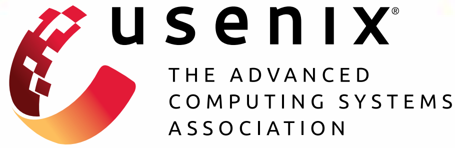


Ensō : A Streaming Interface for NIC-Application Communication

Hugo Sadok and Nirav Atre, *Carnegie Mellon University;* Zhipeng Zhao, *Microsoft;* Daniel S. Berger, *Microsoft Research and University of Washington;* James C. Hoe, *Carnegie Mellon University;* Aurojit Panda, *New York University;* Justine Sherry, *Carnegie Mellon University;* Ren Wang, *Intel*

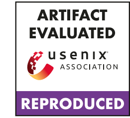
https://www.usenix.org/conference/osdi23/presentation/sadok

This paper is included in the Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-34-2

Open access to the Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation is sponsored by



Ensō: A Streaming Interface for NIC-Application Communication

Hugo Sadok, Nirav Atre, Zhipeng Zhao, Daniel S. Berger
James C. Hoe, Aurojit Panda, Justine Sherry, Ren Wang
Carnegie Mellon University *Intel* *Microsoft* *New York University* *University of Washington*

Abstract

Today, most communication between the NIC and software involves exchanging fixed-size packet buffers. This packetized interface was designed for an era when NICs implemented few offloads and software implemented the logic for translating between application data and packets. However, both NICs and networked software have evolved: modern NICs implement hardware offloads, e.g., TSO, LRO, and serialization offloads that can more efficiently translate between application data and packets. Furthermore, modern software increasingly batches network I/O to reduce overheads. These changes have led to a mismatch between the packetized interface, which assumes that the NIC and software exchange fixed-size buffers, and the features provided by modern NICs and used by modern software. This incongruence between interface and data adds software complexity and I/O overheads, which in turn limits communication performance.

This paper proposes Ensō, a new streaming NIC-to-software interface designed to better support how NICs and software interact today. At its core, Ensō eschews fixed-size buffers, and instead structures communication as a stream that can be used to send arbitrary data sizes. We show that this change reduces software overheads, reduces PCIe bandwidth requirements, and leads to fewer cache misses. These improvements allow an Ensō-based NIC to saturate a 100 Gbps link with minimum-sized packets (forwarding at 148.8 Mpps) using a single core, improve throughput for high-performance network applications by 1.5–6×, *and reduce latency by up to 43%.*

1 Introduction

Network performance dictates application performance for many of today's distributed and cloud computing applications [48]. While growing application demands have led to a rapid increase in link speeds from 100 Mbps links [31] in 2003 to 100 Gbps in 2020 [89] and 200 Gbps in 2022 [58], 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 paper. 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 [19] 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) [20, 39], Large Receive Offloading (LRO) [14], serialization offloads [44, 71, 86], and transport offloads [3, 14, 27, 77] 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 [22], 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 [72, 88].

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 §7, chaotic memory accesses can significantly degrade application performance, particularly those that deal with small requests such as object caches [9, 57] and key-value stores [4, 52].

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 [23].

In this paper, we propose 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 practice. For example, CPU-NIC synchronization can easily lead to poor cache performance: any approach where the NIC and CPU poll for changes at a particular memory location will lead to frequent cache invalidation. Ensō avoids this obstacle by relying on explicit notifications for CPU-NIC synchronization. Unfortunately, explicit notifications require additional metadata to be sent over the CPU-NIC interconnect, which can negate any benefits for interconnect bandwidth utilization. Ensō mitigates this overhead by sending notifications *reactively*. We discuss our synchronization strategy in detail, as well as other challenges to the Ensō design in §4.

To understand its performance, we fully implement Ensō using an FPGA-based SmartNIC. We describe our hardware and software implementations in §5 and how Ensō can be used depending on the functionality offered by the NIC in §6. In §7 we present our evaluation of Ensō, including its use in four applications: the Maglev load balancer [23], a network telemetry application based on NitroSketch [54], the MICA key-value store [52], and a log monitor inspired by AWS CloudWatch Logs [6]. 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 (§7.2.2).

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

2 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 [2], Broadcom [12], Intel [39], Marvell [56], and others. Indeed, prior

[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!

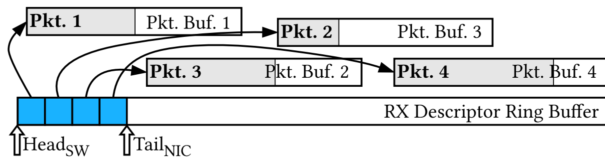


Figure 1: Data structures used to receive packets in a packetized NIC interface. Each packet is placed in a separate buffer that can be arranged arbitrarily in memory.

work [68] 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 [15], DPDK [19] and netmap [73]. In this section, we describe the packetized NIC interface and highlight some of the issues that it brings to high-performance applications.

2.1 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).

Figure 1 shows an example of a packetized NIC interface being used to receive four packets from a particular hardware queue on the NIC. The NIC queue is associated with a set of NIC registers that can be used to control a receive (RX) descriptor ring buffer and a transmit (TX) descriptor ring buffer. Before being able to receive packets, the software informs the NIC of the addresses of multiple available buffers in its pool by enqueueing descriptors pointing to each one in the RX descriptor ring buffer. The NIC can then use DMA to write the incoming packet 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 [32, 67] 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.

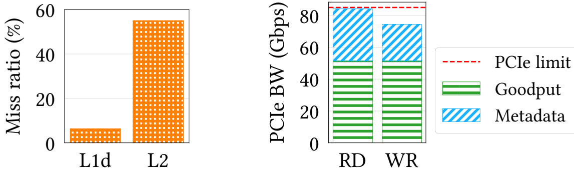
The usage model for a modern high-performance software stack, however, looks very different. Instead of *one* software entity (e.g., kernel, software switch) mediating access to the NIC, there may be many threads or processes with *direct NIC access* (i.e., kernel bypass). High-performance NIC ASICs expose *multiple hardware queues* (as many as thousands [39]) 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 [82], Intel's Flow Director [39], or (for a very rich switching model) Microsoft's AccelNet [28]. In this setting, the multiplexing/demultiplexing capabilities of the packetized NIC interface offer no additional value.

2.2 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 §7, we also show the impact that these issues have on real applications.

Chaotic Memory Access: We experiment with a simple DPDK-based ping/pong program (a description of our testbed is in §7) which receives a packet, increments a byte in the packet payload, and re-transmits it. For this program, we observed maximum throughput of 40 Gbps using a 100 Gb NIC (Intel E810) and a single 3.1 GHz CPU core. When we conduct a top-down analysis [43], we see that the application is backend-bound, primarily due to L1 and L2 cache misses. Figure 2a shows around 6% miss ratio for the L1d and a 55% miss ratio for the L2 cache. This high cache miss ratio is a direct consequence of using per-packet buffers in the packetized NIC interface. First, because *packet buffers themselves are scattered in memory, reads and writes to packet data evade*



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

any potential benefit from shared cache lines or prefetching.[3] Applications like key-value stores [4, 52] or packet processors [18] 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 [55, 81] has repeatedly demonstrated that the size of the working set for packet processing applications often outgrows the amount of cache space dedicated to DDIO [35], negating the benefits of this hardware optimization to bring I/O data directly into the cache. As we discuss in detail in §7.2.3, using a different NIC interface that facilitates sequential memory accesses can drop the miss ratio from 6% to 0.2% for the L1d cache, and from 55% to 9% for the L2 cache.

