overheads of existing NICs with application-specific optimizations in software. TinyNF [76] 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 [52] 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 [23] 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).

4 Nagare: A Programmable Host Interconnect *(Proposed Work)*

We showed with Ensō how having the dataplane do data routing (demultiplexing) can greatly simplify the interface between the NIC and the applications, leading to better performance and flexibility. In this proposed work, we seek to apply some of the same principles that we used when designing Ensō to device communication within a host. Our primary goal is to enable direct communication among devices without CPU intervention.

In §2, we described how existing accelerator interfaces are designed for CPU control. As a result, direct communication between accelerators requires coordination from the CPU. Although there are proprietary techniques, such as GPUDirect [69], that allow direct communication between GPUs, NICs, and storage devices, they are not fundamentally different from the CPU-controlled interface. GPUDirect still uses a similar interface based on shared memory, simply moving the centralized control over the communication from the CPU to the GPU. As a result, GPUDirect still suffers from similar issues as the CPU-centered designs, presenting limited scalability, and not being able to chain multiple devices efficiently. Moreover, GPUDirect's proprietary nature means that it is only applicable to GPUs and not interoperable with other peripheral devices or even non-Nvidia GPUs.

We start by presenting a motivating example with a chain of accelerators and how existing accelerator interfaces prevent us from realizing it efficiently (§4.1), outlining the challenges that prevent direct device communication today (§4.2). We then briefly describe Nagare, our proposal to enable direct device communication using a programmable host interconnect (§4.3) and our plan to evaluate it (§4.4). Finally, we reflect on other use cases that are enabled by a programmable host interconnect (§4.5).

4.1 Motivating Example: Accelerator Chain

Consider the example in Figure 8 where we want to perform machine learning inference on an encrypted image received over the network. In this scenario, we aim for the NIC to send the encrypted image to a decryption accelerator (e.g., Intel QAT [48]), which will then forward the image to a machine learning accelerator (e.g., TPU [51]) to perform inference. Finally, the ML accelerator will send the inference result back to the NIC.

Today, the shared-memory model requires the CPU to mediate the transfers between every accelerator. As a result, data do not move directly between the accelerators in the chain, instead being directed back and forth through the CPU. Table 2 shows the PCIe operations required to implement the accelerator chain using the shared memory model. Note how the CPU is interposed in every transfer, sending commands and data to the device and receiving the output

Figure 8: Example of accelerator chain being used to implement machine learning inference on an encrypted image.

Direction	Data (PCIe Operation)	Latency
NIC → CPU	Encrypted Image (DMA WR)	1/2 PCIe RTT
CPU → QAT	Command Notification (MMIO WR)	1/2 PCIe RTT
CPU → QAT	Command (DMA RD)	1 PCIe RTT
CPU → QAT	Encrypted Image (DMA RD)	1 PCIe RTT
QAT → CPU	Decrypted Image (DMA WR)	1/2 PCIe RTT
CPU → ML Accel.	Command Notification (MMIO WR)	1/2 PCIe RTT
CPU → ML Accel.	Command (DMA RD)	1 PCIe RTT
CPU → ML Accel.	Decrypted Image (DMA RD)	1 PCIe RTT
ML Accel. → CPU	Inference Result (DMA WR)	1/2 PCIe RTT
CPU → NIC	Descriptor Notification (MMIO WR)	1/2 PCIe RTT
CPU → NIC	Descriptor (DMA RD)	1 PCIe RTT
CPU → NIC	Inference Result (DMA RD)	1 PCIe RTT
	Total Latency:	**9 PCIe RTT**

Table 2: Sequence of PCIe operations required to implement the chain in Figure 8 (NIC → QAT → ML Accel. → NIC) with existing CPU-controlled interfaces.

Direction	Data (PCIe Operation)	Latency
NIC → QAT	Encrypted Image (DMA WR)	1/2 PCIe RTT
QAT → ML Accel.	Decrypted Image (DMA WR)	1/2 PCIe RTT
ML Accel. → NIC	Inference Result (DMA WR)	1/2 PCIe RTT
	Total Latency:	**1.5 PCIe RTT**

Table 3: Sequence of PCIe operations ideally required to implement the chain in Figure 8 (NIC → QAT → ML Accel. → NIC).

of the computation back. Besides the overhead of inefficient routing due to CPU interposition, the CPU is also not able to efficiently push data to devices, requiring a combination of a CPU-initiated MMIO write and a device-initiated DMA read to move data to every device. These two factors combined result in the entire chain having a communication latency of 9 PCIe RTT.

