Alleviating Memory and CPU Bottlenecks in Hundred Gigabit Networking

Boris Pismenny

Alleviating Memory and CPU Bottlenecks in Hundred Gigabit Networking

Research Thesis

In Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

Boris Pismenny

Submitted to the Senate of the Technion – Israel Institute of Technology

Kislev 5784 Haifa December 2023

This thesis incorporates the following

1. Boris Pismenny, Haggai Eran, Aviad Yehezkel, Liran Liss, Adam Morrison, and Dan Tsafrir. Autonomous NIC offloads. In ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 18—-35, 2021. https://doi.org/10.1145/3445814.3446732.

2. Boris Pismenny, Liran Liss, Adam Morrison, and Dan Tsafrir. The benefits of general purpose on-NIC memory. In ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 1130–1147, 2022. https://doi.org/10.1145/3503222.3507711

3. Boris Pismenny, Adam Morrison, and Dan Tsafrir. ShRing: Networking with shared receive rings. In USENIX Symposium on Operating System Design and Implementation (OSDI), pages 949-968, 2023.
https://www.usenix.org/conference/osdi23/presentation/pismenny

4. Boris Pismenny, Adam Morrison, and Dan Tsafrir. Disentengling the dual role of NIC receive rings. Work in progress.

Contents

List of Figures

List of Tables

Abstract

The end of Moore's law and the rapid growth in Ethernet network speeds introduce bottlenecks across all host system components, and in particular CPU and memory. Network Interface Controllers (NICs) are ideally positioned to alleviate these bottlenecks as all network data flows through them in any case. Furthermore, hardware pricing trends indicate that NIC upgrades are more cost efficient than CPU upgrades, and that NIC functionality upgrades that preserve NIC speed and port number are essentially free. In this dissertation, we therefore offload CPU-intensive operations to the NIC, or alleviate the CPU when offloading is not possible.

For a class of applications that process packets based on the headers—rather than the data—of incoming packets, needlessly moving data from the NIC to host memory introduces NIC, PCIe, and memory bandwidth bottlenecks. We therefore propose to expose the on-NIC memory (nicmem) that is often underutilized for storing packet data. We demonstrate this using two applications: network functions (NFs) and key-value stores (KVS). Our approach improves latency by up to 43% and increases throughput by up to 80%.

CPU-intensive layer-5 protocol logic (e.g., encryption) built on top of TCP can benefit from NIC offloading, but this typically entails offloading the underlying layer$\leq$4 network stack. We propose an approach that breaks this dependence, allowing offloading only data-intensive computations. A main challenge our approach faces is coping with out-of-sequence packets. We implement our approach for two protocols: (i) NVMe-over-TCP zero-copy and CRC computation, and (ii) HTTPS encryption. Our approach increases throughput by up to 3.3x, and it improves CPU and latency by up to 0.4x and 0.7x, respectively.

The NIC interface of multicore CPUs causes high memory bandwidth when using too many private per-core reception buffers which absorb bursts but exceed the last-level cache to do so. We therefore study two ways to share these buffers between cores: shRing and rxBisect.

ShRing shares a ring between cores using per-core completion rings (CRs). CRs allow cores to process packets on the shared ring independently. But, synchronization is still necessary when advancing the shared ring's head. ShRing is therefore beneficial only when memory contention is greater than the cost of synchronization, and we use a heuristic to enable shRing only in these cases. Our implementation runs on DPDK using NVIDIA ConnectX NICs. Our results show that shRing improves throughput by up to 1.27x and it reduces latency by up to 38x.

RxBisect improves upon shRing by redesigning rings to avoid their inherent producer-consumer entanglement that causes synchronization: (1) memory allocation, whereby the core “produces” empty buffers that the NIC “consumes” for storing packets; and (2) packet delivery, whereby the NIC “produces” incoming packets that the core “consumes” (receives). RxBisect substitutes each ring with two rings: one for each producer-consumer functionality. We implement rxBisect on DPDK using NIC emulation. Our implementation improves throughput by up to 20% and latency by up to 11x. The significant latency gains of both shRing and rxBisect occur when they meet line rate load whereas the baseline fails to do so.

Chapter 1

Introduction

The Internet services which we enjoy in our day-to-day lives—search, social networking, online maps, video sharing, online shopping—run on Data Centers (DCs). DCs are warehouse scale computers that consist of thousands of machines which are interconnected via fast networks. Building and maintaining DCs is tremendously expensive, for example, Amazon's DC in Tel-Aviv spans over 100,000 square feet and they estimate that building each DC costs approximately 2.37 billion USD where only 280 million USD are designated for the land and buildings and the rest is for computing infrastructure [10, 11].

Technology companies seeking to maximize their return on investment (ROI) must efficiently utilize their DCs which is particularly challenging because rapid computer technology changes shift the bottleneck component in DCs. Our work is motivated by a number of recent technology trends in DC servers: (1) Network Interface Controller (NIC) upgrades are cheaper than CPU upgrades (§1.1.1); (2) NIC offloads are essentially free as long as NIC speed and port number remain the same (§1.1.2); (3) NIC speeds are greater and grow faster than the speeds at which CPUs can drive NICs (§1.1.3); and (4) memory bandwidth is an oversubscribed resource shared by both CPUs and NICs, and as memory bandwidth contention increases memory accesses for both slows down linearly at first and then exponentially (§1.1.4).

Based on these trends, in this dissertation, we propose to improve DC ROI by alleviating CPU and memory using the NIC whenever possible: avoiding needless data transfer between the CPU and the NIC (§1.2.1), and offloading data-intensive layer-5 protocol computations without necessitating the offload of all layer$\leq$4 protocols (§1.2.2). When offloading is not an option, we propose to make better use of CPUs. We observe that the way NICs interface to multicore CPUs results in $\geq$1Ki receive buffers per-core being used to absorb bursts, and that these buffers exceed the last-level cache causing high memory bandwidth (§1.2.3). We consider two solutions to this problem that use less receive buffers while still absorbing bursts. Our two approaches achieve this by sharing buffers between cores. Our first approach shares a ring between multiple cores, which imposes synchronization when updating the shared ring, and we therefore use a heuristic to enable it only when memory contention justifies it (§1.2.4). Our second approach proposes a way to share buffers without synchronization by disentangling the ring's packet-reception functionality from its buffer-allocation func-

Table 1.1: The four topics covered in this dissertation. Each topic is introduced in this chapter (2–3 pages per topic) and is later covered by a dedicated chapter that is mostly based on the associated paper.

#	*topic*	*preview of results*		*chapter*		*referred paper*	
		sect.	page	chap.	page	ref.	venue
1	NIC memory offloads	1.2.1	9	2	18	[271]	ACM International Conference on Architectural Support for Programming Languages and Operating Systems **(ASPLOS'22)**
2	Autonomous L5P offloads	1.2.2	11	3	40	[270]	ACM International Conference on Architectural Support for Programming Languages and Operating Systems **(ASPLOS'21)**
3	Shared receive rings	1.2.4	15	5	68	[268]	USENIX Symposium on Operating System Design and Implementation **(OSDI'23)**
4	Disentangled receive rings	1.2.5	16	6	87		Work in progress.

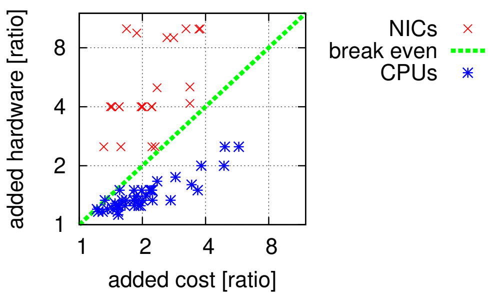


Figure 1.1: CPU upgrades are costlier than NIC upgrades. CPU data is for Intel CPUs and NIC data is for multiple vendors.

tionality and assigning a dedicated ring for each—two rings per-core with the buffer-allocating rings sharing their buffers with multiple buffer-receiving rings (§1.2.5).

Dissertation roadmap This dissertation is structured around the aforementioned four topics: host computations and data movement offloads and two CPU-NIC interface optimizations for when offloading is not possible; as shown in Table **??**. Each topic is associated with a separate chapter, which in turn is largely based on a corresponding paper. Other than these four chapters, this dissertation includes another three: this chapter, which introduces our work, a chapter describing the I/O working set problem which our proposed CPU-NIC interfaces solve, and a chapter with our conclusions.

1.1 Motivation

1.1.1 NICs are cheaper than CPUs

Hardware upgrades are a common method of improving data center performance. When upgrading hardware, one must consider the cost of the various alternatives.

Here, we compare CPU and NIC upgrades using data and analysis from Kuperman et al. [12]. The results in Figure 1.1 show that NIC upgrades are always more advantageous relative to CPU upgrades.

The CPU data corresponds to Intel CPUs. Each point corresponds to a pair of CPUs (c_1, c_2) such that: (1) the number of cores in c_1 is smaller than that of c_2; (2) the cache size (MB), power (W), and QPI speed (GT/s) of c_1 are proportionally smaller than that of c_2; and (3) the series, version, speed (GHz), and process node (nm) of the two are identical. For example, the following two CPUs are adjacent:

c_1: $612 6-core E5-2630L v2 2.40GHz 12T 15.00MB 60W 7.20GT/sec QPI 22nm
c_2: $2336 12-core E5-2695 v2 2.40GHz 24T 30.00MB 115W 8.00GT/sec QPI 22nm

The data points (x, y) in Figure 1.1 represents the relative cost of upgrading from c_1 to c_2 and the resulting relative number of cores added: $x = \frac{\$2336}{\$612} \approx 3.8$ and $y = \frac{12-core}{6-core} = 2$.

Similarly, the NIC points in Figure 1.1 correspond to NICs from Chelsio, Dell, Emulex, HotLava, Intel, Mellanox, and SolarFlare. Two NICs (n_1, n_2) are adjacent if: (1) the speed of n_1 is smaller than that of n_2; (2) the power (W) and PCIe generation and number of lanes of n_1 are proportionally smaller than that of n_2; and (3) the vendor, product series, port number, form factor, and connector types are the same. For example, the following two NICs are adjacent:

n_1: $244 10GbE ConnectX-4 Lx EN MCX4111A-XCAT single SFP+ port PCIe 3.0 x8
n_2: $482 40GbE ConnectX-4 Lx EN MCX4131A-BCAT single QSFP28 port PCIe 3.0 x8

The corresponding (x, y) data points in Figure 1.1 represents the relative cost of upgrading from n_1 to n_2 and the resulting relative bandwidth added: $x = \frac{\$482}{\$244} \approx 1.97$ and $y = \frac{40GbE}{10GbE} = 4$.

The diagonal line marks the break even point, in other words, where the relative cost of added hardware is equal to the relative amount of added hardware. Since all CPU points are below the line whereas all NIC points are above the line, we conclude that NIC upgrades are more cost effective compared to CPU upgrades.

One possible reason underlying this trend might be Intel's dominance in the CPU market at the time in question, which may enable Intel to push CPU prices up, while the NIC market has more competition. Regardless of the reason, the aforementioned trend suggests that making better use of NICs will improve data-center ROI.

1.1.2 NIC Offloads are Essentially Free

When considering to modify or add some NIC functionality, it makes sense to evaluate the cost of the relevant alternatives and see which is preferable in terms of price, with the overall goal of maximizing performance per dollar [12–16].

We study the cost of various changes in application-specific integrated circuit (ASIC) NICs. We find that such changes are cost-effective for clients relative to any potential alternative, because commercial NIC pricing data indicates that clients get ASIC NIC offloads essentially for free.

Table 1.2: Generations of the Mellanox ConnectX NIC over the last decade, and some of the capabilities they introduced.

gen.	*year*	*added offloads*
3 [17]	2011	stateless checksum [18], Large Segmentation Offload (LSO) for TCP over VXLAN and NVGRE [19]
4 [20]	2014	streaming/striding receive queue interface [21,22], Large Receive Offload (LRO) [23], Receive Side Scaling (RSS) [24], VLAN insertion/stripping [25], accelerated receive flow steering (ARFS) [26], on-demand paging (ODP) [27], T10-DIF signature offload (T10-DIF) [28]
5 [29]	2016	header rewrite [30], adaptive routing for RDMA [31], NVMe over fabric [32], host chaining support [33], MPI tag matching and rendezvous [34], UDP Segmentation Offload (USO) [35]
6 [36]	2019	block-level AES-XTS 256/512 bit [37]

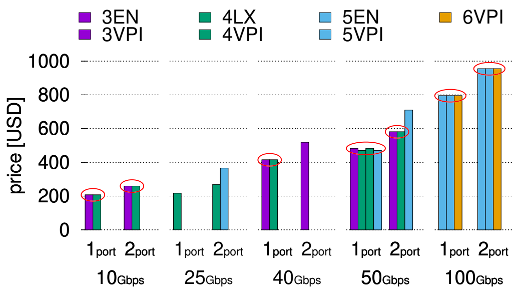


Figure 1.2: Prices of standard ConnectX NICs available via Mellanox's pricing list on March 2020 [1]. The legend numbers (3–6) indicate the ConnectX NIC generation. EN and LX models support Ethernet. VPI models support Ethernet and InfiniBand. Prices of older NICs are typically similar to prices of newer NICs that agree on throughput and number of ports (ellipses), even though the latter provide additional capabilities.