Metadata Bandwidth Overhead: We observe that the packetized NIC interface requires the CPU and the NIC to exchange *both* descriptors and packet buffers. This leads to the second problem with the packetized interface: *up to 39% of the CPU to NIC interconnect bandwidth is spent transferring descriptors* (Figure 2b). While NIC-CPU interconnect line rates are typically higher than network line rates, the gap between them is relatively narrow. This is particularly problematic for small transfers as the PCIe theoretical limit drops to only 85 Gbps with 64-byte transfers [62]. 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 [1].

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

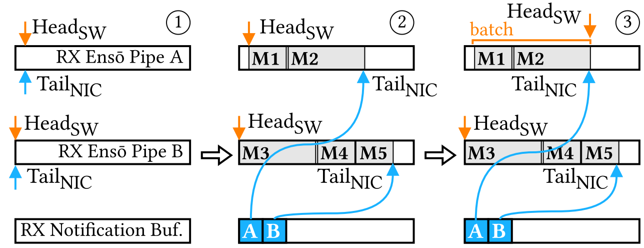


Figure 3: Steps to *receive* batches of messages in two Ensō Pipes.

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 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 3 we show how an application, with two Ensō Pipes, receives messages. Initially, the Ensō Pipes are empty, and the $Head_{SW}$ and $Tail_{NIC}$ point to the same location in the buffer ①. When the NIC receives messages, it uses DMA to enqueue them in contiguous memory owned by the Ensō Pipes ②. In the figure, the NIC enqueues two messages in Ensō Pipe A's memory, and three in Ensō Pipe B's memory. The NIC informs the software about this by also 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 3 shows the steps to send *messages*, because Ensō Pipes are opaque, they can be used to transmit arbitrary chunks of data. These can be raw packets, messages composed of multiple MTU-sized packets, or even an unbounded bytestream. The format of the data is dictated by the application and the offloads running on the NIC. Moreover, Ensō Pipes' opaqueness means that they can be mapped to any pinned memory within the application's memory space. Thus, by mapping both the RX and TX Ensō Pipes to the same region, network functions and other forwarding applications can avoid copying packets. In our evaluation (§7) 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 3, inlining data in an Ensō Pipe removes the need for per-packet descriptors, thus reducing the amount of metadata exchanged over the PCIe bus, and reducing cycles spent managing (i.e., allocating and freeing) packet buffers.

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

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

In §4.1, we discuss how naïve approaches to notification can stress worst-case performance of MMIO and DMA. In particular, concurrent accesses to the same memory address can create cache contention between the CPU and the NIC. Ensō uses dedicated *notification buffers* to synchronize updates to head and tail pointers; when combined with batching and multiqueue processing, the notification buffer approach reduces the threat of cache contention.

Notification pacing: Ensō Pipes are designed so that notifications for multiple packets can be combined, reducing the amount of metadata transferred between the CPU and the NIC. However, it is still important to decide *when* to send notifications: when sent too frequently they waste PCIe bandwidth and add software overheads, but if sent too infrequently the core might be idle waiting for notification, thus reducing throughput. Ensō includes two mechanisms, *reactive notifications* and *notification prefetching* (§4.2), that control when notifications are sent. These mechanisms are naturally adaptive, i.e., they minimize the number of notifications sent without limiting throughput, and can be implemented without adding hardware complexity.

Low hardware complexity and state: Because the design of Ensō involves both hardware and software, we must be careful to not pay for software simplicity with hardware complexity. Ensō favors coordination mechanisms that require little NIC state. We aim for a design that is simple and easily parallelized. We present Ensō's hardware design in §5.

Target applications: Ensō implements a streaming interface that is optimized for cases where software processes received data in order. Our evaluation (§7) shows that this covers a wide range of network-intensive applications.

One might expect that the resulting design is ill-suited for applications that need to multiplex and demultiplex packets (e.g., virtual switches like Open vSwitch [67] and BESS [32]), as such applications require additional copies with Ensō.[4] However, perhaps surprisingly, Ensō outperforms the packetized interface *even when it requires such additional copies*. As we show in §7.2.2, 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 [13] (average packet size of 462 B), we find that Ensō's throughput is still 28% higher than DPDK's (92.6 Gbps vs. 72.6 Gbps). We discuss how Ensō can be used depending on the applications and the functionality offered by the NIC in §6.

4 Efficient Notifications

The key challenges in Ensō arise from efficiently coordinating Ensō Pipes between the CPU and the NIC. In this section, we describe how Ensō efficiently coordinates pipes using notifications (§4.1) and how it paces such notifications (§4.2).

4.1 Efficient Ensō Pipe Coordination

Recall from Figure 3 that the software and the NIC coordinate access to RX Ensō Pipes using Head_{SW} and Tail_{NIC} and to TX Ensō Pipes using Head_{NIC} and Tail_{SW}. How should software and the NIC communicate pointer updates?

In the descriptor ring buffers employed by the packetized NIC interface, software communicates pointer updates to the NIC using MMIO writes, and the NIC communicates pointer updates via inline signaling in the descriptor buffer itself [25, 39], avoiding the overheads of MMIO reads. Because the descriptor's format is defined by the NIC, the NIC

[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.

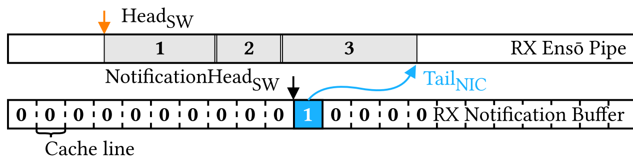


Figure 4: Notification buffer being used to notify the arrival of three chunks of data in an RX Ensō Pipe. Notifications contain the latest Tail_{NIC} for a given Ensō Pipe as well as a flag signal.

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

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

4.1.1 Notification Buffer

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

Figure 4 shows an RX Ensō Pipe with its corresponding notification buffer. It shows how a single notification indicates the presence of multiple sequential chunks of data at the same time; we discuss how notifications are 'batched' or coalesced in §4.2. Notifications contain the latest Tail_{NIC} for a given RX Ensō Pipe as well as a flag signal that software can use to check if the next notification is ready to be consumed. As is done typically with descriptor ring buffers, software advances the $\text{NotificationHead}_{SW}$ using an MMIO write after consuming a notification. Like an RX Ensō Pipe, a TX Ensō Pipe also uses a separate notification buffer to synchronize pointer updates. But software enqueues new notifications when it wants to transmit a chunk of data and the NIC overwrites the transmission notification with a completion notification once it is done transmitting the corresponding batch. Completions flip a flag signal, so that software can check if the following cache line corresponds to a completion or a pending TX notification.

4.1.2 Multiplexing and Scaling

Within a single thread: To let a single thread efficiently access multiple Ensō Pipes, we associate multiple Ensō Pipes with the same notification buffer. To accomplish this, notifications include an *Ensō Pipe ID* alongside the Tail_{NIC} and the flag signal that we discussed before. As a result, software can probe a *single* notification buffer to retrieve updates from *multiple* Ensō Pipes. This avoids the known scalability issues from needing to poll multiple queues [59, 76].

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

Among multiple applications: In addition to using independent notification buffers, Ensō ensures that applications only have access to their own subset of Ensō Pipes and notification buffers. Each queue's MMIO pointer register pair is kept in its own dedicated page-aligned block of memory [22]. This lets the kernel map the pointer registers at a per-queue granularity to the address space of the application that requested it.