If, instead of the CPU-controlled shared memory model, we let the devices communicate directly, we can reduce the total communication latency of the same example chain to 1.5 PCIe RTT (as shown in Table 3). To achieve this ideal communication latency, devices should be able to push data directly to each other without CPU mediation. In the following section, we discuss the challenges that prevent this from happening today and how an alternative based on streaming can help.

4.2 Challenges Preventing Direct Communication Between Devices

Although the PCIe standard supports peer-to-peer communication among PCIe devices, the shared memory model requires centralized coordination to decide *where* in memory to store the data and what functionality will run in the device. When we assume that both the input and output data reside in the host memory and will be accessed by the CPU, it makes sense to have the CPU decide where to place the data. But this model imposes many challenges when the

goal is to have the data traverse *different* devices. These challenges prevent us from achieving a design that removes the CPU from the datapath. Here, we elaborate on each of these challenges and argue how streaming can be used to address them:

Efficient data push: There are two main ways of moving data using the shared memory model: "pull" or "push." We can either have the sender transmit the address of the data to the receiver device and have the receiver device fetch the data from this address (we refer to this as "pull"); or have the receiver transmit a buffer address to the sender and have the sender device store the data at this location (we refer to this as "push"). None of these strategies is ideal. The pull strategy imposes an interconnect RTT as soon as we are ready to transfer the data. The push strategy can mask the interconnect RTT by sending buffers ahead of time but leads to fragmentation when we do not know the size of the data beforehand. Fragmentation adds overhead due to unpredictable memory accesses as well as the need for the receiver to recombine the data before processing. In addition, both strategies require that we exchange metadata over the interconnect to coordinate pointers, which, as we saw in §3, can consume a significant fraction of the interconnect bandwidth.

Solution – Streaming buffer: With streaming, we can implement data push without fragmentation or the need to coordinate every piece of data. The sender does not need to know the buffer address where it needs to send the data, it simply sends the data to a unified input buffer (streaming buffer) in the device. Because data is always pushed after the previous one in the same buffer, there is no need to coordinate a separate buffer for every piece of data or to fragment the data.

Policy-Based Routing: Routing decisions between accelerators can be complex. As such, having the CPU interposed in every transfer allows programmers to implement rich glue logic and forwarding decisions. To be able to remove the CPU from the datapath, we should be able to replicate similarly complex decisions without the CPU.

Solution – Streaming routing: With streaming, we can make complex routing decisions *per stream*, instead of for every piece of data. The CPU is only used when establishing a new stream, all subsequent data in the same stream is routed in the same way in the interconnect itself. This preserves the richness of CPU-based routing, without requiring the CPU to be interposed in every transfer. However, the ability to do routing on the dataplane, even if simple, requires architectural changes to the existing host interconnect.

Device Control: Besides deciding what accelerator to send an output, we also need to decide which commands should execute on the given accelerator and which state, if any, should be available to the device. With CPU control, the CPU is also responsible for sending the right command and state to the device.

Solution – Streaming state: Similarly to what we do when routing data, we can involve the CPU only once per stream. When establishing a stream, the CPU decides not only the routing, i.e., the sequence of accelerators that the data should traverse, but also the *functionality* that each of these devices should run. The dataplane will then store the sequence of commands that it should send to each accelerator for a particular stream. The data plane should also keep *state* associated with each stream that it can feed directly to accelerators with the corresponding data

and commands.

Compatibility: The last challenge in removing the CPU from the datapath is that each device today exposes a slightly different interface, with their own data formats and sequence of required transfers. Therefore, the role of the CPU extends beyond data routing and device control, being responsible for glueing the different device interfaces together. To replace the CPU, the dataplane should be flexible enough to allow different accelerators with slightly different interfaces to communicate.