To back the above claim, Figure 1.2 shows the prices of different Mellanox NIC generations in the last decade, as specified in the Mellanox website [38]. Each NIC generation features additional capabilities, listed in Table 1.1. The figure uses different colors for different NIC generations. It reveals that prices are typically determined by the NIC's throughput and number of ports, such that NICs from different generations usually have a similar price if they agree on these two properties. Price similarity exists despite the fact that NIC capabilities substantially improve across generations, so customers do not have to pay more to enjoy additional capabilities.

We note that, for readability, we omit prices of (1) NIC bundles that add some hardware component to the basic NIC, (2) NICs suitable for the Open Compute Project board [39], and (3) NICs that have PCIe connectivity far exceeding their throughput. Including these does not change the conclusion.

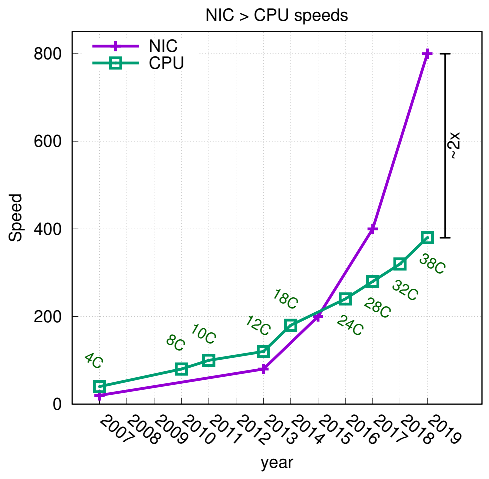


Figure 1.3: The bandwidth of the NIC exceeds what a single CPU could use. Labels show the number of cores per CPU. (Data taken from various sources corresponding to Intel/AMD CPUs [2–4] and Mellanox and Intel NICs [2, 5–9].)

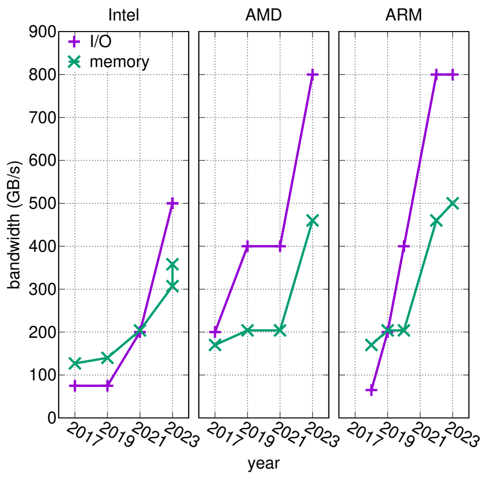


Figure 1.4: Server I/O bandwidth exceeds server memory bandwidth. Data taken from Intel and AMD for x86 and from NVIDIA, Amazon, and Ampere for ARM.

1.1.3 NIC > CPU Speeds

NICs are not only more cost-effective than CPUs, but, as we show next, the growth in the speed of NICs has outpaced the speed of CPUs for the past several years. Figure 1.3 shows this trend using the past progression of the maximal bandwidth a single NIC can support compared to the network bandwidth a single CPU may consume.

The NIC line corresponds to the full-duplex throughput of a single-port NIC. The CPU line shows a relatively high per-core rate of 10 Gb/s TCP, which is about 50% of a core's cycles in a bare-metal setup when running the canonical netperf benchmark [40]; let us assume the other 50% is needed for computation, as netperf does not do anything useful. The number of cores shown reflects the highest per-CPU core-count available from Intel and AMD for the corresponding year. We multiply the assumed maximal per-core bandwidth with the highest core count and display the product as the maximal throughput that one CPU may produce/consume (optimistically assuming that OSes can provide linear scaling when all CPU cores simultaneously do I/O). The figure indicates that CPU speeds must double to satisfy the speed provided by a single NIC. This suggests that offloading CPUs to NICs is not only beneficial but it will also improve the utilization of the most expensive component, i.e., the CPU.

1.1.4 Memory Bandwidth Bottlenecks

One reason for CPU slowness is memory bandwidth contention. Memory is a key resource that is shared by the NIC and the CPU. CPUs read/write memory when processing data and NICs send/receive data using memory to store data between the CPU and the network. Server memory bandwidth is typically oversubscribed: (1) I/O traf-

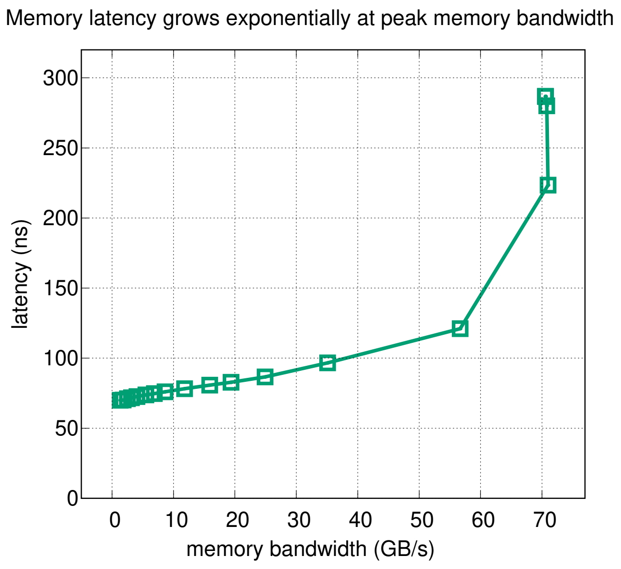


Figure 1.5: CPU memory access latency increases linearly with low memory bandwidth and exponentially with high memory bandwidth.

fic exceeds available memory bandwidth in recent servers; and (2) host-local traffic has exceeded available memory bandwidth by at least 3x for the last several years [41].

Figure 1.4 demonstrates the rapid growth in server I/O bandwidth relative to memory bandwidth. We collect data from Intel, AMD, NVIDIA, Amazon, and Ampere server CPUs between 2017 and 2023. We then calculate I/O bandwidth based on the available PCIe lanes and their generation. For example, an Intel Sapphire Rapids CPU has 80 PCIe generation 5 lanes—about 500 GB/s of bidirectional data (250GB/s per-direction) can be transferred after subtracting PCIe protocol overheads. We calculate memory bandwidth based on the available memory channels and supported memory speed. For example, the same Intel processor supports 8 memory channels each capable of 4800 MT/s of 8 bytes—about 307 GB/s of data. Hence, in our example, I/O bandwidth exceeds memory bandwidth by $\frac{500}{307} \approx 1.62$.

The figure shows that for all servers, today's I/O bandwidth is greater than memory bandwidth: for AMD and ARM servers I/O bandwidth is double the memory bandwidth, and it is 1.67x higher for Intel servers. We speculate that Intel is an exception only because its process node technology is lagging behind TSMC, which the other vendors use.

Memory, like I/O, can suffer from queueing and high access latency as load increases. The memory bus slows down linearly at first, but as load builds up and approaches its peak, memory bandwidth slows down exponentially. Figure 1.5 shows this phenomena by measuring CPU memory access latency with varying background memory bandwidth using Intel's memory latency checker [42] on a Xeon Silver 4216 CPU with 128 GiB (=4x16 GiB) 2933 MHz DDR4 memory. The results show that increasing memory bandwidth increases access latency linearly at first and then exponentially as memory bandwidth approaches saturation.

To reduce memory bandwidth that is caused by I/O, CPUs offer direct cache access technologies, such as Intel Direct Data I/O (DDIO) [43][1], that satisfy NIC direct memory access (DMA) operations from the last level cache (LLC) rather than main memory, which is faster/cheaper and may thus also improve throughput and latency. Specifically, DDIO services DMA reads from the LLC if the target data is already there, which, in addition to being faster, also reduces memory bandwidth contention. Symmetrically, DDIO can perform DMA writes directly to the LLC instead of to main memory by either overwriting existing LLC lines, if they reside in the LLC, or by allocating new lines in up to two LLC ways.

1.2 Contributions

In this thesis, we propose to improve data center efficiency by alleviating CPU and memory using the NIC whenever possible: in §1.2.1, we eliminate needless data transfers between the CPU and the NIC, and, in §1.2.2, we offload data-intensive layer-5 protocol computations to the NIC without requiring the offload of layer$\leq$4 protocols. When the NIC cannot alleviate the CPU, we propose to make better use of CPU resources. In §1.2.3, we observe that the NIC interface to multicore CPUs needs $\geq$1Ki receive buffers per-core to absorb incoming bursts, and that the larger than LLC size of these buffers induces high memory bandwidth that could have been avoided otherwise. To solve this problem, we propose two solutions that share receive buffers between cores. In §1.2.4, we share 1Ki receive buffers between several cores by repurposing existing interfaces as much as possible, and this results in some synchronization between the cores that we only partially eliminate. We therefore use a heuristic to enable our approach only when the cost of synchronization is lower than the cost of memory contention. In §1.2.5, we eliminate the need for synchronization when sharing receive buffers by separating the interface for buffer "production" on the CPU from the interface for buffer "consumption" on the NIC, and then multiple buffer production interfaces can feed one buffer consumption interface without synchronization.

1.2.1 The Benefits of General-Purpose On-NIC Memory

In Chapter 2, we propose an effective, previously unnoticed type of high-throughput networking optimization that eliminates superfluous data movement between host memory and the network. Our new optimization rests on three observations. Firstly, there exists a canonical class of applications that are tasked with moving messages around, from some source to some destination, by exclusively operating on the *metadata* of messages. In contrast, the associated data is not used by such applications except for the purpose of moving it, as is. We denote this type of applications as *data movers* (§2.1).

A notable example of data movers is found in network function virtualization (NFV) workloads such as network address translation (NAT) [46] and load balancing (LB) [47]. In accordance with our above definition of data movers, such network functions are characterized by the property that (1) they make decision based on packet headers

[1]Other processor vendors support similar technologies [44,45]

and may additionally modify these headers, but (2) they neither modify nor use packet payloads save for delivering them to their destinations. Key-value stores such as Memcached [48] and Redis [49] constitute another notable example of a data mover family of applications. In this type of workloads, the key and value are the metadata and data, respectively.

The second observation that underlies our proposed optimization is that all major networking vendors, including Broadcom, Intel, and NVIDIA, increasingly equip their NICs with a small, fast, internal memory [29, 36, 50–55]. For example, the newest NVIDIA NIC (ConnectX6-Dx) is equipped with 4 MiB SRAM. Ordinarily, this memory is designated to be used by various offloading, acceleration, and transport functionalities that the NIC supports and that its users may employ.

The third observation that motivates our work is that this on-NIC memory is typically underutilized. The default setting of, e.g., the aforementioned NIC uses less than 15% of the internal memory, and NVIDIA usage data indicates that clients seldom configure their NICs to use significantly more. Moreover, NVIDIA NIC designers acknowledge that it would be reasonably easy to increase the size of the NIC's internal SRAM (and/or add bigger/slower/cheaper DRAM) provided a compelling use case that needs the additional memory.

In light of the above observations, rather than keeping the on-NIC memory internal, we suggest to expose its unused part to software, to be utilized for general purposes, as regular memory, through memory-mapped I/O (MMIO). We denote this exposed part as "nicmem," and we propose to leverage it for optimizing data mover applications.

We assign the name "nmNFV" (short for "nicmem NFV") to our system that optimizes for NFV data movers with the help of nicmem. In implementing it, we rely on the ability of existing NICs to split each incoming packet into two different buffers that store the packet's header and payload [52, 56, 57]. When arming the receive (Rx) ring with memory buffers that absorb the incoming traffic, the NIC's packet-splitting ability allows nmNFV software to use nicmem for storing payload buffers, simply by populating the relevant Rx ring fields with nicmem pointers. In parallel, nmNFV software uses pointers to regular host memory for header buffers. Consequently, when a packet arrives, its header is placed in host memory, but its payload remains on the NIC, thus reducing PCIe traffic, host memory traffic, and hence latency. For data mover network functions (NFs), the header provides all the information required, so the NF does not mind that the payload is remote. When the NF finishes operating on the header, it transmits the packet using the same payload (nicmem) pointer it received, thus further reducing PCIe and host memory traffic and latency.

Splitting the header and payload of packets between nicmem and host memory ("hostmem") allows us to incorporate a second optimization in nmNFV. Let p denote an incoming (or outgoing) packet, and let h denote its header. In the baseline system, p is stored in its entirety somewhere in hostmem, and this memory location is pointed to by some Rx (or Tx) ring entry. But in nmNFV, only h is stored in hostmem, so instead of pointing to h's location, nmNFV can write h's content to the associated ring entry, leveraging the fact that packet headers are relatively small. We call this optimization "header inlining." We find that it is effective because it improves data locality and reduces the number of CPU cycles and PCIe roundtrips required to process p.