4.1.3 Notifications: Contention and Overhead

Allowing multiple Ensō Pipes to share the same notification queue (§4.1.2), and having notifications arrive only for larger batches of data (§4.2) naturally prevents contention by keeping the NIC 'ahead' of the CPU in updating the notification buffer, and also reduces the PCIe overhead of communicating these notifications. As the CPU reads in data for one Ensō Pipe, the NIC is writing new entries for subsequent Ensō Pipes. Because the CPU is processing larger batches of data, it is busier for longer before it needs to check the notification buffer. Hence, as line rates go up, the two are unlikely to be accessing the same cache line simultaneously.

4.2 Pacing Notifications

As mentioned above, Ensō batches notifications aiming to reduce metadata bandwidth consumption and to keep CPUs busy processing data, rather than waiting for notifications. Using the wrong batch size, however, is problematic: a system that uses batch sizes that are too small would unnecessarily transmit extra metadata, and a system that uses batch

```
func onPktArrival()            func onRxUpdate(Head_SW)
    if status = 0 then             if Head_SW = Tail_NIC then
        notify (Tail_NIC)              status ← 0
        status ← 1                 else
                                       notify (Tail_NIC)
```

Figure 5: Reactive notification mechanism for an Ensō Pipe. It only sends notifications when new data arrives and status = 0, or if software updates $Head_{SW}$ and it is different from $Tail_{NIC}$.

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

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

Operation: Figure 5 summarizes the reactive notification mechanism. Whenever data arrive at an RX Ensō Pipe (`onPktArrival`), we check the status bit, only sending a notification if status = 0 (indicating that the buffer is empty). We also potentially send notifications whenever software updates $Head_{SW}$ (`onRxUpdate`). If the new $Head_{SW}$ is the same as $Tail_{NIC}$, it means that the buffer is now empty and we can reset the status back to '0' without sending a new notification. Otherwise, it indicates that software is unaware of some of the latest data in the buffer, which triggers a new notification.

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

Reactive notifications ensure that every piece of data is notified, but as we will see in §7.2.5, they can impose a small latency overhead. This overhead occurs if packet arrivals are known to the NIC but have not yet been communicated in the notification buffer. In this case, when $Head_{SW}$ reaches $Tail_{NIC}$, software has to wait for a PCIe RTT before it is notified of the waiting packets. While an extra PCIe RTT is unlikely to be an issue for Internet-facing applications, it might be an issue for some latency-sensitive applications [7].

Notification prefetching: To improve latency for latency-sensitive applications, Ensō also implements a notification prefetching mechanism. Notification prefetching allows software to explicitly request a new notification to the NIC. Applications can use notification prefetching either explicitly, by calling a function to prefetch notifications for a given Ensō Pipe, or transparently, by compiling the Ensō library in low-latency mode. When compiled in low-latency mode, the library always prefetches notifications for the next Ensō Pipe before returning data for the current one. We evaluate the impact of using notification prefetching in §7.2.5.

5 Ensō Implementation

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

5.1 Software Implementation

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

To ensure a consistent abstraction of unbounded Ensō Pipes we must also deal with corner cases that arise when data wrap around the buffer limit. To prevent breaking received data that wrap around, we map the same Ensō Pipe's physical address twice in the application's virtual memory address space. This means that, to the application, the buffer appears to be twice its actual size, with the second half always mirroring the first one. The application can then consume up to the full size of the buffer of data at once, regardless of where in the buffer the current $Head_{SW}$ is. To transmit data that wrap around the buffer limit, the library checks if

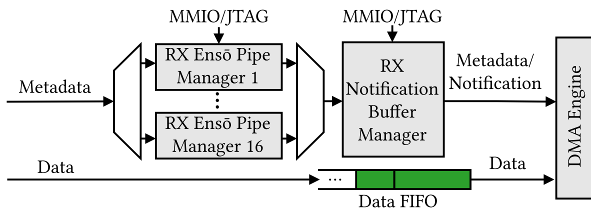


Figure 6: NIC RX Datapath.

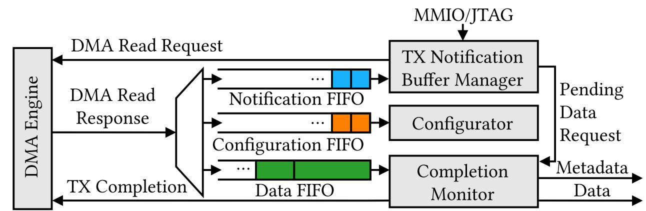


Figure 7: NIC TX Datapath.

the transfer goes around the Ensō Pipe boundary and automatically partitions it into several transfers, each of which is smaller than the overall buffer size.

5.2 Hardware Implementation

We now describe key RX and TX hardware modules.

5.2.1 RX Datapath

Figure 6 illustrates the RX datapath. It receives as input data and metadata, which includes the Ensō Pipe ID and the size of the corresponding data that is being enqueued. Metadata is handled separately from the data, which allows for smaller queues between modules. The RX datapath is composed of the following modules:

RX Ensō Pipe Managers: The Ensō Pipe managers are responsible for keeping Ensō Pipe state such as the buffer's physical address, Head_{SW}, Tail_{NIC}, notification buffer ID, and notification status bit. When metadata arrive for Ensō Pipe i, the manager checks for sufficient space in i and advances i's Tail_{NIC}. The manager also determines whether to trigger a notification according to the reactive notification strategy (§4.2). To trigger a notification, the manager includes the notification buffer ID and sets a bit in the metadata that is sent to the Notification Buffer Manager.

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

RX Notification Buffer Manager: This module issues notifications when needed. It spends one cycle for every request and an extra cycle for those that require a notification. If we can suppress notifications for at least 20% of requests, we only need a single notification manager at a 250 MHz clock. A single notification manager is also sufficient since we use fewer notification buffers than Ensō Pipes, e.g., one notification buffer per CPU core. Our implementation defaults to 128 notification buffers but also meets timing with 1024.

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

5.2.2 TX Datapath and Configuration Path

The TX datapath (Figure 7) is composed of:

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

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

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

6 Using Ensō

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

How one uses Ensō depends on how features are split between hardware and software. We consider three settings: (1) Traditional NICs which implement simple offloads, such as checksum and RSS [82], and rely on software implementation for the rest. These can use Ensō Pipes to deliver raw

packets to/from a network stack implemented in software. (2) NICs that implement transport offloads [10, 14, 77, 78] in hardware. These can deliver application-level messages or reassembled bytestreams through the Ensō Pipes. And finally, (3) NICs that implement application logic [41, 49], which can use Ensō Pipes to exchange application data. We elaborate on each of these settings below.

Traditional NICs: For traditional NICs that perform simple offloads such as checksum and segmentation, using Ensō is not significantly different than using a packetized interface. In both cases, the network stack is implemented in software and only needs to be changed to use Ensō Pipes. Ensō is designed to support several Ensō Pipes, and for most applications this change does not induce any additional software overheads. Furthermore, high-performance packet processing applications that process packets in order, as is the case for most network functions [68] and applications that use UDP, can consume and transmit raw packets through an Ensō Pipe without copies.

However, applications such as virtual switches [32, 67], that multiplex and demultiplex packets (but do not perform reassembly or other functions that require re-ordering packets, where both interfaces require copies), need to perform an additional copy when forwarding packets to their destination. As we show in §7.2.2, Ensō's performance advantages can outweigh the cost of copies even for such applications.