Solution – Common streaming interface and dataplane programmability: Solving the compatibility issue requires two solutions. The first solution is to define a common streaming interface. This will allow future accelerators to be more easily interoperable without glue logic. The second solution is to allow programmability in the dataplane, allowing the dataplane to act differently depending on the devices that we need to connect. This lets existing devices enjoy some of the benefits of streaming without requiring hardware changes to the devices.

4.3 Achieving Streaming Through a Programmable Host Interconnect

Our proposal to enable efficient communication between peripheral devices is to make routing decisions directly on the dataplane, in the host interconnect itself. The idea is to have devices push data to the interconnect through streams and make the interconnect responsible for directing the data to the right accelerator based on user-defined policies. Operators can specify chains of accelerators using policy graphs, similar to those used to specify chains of network functions in NFV [50, 59, 70].

Our system named Nagare relies on a programmable PCIe switch to implement policy-based routing in the dataplane, relying only on the CPU to configure new streams. This switch exposes a stream abstraction to devices. Its programmable nature should also allow us to integrate existing peripheral devices that were not originally designed for streaming.

Architecture: We envision a PCIe switch architecture similar to RMT [11], where messages traverse multiple pipeline stages and each stage can perform simple instructions with a fixed number of cycles. These instructions can use message fields or registers as input or output. We can use this architecture to process incoming PCIe messages, converting them to a different kind or changing message fields, e.g., destination, depending on the desired functionality. The switch should also be able to issue new messages as needed.

This architecture can guarantee a fixed latency overhead that depends on the number of stages in the pipeline. For instance, if each stage of the pipeline requires 10 clock cycles, an architecture with 10 stages running at 3 GHz would add a fixed latency overhead of only 33 ns to existing PCIe switches. This is only a 9% increase in the one-way intra-host PCIe latency of 379 ns reported by previous work [39].

Push primitive: To allow devices to send data directly to each other, without CPU intervention, the switch exposes a push primitive. This primitive allows devices to send data directly to each other without having to be aware of the policy graph currently configured in the switch. To implement the push primitive, the switch dedicates an addresses in the host address space to each established stream. Devices simply need to send DMA writes to the corresponding address

to push data to the right stream. The switch can then look at the incoming PCIe messages and use the destination address to match the message to one of the open streams. Once the switch determines the stream, it is able to route the message to the appropriate device following the policy graph.

Working with existing accelerator interfaces: While the new push primitive allows new devices to adopt streaming, it has limited applicability today as it does not help existing accelerators. As discussed in §2.1, accelerators typically rely on a command queue stored in host memory to allow the CPU to submit commands, as well as pointers to input and output data. To be able to send data directly to existing devices, the switch needs to be able to use this interface.

Having a programmable PCIe switch allows us to bring streaming abstractions to existing devices, without requiring changes to the devices themselves. The switch can serve as a glue between different devices that expose different interfaces. Note that the switch operation required to interface with existing devices is not as complex as it may appear at first glance. That is because the switch only needs to perform dataplane operations (i.e., sending data and commands to the devices and routing the output to the next accelerator). Device configuration should still be done from the CPU.

4.4 Evaluation Plan

We plan to evaluate Nagare using a combination of cycle-accurate simulation and ASIC synthesis. Using cycle-accurate simulation helps us to understand the performance when integrating the programmable switch with existing open-source accelerators [14, 54, 68] and NICs [32, 85]. ASIC synthesis will let us know the maximum clock frequency achievable, as well as the cost of the design [86] (in terms of power and silicon die area).

Cycle-accurate simulator: We plan to use an existing PCIe simulation framework that implements different components of the PCIe fabric [31], including the root complex, switches, and devices. The framework also integrates with PCIe core implementations for both Intel and AMD (Xilinx) FPGAs. This simplifies integration with existing accelerator and NIC designs. Moreover, the framework is based on cocotb [16], which will allow us to quickly prototype different design options.

ASIC synthesis: We plan to implement only the programmable component of the Nagare switch in RTL. Then, we can synthesize the ASIC to understand the cost and performance of the additional programmable component, without the need to reimplement an entire PCIe switch. We have performed ASIC synthesis in a previous project [6] and plan to follow a similar methodology.

4.5 Beyond Accelerator Chaining

While our main goal in this project is to implement accelerator chaining, a programmable host interconnect enables many useful capabilities that extend beyond more efficient accelerator communication. In particular, the same programmable architecture can also be deployed in the PCIe root complex; this enables low-latency access to host memory and allows us to implement dataplane functionality interposed between the CPU and other devices (e.g., policies [84]).