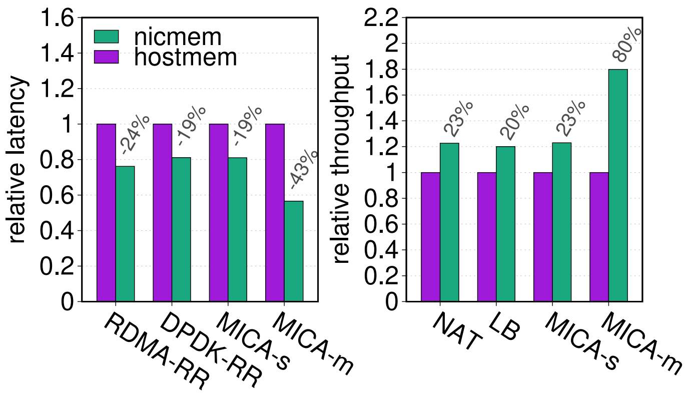


Figure 1.6: Preview of experimental results.

In dealing with key-value stores (KVS), we use the name "nmKVS" (short for "nicmem KVS") to describe our system that optimizes KVS workloads with the help of nicmem. The nmKVS infrastructure accelerates KVS data mover applications by letting them store popular values in nicmem. When incoming requests target such values, the data mover induces smaller PCIe and hostmem traffic overheads similarly to nmNFV, which likewise results in lower latency and higher throughput. KVS workloads are commonly skewed, exhibiting Zipf distributions [58–60]. Because nicmem is smaller than hostmem, such workloads are most suitable for nmKVS.

We describe our design of nmNFV and nmKVS (§2.2), and we explain how we implement a realistic prototype of the two using the NVIDIA ConnectX-5 NIC and its nicmem (§2.3). We experimentally evaluate our prototype using micro- and macro-benchmarks (§2.4). Figure 1.6 provides a preview of some of these results, using: two request-response ("RR") implementations [61, 62] that ping-pong a small message between them; the MICA [63] key-value store accelerated with nmKVS and serving a single ("s") or multiple ("m") clients; and the aforementioned NAT and LB network functions accelerated with nmNFV. As can be seen, our approach improves latency and throughput by up to 43% and 80%, respectively.

1.2.2 Autonomous NIC Offloads

In Chapter 3, we propose an architecture for offloading layer-5 network protocol (L5Ps) data-intensive computations to NICs without necessitating the offload of any layer$\leq$4 protocols. L5Ps built on top of TCP are commonplace and widely used. Examples include: (1) the transport layer security (TLS) cryptographic protocol [64, 65], which provides secure communications for, e.g., browsers via https [66]; (2) storage protocols like NVMe-TCP [67], which allow systems to use remote disk drives as local block devices; (3) remote procedure call (RPC) protocols, such as Thrift [68] and gRPC [69]; and (4) key-value store protocols, such as Memcached [48] and MongoDB [70].

L5Ps are frequently *data-intensive*, as it is in their nature to move bytes of network streams to/from memory while possibly transforming or computing some useful function over them. Such processing tends to be computationally heavy and therefore might adversely affect the performance of the applications that utilize and depend on the L5Ps. In most cases [71], the data-intensive processing consists of: encryption/decryption, copying, hashing, (de)serialization, and/or (de)compression. It is consequently beneficial to accelerate these operations and thereby improve the performance of the corresponding applications.

We classify prior approaches to accelerate L5P processing into three categories (§3.1). The first is *software-based*. It includes in-kernel L5P implementations, such as that of NVMe-TCP [72, 73] and TLS [74]. It also includes specialized software stacks that bypass the kernel and leverage direct hardware access [75, 76]. These type of techniques are good for reducing the cost of system software abstractions. But they are largely irrelevant for accelerating the actual data-intensive operations.

The second category consists of *on-CPU acceleration*. It encompasses specialized data-processing CPU instructions, such as those supporting the advanced encryption standard (AES) [77, 78], secure hash algorithms (SHA) [79, 80], and cyclic redundancy check (CRC) error detection [81, 82]. These instructions can be effective in accelerating L5Ps. But then they themselves become responsible for most of the L5P processing cycles, motivating the use of *off-CPU accelerators*, which comprise the third category, and which we further subdivide into two subcategories: accelerators that are off and on the networking path.

Off-path computational accelerators include various devices that may encrypt, decrypt, (de)compress, digest, and pattern-match the data [83, 84]. Their goal is to allow the CPU to offload much of the data-intensive computation onto them to make the code run faster. The problem is: (1) that the CPU must still spend valuable cycles on feeding the accelerators and on retrieving the results; (2) that developers might need to rewrite applications in nontrivial ways to effectively exploit the accelerators; and (3) that, regardless, the accelerators consume memory bandwidth and increase latency because the CPU must communicate with them via mechanisms such as direct memory access (DMA). Therefore, the outcome of using off-path accelerators might be suboptimal [85].

NICs are *on-path* accelerators, and they do not suffer from the above problems. Because L5Ps are, in fact, network protocols, they necessarily operate NICs in any case, and so driving them (as accelerators) does not incur any additional overhead costs. Moreover, NICs are ideally situated for L5P acceleration, as they process the data while it flows through them, avoiding the aforementioned latency increase and additional memory bandwidth consumption associated with off-path accelerators. NICs already routinely seamlessly handle offloaded computation for the underlying layer$\leq$4 protocols, such as packet segmentation, aggregation, and checksum computation and verification [18, 19, 23].

Despite their seemingly ideal suitability, L5P NIC offloads are not pervasive. The reason: existing designs assign NICs with the role of handling the L5P, which in turn necessitates that the NICs also handle layer$\leq$4 functionality upon which the L5P depends, most notably that of TCP/IP [14, 86–88]. Such *offload dependence* is undesirable, as implementing TCP in hardware encumbers innovation in the network stack [89], and it slows down fixes when robustness [90] or security issues [91, 92] arise. For these reasons, Linux does not support TCP offloads [93, 94], and Windows recently deprecated such support [95].

We propose autonomous NIC offloads to address the above shortcomings and eliminate the aforementioned undesirable dependence. Autonomous offloads facilitate a hardware-software collaboration for moving data between L5P memory and TCP packets, optionally transforming or computing some function over the transmitted bytes. Autonomous offloads allow L5P software to outsource its data-intensive operations to the NIC, while leveraging the existing TCP/IP stack rather than subsuming it, and thus ridding NIC designers form the need to migrate the entire TCP/IP stack into the NIC. Autonomous offloads are applicable to setups where the L5P and NIC driver can communicate directly, as is the case with, for example, in-kernel L5P implementations or when userspace TCP/IP stacks are utilized.

The idea underlying autonomous offloads is for the L5P and NIC to jointly process L5P messages (which may consist of multiple TCP segments) in a manner that is transparent to the intermediating TCP/IP stack. When sending a message, the L5P code "skips" performing the offloaded operation, thereby passing the "wrong" bytes down the stack to the NIC. The NIC then performs the said skipped operation, resulting in a correct message being sent on the wire. In the reverse direction, under normal conditions, the NIC parses incoming messages and likewise performs the offloaded operation instead of the L5P while keeping the TCP/IP stack unaware.

A major challenge that we tackle when designing autonomous offloads is handling out-of-sequence traffic, which occurs when TCP packets are lost, duplicated, retransmitted, or reordered. We use the following three principles to address this challenge: (1) we optimize for the common case by maintaining a small context at the NIC to process the next in-order TCP packet; (2) we fall back on L5P software processing upon out-of-sequence packets; in which case (3) we employ a minimalist NIC-L5P interface that allows the L5P software to help the NIC hardware and driver to resynchronize and reconstruct the aforementioned context.

Context reconstruction for incoming traffic (which is harder than that of outgoing traffic) is driven by the NIC hardware and consists of: (1) speculatively identifying an L5P message header in the incoming data using some "magic pattern" characteristic of the L5P; (2) asking the L5P software to confirm this speculation using the aforementioned interface; (3) tracking the speculated L5P messages while waiting for confirmation; and (4) seamlessly resuming offloading activity once confirmation arrives.

Not all common L5Ps can be autonomously offloaded. In §3.2, we highlight the main ideas underlying autonomous offloads, and, importantly, we identify the properties that L5Ps should have to be autonomously offloadable. Then, in §3.3, we describe the software and hardware design of autonomous offloads in general, and, in §3.4, we present our concrete implementation for two L5Ps: NVMe-TCP and TLS, as well as their combination.

Our TLS autonomous offload is already implemented in the latest generation of Mellanox ConnectX ASIC NICs [53]; it offloads TLS authentication, encryption, and decryption functionalities. Our NVMe-TCP autonomous offload implementation will become available in a subsequent model; it offloads data placement at the receiving end (which thus becomes zero-copy) and also CRC computation and verification at both ends. Linux kernel support for the former has been upstreamed, while the latter is currently under review.

L5P	*application*	*offloads*	*max improvement*		
			throughput	*CPU utilization*	*latency*
NVMe-TCP	fio	copy, CRC	2.2x	2.0x	1.3x
TLS	iperf	crypto	3.3x	2.4x	N/A
NVMe-TCP	nginx/http	copy, CRC	1.4x	1.3x	1.2x
TLS	nginx/https	crypto	2.7x	1.3x	1.2x
both (NVMe-TLS)	nginx/https	all above	2.8x	1.7x	1.4x
both (NVMe-TLS)	redis	all above	2.3x	1.9x	N/A

Table 1.3: Summary of evaluation results. Data is served either from memory (TLS), or from a remote disk over NVMe-TCP, which can also be encrypted with TLS (NVMe-TLS). In NVMe-TLS, crypto is offloaded for both sent HTTPS and received NVMe-TLS traffic. "Crypto" is AES128-GCM encryption/decryption and authentication.

We experimentally evaluate the two offloads and their combination in §3.5, and we find that they provide throughput that is up to 3.3x higher and latency that is as low as 1.4x. When I/O devices become saturated, we show that our autonomous offloads provide CPU consumption that is as low as 2.4x. Table 1.2 summarizes the results. We further show that our offloads are resilient to loss, reordering, and performance cliffs when scaling to thousands of flows that far exceed NIC cache capacities. Finally, in §3.6, we expand on the applicability of autonomous offloads.

1.2.3 The I/O Working Set Problem

In Chapter 4, we describe today's NIC-CPU interface and present the I/O working set problem. Let the *I/O working set* [96] be the memory area that an I/O device (e.g., NIC) reads/writes via DMA in a given time interval (§4.2). For networking intensive applications, this set should preferably fit in the LLC due to DDIO, as otherwise I/O-related data competes for cache capacity and DMAs are increasingly served by main memory instead of the LLC which causes memory bandwidth bottlenecks as a result [75,97–101] (§4.2.1). But, the combination of (1) slow growth in LLC size compared to the number of CPU cores and (2) the interface NICs use to interact with software pushes the I/O working set size beyond the LLC for all high-bandwidth networking applications.

NICs deliver packets from the network to software via queues of pre-allocated buffers, that are called "rings" (§4.1). By default each receive (Rx) ring consists of $\geq$1Ki buffers [102–106,106,107], each large enough to store Ethernet's maximum transmission unit (MTU) of 1500 B [108]. A typical Rx ring thus requires (1Ki $\times$ 1500B $\approx$) 1.5MiB. NICs support hundreds of such rings [36, 51, 52, 109], which software uses for synchronization-free parallelism, assigning different rings to different cores in both kernel [24, 26, 110–114] and user [115–119] network stacks. The combined size of all Rx buffers across all cores constitutes a lower bound for the size of the I/O working set, as the NIC sequentially operates on all Rx buffers, one after the other, so all buffers in the circle must be used before they can be re-used.

To demonstrate the underlying technology trends, we collected the maximal number of rings (N) and the ring's default size (R) from the datasheet and driver, respectively, of every Intel NIC model released during 2000–2022. Table 1.3 shows a representative summary; to conserve space, we only include the first NIC of each Ethernet generation with increasing throughput.

year	*Intel NIC*	*gen. (GbE)*	*max ring num. (r_m)*	*default size (s)*	*Xeon CPU*	*LLC*	*cores*
2001	82544EI [122]	1	1	256	Xeon-2.0 [123]	256 KiB	1
2007	82598 [124]	10	64	512	Xeon-X5482 [125]	12 MiB	4
2014	X710-AM2 [126]	40	1536	512	Xeon-E72880 [127]	38 MiB	15
2020	E810-CAM1 [128]	100	2048	2048	Xeon-9282 [129]	77 MiB	56

Table 1.4: The first Intel NIC model in each GbE generations shown alongside the Intel CPU launched at the same year whose LLC was the largest in that year. The number of supported NIC rings and the default ring size are increasing.

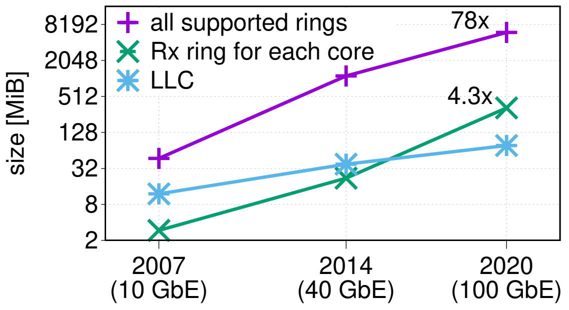


Figure 1.7: aggregate Rx size ($N \times R \times 1500$ B) grows faster than LLC size and has already exceeded it in even the most minimalist configuration (based on data from table 1.3).