NICs with transport offload: NICs that implement message-based transports (e.g., SCTP, Homa [60]) or streaming-based transports (e.g., TCP) may also choose to use Ensō Pipes to deliver messages or reassembled bytestreams directly to the application without copies.

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

7 Evaluation

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

7.1 Setup and Methodology

Device Under Test (DUT): We synthesize and run the Ensō NIC on an Intel Stratix 10 MX FPGA NIC [42] 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 [40] 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 [38] 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.

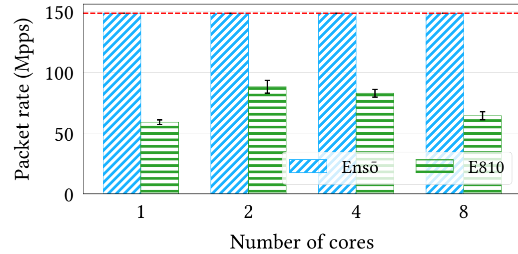


Figure 8: Raw packet rate. Ensō is bottlenecked by Ethernet while the E810 does not scale beyond two cores. The dashed line represents the 100 Gb Ethernet limit.

Packet generator: The packet generator machine is equipped with an Intel Core i7-7820X CPU [37] with 8 cores running at 3.6 GHz base frequency, 11 MB of LLC, and PCIe 3.0. It includes another Stratix 10 MX FPGA connected back to back to the E810 and the FPGA on the DUT machine.

We found that existing high-performance packet generators such as DPDK Pktgen [85] and Moongen [24] are unable to keep up with Ensō's packet rate because their performance is limited by the packetized NIC interface. We thus implement EnsōGen, a packet generator based on Ensō. EnsōGen generates packets from a pcap file, and can send and receive arbitrary-sized packets at 100 Gbps line rate using a *single* CPU core. We describe EnsōGen in more detail in Appendix B. We use EnsōGen in all experiments except for MICA, where we send requests from a MICA client (§7.3.3).

Methodology: We measure zero-loss throughput as defined in RFC 2544 [11, 61] 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 [17] and we obtain other CPU counters using perf [65]. To evaluate MICA, we use the same methodology as the original paper [52] for consistency.

7.2 Microbenchmarks

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

7.2.1 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,

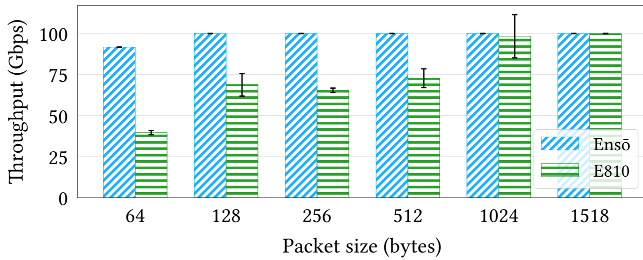


Figure 9: Throughput when forwarding packets between two queues. Ensō outperforms zero-copy E810 even when it needs to copy packets.

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 8 compares the packet rate for Ensō and DPDK for different numbers of cores. Even with a single core, Ensō is bottlenecked by Ethernet, achieving 148.8 Mpps. In contrast, the E810 with DPDK achieves 59 Mpps with a single core and does not scale beyond two cores, where it peaks at 88 Mpps. Beyond two cores, the experiments with the E810 are bottlenecked by PCIe bandwidth, which is insufficient for transferring packet data and descriptor metadata (§7.2.4). 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.

7.2.2 Packet Forwarding with Copies

As we discussed in §3, Ensō's streaming interface targets applications that process received data in order. With Ensō, applications that multiplex and demultiplex data, such as virtual switches [32, 67], need to copy packets to forward them to their destination. However, as we will see next, Ensō outperforms the E810 even in this scenario.

To quantify the overhead of multiplexing and demultiplexing data, we implement a simple packet forwarding application that swaps MAC addresses and copies incoming packets to a different TX Ensō Pipe. We compare this application against an equivalent zero-copy DPDK implementation. Figure 9 shows the throughput when forwarding packets of different sizes using a single CPU core. Packet copies add overhead to Ensō, which can no longer forward 64-byte packets at 100 Gbps (148.8 Mpps). However, Ensō's throughput with 64-byte packets (91.7 Gbps) is still more than 2× that achieved by the E810 with DPDK *without* copies (39.6 Gbps). For other packet sizes, Ensō achieves line rate. We also evaluate the throughput for the same application when sending packets from a CAIDA trace [13]. For this trace Ensō's throughput is 92.6 Gbps, compared to 72.6 Gbps for the E810 with DPDK without copies.

This result came as a surprise to us as our goal with Ensō was never to target multiplexing/demultiplexing in software.

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

Table 1: Effect of chaotic memory accesses on throughput and cache misses. Results are the average of 10 runs, each lasting 10 s.

It also puts into question the usefulness of a packetized interface, as its overheads can be greater than those imposed by packet copies.

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

7.2.3 Effect of Chaotic Memory Accesses on Cache

We now show that placing messages sequentially in an Ensō Pipe is important for Ensō's performance. As we discussed previously, not using sequential buffers results in chaotic memory accesses, which reduces the effectiveness of hardware prefetchers (e.g., streaming prefetcher [36]). To evaluate this claim, we built a modified version of Ensō, which we call Chaotic Ensō, that changes the gaps in memory between subsequent messages. We compute the gap deterministically based on the message's current position in an Ensō Pipe, ensuring that we add no additional software overhead when using Chaotic Ensō.

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

We show the results in Table 1. Observe that Chaotic Ensō achieves a lower throughput of 56.5 Mpps than even the E810. The number of cache misses reveals why: despite processing fewer packets per second (and hence having fewer cache accesses), Chaotic Ensō has an order-of-magnitude (557 million vs. 22.7 million) more L1d cache misses than Ensō. This, in turn, leads to an order of magnitude (558 million vs. 22.8 million) more accesses to L2 cache, thus increasing packet processing overheads. We also observe that the E810 has more cache accesses than either Ensō or chaotic Ensō, this is because it uses a descriptor per packet and thus requires the application to read more data. Finally, we observed that LLC misses were rare in all three configurations.

7.2.4 PCIe Bandwidth

Next, we evaluate the importance of reducing packet metadata by eliminating descriptors. We do so by measuring PCIe bandwidth when using the echo server described in §7.2.1 with 64-byte packets. In what follows, PCIe writes refer to DMA transfers from NIC to host memory, while PCIe reads refer to DMA transfers from the host memory to the NIC.