Data push from CPU: Having programmability on the PCIe root complex also allows us to expose the data push primitive to the CPU itself. One of the reasons for the many round trips required by existing accelerator interfaces (e.g., Table 2) is that it is expensive for the CPU to push data to devices using MMIO writes. As a result, the software running on the CPU usually sends an MMIO write to inform the device that there are data available (doorbell). The device then needs to explicitly pull the data using a DMA read. By implementing our programmable architecture in the PCIe root complex, we can have the fabric itself send the data to the devices in response to an MMIO write, reducing latency without adding CPU overhead. In addition, the same push primitive can be used for efficient data copies, even when both the source and destination addresses reside in host memory.

Interposed dataplane policies: The fact that the PCIe fabric interposes between every accelerator allows us to implement policies such as access control efficiently. An operator may restrict which devices are reachable from each other. For instance, an operator may not want a particular device to be able to access the network or a storage device. Being able to implement these kinds of policies in the host dataplane also simplifies manageability as it is easier to reason about compared to enforcing the same policies from the devices themselves.

5 Thesis Timeline

Semester	Plan
Spring 2024	• Thesis proposal. • Complete speaking and writing skills. • Finish implementing Nagare in simulation.
Summer 2024	• Finish implementing Nagare in RTL. • Submit Nagare short paper to HotNets '24. • Nagare experiments.
Fall 2024	• Paper writing. • Submit Nagare to OSDI '25.
Spring 2025	• Work on camera ready or resubmission to SOSP '25.
Summer 2025	• Thesis writing. • Thesis defense.

Table 4: Proposed timeline to finish thesis.

References

[1] 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 978-1-72814-661-4. §1

[2] Saksham Agarwal, Rachit Agarwal, Behnam Montazeri, Masoud Moshref, Khaled Elmeleegy, 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. ISBN 978-1-4503-9899-2. §3.1.2

[3] Amazon. DPDK driver for elastic network adapter (ENA). `https://github.com/amzn/amzn-drivers/tree/master/userspace/dpdk`, 2022. §3.1

[4] 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. ISBN 978-1-939133-13-7. §1 , §3

[5] 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. ISBN 978-1-4503-1097-0. §3 , §3.1.2

[6] 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, Santa Clara, CA, April 2024. USENIX Association. §4.4

[7] AWS. What is Amazon CloudWatch Logs?, 2022. `https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/WhatIsCloudWatchLogs.html`. §3 , §3.3.3

[8] 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. ISBN 978-1-931971-16-4. §3.4

[9] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, 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, November 2020. ISBN 978-1-939133-19-9. §3

[10] Corinne Bernstein. What is ISA (industry standard architecture)?, 2024. https://www.techtarget.com/searchwindowsserver/definition/ISA-Industry-Standard-Architecture. §1

[11] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 99–110, New York, NY, USA, 2013. ISBN 978-1-4503-2056-6. §4.3

[12] S. Bradner and J. McQuaid. Benchmarking methodology for network interconnect devices. RFC 2544, March 1999. URL https://www.rfc-editor.org/info/rfc2544. §3.3.1

[13] Broadcom. BNXT poll mode driver. https://doc.dpdk.org/guides/nics/bnxt.html, 2022. §3.1

[14] Jeff Bush et al. Nyuzi Processor, 2024. https://github.com/jbush001/NyuziProcessor. §4.4

[15] CAIDA. Anonymized internet traces 2016. https://catalog.caida.org/dataset/passive_2016_pcap, 2016. §3.2

[16] cocotb. cocotb | Python verification framework, 2024. https://www.cocotb.org/. §4.4

[17] Chelsio Communications. Terminator 5 ASIC, 2021. https://www.chelsio.com/terminator-5-asic/. §1 , §3

[18] Jonathan Corbet. Ringing in a new asynchronous I/O API, 2019. https://lwn.net/Articles/776703/. §3.1

[19] Roman Dementiev et al. Processor Counter Monitor (PCM), 2022. https://github.com/opcm/pcm. §3.3.1

[20] 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. ISBN 978-1-60558-752-3. §3.1.2