Early 1GbE NICs supported only a single ring, but as multicore CPUs became more common, subsequent 1GbE NICs supported up to 16 rings (not shown). Later, the first generation of 10GbE, 40GbE, and 100GbE respectively introduced support for 64, 1536, and 2048 rings.[2] The default ring size likewise increased from 256 to 2048. Network stacks and libraries adopt similar sizes. For instance, the default Rx ring size in all sample apps in the DPDK library is currently 1024 [104].

The right side of Table 1.3 matches each NIC with an Intel CPU model launched at that year, whose LLC was the largest at the time. Using this data, Figure 1.7 plots the size of the LLC and the minimal and maximal $|Rx|$ of the associated NIC. We see that the maximal aggregate Rx size (assuming all N supported rings are used) was always too big to fit in the LLC size in this time range. But in 2020, the aggregate Rx size of even the most minimalist configuration—just one Rx per core—became too big. This is the source of the problem.

The situation is exacerbated if considering logical, rather than physical cores (the 4.3x in the figure would have become 8.6x). We predict that this trend will continue, as upcoming NICs will bring more features (and queues), with speeds of up to 800 GbE expected in 2025 [130, 131].

We consider addressing the I/O working set problem by reducing the size of Rx rings, as proposed by Tootoonchian et. al. [97] (§4.2.2). We find that a size smaller than

[2]It makes sense for N to be much bigger than CPU core number in order to support, e.g.: per-application rings [119, 120]; per-container rings [117, 118]; a ring for every SRIOV [121] instance of every virtual CPU of every virtual machine that runs on the host machine [111, 112]; and a hypervisor ring per VM ring for fallback when flow rule offloading is not yet configured [113, 114].

1Ki might cause a core to experience many more packet drops when the incoming traffic targets this specific core. For example, a core may sustain 2x more packets without drops using 1Ki entries instead of 128. (Increasing Rx sizes beyond 1Ki has no benefit in our workloads.) In contrast, in multicore setups, using 128 entries per Rx ring reduces the I/O working set size without incurring additional drops, provided the incoming traffic is evenly spread between the cores, which curbs the traffic and bursts that each individual Rx ring experiences.

Motivated by this finding, we propose two systems that share Rx buffers: shRing (§1.2.4) and rxBisect (§1.2.5). By sharing buffers, our systems satisfy the simultaneous needs of all sharing cores when incoming traffic is even or uneven. Sharing balances buffer usage, allowing cores that sustain heavier traffic to utilize more Rx entries at the expense of cores sustaining lighter traffic while keeping the I/O working set small.

1.2.4 ShRing: Networking with Shared Receive Rings

In Chapter 5, we present shRing, shRing is advantageous if (1) cache misses due to ineffective DDIO usage cause non-negligible overhead, and (2) the workload avoids pathologically imbalanced conditions, where a subset of the sharing cores are continuously overloaded while their peers are underloaded. (NFV studies commonly assume non-pathological conditions [99,132–139], which might indicate the system is misconfigured.) If DDIO usage *is* effective, then shRing's synchronization overhead might degrade the performance, and if the workload *is* pathologically imbalanced, then the overloaded cores might monopolize all the entries of the shared ring. ShRing thus dynamically identifies the above two conditions, and it turns itself on or off accordingly.

When operational, shRing boosts LLC hits by shrinking the working set, which reduces the per-packet processing time (P_t) and thus increases throughput. If shRing's shorter P_t becomes smaller than packet interarrival time (I_t), queuing theory dictates that ring occupancy drops from full to empty, dramatically shortening latency from linear in the ring size to essentially P_t. But even if shRing's P_t remains greater than I_t (ring fully occupied, so latency is linear in ring size), latency still improves by a factor of $1/N$, as the per-core Rx ring size is effectively $1/N$ smaller, being shared by N cores.

Shared data structures commonly underperform due to software synchronization overhead [26,132,140–143]. ShRing reduces this overhead by avoiding synchronization when deciding which core will process which newly arriving packet. By using per-core completion rings (CRs), the NIC spreads incoming packets between cores, adding the integer index of each packet's entry to the CR of the core that owns the packet [144]. Cores still require synchronization when notifying the NIC that ring entries can be reused. ShRing bounds this overhead by limiting N, the number of sharing cores. We use N=8, but other values may be preferable in other setups.

We explore two shRing variants. The first, "RxArr," is a shared cyclic Rx array structured similarly to a private ring. Because it is shared, its packet buffers routinely become ready for reuse out of (array) order, as they are processed by different cores. The problem is that, for correctness, RxArr is permitted to notify the NIC that entry i can be reused only after all preceding entries (such as i-1) are likewise made reusable. This constraint necessitates coordination between cores, which increases the overhead of synchronization.

Our second shRing variant, "RxList," simplifies coordination by turning the shared ring into a linked list using a "next" field added to Rx entries. When storing incoming packets, the NIC follows list (rather than array) order. This change allows cores to make entries immediately available for NIC reuse; they no longer have to wait for preceding entries. We find, alas, that RxList performs poorly, as the linked list structure undermines the NIC's ability to prefetch Rx entries, ruling this design out for the time being. We propose a modest NIC ASIC modification that resolves this problem (but prevents us from experimentally evaluating this improved design).

We demonstrate in §5.2 that RxArr shRing works as expected, improving NFV macrobenchmark throughput by up to 1.27x and latency by up to 38x. In §5.3, we experimentally show that our findings are also applicable to more traditional applications that use kernel-based TCP sockets. Finally, we discuss related work in §5.4.

1.2.5 Disentangling the Dual Role of NIC Receive Rings

In Chapter 6, we observe that the root cause for all the undesirable trade-offs in shRing and small private rings is the canonical Rx interface, which needlessly entangles two orthogonal producer-consumer functionalities: (1) memory allocation, where software produces empty buffers for the NIC to consume by storing incoming packets; and (2) packet reception, where the NIC produces incoming packets for software to consume by processing them.

Because of this entanglement, on one hand, we cannot reduce the number of Rx buffers per core without simultaneously reducing a core's ability to absorb packet bursts; and on the other hand, we cannot share the system's Rx buffer pool among the cores without also sharing the cores' packet reception capacity.

We propose rxBisect which solves this problem by redesigning the Rx NIC interface to disentangle packet allocation from reception (§6.1). RxBisect splits the traditional circular Rx array into separate allocation (Ax) and bisected reception (Bx) rings, which are independent of each other and may therefore have different sizes. Crucially, rxBisect supports cross-core receive buffer sharing by allowing the NIC to consume an allocated buffer from any Ax ring to store an incoming packet, regardless of the packet's destination Bx ring, provided that both Ax and Bx rings belong to the same software entity/domain. (And provided that the allocation and reception occur on the same NUMA node, to avoid NUMA effects.)

To replenish allocated buffers, the NIC posts a notification to a core's Bx ring whenever it consumes a buffer from that core's Ax ring. In the common case of the Ax and Bx ring belonging to the same core, this notification is included in the new packet's descriptor. When processing a Bx descriptor with said notification, the core places a fresh empty buffer into its Ax ring.

With rxBisect, each core can employ a 1Ki Bx ring, which may be empty or full if, respectively, none or most of the incoming traffic is directed at that core. Allocation rings can then be defined to be much smaller, to reduce the overall I/O working set and achieve the desired effect. RxBisect's sharing of per-core buffers, by quickly moving them around between Bx rings as needed, enables a small I/O working set that can nevertheless accommodate the needs of each individual core, even when much of the network load is mostly directed at it.

workload	*application*	NIC interface	*improvement* *throughput*	*latency*
balanced	network address translation	shRing	25%	25.0x
balanced	network address translation	rxBisect	24%	25.0x
balanced	load balancer	shRing	20%	11.1x
balanced	load balancer	rxBisect	20%	11.1x
overloaded core	synthetic network function	shRing	-58%	N/A
overloaded core	synthetic network function	dynamic shRing	0%	N/A
overloaded core	synthetic network function	rxBisect	11%	N/A

Table 1.5: Main shRing and rxBisect evaluation results. The workload is 200 Gbps that is either *balanced* among all cores or split between one *overloaded core* receiving 1 Gbps with the other 199 Gbps are balanced between the remaining cores. All approaches work well for balanced workloads. When a core is overloaded, shRing performance degrades according to how slow is the overloaded core. Dynamic shRing overcomes this by heuristically switching to the baseline. In contrast, rxBisect achieves line-rate speed even when cores are overloaded.

The rxBisect prototype we evaluate implements the hardware side of rxBisect by means of software emulation (§6.2), as a full implementation would require changing the NIC ASIC. Our implementation dedicates a single core that emulates an rxBisect NIC by consuming packets from the real NIC and producing them to packet-processing cores using the rxBisect protocol. This "software rxBisect NIC" runs on a separate NUMA node from the one housing the processing cores, packet buffers, and the hardware NIC. Our emulated system thus enjoys the benefit of DDIO in the same way a real non-emulated system would.

We compare our emulated rxBisect to bare metal implementations (without emulation) of the privRing and shRing designs. Despite penalizing only rxBisect with the cost of emulation, our evaluation with dual 100 Gbps NICs (§6.3) shows that rxBisect improves network function (NF) throughput by up to 20% and reduces latency by up to 11x compared to default-sized per-core rings. We also show that rxBisect performs well where shRing struggles, including outperforming the "dynamic shRing" that reverts to standard per-core rings when workload conditions are unfavorable for shRing.

Table **??** summarizes the main evaluation results from shRing and rxBisect: they improve performance similarly when traffic is balanced, but only rxBisect performs well when cores are overloaded. We remark that shRing shows even greater improvements in throughput and latency with 190 Gbps balanced workload.

Chapter 2

The Benefits of General-Purpose On-NIC Memory

We propose to use the small, newly available on-NIC memory ("nicmem") to keep pace with the rapidly increasing performance of NICs. We motivate our proposal by accelerating two types of workload classes: NFV and key-value stores. As NFV workloads frequently operate on headers—rather than data—of incoming packets, we introduce a new packet-processing architecture that splits between the two, keeping the data on nicmem when possible and thus reducing PCIe traffic, memory bandwidth, and CPU processing time. Our approach consequently shortens NFV latency by up to 23% and increases its throughput by up to 19%. Similarly, because key-value stores commonly exhibit skewed distributions, we introduce a new network stack mechanism that lets applications keep frequently accessed items on nicmem. Our design shortens memcached latency by up to 43% and increases its throughput by up to 80%.

2.1 Superflous Data Movement

To motivate the benefit of nicmem for "data movers," network applications that move unchanged data to its destination exclusively based on the associated metadata, we focus on two types of data mover workloads: network function virtualization (NFV) applications and key-value stores (KVS). In this section, after we discuss these workloads in more detail (§2.1.1), we exemplify the latency cost that they pay due to superfluously moving data from NIC to host memory and back (§2.1.2). We then enumerate system bottlenecks triggered by this superfluous activity (§2.1.3), and highlight why direct data I/O (DDIO) caching technology cannot eliminate this problem (§2.1.4). Finally, we present the technological trends behind on-NIC memory (§2.1.5).

2.1.1 Data-Mover Workloads

NFV A lot of effort has been put into studying NFV [47, 132, 135, 145–149]. In this class of applications (called "network functions" or NFs), flexible software and off-the-shelf hardware replace rigid proprietary network equipment. Common NFs include firewalls, virtual private networks (VPN), deep packet inspectors (DPI), routers and forwarders, network address translators (NAT), load balancers (LB), flow monitors, and rate limiters. Aside from VPN and DPI, all the above NFs are data movers, operating on metadata only (packet headers and per-flow state) before delivering the packets to their next destination. NAT and LB are two particularly important data movers: a study of data center NFs showed that up to ≈60% of total traffic goes through one or both [150]. As end-to-end encryption prevails, NF access to packet payload is diminished, making data movers more important [151,152].

KVS Key-value stores like Memcached [48] and Redis [49] underlie key cloud and data center infrastructures and drive much of their network traffic [153,154]. Significant research effort has thus been dedicated to improving them, both in software [63,75,76] and in hardware [155–157]. "Get" KVS operations are data movers: clients send keys (metadata) and servers return the matching values (data). KVS access patterns are commonly highly skewed [58–60], so improving the performance of a small set of hot key-value pairs can improve overall performance significantly.

Importantly, in this work, we use the term KVS more broadly than typical, also associating it with such applications as web servers (like Apache [158]) when serving static files.

2.1.2 Latency Cost

In high-throughput workloads, when traffic approaches line rate, we show that superfluous data movement between NIC and host memory causes systems to bottleneck (§2.1.3). But superfluous movement is also disadvantageous in underloaded, non-bottlenecked conditions, because it increases latency.