Unlike other microbenchmarks, we do not limit ourselves to zero-loss throughput for this experiment, and instead send

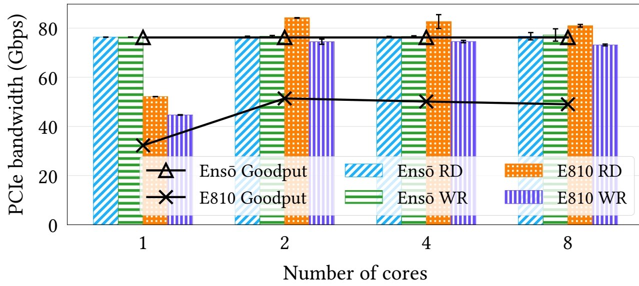


Figure 10: PCIe bandwidth utilization for read (RD) and write (WR) transactions. The bars represent the overall PCIe bandwidth utilization, while the lines represent the goodput, i.e., the amount of PCIe bandwidth used to transmit packet data. Ensō uses little PCIe bandwidth for metadata, causing it to achieve a higher goodput while consuming less overall PCIe bandwidth.

packets at line rate. This ensures that software overheads do not limit observed PCIe bandwidth, since drops due to queuing do not reduce throughput. We measure PCIe bandwidth and the rate at which the packet generator receives packets. We use this to calculate the fraction of PCIe bandwidth used for actual packet data (goodput).[5]

We report the results in Figure 10, where we show both PCIe goodput and total PCIe bandwidth for read (RD) and write (WR) directions. We draw four conclusions from it:

1. With one core, the E810 is CPU-bound, and the NIC drops incoming packets because of a lack of buffers in host memory. This reduces goodput and PCIe bandwidth utilization.
2. Beyond two cores, the E810 becomes PCIe bound, and consumes up to 84.1 Gbps of PCIe read bandwidth. This is close to 85 Gbps, the theoretical limit for PCIe Gen3 x16 with 64-byte transfers [62].
3. Even though the E810 has a lower goodput, it consumes more PCIe bandwidth due to metadata. This overhead accounts for up to 39% of the PCIe read bandwidth. In contrast Ensō's metadata overhead remains below 1.2%.
4. Although barely noticeable in the plot, Ensō's PCIe write bandwidth utilization increases as we increase the number of cores: going up from 76.4 Gbps (one core) to 77.2 Gbps (eight cores). This is because the NIC sends reactive notifications more frequently when software consumes packets faster.

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

7.2.5 Reactive Notifications and Latency

As we discussed in §4.2, Ensō is able to reduce metadata overhead by sending notifications reactively. We measure

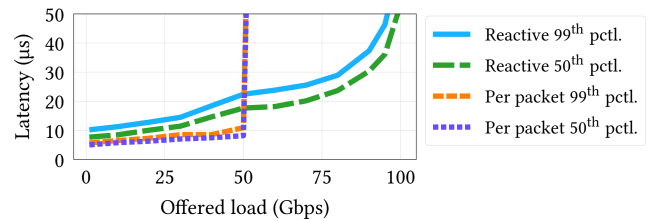


Figure 11: RTT for different loads when using a notification per packet or reactive notifications without notification prefetching.

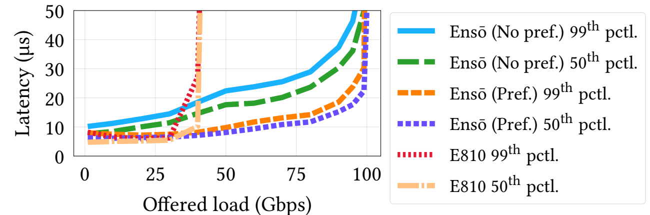


Figure 12: RTT for different loads for the E810 as well as Ensō with and without notification prefetching.

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

We use notification prefetching to minimize latency under high loads.[6] When using notification prefetching, the software explicitly sends the NIC a request for notifications pertaining to the next Ensō Pipe, while consuming data from the current Ensō Pipe. This effectively doubles 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 12 shows the RTT with an increasing load for Ensō with and without notification prefetching and for an E810 NIC with DPDK. We observe that notification prefetching significantly reduces Ensō's latency, and allows us to achieve latency comparable to the E810, while still sustaining 100 Gbps.

7.2.6 Sensitivity Analysis

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

Impact of the number of Ensō Pipes: We measure the impact of increasing the number of Ensō Pipes by varying the number of active Ensō Pipes, and using a workload where each incoming packet goes to a different Ensō Pipe. We partition Ensō Pipes evenly across all cores. Our results in Figure 13 show that (a) we need at least two Ensō Pipes to

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

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

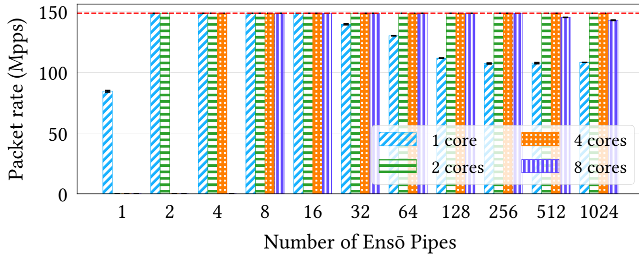


Figure 13: Packet rate for different numbers of Ensō Pipes and core counts. The dashed line represents the 100 Gb Ethernet limit.

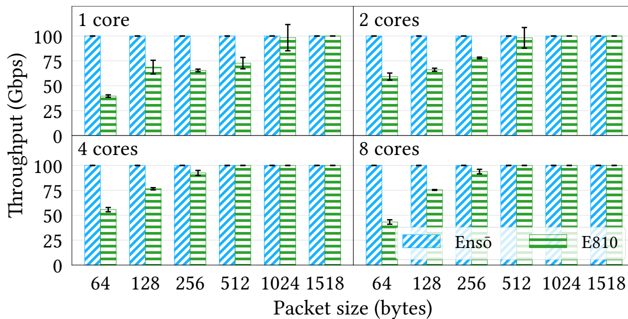


Figure 14: Sensitivity analysis with different numbers of cores and packet sizes. Ensō is bottlenecked by 100 Gb Ethernet in all scenarios.

achieve line rate, since this allows us to mask notification latency; and (b) that throughput drops when a core has more than 32 Ensō Pipes, or eight cores have more than 512. Note that this is a pessimal workload, and realistic workloads are likely to perform better even with many queues [30].

Impact of packet sizes and cores: In Figure 14 we measure the impact of varying packet size and number of cores, and find that Ensō can always sustain full line rate, regardless of packet size and core count.

7.3 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 [23], a network-telemetry application based on NitroSketch [54], MICA Key Value Store [52], and a log monitor inspired by AWS CloudWatch Logs [6]. 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 (§6).

7.3.1 Maglev Load Balancer

We implemented Google's Maglev load balancer [23] as follows. We replicate the consistent-hashing algorithm pro-

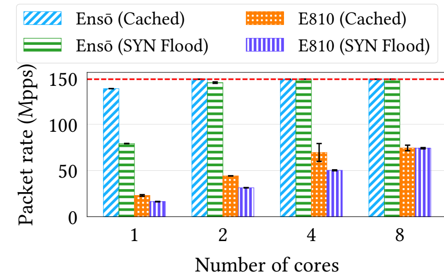


Figure 15: Packet rate for the Maglev load balancer under two types of workloads. The dashed line is the 100 Gb Ethernet limit.

posed in the Maglev paper, caching recent flows in a flow table, as also suggested in the paper. The load balancer ultimately determines a backend server for every incoming packet, rewriting the packet's destination IP to that of the chosen backend server. To steer packets among different cores, we use a hash of the 5-tuple (RSS) in both systems. We evaluate our implementation using two extreme types of workloads: one with only 16 flows, which means that packets always hit the flow cache; and another with a SYN flood, which means that packets always miss the flow cache. In both cases, we use small 64-byte packets as Maglev is motivated by the need to load balance small requests [23, §3.2]. For Ensō, we use unified RX/TX Ensō Pipes to avoid copying the packet when forwarding it back.

Figure 15 shows the packet rate with both the E810 NIC using DPDK and Ensō as we scale the number of cores. With a single core and the cached workload, Ensō achieves a packet rate of 138 Mpps, approximately 6× the 23 Mpps achieved by the E810. With the SYN flood workload, Ensō achieves 79 Mpps, approximately 5× the 16 Mpps achieved by the E810.[7] With four cores, Ensō becomes bottlenecked by Ethernet for both workloads; and with eight cores, the DPDK implementation becomes bottlenecked by PCIe (§7.2.4).