[21] DPDK. Data Plane Development Kit, 2022. https://dpdk.org. §3 , §3.1

[22] DPDK. NVIDIA MLX5 ethernet driver, 2022. https://doc.dpdk.org/guides/nics/mlx5.html. §1 , §3 , §3.4

[23] 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. ISBN 978-1-931971-09-6. §3.4

[24] 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. ISBN 0-89791-682-4. §1 , §3.4

[25] 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. ISBN 978-1-931971-29-4. §3 , §3.3.3

[26] 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. ISBN 978-1-4503-3848-6. §3.3.1

[27] D. R. Engler, M. F. Kaashoek, and J. 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. ISBN 0-89791-715-4. §3.4

[28] 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. ISBN 978-1-939133-03-8. §1 , §3 , §3.4

[29] 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. ISBN 978-1-939133-01-4. §3.1.1

[30] 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. ISBN 978-1-931971-01-0. §3.4

[31] Alex Forencich. PCI express simulation framework for Cocotb, 2024. `https://github.com/alexforencich/cocotbext-pcie`. §4.4

[32] Alex Forencich, Alex C. Snoeren, George Porter, and George Papen. Corundum: An open-source 100-Gbps NIC. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines*, FCCM '20, pages 38–46. IEEE, 2020. ISBN 978-1-72815-803-7. §4.4

[33] 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. ISBN 1-58113-757-5. §3

[34] 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 9798400700958. §1

[35] PCI Special Interest Group. PCI Local Bus Specification (Revision 2.2). Technical report, PCI Special Interest Group, December 1998. §1

[36] PCI Special Interest Group. PCI Express Base Specification Revision 1.0. Technical report, PCI Special Interest Group, April 2002. §1

[37] 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. §3.1.1 , §3.2

[38] 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. ISBN 978-1-4503-6080-7. §3.4

[39] Wentao Hou, Jie Zhang, Zeke Wang, and Ming Liu. Understanding routable PCIe performance for composable infrastructures. In *21st USENIX Symposium on Networked Systems Design and Implementation*, NSDI '24, Santa Clara, CA, April 2024. USENIX Association. §4.3

[40] Intel. Intel data direct I/O technology (Intel DDIO): A primer. Technical report, Intel, February 2012. §3.1.2

[41] Intel. Intel Ethernet Controller E810. Technical Report 613875-006, Intel, March 2022. §1 , §3 , §3.1 , §3.1.1

[42] Intel. Intel Ethernet Network Adapter E810-CQDA2, 2022. `https://ark.intel.com/content/www/us/en/ark/products/210969/intel-ethernet-network-adapter-e8102cqda2.html`. §3.3.1

[43] Intel. Intel Core i7-7820X X-series Processor, 2022. `https://ark.intel.com/content/www/us/en/ark/products/123767/intel-core-i77820x-xseries-processor-11m-cache-up-to-4-30-ghz.html`. §3.3.1

[44] Intel. Intel Core i9-9960X X-series Processor, 2022. `https://ark.intel.com/content/www/us/en/ark/products/189123/intel-core-i99960x-xseries-processor-22m-cache-up-to-4-50-ghz.html`. §3.3.1

[45] Intel. Intel infrastructure processing unit (Intel IPU) platform (codename: Oak Springs Canyon), 2022. `https://www.intel.com/content/www/us/en/products/platforms/details/oak-springs-canyon.html`. §1

[46] Intel. Intel Stratix 10 MX 2100 FPGA, 2022. `https://ark.intel.com/content/www/us/en/ark/products/210297/intel-stratix-10-mx-2100-fpga`

.html. §3.3.1

[47] 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, 2022. §3.1.2

[48] Intel. Intel QuickAssist Adapter Family for Servers, 2024. https://www.intel.com/content/www/us/en/products/docs/network-io/ethernet/10-25-40-gigabit-adapters/quickassist-adapter-for-servers.html. §1 , §2.1 , §4.1

[49] 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 978-1-72814-661-4. §3

[50] Dilip A. Joseph, Arsalan Tavakoli, and Ion Stoica. A policy-aware switching layer for data centers. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, pages 51–62, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 978-1-60558-175-0. §4.3