To illustrate, Figure 2.1 (left) shows the latency breakdown of Data Plane Development Kit (DPDK) ping-pong [61], which sends 64B and 1500B (MTU) packets over the ICMP protocol back and forth between two machines. The first bar ("host") corresponds to the baseline system, which delivers entire packets to host memory, whereas the second bar ("nic") corresponds to storing payloads in nicmem. The third and fourth bars respectively add the header inlining ("inl") optimization, storing headers in NIC rings as outlined in §2. For 1500B packets, nicmem shortens latency by 8% and 15% without and with inlining. For 64B packets, latency is shortened by 19% due to inlining only (nicmem does not play a part as the entire packet is inlined).

Observe that 64B latency is improved by our optimizations (19%) more than 1500B (15%), which is counterintuitive, as 64B benefits from only inlining, whereas 1500 also benefits from nicmem. We hypothesize that this happens because packet-splitting occurs only for 1500B, requiring software to process *two* ring entries when sending and receiving. We corroborate our hypothesis by repeating the ping-pong experiment using RDMA unreliable datagram (UD) [62], as RDMA rids software from having to handle headers. Figure 2.1 (right) shows that in this case the benefit for 1500B is indeed greater.

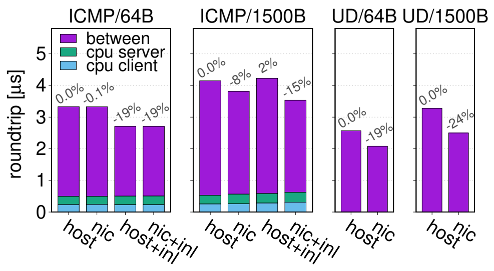


Figure 2.1: Superfluous data movement between NIC and host memory degrades performance even in underloaded conditions.

2.1.3 Bottlenecks

When high-throughput applications stress the network subsystem, the superfluous data movement we identify can bottleneck the three main components that are involved in accommodating this traffic: NIC, PCIe interconnect, and host memory. To exemplify, we run the DPDK Layer-3 forwarding benchmark (called l3fwd [159]) under three gradually intensifying setups configured to forward 1500B packets. The l3fwd server machine is connected back-to-back to a single client load generator machine running the Cisco T-Rex load generator [160]. Full details of our evaluation setup appear in §2.4.1.

NIC The first experiment utilizes a single core driving a single 100 Gbps NIC. The average results are shown in Figure 2.2 (top), which depicts: (i) throughput; (ii) roundtrip latency; (iii) idle CPU cycles ("idleness"); (iv) PCIe bus traffic flowing from NIC to hostmem as observed by the NIC, expressed as percentage out of the maximal PCIe bandwidth available to the NIC, which is 125 Gbps ("PCIe out"); (v) PCIe traffic in the opposite direction ("PCIe in"); (vi) number of occupied Tx ring entries, as measured by the CPU whenever it enqueues packets, expressed as percentage of the ring size, which is 1024 ("Tx fullness"); and (vii) host DRAM bandwidth as measure by Intel pcm [161] ("mem bw"). We measure NIC PCIe utilization with NVIDIA NEO-Host [162].

Examining Figure 2.2 (top), we see that the baseline system is unable to achieve line rate (Fig. i), and that is suffers from high latency (Fig. ii) due to two bottlenecks: PCIe (Fig. iv) and Tx fullness (Fig. vi). We focus on the latter as it is unique to one core/ring processing and has been observed by others attempting to achieve one core/ring 100 Gbps [163–165]. We also remark that single ring bottlenecks are not unique to NICs as we also observe similar issues in NVMe SSDs.

When l3fwd tries to transmit packets that it just processed only to find that the Tx ring full, it drops the packets, which is why it is unable to achieve line rate. We know that the NIC is fast enough to sustain 100 Gbps line rate, so the question is: why is the NIC failing to consume Tx ring entries fast enough and thus causing the Tx ring to reach 100% fullness? NVIDIA performance engineers helped find the answer.

The NIC's transmit engine gathers packets from Tx ring r over PCIe to stream them via the outgoing wire. PCIe is speedier (has higher throughput) than the wire, so r's packets accumulate in an internal NIC buffer b, until unavoidably b gets full. The NIC then reacts by de-scheduling transmission from r for a timeout duration of t, which, for

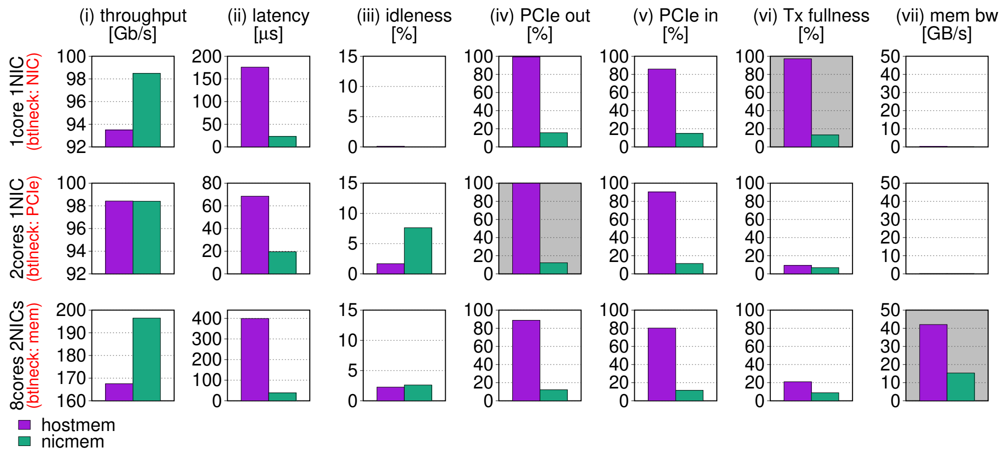


Figure 2.2: Bottlenecks triggered by superfluous traffic between NIC and host memory when running DPDK l3fwd.

reasons outside our scope, is set to be proportional to how long it takes to read another byte from r (≈PCIe roundtrip). The NIC assumes that other Tx rings will keep it busy during this timeout. But no such rings exist in our setup, and t is longer than b's drain time, so the NIC is left with no work, even though packets are awaiting in r. From the CPU perspective, the NIC appears to temporarily stop transmitting, causing r to fill up as observed. Nicmem does not suffer from this problem because for it b contains only headers, so the NIC has a lot more packets to send during t.

PCIe The results of our second experiment are shown in the middle of Figure 2.2. Here, we use two cores (and hence two rings) instead of one. As expected, this eliminates the NIC bottleneck, allowing the baseline to achieve 100 Gbps. The bottleneck that remains is PCIe outbound, which is 99.8% saturated. NIC PCIe out operations are thus stalled and increase latency considerably (Fig. ii), and packets get discarded. We verified that PCIe out is indeed the bottleneck to blame by curbing the client to deliver 90 Gbps, which reduced server PCIe out bandwidth somewhat and resolved the problem.

Nicmem does not exhibit the problem, consuming much less PCIe traffic because packet payloads do not traverse it.

We remark that PCIe out exceeds PCIe in because transmitted packets and associated Tx ring entries are easier to batch than incoming packets and associated completions. Each PCIe transaction incurs some overhead in the form of PCIe headers. With batching, one PCIe transaction handles multiple descriptors, thus batching reduces PCIe link utilization.

Host Memory Our third experiment resulted in the bottom row of Figure 2.2. Here, we use eight cores to handle double the throughput, utilizing two 100 Gbps NICs instead of one in both the client and the server. Additionally, to approximate a memory intensive NF, we configure l3fwd to perform 250 random memory reads per packet from a 8 MiB buffer. Although the baseline system offers an incoming load of 200 Gbps, the server is able to accommodate only ≈170 Gbps from it (Fig. i) with high latency (Fig. ii). This performance drop is caused by running out of DRAM bandwidth (Fig. vii), as the NIC reads and writes payload data that compete with the NF's memory activity, which prolongs its per-packet processing time. (Later, in Figure 2.5, we show that less than 10 per-packet memory reads are enough to bottleneck DRAM.) Nicmem does not suffer from these problems.

2.1.4 DDIO Limitations

The bottleneck resource of applications that exhibit poor memory locality is DRAM [166, 167]. As shown above, I/O-intensive applications might suffer from the same problem, because they involve high-throughput direct memory access (DMA) activity performed by I/O devices—an activity that contends for the same DRAM bandwidth resource [75, 100]. This issue also affects data movers like network functions, which consequently suffer from lower throughput, longer latency, and higher variability [97–99, 168–170]. The problem occurs even in "balanced" systems whereby, on paper, DRAM capacity exceeds I/O bandwidth. The reason is that, as memory utilization increases, access latency likewise increases: linearly at first, and then exponentially when nearing capacity [97].

Direct data I/O (DDIO) technology [43] can avoid or alleviate the problem, as it serves DMA reads from the last level cache (LLC) if the data is there, and it allows DMA writes to allocate up to two LLC ways by default, thereby bypassing DRAM. The effectiveness of DDIO, however, is inherently limited by LLC capacity dedicated to DMA writes [97, 171, 172]. Notably, at a fast enough rate, DDIO writes might evict still-unprocessed packets to DRAM (a.k.a. the "leaky DMA problem"), implying that for DDIO to be effective, the combined size of the buffers pointed by Rx rings should not exceed the LLC size dedicated to DDIO allocations [97].

Ideally, a handful of DDIO allocated cache lines would be enough to fit all receive buffers. However, multi-core packet processing requires a receive ring per core, and each ring entry must be large enough to store the maximum packet size (1500B). For example, an 1024-sized ring stores up to 2MiB of payload buffers which is as large as an entire LLC way on our system. To avoid exceeding DDIO capacity, one may consider to decrease ring sizes [97].

Unfortunately, one cannot arbitrarily reduce the size of rings to accommodate DDIO without negative implications, as we show in §4.2.2. To accommodate high I/O rates, in addition to increasing ring sizes, researchers proposed to increase the number of LLC ways available for DDIO DMA writes [164, 169]. In both cases, the problem is that I/O and CPU potentially contend for the same LLC resource. Using Nicmem alleviates this problem.

2.1.5 On-NIC Memory Today

Our proposal hinges on several technological trends: (1) On-NIC memory *already* exists to support various NIC functionalities, it just is not available for software use; (2) the

NIC's demand for on-NIC memory is limited; and (3) on-NIC memory size can be increased to support data mover usage.

Many commercial ASIC NICs contain on-NIC SRAM [29, 36, 50–55] to support various optional NIC offloading, acceleration, and transport functionalities. For instance, on-NIC memory is used to cache packet steering rules [165, 173]. Due to its current target use case, on-NIC memory is relatively small. For example, the latest NIC model of NVIDIA (ConnectX-6Dx) is equipped with 4 MiB of on-NIC SRAM memory. But this size is dictated by the current use cases, not by technological constraints. NVIDIA architects acknowledge that it is feasible and cost-effective to increase on-memory NIC size to several MiBs—roughly, the equivalent of a CPU LLC size—given a compelling use case.

Moreover, even the limited on-NIC memory is not fully utilized today, because applications and OSs typically do not enable the advanced NIC functionalities that use this memory. For instance, NVIDIA usage data indicates less than 15% of on-NIC memory is typically used.

When adding SRAM we increase NIC die size, each bit cell spans 0.3 μm^2 of silicon [174] which translates to 0.21 \$ per MB at estimated 7 nm process wafer prices [175, 176]. For 16\$ (2% of the price of the cheapest 100 GbE NIC, 795\$) [177], we can obtain 80 MBs which exceeds the LLC of the most powerful Intel 3rd gen scalable processor [178], and can sustain 37 NIC queues with 1024 entries each. Furthermore, we speculate that SRAM die size and price will decrease as 3D stacking technology unlocks hundreds of MBs of SRAM [179].

Because it is cost-effective and it simplifies our implementation, we assume nicmem is as large as CPU LLC. In our design, we show that it is possible to overcome this limitation (§2.2.1). In our evaluation, we show that even small amounts of nicmem are useful (§2.4.4).

2.2 Design

We propose to improve the latency and throughput of data mover applications by equipping ASIC NICs with nicmem, which is on-NIC memory (SRAM and/or DRAM) that the NIC exposes for use by data movers to hold their data, and thereby improve latency, save host memory bandwidth, and reduce DDIO/LLC contention.

We describe the required NIC hardware changes in §2.2.1. We then demonstrate nicmem's utility by designing nicmem-based systems for accelerating NFV and KVS applications (§2.2.2).

2.2.1 Nicmem Hardware

At a high-level, the nicmem design consists of providing large on-NIC SRAM, which is partitioned into two regions. Most of the SRAM is called *nicmem* and is exposed to software, allowing data mover applications to use it for data storage and thereby accelerate data transfer. The rest of the SRAM is not exposed to software and is used by NIC hardware to support various functionalities, as is the case today.

NIC hardware supports identifying packet descriptors (either Rx or Tx) whose payload is located in nicmem. When writing (Rx) or reading (Tx) such a descriptor's payload, the NIC directly accesses its SRAM instead of going through the PCIe interconnect. This mechanism allows data movers to reap nicmem benefits via pure software techniques, by allocating their payloads on the nicmem and thus avoiding hostmem and PCIe traffic for data transfers (as we demonstrate in §2.2.2).