7.3.2 Network Telemetry

Sketching algorithms are popular primitives for many network telemetry tasks (e.g., heavy-hitter detection, flow count estimation) because of their small memory footprint and theoretical accuracy guarantees. NitroSketch [54] is a sketching framework that enables *software sketches* to achieve high performance on commodity servers without sacrificing accuracy. To evaluate this class of applications, we implemented a Count-Min Sketch (CMS) using the NitroSketch framework and Ensō. As in Maglev, we use unified RX/TX Ensō Pipes.

We benchmarked our implementation using two workloads: 64B packets (emulating the stress-test performed in [54]), and a busy period sampled from the 2016 CAIDA Equinix 10G dataset [13], with an average packet size of 462B.

[7] We note that DPDK's packet rate of 16 Mpps with a single core is in fact a good packet rate for DPDK. For instance, NetBricks' Maglev implementation achieves 9.2 Mpps with a single core [64]. We attribute the improvement in our DPDK numbers to NetBricks' unoptimized implementation and our use of a newer CPU with better single-thread performance.

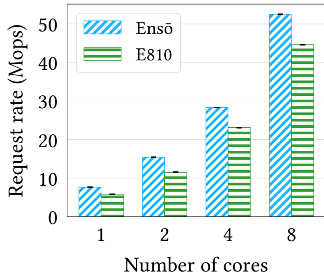


Figure 16: MICA throughput with 8B keys and values.

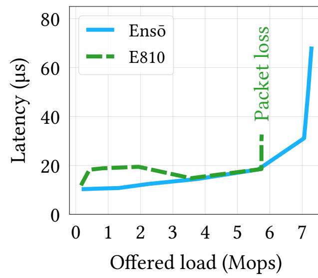


Figure 17: Mean RTT for MICA as a function of offered load (1 core).

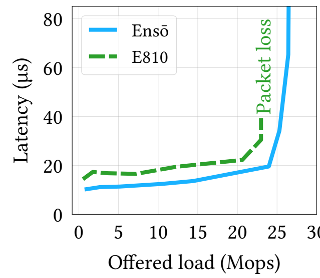


Figure 18: Mean RTT for MICA as a function of offered load (4 cores).

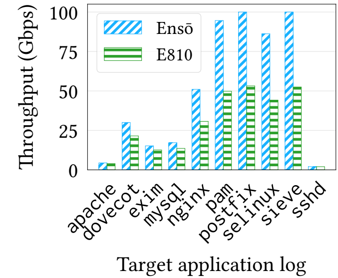


Figure 19: Log monitor throughput for different target applications.

With a single core, Ensō sustains a zero-loss throughput of 81.5 Gbps (121 Mpps), approximately 3.5× that achieved by the E810 (22.8 Gbps or 33.9 Mpps) on 64B packets. On the CAIDA trace, Ensō achieves 94.9 Gbps, a 21% improvement over the E810 (78.3 Gbps); the application remains compute-bound in this setting, but Ensō shrinks the fraction of time spent performing network I/O, which improves performance.

7.3.3 MICA Key-Value Store

MICA [52] is a state-of-the-art key-value (KV) store which is also a popular benchmark in the literature [34,46,47,51,69,79]. Different from Maglev and the Network Telemetry application, that operate on raw packets and forward them back with modifications, MICA represents a typical message-based application whose responses must be constructed separately from the incoming request. Also different from these applications, latency is typically more critical for key-value stores [7]. MICA also entails significantly more work per packet (in terms of both compute and memory accesses) and is, therefore, less likely to become network-bound.

For the following experiments, we set the size for both keys and values to 8B (corresponding to the 'tiny' configuration in [52]). We report results for operations skewed 50% towards `GET` requests and with a uniform distribution of key popularity, but these generalize to other configurations as well. We use the same throughput metric as described in [52] (tolerating <1% loss at the NIC), and the same methodology for measuring end-to-end latency (the client tags each request with an 8B timestamp, then computes latency based on the arrival time of the corresponding response).

Figure 16 shows the steady-state throughput in millions of operations per second (Mops) achieved by MICA for both E810 with DPDK and Ensō for different numbers of cores. With a single core, Ensō achieves 7.65 Mops, a 31% improvement over the E810.[8] While this might seem modest at first (compared to the 6× speedup on Maglev), note that MICA is significantly more compute- and memory-intensive than Maglev. Thus, while DPDK adds considerable CPU overhead, it corresponds to a smaller fraction of the overall compute time.

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

With 2 and 4 cores, we see throughput speedups of 33% and 23%, respectively. At 8 cores, the bottleneck moves to a different part of the system (i.e., memory). We report numbers for the 'tiny' configuration since it represents a significant fraction of requests found in real workloads [4, 57], while also being the most challenging workload for MICA. We also tested other configurations with larger keys and values, obtaining up to 29% improvement for the 'small' workload (16B keys and 64B values) and up to 12% for the 'large' workload (128B keys and 1024B values).

We also evaluate latency, plotting the average request latency as a function of the offered load when using a server with a single core (Figure 17) or four cores (Figure 18). For both configurations, we find that Ensō outperforms the E810 in terms of both throughput and latency. With a single core, Ensō reduces latency by up to 8 µs (43% reduction) before the queues start to build up and, with four cores, Ensō reduces latency by up to 6 µs (36% reduction).

7.3.4 Log Monitor

To understand Ensō's effect on streaming applications, we implemented a log monitor. The log monitor is inspired by AWS CloudWatch Logs [6], which lets users centralize logs from different hosts and craft queries to look for specific patterns and raise alarms. Like, AWS CloudWatch Logs Insights [5], the log monitor also supports Hyperscan [84] to search for multiple regular expressions. We use Hyperscan in streaming mode for both Ensō and DPDK implementations. To feed the system, we use MTU-sized packets, carrying system logs extracted from long-running Linux hosts. We also configured the log monitor to look for regular expressions extracted from Fail2ban [45]. We run experiments targeting each of the ten most popular applications supported by Fail2ban according to the Debian package statistics [16].

Figure 19 shows the throughput we achieve when targeting each of the ten applications. Performance is dictated primarily by the number and complexity of the regular expressions that are required by each target. Ensō's throughput is higher across all targets but the gap is more noticeable for those with simpler or fewer regular expressions, with almost double the throughput when targeting `postfix`, `selinux`, or `sieve`. The reasons for Ensō's improvement in performance are twofold: First, Hyperscan performs better when

larger chunks of data are handed to it at once. With Ensō, we can invoke Hyperscan with large chunks of contiguous logs delivered from the NIC but with DPDK we need to invoke Hyperscan for every DPDK `mbuf`. Second, as demonstrated in §7.2.3, Ensō's memory access patterns are sequential, making better use of the CPU prefetcher.

8 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, 22, 26, 33, 50, 66, 73, 75, 80, 87, 88], 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 [22], 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 [20] 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 [70]. 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 [81].

In terms of abstraction, U-Net [83] and, more recently, NICA [27] allow the NIC to exchange application-level messages directly. U-Net proposes a communication abstraction that resembles part of what is now `libibverbs` (RDMA) [53] 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 [49] quantifies the benefits that custom NIC interfaces could have to different applications. NIQ [29] 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 [69] stores packet payloads on NIC memory, sending only the packet headers inlined inside descriptors, which is useful for network functions that only need to modify the header. This is orthogonal to Ensō's interface changes and could also be used in conjunction with it.

Application-specific software optimizations: Some proposals avoid part of the overheads of existing NICs with application-specific optimizations in software. TinyNF [68] 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 [46] 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 [21] 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).