[51] 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, aug 2018. ISSN 0001-0782. doi: 10.1145/3154484. §1 , §4.1

[52] 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. ISBN 978-1-931971-49-2. §3.4

[53] 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. ISBN 978-1-4503-3402-0. §3

[54] 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. §1 , §4.4

[55] 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. ISBN 978-1-4503-4091-5. §3.4

[56] 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. ISBN 978-1-

4503-6281-8. §3.4

[57] 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. ISBN 978-1-931971-09-6. §3 , §3.1.2 , §3.3.1 , §3.3.3

[58] Linux RDMA. RDMA core userspace libraries and daemons, 2022. `https://github.com/linux-rdma/rdma-core`. §3.4

[59] 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. §4.3

[60] 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. ISBN 978-1-4503-5956-6. §3 , §3.3.3

[61] 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. ISBN 978-1-4503-7955-7. §3.1.2

[62] Marvell. DPDK marvell. `https://github.com/MarvellEmbeddedProcessors/marvell-dpdk`, 2024. §3.1

[63] 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. ISBN 978-1-4503-8709-5. §3

[64] 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. ISBN 978-1-4503-9420-8. §3

[65] Al Morton. RFC Errata, Erratum ID 412, RFC 2544, November 2006. `https://www.rfc-editor.org/errata/eid422`. §3.3.1

[66] 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. ISBN 978-1-4503-5567-4. §3.1.2

[67] Nvdia. Nvidia Mellanox Innova-2 Flex open adapter card, 2024. https://docs.nvidia.com/networking/display/innova2flex. §1

[68] Nvdia. Nvidia deep learning accelerator, 2024. http://nvdla.org/. §4.4

[69] Nvidia. GPUDirect, 2024. https://developer.nvidia.com/gpudirect. §4

[70] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. E2: A framework for NFV applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 121–136, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 978-1-4503-3834-9. §4.3

[71] Pensando. Pensando DSC-100 distributed services card. Technical Report PPB19002. Rev 6, Pensando, 2021. §1

[72] Perf. perf: Linux profiling with performance counters, 2022. https://perf.wiki.kernel.org. §3.3.1

[73] 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. ISBN 978-1-931971-16-4. §3.4

[74] 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. ISBN 978-1-931971-21-8. §3.1.1 , §3.2

[75] 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. §1

[76] 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, November 2020. ISBN 978-1-939133-19-9. §3.1 , §3.4

[77] Boris Pismenny, Liran Liss, Adam Morrison, and Dan Tsafrir. 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, New York, NY, USA, 2022. §3.4

[78] Boris Pismenny, Adam Morrison, and Dan Tsafrir. ShRing: Networking with shared receive rings. In *17th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '23, Boston, MA, July 2023. §3.4

[79] 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. ISBN 978-1-4503-7102-5. §3

[80] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis,

Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Plasticine: A reconfigurable architecture for parallel paterns. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 389–402, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 978-1-4503-4892-8. §1

[81] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 13–24, Minneapolis, Minnesota, USA, 2014. IEEE Press. ISBN 978-1-4799-4394-4. §1

[82] 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. ISBN 978-1-4503-8438-4. §3

[83] 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. ISBN 978-931971-93-5. §3.1 , §3.4

[84] 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. ISBN 978-1-4503-8438-4. §3.4 , §4.5

[85] 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. §3.3 , §4.4

[86] 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 9798400704154. §4.4

[87] 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. ISBN 978-1-939133-27-4. §1 , §3

[88] 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. ISBN 978-1-4503-7102-5. §1

[89] 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. ISBN 0-89791-619-0. §3.4

[90] 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. ISBN 978-1-939133-01-4. §3.1.2 , §3.4

[91] Amy Viviano. Introduction to receive side scaling, 2022. `https://docs.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling`. §3.1.1

[92] 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. ISBN 0-89791-715-4. §3.4

[93] Keith Wiles et al. The Pktgen application, 2022. `https://pktgen-dpdk.readthedocs.io/`. §3.3.1

[94] 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. ISBN 978-1-4503-8438-4. §3

[95] 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. ISBN 978-1-931971-30-0. §3.4

[96] 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. ISBN 978-1-4503-8709-5. §3 , §3.4

[97] 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. ISBN 978-1-4503-7102-5. §3