Using nicmem for Rx traffic poses a challenge, however: because nicmem is limited, it may not suffice to hold large packet buffer pools, which are required to support bursty and/or high-throughput traffic. To address this, we design a *split* Rx queue mechanism, in which the NIC can use a secondary Rx queue, located in hostmem, to absorb Rx traffic when nicmem resources are exhausted.

In the following, we describe the design in more detail.

Exposing nicmem NIC firmware carves out a portion of the on-board SRAM and isolates it from the internal NIC functionalities. This step takes place after the NIC driver has initialized and configured all desired NIC functionality, to ensure that all SRAM resources needed for NIC operation are available. The firmware then exposes the nicmem as a memory mapped I/O range on the NIC. The OS identifies this range as a nicmem through the NIC capabilities and makes it available to applications through the `mmap` system call interface. Applications can then map nicmem regions into their address space and subsequently access it through CPU load/store instructions that get routed to the nicmem over PCIe. Since the OS intermediates nicmem mapping, it can restrict different applications to disjoint nicmem ranges. Applications can also register mapped nicmem address regions with the NIC and then use it through NIC queue descriptors. Similarly to the CPU, because NIC hardware interposes between queues and registered nicmem, it can control the access to disjoint nicmem ranges.

Identifying nicmem The benefit of nicmem is that the NIC can access it without going out to the PCIe interconnect. To reap this benefit, the NIC must identify when packet descriptors (created by software) have their payload located in a nicmem address. This is achieved by software setting a flag in the descriptor, which tells the NIC that the address corresponds to a nicmem address.

Receiving traffic into nicmem Typical NIC receive flow (§4.1) makes it challenging to use nicmem for Rx traffic. Since nicmem size is limited, at high networking rates it might not suffice to hold a burst of incoming traffic. As a result, an Rx ring containing only nicmem buffers may become empty during such a burst, leading to packet drops.

To address this problem, we propose to employ a *split rings* mechanism, which is inspired by network page faults [27]. In this design, NIC receive rings are split in two: primary and secondary receive rings. When a packets arrives, the NIC tries to consume a buffer from the primary ring, which holds nicmem buffers. If the primary ring is empty, the NIC proceeds to consume a buffer from the secondary queue (Figure 2.3). Completion entries are stored in a single queue, as before, but with the entries indicating the order and location of received packets, i.e., primary or secondary ring. Packet buffers are subsequently returned by software to their original ring.

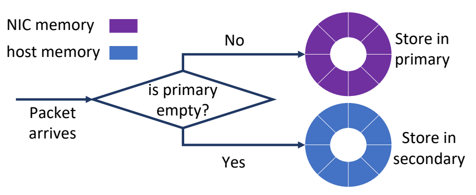


Figure 2.3: The split rings approach.

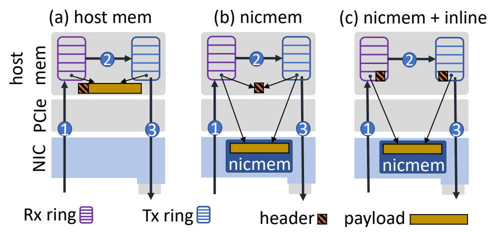


Figure 2.4: Host memory based packet transmission compared to data on nicmem with and without header inlining.

The split rings approach guarantees that as long as the working set of incoming packets is smaller than the nicmem size, then all received packets are served by nicmem from the primary ring. The split rings design introduces only negligible latency to the NIC's receive pipeline, since checking ring occupancy is based on ring producer and consumer indexes that are stored on nicmem. It does, however, double the number of queues in the system, but we believe that this overhead is acceptable because per-queue state is very small.

Beyond SRAM Nothing in the above design is SRAM-specific. Indeed, nicmem can be extended with DRAM to provide value for applications with memory demands beyond those that can be satisfied by SRAM. On-NIC DRAM is faster for the NIC to access compared to host DRAM, as it can be accessed without a CPU interconnect trip.

2.2.2 Leveraging Nicmem in Data Movers

To improve performance using nicmem, software must navigate the trade-off that nicmem is fast for the NIC to access but slow for the CPU to access, as CPU accesses are routed over the PCIe interconnect to the NIC. We thus observe that data mover applications can significantly benefit from nicmem. A data mover can use nicmem to hold its data and rely on hostmem only for the metadata. This approach saves the CPU cycles and memory bandwidth that would otherwise be required to transfer the data to/from the network from/to hostmem.

In the following, we describe designs that use the above idea to accelerate NFV (§2.2.2) and KVS (§2.2.2) applications. Our designs assume that the application can safely manipulate NIC Rx/Tx rings directly and does not require OS intervention to send/receive traffic. This is the case, for example, in applications using DPDK which offers a packet processing programming model that is based on kernel bypass and direct hardware access for efficiency.

NFV Acceleration (NmNFV)

Our design for accelerating NFV data movers with nicmem is named *nmNFV*. NmNFV mitigates memory bandwidth, DDIO and LLC contention caused by copying of packet payloads into hostmem for NF operations, as shown in Figure 2.4(a). Without nicmem, each incoming packet is (1) DMAed to hostmem by the NIC, (2) operated on by the

NF; and finally (3) transmitted, which requires the NIC to read the header and payload from hostmem with DMA again. Crucially, however, packet payloads are completely ignored by most NFs. The waste of copying payloads into hostmem is compounded by the fact that payloads are typically an order of magnitude larger than headers: network traffic characteristics studies show that packet sizes in data centers, universities, and on the Internet follow a bimodal clustering pattern around small $\approx$ 200 B and large $\approx$ 1400 B packets [180–184].

The basic idea of nmNFV is thus to simply keep packet payloads on the nicmem. To realize this idea, we use several techniques.

First, we rely on the pre-existing capability of the NIC to write an incoming packet's header and payload into different buffers [52,56,57]. NmNFV uses this packet-splitting functionality to configure the Rx ring with Rx descriptors that instruct the NIC to write headers into hostmem and payloads into nicmem (Figure 2.4(b)). Consequently, when a packet arrives, its payload remains on the NIC. Only the header is written to hostmem, which suffices for the NF to perform its operation. Finally, on transmit, the NIC already has the packet's payload in nicmem.

The trade-off in splitting packet headers and payload is that it introduce some overhead to packet processing. The NIC's Rx/Tx rings require twice the number of scatter-gather elements to hold the same number of packets. Not only does it increase the ring's size, but it increases the number of scatter-gather operations the NIC must perform per packet. Moreover, these two pointers must propagate from the application level, which means that book-keeping structures increase in size and more CPU work is required to construct them.

To address this overhead and to further optimize the NF flow, we propose to store a packet's header in its descriptor instead of in an independent hostmem buffer (Figure 2.4(c)). We call this optimization *header inlining*. It leverages pre-existing NIC inlining functionality by which descriptor flags can instruct the NIC to read/write a small range of packet data from/to the descriptor. Header inlining reduces the number of scatter-gather entries required to represent a packet back to one (for the payload). More importantly, it enables the NIC to fetch only the descriptor when sending/receiving data. This optimization thus reduces both the amount of data fetched from hostmem as well as the number of PCIe roundtrips required to do so, because in the non-optimized case the NIC must first read the descriptor in order to obtain the header's address in hostmem.

Header inlining does require an NF to copy the packet's header from its Rx to its Tx descriptor, but the related CPU overhead is low, because the headers are hot in the cache following the NF's processing of the header.

KVS Acceleration (NmKVS)

In theory, a KVS could leverage nicmem by serving its item set from nicmem. The KVS would store the values (data) associated with the keys (metadata) in nicmem and each read request would be answered with a response whose payload is in the relevant item's nicmem. This approach is not viable in general, however, because the size of KVS item sets are as large as host DRAM [185,186], which dwarfs the size of the multi-MiB nicmem.

We address this problem by leveraging the property that KVS workloads are commonly skewed, exhibiting a Zipf distribution [58–60]. We therefore propose *nmKVS*, which is a KVS design that stores a subset of *hot* items on nicmem and serves them from it. Our design focuses on the mechanism of serving hot items out of nicmem and not on identifying hot items in the first place. That is, we assume that a KVS can efficiently identify the hottest items—e.g., using a heavy hitters algorithm [187–189]—and move them to nicmem, while evicting "colder" items back to hostmem.

NmKVS relies on header-data splitting (§2.2.2) to perform zero-copy sends of values residing on nicmem. The basic idea is straightforward, but it creates a concurrency challenge. Suppose that a response containing an item is posted to the NIC's Tx queue, but has not been transmitted yet. In the meantime, the KVS receives an update operation of that value and the CPU starts overwriting the old value. Because the value is updated in place, if the NIC now begins transmission of the queued response packet, it might read (and transmit) an inconsistent mix of the old and new values, since it reads the value concurrently to the CPU updating it.

We handle this race by avoiding in-place data overwrites for "hot" items served directly (zero-copy) from nicmem. Instead, we maintain two buffers for each such item. One buffer, called the stable buffer, resides in nicmem and holds data that may be transmitted by the NIC. This buffer is guaranteed to not be overwritten concurrently to a NIC access. The second buffer, called the pending buffer, holds new data written by an update operation. After an update overwrites the pending buffer, it invalidates the stable buffer by clearing a "valid" bit in its structure. The stable buffer gets updated later, lazily, by some get operation.

To safely update the stable buffer, its structure contains a reference count indicating the number of outstanding Tx descriptors referencing it. The KVS services a get operation for a "hot" item as follows. If the stable buffer is valid, the KVS increments its reference count and uses the stable buffer as the response packet's payload (zero-copy). (The reference count is decremented when processing the NIC's completion event of this packet's transmission.) If the pending buffer is invalid, the KVS checks whether its reference count is zero. If so, the KVS overwrites the stable buffer with the contents of the pending buffer, and transmits a zero-copy response, as before. Otherwise, the KVS transmits a response whose payload is a copy of the pending buffer.

2.3 Implementation

We elaborate on nicmem system software and hardware, and describe our implementation of nmNFV and nmKVS in the DPDK framework, targeting NVIDIA ConnectX NICs.

Kernel API Hardware exposes nicmem to the kernel which manages its allocation to processes using Linux RDMA verbs APIs. Processes obtain nicmem by: (1) requesting the kernel for an allocation of the desired length; and (2) calling `mmap` to map it to virtual memory using write-combined memory pages. Using virtual memory nicmem can be shared or isolated between different processes in the system.

DPDK API To expose nicmem to DPDK applications, we introduce a new API for DPDK NICs (Listing 2.1): `alloc_nicmem` and `dealloc_nicmem` allocate and free NIC memory, respectively. Applications manage this memory using standard DPDK memory allocator APIs such as packet memory buffer pools.

NVIDIA NICs use an on-NIC IOMMU to translate all memory accesses and isolate between applications. To use memory with the NIC it must be registered with the kernel to create a memory key (mkey) that is associated with the application. Then, to send or receive data via application NIC queues, the mkey is provided alongside memory addresses. Nicmem references use an mkey too. Therefore, nicmem is isolated from other application using the NIC.

When DPDK posts receive or transmit descriptors on NIC queues, the driver looks up the mkey corresponding to packet buffer memory. Host-memory usually requires only one mkey while nicmem requires another. To optimize these lookups the drivers caches the most recently used mkeys in order; this optimization is weakened when splitting packets when two mkeys are used per-packet.

NmNFV We implement nmNFV in the DPDK l3fwd [159] application and in the modular FastClick [116, 190] NF composition framework. Our implementation closely follows the nmNFV design. After allocating and mapping nicmem, the NF creates a packet buffer pool on top of nicmem. Next, it configures receive rings to split packets at a 64B offset into header and data buffers residing in hostmem and nicmem buffer pools, respectively, and to inline the headers. Split packets consist of two DPDK `mbuf` structures chained together: one that holds the header and another that points to the data which is either in hostmem or in nicmem.

Importantly, all changes related to nicmem are in DPDK's control-path, which means that application data-path operations are unmodified. As a result, we expect applications which follow DPDK APIs to adopt our approach easily and with no risky modifications to performance critical code. However, we find that some DPDK applications ignore DPDK's APIs and make assumptions about packet buffer structure. In particular, we observed that FastClick accesses packet buffers directly and assumes that there is only one buffer per `mbuf`. Therefore, we modify its data-path elements to support our split packets.

NmKVS We implement nmKVS on top of MICA [63], a highly optimized DPDK-based KVS that is built to achieve the highest performance on CPUs [191]. However, MICA get operations do copy item data twice: once from the KVS table to the stack and again from the stack to the response packet. We speculate that the reason behind this implementation is that copy semantics greatly simplify the design and implementation of the system and/or that it was forced by missing DPDK features, such as a callback upon completion of a packet transmission. Our implementation extends DPDK to support these features.

Our nmKVS implementation modifies MICA to serve a set of hot items directly from nicmem with the zero-copy design described in §2.2.2. We allocate stable buffers for hot items according to available nicmem and use `memcpy` to overwrite values on nicmem as needed. We additionally introduce a DPDK callback on transmit completion

```
void *alloc_nicmem(device, len);
void dealloc_nicmem(addr);
```

Listing 2.1: routines to control NIC memory

to decrement the stable buffer's reference count. Such a callback was not available in DPDK before, and so we modify DPDK and NVIDIA drivers to support it.