9 Conclusion

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

Acknowledgments

We thank our shepherd, Jon Crowcroft, and the anonymous OSDI '23 reviewers for their great comments. We thank Francisco Pereira and Adithya Abraham Philip for their comments on this and earlier drafts of this paper, and Ilias Marinos for the discussions and feedback regarding applications. We also thank the people from Intel and VMware that gave us feedback throughout this work, including Roger Chien, David Ott, Ben Pfaff, Yipeng Wang, and Gerd Zellweger.

This work was supported in part by Intel and VMware through the Intel/VMware Crossroads 3D-FPGA Academic Research Center, by a Google Faculty Research Award, and by ERDF through the COMPETE 2020 program as part of the project AIDA (POCI-01-0247-FEDER-045907). Nirav Atre was supported by a CyLab Presidential Fellowship.

References

[1] 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. 2.2

[2] Amazon. DPDK driver for elastic network adapter (ENA). https://github.com/amzn/amzn-drivers/tree/master/userspace/dpdk, 2022. 2

[3] 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. 1

[4] 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. 1, 2.2, 7.3.3

[5] AWS. CloudWatch Logs Insights query syntax, 2022. https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/CWL_QuerySyntax.html. 7.3.4

[6] AWS. What is Amazon CloudWatch Logs?, 2022. https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/WhatIsCloudWatchLogs.html. 1, 7.3, 7.3.4

[7] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Communications of the ACM*, 60(4):48–54, March 2017. 4.2, 7.3.3

[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. 8

[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. 1

[10] Junehyuk Boo, Yujin Chung, Eunjin Baek, Seongmin Na, Changsu Kim, and Jangwoo Kim. F4T: A fast and flexible FPGA-based full-stack TCP acceleration framework. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, Orlando, FL, USA, 2023. 6

[11] S. Bradner and J. McQuaid. Benchmarking methodology for network interconnect devices. RFC 2544, March 1999. 7.1

[12] Broadcom. BNXT poll mode driver. https://doc.dpdk.org/guides/nics/bnxt.html, 2022. 2

[13] CAIDA. Anonymized internet traces 2016. https://catalog.caida.org/dataset/passive_2016_pcap. 3, 7.2.2, 7.3.2

[14] Chelsio Communications. Terminator 5 ASIC, 2021. https://www.chelsio.com/terminator-5-asic/. 1, 6

[15] Jonathan Corbet. Ringing in a new asynchronous I/O API, 2019. https://lwn.net/Articles/776703/. 2

[16] Debian. Debian popularity contest: Statistics by source packages (sum) sorted by fields, 2022. https://popcon.debian.org/source/by_inst. 7.3.4

[17] Roman Dementiev et al. Processor Counter Monitor (PCM), 2022. https://github.com/opcm/pcm. 7.1

[18] 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. 2.2

[19] DPDK. Data Plane Development Kit, 2022. https://dpdk.org. 1, 2

[20] DPDK. NVIDIA MLX5 ethernet driver, 2022. https://doc.dpdk.org/guides/nics/mlx5.html. 1, 8

[21] 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. 8

[22] 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. 1, 4.1.2, 8

[23] 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. 1, 7.3, 7.3.1

[24] 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. 7.1, B

[25] Paul Emmerich, Maximilian Pudelko, Simon Bauer, Stefan Huber, Thomas Zwickl, and Georg Carle. User space network drivers. In *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '19, pages 1–12, Cambridge, UK, 2019. IEEE. 4.1, A

[26] 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. 8

[27] 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. 1, 8

[28] 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. 2.1

[29] 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. 8

[30] Hamid Ghasemirahni, Tom Barbette, Georgios P. Katsikas, Alireza Farshin, Amir Roozbeh, Massimo Girondi, Marco Chiesa, Gerald Q. Maguire Jr., and Dejan Kostić. Packet order matters! Improving application performance by deliberately delaying packets. In *19th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '22, pages 807–827, Renton, WA, April 2022. 7.2.6

[31] 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. 1

[32] 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. 2.1, 3, 6, 7.2.2

[33] 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. 8

[34] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Changhoon Kim, and Nick McKeown. The nanoPU: A nanosecond network stack for datacenters. In *15th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '21, pages 239–256, July 2021. 7.3.3

[35] Intel. Intel data direct I/O technology (Intel DDIO): A primer. Technical report, Intel, February 2012. 2.2

[36] Intel. Intel 64 and IA-32 architectures optimization reference manual. Technical Report 248966-045, Intel, 2022. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html. 7.2.3

[37] 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. 7.1

[38] 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. 7.1

[39] Intel. Intel Ethernet Controller E810. Technical Report 613875-006, Intel, March 2022. 1, 2, 2.1, 4.1, A, B

[40] 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. 7.1

[41] 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. 6

[42] 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. 7.1, B

[43] 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. 2.2

[44] 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. 1

[45] Cyril Jaquier et al. Fail2ban, 2022. https://www.fail2ban.org/. 7.3.4

[46] 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. 7.3.3, 8

[47] Anuj Kalia, Michael Kaminsky, and David G. Andersen. FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '16, pages 185–201, Savannah, GA, November 2016. 7.3.3

[48] 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. 1

[49] 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. 6, 8

[50] 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. 8

[51] Nikita Lazarev, Shaojie Xiang, Neil Adit, Zhiru Zhang, and Christina Delimitrou. Dagger: Efficient and fast RPCs in cloud microservices with near-Memory reconfigurable NICs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, pages 36–51, New York, NY, USA, 2021. 7.3.3

[52] 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. 1, 2.2, 7.1, 7.3, 7.3.3

[53] Linux RDMA. RDMA core userspace libraries and daemons, 2022. https://github.com/linux-rdma/rdma-core. 8

[54] 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. 1, 7.3, 7.3.2

[55] 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. 2.2

[56] Marvell. DPDK marvell. https://github.com/MarvellEmbeddedProcessors/dpdk-marvell, 2022. 2

[57] 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. 1, 7.3.3

[58] 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. 1

[59] Amirhossein Mirhosseini, Hossein Golestani, and Thomas F. Wenisch. HyperPlane: A scalable low-latency notification accelerator for software data planes. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '20, pages 852–867, 2020. 4.1.2

[60] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 221–235, New York, NY, USA, 2018. 6

[61] Al Morton. RFC Errata, Erratum ID 412, RFC 2544, November 2006. https://www.rfc-editor.org/errata/eid422. 7.1

[62] 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. 2.2, 2

[63] Nvidia. NVIDIA BlueField-3 DPU, 2022. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-3-dpu.pdf. B

[64] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. NetBricks: Taking the V out of NFV. In *12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '16, pages 203–216, Savannah, GA, November 2016. 7

[65] Perf. `perf`: Linux profiling with performance counters, 2022. https://perf.wiki.kernel.org. 7.1

[66] 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. 8

[67] 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. 2.1, 3, 6, 7.2.2

[68] 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. 2, 6, 8

[69] 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. 7.3.3, 8

[70] 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. 8

[71] 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. 1

[72] 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. 1

[73] 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. 2, 8

[74] Hugo Sadok, Miguel Elias M. Campista, and Luís Henrique M. K. Costa. A case for spraying packets in software middleboxes. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, HotNets '18, pages 127–133, New York, NY, USA, 2018. B

[75] 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. 8

[76] Alireza Sanaee, Farbod Shahinfar, Gianni Antichi, and Brent E. Stephens. Backdraft: A lossless virtual switch that prevents the slow receiver problem. In *19th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '22, pages 1375–1392, Renton, WA, April 2022. 4.1.2