Hardware limitations Available hardware imposes some limitation on our implementation. First, our NIC firmware exposes only 256 KiB of its available SRAM. Second, our NIC requires hardware modifications to support the split rings approach. To overcome these and support real applications, we emulate a large nicmem by reusing the provided memory buffer for storing the data of multiple packets, which thus override each other. This methodological technique works as data mover applications and benchmarks do not inspect their payloads. We verified that this methodology does not affect performance by observing no measurable difference in DPDK l3fwd performance with and without reusing nicmem on the available hardware. Third, our NIC also supports only transmit-side inlining, and therefore we still suffer the cost of splitting on receive. Finally, our NIC does not split packets according to hardware parsing which restricts us to use suboptimal hard-coded header split offsets. We expect that future devices will remove this limitation.

Implementation effort To support NIC memory we change 404 lines of code (LoC) in DPDK 20.08 NVIDIA's driver; and 329 LoC in PCIe and Ethernet device infrastructure code. For nmNFV, we modify 194 LoC in FastClick's DPDK binding, and another 25 LoC to support split packets in IP, TCP, and UDP element code. NmKVS support in MICA relies on transmit completion callbacks (64 LoC), zero-copy support (282 LoC), and nicmem support (125 LoC).

2.4 Evaluation

We use microbenchmarks and macrobenchmarks to evaluate nicmem performance for KVS and NFV workloads. After introducing our methodology (§2.4.1), we evaluate NF performance with a syntactic microbenchmark (§2.4.2) and then real NF applications: network address translation (NAT) and load-balancer (LB). Based on these, we quantify the number of cores required to saturate 200 Gbps and measure the impact of various packet and NIC receive ring sizes and DDIO way allocations (§2.4.3). We then measure the impact of split-ring spilling to hostmem by varying the nicmem available in NAT and LB NFs (§2.4.4). We next measure the cost of accessing nicmem from the CPU (§2.4.5) and quantify KVS performance using nicmem (§2.4.6).

2.4.1 Methodology

Our setup consists of a pair of Dell PowerEdge R640 servers, one of which is the system under test and the other is the load generator. Both have 16-core 2.1 GHz Xeon Silver

4216 CPUs, 128 GiB (=4x16 GiB) 2933 MHz DDR4 memory, 22 MiB LLC split across 11 ways. They run Ubuntu 18.04 (Linux 5.6.0) with hyperthreading and Turbo Boost off. The machines are connected back-to-back via two 100 GbE NVIDIA ConnectX-5 NICs [29]. All the results presented are trimmed means of ten runs; the minimum and maximum are discarded. The standard deviation is always below 5%.

NF Benchmarking On the load generator machine, we run the stateless Cisco T-Rex packet generator [160], which we modify to improve latency measurement accuracy from 10-100μs to 1μs (similarly to Primorac [192]). Unless stated otherwise, we send packets at 200 Gbps using our two NICs.

For macrobenchmarking, on the server, we run FastClick [116] based NAT and LB using FastClick's DPDK mbuf pool to avoid unnecessary packet metadata copies. Unless stated otherwise: we disable pause frames; use 1024 Rx and Tx ring descriptors (default); two DDIO LLC ways (default); and 14 cores (as our experience with NAT and LB shows that 14 cores are needed to process 200 Gbps; see Figure 2.6). To maximize CPU efficiency and reach line rate speeds, we spread load equally among all cores using a different flow per packet. We use large 1500B UDP packets unless stated otherwise, as this is a common use case (§2.2.2), and because it helps us sustain 200 Gbps processing on our setup, which generates the highest load on PCIe, DDIO, and memory bandwidth.

We evaluate the following NF processing configurations: (1) "host" employs the baseline DPDK host memory; (2) "split" demonstrates the overhead introduced by splitting packet headers and data (before reducing host memory copies); (3) "nmNFV-" improves performance by placing data on nicmem, thus removing data copies; and (4) "nmNFV" further improves it by inlining headers inside Tx descriptors.

As in §2.1.3, we measure NIC PCIe utilization using NVIDIA NEO-Host [162] and CPU core and unncore counters using Intel pcm [161].

KVS Benchmarking We evaluate the performance of nmKVS using MICA [63] executing on 4 cores. MICA's client is the load generator, using 800K large key-value pairs (128B keys and 1024B values), which we access uniformly at random. As noted, large values ease CPU processing and are common in real workloads [58, 185]. We evaluate two server configurations: (C1) 256 KiB hot area cache that corresponds to the size of nicmem available on our NICs, and (C2) 64 MiB hot area corresponding to a future device that we emulate (§2.3).

2.4.2 NF Microbenchmarks

We use a synthetic NF to explore how memory subsystem contention affects CPU efficiency and, in turn, the throughput and latency of NFs with different attributes. To control NF memory intensity we run layer-2 forwarding followed by the WorkPackage FastClick element, which performs a number of random memory reads from preallocated buffers. We perform 480 runs, covering the space of the following parameters: Rx ring size: 256, 512, 1024, or 2048; accessed memory buffer size: 1, 2, 4, 8, 16, or 32; memory reads per packet: 2, 4, 6, 8, or 10; and DDIO ways: 0, 2, 8, or 11. For each NF

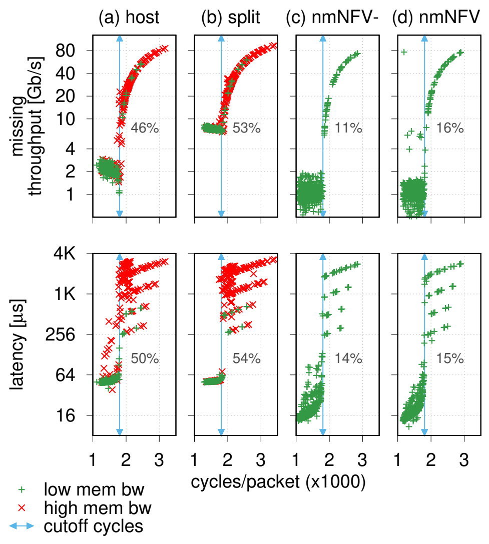


Figure 2.5: Synthetic NF performance. The points are runs with varying Rx ring size, NF memory intensity, and DDIO ways. Percentages show the number of runs past the cutoff.

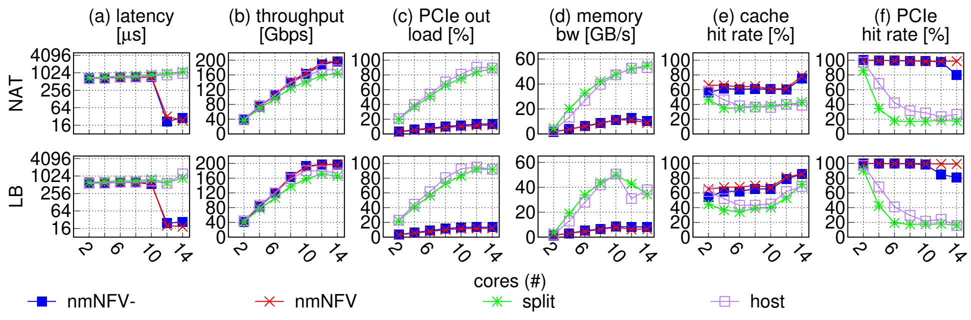


Figure 2.6: To handle 200 Gbps loads NAT and LB need (1) at least 12 cores and (2) to reduce memory and PCIe load.

processing configuration, we plot the missing throughput (200Gbps - measured) and latency of all 480 runs in a scatter-plot in Figure 2.5.

After considering various parameters that may influence performance, we find that NF processing time per packet is most meaningful. We then calculate the per-core budget and mark it as the "cutoff" point. We use 14 cores with frequency 2.1 GHz, and packet arrival rate of 16.26 MPPS: $(14 \times 2.1 \times 10^9)/(16.26 \times 10^6)$ gives us a budget of 1808 cycles per packet before the cutoff point. We observe that in the host configuration, which has to copy packet data to memory, this point is passed for at least 46% of the NFs. Meanwhile, nmNFV only passes its cutoff point for at most 16% of these same NFs.

We mark runs with less than 30 GB/s memory bandwidth with "+" and the rest with "x". We observe that both nmNFV variants eliminate memory bandwidth contention (all are below 30 GB/s), while base and split suffer from the leaky DMA prob-

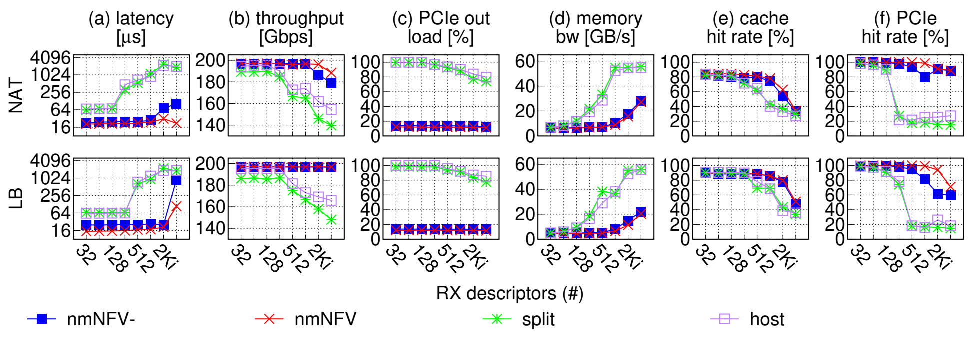


Figure 2.7: Small receive ring size can alleviate memory bandwidth bottlenecks, increasing throughput. But, these are susceptible to packet loss during bursts.

lem and high memory bandwidth contention: at least 60% of runs have more than 30 GB/s memory bandwidth, and in fact at least 31% exceed 40 GB/s. Consequently, both nmNFV variants have as much as 42% more runs within the cutoff budget. Pleasingly, the majority of both nmNFV variants results below the cutoff also have better throughput and latency.

As expected, the results also show that nmNFV consumes more cycles than nmNFV- and thus performs slightly worse when CPU cycles are scarce. However, this is part of a trade-off: nmNFV has better 99th percentile tail latency compared to nmNFV-, i.e., 58% of nmNFV runs are lower than 128 µs compared to only 40% in nmNFV- (not shown).

We observe that when cycles per packet are greater than the budget, workloads with the same cycles per packet still show different latencies. Furthermore these latencies can be grouped into four clusters that correspond to the various Rx ring sizes (256, 512, 1024, and 2048). The reason is that once an NF exceeds the budget it will never process packets before more packets build up in its Rx ring. Therefore, receive rings are always full and each packet will wait until all preceding packets in the ring are processed, thus latency increases with ring size.

2.4.3 NF Macrobenchmarks

We use two stateful NFs to evaluate the performance of both nmNFV variants: LB and NAT using 200 Gbps. These applications cache up to 10 M flows using a per core cuckoo hash table to avoid needless cache contention. LB assigns each flow, using its 5-tuple, to one of 32 destination servers, and stores this pairing to consistently hash and forward subsequent packets of that 5-tuple to the same server. If no match is found, LB uses round-robin to assign a new destination server to the flow. Similarly, NAT identifies existing flows using their 5-tuples and rewrites packet source IP and port consistently. New flows are assigned one of the available source ports.

Next, using 200 Gbps and 1500 B packets, we measure the number of cores required to meet this load, and the impact of various packet and NIC Rx ring sizes and DDIO ways.

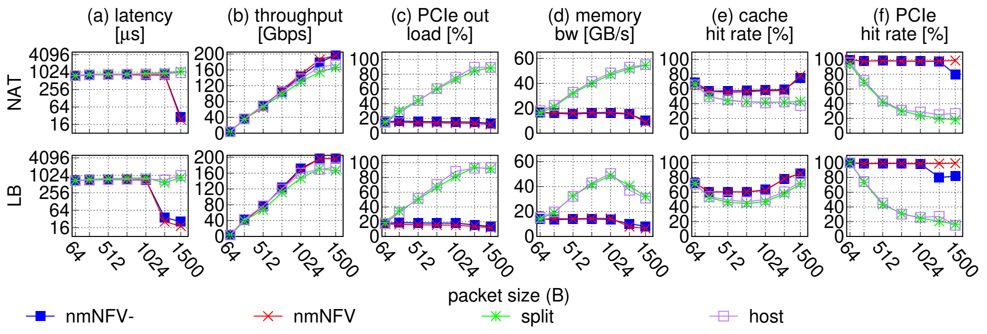


Figure 2.8: Our approach enables efficient 200 Gbps processing for large packets. Small packet workloads are always CPU bound.