[77] 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. 1, 6

[78] Arjun Singhvi, Aditya Akella, Dan Gibson, Thomas F. Wenisch, Monica Wong-Chan, Sean Clark, Milo M. K. Martin, Moray McLaren, Prashant Chandra, Rob Cauble, Hassan M. G. Wassel, Behnam Montazeri, Simon L. Sabato, Joel Scherpelz, and Amin Vahdat. 1RMA: Re-envisioning remote memory access for multi-tenant datacenters. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, pages 708–721, New York, NY, USA, 2020. 6

[79] Mark Sutherland, Siddharth Gupta, Babak Falsafi, Virendra Marathe, Dionisios Pnevmatikatos, and Alexandres Daglis. The NeBuLa RPC-optimized architecture. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ISCA '20, pages 199–212, Virtual Event, 2020. IEEE Press. 7.3.3

[80] 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. 8

[81] 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. 2.2, 8

[82] Amy Viviano. Introduction to receive side scaling, 2022. `https://docs.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling`. 2.1, 6

[83] 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. 8

[84] Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park, Geoff Langdale, Jiayu Hu, and Heqing Zhu. Hyperscan: A fast multi-pattern regex matcher for modern CPUs. In *16th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '19, pages 631–648, Boston, MA, February 2019. 7.3.4

[85] Keith Wiles et al. The Pktgen application, 2022. `https://pktgen-dpdk.readthedocs.io/`. 7.1

[86] 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. 1

[87] 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. 8

[88] 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. 1, 8

[89] 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. 1

Appendix A Rejected Designs for Pointer Updates

Unfortunately, the simplest alternatives for communicating pointer updates from the NIC to software perform poorly. We considered three potential alternative designs to let the NIC inform pointer updates to software. While these designs might seem to suit our needs at first, we ultimately discarded them since they perform poorly due to architectural details of PCIe or the CPU:

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

Shared memory synchronization: A second simple alternative would be to dedicate an address in host memory to hold the pointer value. The NIC can issue a DMA write whenever it needs to update the pointer value. Software can then periodically poll the same address to figure out if the NIC advanced the pointer. The issue with this design is that it makes the software and the NIC contend for the same cache line, which requires a slow ownership transfer whenever the NIC or the CPU access the cache line. To verify this, we implemented this design and obtained a throughput of less than 5 Gbps when enqueueing 64-byte packets using the same setup as described in §7.

Inline synchronization: The last discarded design is motivated by traditional NIC descriptor ring buffers which contain constant-sized descriptors with metadata in a format defined by the NIC. These designs designate a bit of the descriptor as a "flag bit" [25, 39]. Initially, all the descriptor slots in the buffer have their flag bit zeroed. Whenever the NIC DMA writes a new descriptor to host memory, it overwrites the old '0' flag with a '1' flag. To figure out if a new descriptor is ready to be consumed, software simply needs to check if the next slot's flag is set. After consuming a descriptor, software sets the flag back to '0.' Because many descriptors are likely to be present in the buffer, this reduces the chance that software and the NIC will contend for the same cache line.

Since Ensō Pipes are opaque, implementing the same strategy used in the descriptor ring buffer is impossible. Even if we zeroed the entire buffer after the data is consumed, we do not know what data to expect in the buffer—the next incoming data might also be zero. Therefore we tested an alternative design: we picked a 128-bit random cookie, to make the chance of collision with incoming data negligible and placed it at the beginning of every cache line of the Ensō Pipe. Software now only needs to check if the next 128 bits match the cookie. Unfortunately, filling the buffer with cookies whenever the data is consumed imposes considerable overhead. For this reason, this design worked well for small data transfers but poorly when using large chunks of data. This design also prevents software from detecting unaligned writes.

Appendix B EnsōGen Packet Generator

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

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

Simple offloads: We implemented hardware support for timestamping and rate limiting, which helps EnsōGen achieve cycle-accurate precision while saving CPU cycles. These features are also commonly offered in existing NICs [39, 63] and are leveraged by some software packet generators [24]. When hardware timestamping is enabled, EnsōGen keeps a histogram in host memory with the RTT of every received packet with 5 ns granularity (the same precision as the hardware timestamper, which operates at 200 MHz). To spread the load equally among the RX Ensō Pipes regardless of the workload, EnsōGen also configures the hardware to direct packets to pipes in a round-robin fashion [74].

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

Artifact Appendix

Abstract

The paper artifact is composed of Ensō's hardware and software implementations, the applications used in the evaluation, the EnsōGen packet generator, as well as the code to automatically run most of the experiments. We also include documentation describing how to set up the environment, compile the code, synthesize the hardware, and use Ensō for other purposes—including a detailed description of the software API.

Scope

The artifact has two main goals: The first is to allow the main claims in the paper to be validated. The second is to allow others to build upon Ensō for their own projects.

We include code to automatically reproduce Figures 8, 11, 12, 13, 14, and 15. We also include the source code for all the applications that we evaluate in §7.3 and for the EnsōGen packet generator.

Contents

The artifact is split between two git repositories.

Ensō Repository

This repository includes Ensō's source code as well as documentation and example applications. It is structured as follows:

`hardware/`: Source code for the hardware component and scripts to automatically generate all the required IPs.

`software/`: Source code for the software component, which includes both the library and the kernel module. It also includes example applications and EnsōGen under `software/examples`.

`frontend/`: Frontend to programmatically load and configure the hardware from Python as well as a command line interface based on this frontend.

`docs/`: Documentation detailing how to set up the system, compile the software and the hardware, and how to use Ensō's primitives (RX Pipes, TX Pipes, and RX/TX Pipes) from an application.

Ensō Evaluation Repository

This repository includes code to automatically run experiments to verify the main claims in the paper and the applications that we evaluate in §7.3. Here we briefly describe the main files and directories:

`experiment.py`: Script to automatically run the experiments to verify the main claims in the paper.

`paper_plots.py`: Script to produce all the plots in the paper.

`setup.sh`: Script to automatically setup the experiment environment.

`maglev/`: Maglev Load Balancer used in §7.3.1.

`nitrosketch/`: Network telemetry application based on NitroSketch used in §7.3.2.

`mica2/`: MICA Key-Value store used in §7.3.3.

`log_monitor/`: Log monitor application used in §7.3.4.

Hosting

Both repositories are hosted on GitHub and archived using Zenodo with a permanent DOI. The documentation contained in the Ensō Repository is also automatically deployed using GitHub actions for easy access.

Ensō Repository

- Repository: `https://github.com/crossroadsfpga/enso`
- Commit: `093dca77836fbe10409af7f0ec3b28232fc25f44`
- Zenodo Archive: `https://zenodo.org/record/7860872`
- DOI: `https://doi.org/10.5281/zenodo.7860872`

Ensō Evaluation Repository

- Repository: `https://github.com/crossroadsfpga/enso_eval`
- Commit: `1a100cb38577930b9124fc6fefced3f0a6da7da4`
- Zenodo Archive: `https://zenodo.org/record/7860936`
- DOI: `https://doi.org/10.5281/zenodo.7860936`

Ensō Documentation

- Link: `https://enso.cs.cmu.edu`

Requirements

Running Ensō requires a host equipped with an Intel Stratix 10 MX FPGA [42] and an x86-64 CPU. The software component also assumes that the host is running Linux. §7 details the exact environment we used in our experiments.