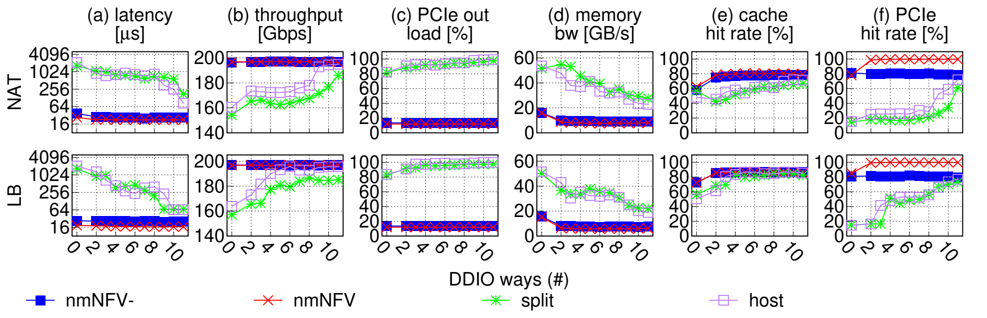


Figure 2.9: A system with DDIO disabled and nicmem enabled outperforms the same system with maximum DDIO and no nicmem.

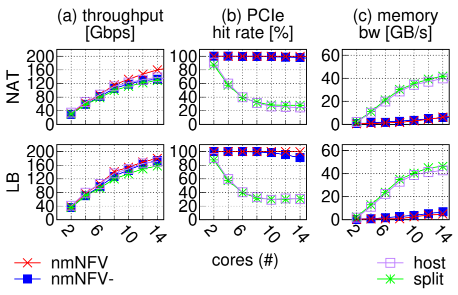


Figure 2.10: Performance with real packet trace from CAIDA.

Cores Figure 2.6 shows the results for LB and NAT scalability from 2 to 14 cores. Host and split fall short of reaching line rate throughput and as a result their latency increases with the number of cores. The reason is DDIO thrashing of the LLC due to the leaky DMA problem. The DDIO hit rate declines and memory bandwidth increases as we increase the core count.

Both nmNFV variants, in contrast, achieve line-rate throughput at 12 and 14 cores for LB and NAT, respectively. When approaching line-rate, improvements manifest in reduced latency. As expected, both variants improve PCIe hit rate, PCIe outbound utilization, memory bandwidth, and CPU cache hit rate. We remark that split and nmNFV- use two scatter-gather entries compared to one for nmNFV and host, and as result their performance is lower.

Real trace We repeated the experiment above with the first million packets from a 2019 real-world CAIDA packet trace form the Equinix NYC monitor [182]. The trace we used contains 43261 unique source IPs and 58533 unique destination IPs with an average packet size of 916 bytes (small and large packet clusters). Figure 2.10 presents the results. Due to limitations in our load generator (T-Rex), we cannot measure latency so we focus on throughput. Both nmNFV- variants outperform base by up to 28%. The results are similar to Figure 2.6 with lower throughput for all as small packets increase the load on the CPU without benefiting from nicmem.

Rx Descriptors To examine the performance impact of growing Rx ring sizes, which are necessary to handle packet bursts (see §2.1.4), we measure the performance with Rx ring sizes between 32 and 4096 (Figure 2.7). We observe that increasing ring size decreases throughput by up to 15% and 20% for LB and NAT, respectively. Latency grows exponentially as LB and NAT fail to meet the offered load at 256 and 128 Rx descriptors, respectively. This is preceded by a sharp decline in PCIe hit rate, as the total Rx ring buffers exceed available LLC space for DDIO: $256 \times 14 \times 1500 \approx 5\,MiB > 4\,MiB$ available to DDIO. Interestingly, host and split NAT performance diminishes before exceeding DDIO LLC capacity. We observe that NAT's higher LLC access rate and occupancy are responsible, as NAT uses two cache entries per flow, i.e., one for each direction and LB uses only one. Base and split application cache hit rate plummets from 83% to 27% and memory bandwidth grows from 5 GB/s to 55 GB/s, in correlation with PCIe hit rate, which reaffirms the importance of LLC locality and low memory bandwidth to NF performance.

Packet Size Figure 2.8 shows the performance with packet sizes between 64B and 1500B. We observe that for both nmNFV variants throughput and latency is similar or better than host and split for all packet sizes. Both variants achieve better throughput for packets larger than 1024 B. Both variants also improve memory bandwidth, PCIe utilization, and PCIe hit rate for all packet sizes.

DDIO Figure 2.9 shows performance with various DDIO cache way allocations. To control DDIO cache ways, we use the DDIOTune fastclick element developed by Farshin et al. [164]. The results show that a system with DDIO disabled and nicmem enabled outperforms the same system with maximum DDIO assigned LLC ways and no nicmem in latency (22 µs vs. 84 µs) and throughput (197 Gbps vs. 195 Gbps).

As expected, adding DDIO ways improves the performance of host and split; host achieves line-rate at 5 and 9 cache ways for LB and NAT, respectively. We observe that even though host and split reach line-rate, their latency remains as high as 64 µs, while the latency of nmNFV- and nmNFV is 26 µs and 22 µs, respectively. Nicmem improves latency due to its lower PCIe utilization, and inlining improves latency further by avoiding an extra PCIe round-trip to fetch the header.

Curiously, nmNFV- PCIe hit rate is constant at 80% for all DDIO cache way settings. Meanwhile, nmNFV benefits from 100% PCIe hit rate. This suggests that packet header buffers are evicted from the cache before they are reused by DDIO; inlining avoids this problem as it reduces the number of buffers in-use.

2.4.4 Insufficient NIC Memory Capacity

Nicmem capacity changes between devices and it may not suffice to feed all per-CPU queues and even the split rings approach may spill over data into hostmem queues. We therefore re-test NAT performance when varying the available nicmem by controlling the number of nicmem queues.

Figure 2.11 presents the results. We observe that a single nicmem queue (out of 7 in total per NIC) drastically improves latency and throughput as it eliminates the PCIe bottleneck discussed in §2.1.3. Meanwhile, increasing nicmem queues further reduces memory bandwidth, DDIO contention, and improves application LLC hit rate (not shown).

2.4.5 Cost of Accessing NIC Memory

In this section, we compare the cost of CPU access to nicmem in comparison to hostmem. Figure 2.12 compares the copy rate within hostmem with the copy rate from hostmem to nicmem, and vice versa. Our experiment measures the throughput of 100 copy iterations within hostmem to the same copy loop with source/destination in nicmem.

We observe that the results differ greatly between the two. On the one hand, the rate of copy into nicmem decreases as the source buffer grows in size, from 4.0x for buffers

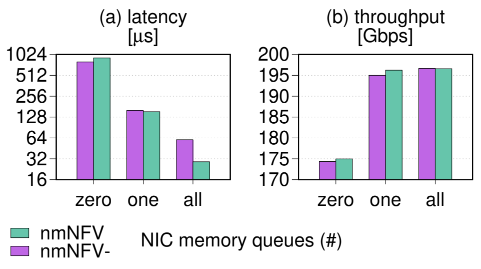


Figure 2.11: NFV performance improves with 1/7 nicmem queues.

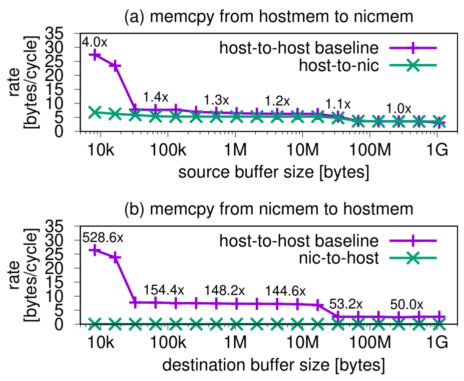


Figure 2.12: Cost of copy between hostmem and nicmem.

in L1 (32 KiB) to 1.0x for non-cached data. On the other hand, the rate of copy from nicmem to hostmem incurs between 528x and 50x overhead. This is because nicmem is marked for write-combined caching, which permits the caching of writes, but prevents the caching of reads.

2.4.6 Key-Value Store

We use two workloads to evaluate nmKVS using 4 cores: 100% get requests (best case scenario), and various get/set ratios to show the affect of costly nicmem sets.

100% Get Workload Figure 2.13 shows nmKVS performance with 100% get load, varying the load directed at hot items. This is the best-case scenario, as nicmem is never accessed directly by the CPU when processing get requests (§2.2.2); response packet descriptors only reference data in nicmem that the NIC fetches when sending packets to the wire.

The results show that increasing the portion of requests directed at the hot items increases the benefit of nicmem, and larger nicmem provides greater benefits. We observe that (C2) outperforms (C1) for two reasons: (1) the 256 KiB hot area causes an imbalanced load distribution between the 4 server cores, underutilizing one core, and (2) the 64 MiB hot area exceeds the size of host LLC and therefore, in this case, hostmem does not benefit from caching. Overall, nmKVS improves MICA throughput by up to 21% in (C1) and 79% in (C2), improves latency by 14% in (C1) and 43% in (C2), and tail latency by 21% in (C1) and 42% in (C2). We also measure unloaded nmKVS latency using a modified closed-loop MICA client (not shown). We observe analogous results, nmKVS improves latency and throughput by up to 6% and 19% for (C1) and (C2), respectively.

We remark that nmKVS improves throughput more than nmNFV while using nicmem only on transmit. The reason is that MICA must copy data to avoid zero-copy races (§2.2.2), an overhead we avoid in nmKVS. Meanwhile, in NFV systems, the baseline performs no copies and therefore the gap is smaller.

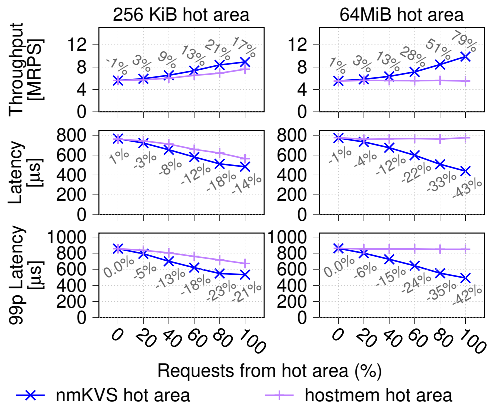


Figure 2.13: MICA 100% get throughput and latency. Labels show improvement over hostmem.

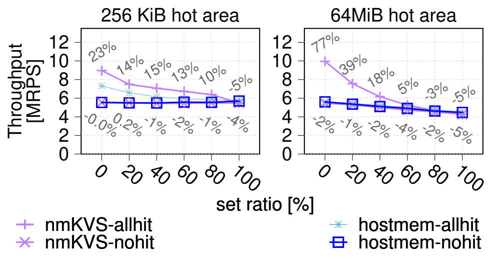


Figure 2.14: MICA set+get throughput using 4 cores. Labels show nmKVS relative to the corresponding baseline.

Mixed Workload Figure 2.14 shows the throughput of nmKVS under various get/set request ratios. Recall that nmKVS sets are more costly as they need to write data in both hostmem and nicmem to avoid zero-copy races, therefore 100% sets is the worst case scenario for nmKVS. To show this scenario, we direct all sets to the hot area. We then consider two types of workloads, one in which all gets are served from the hot area (best case), denoted “allhit”, and another where all gets go to non-hot area (worst case), denoted “nohit”. Then, we compare (C1) and (C2) as above.

We observe that the nmKVS is no more than 5% worse in both (C1) and (C2) indicating that most set operations write into non-cached memory, which we confirm by observing 70% cache misses using 100% sets. In the best case, throughput improves by up to 23% and 77% for (C1) and (C2), respectively. In (C1), serving gets from hot hostmem area improves throughput by up to 31%, in contrast to (C2) which performs the same regardless of whether gets are served from the hot area. This is due to the larger than LLC hot area in (C2).

2.5 NIC Memory and NFV Acceleration

In this section we contrast our approach to the common use of NIC memory today in the context of data-mover NFV applications (i.e., NFV acceleration) to show the tradeoff between the two approaches as a function of the number of flows.

Today, in NFV acceleration, NIC memory stores per-flow state, such as packet steering rules used for NFV acceleration. In particular, products such as NVIDIA ASAP2 [193] will group packets to flows, apply actions such as count, modify, encapsulate, and decapsulae packet headers, and then send packets out (i.e., hairpin); all in ASIC without software involvement. This approach works best when all per-flow states fit inside NIC memory, but performance degrades as the number of flows grows. In contrast, nmNFV NIC memory utilization is independent of the number of flows, and it scales as well as baseline CPU based NFs while improving their performance.

Figure 2.15: NFV scalability to large numbers of flows. The labels show the difference between nmNFV and accelNFV.

To compare the performance of ASAP2 per-flow acceleration with nmNFV, we run an NF that counts the number of bytes and packets for each flow, while varying the number of flows. We implement this NF by modifying DPDK's l3fwd, and run it on two CPU cores. We also implement and run this NF in NIC ASIC by using DPDK's `rte_flow` match and action rules together with two pairs of queues operated by NIC hardware in hairpin mode; we call this accelNFV.

Figure 2.15 shows the resulting throughput, latency, CPU utilization, and NIC cache misses. The figure shows that accelNFV is idle even when processing 100Gbps, as NIC ASIC processes packets without interfering with the CPU. We also observe that increasing the number of flows beyond on-NIC memory capacity, increases the time to process packets as the number of NIC context misses requires fetching and also evicting contexts to hostmem. When packets are processed too slowly, the Rx ring overflows causing significant packet loss and increased latency. Increasing the number of rings would not mitigate this problem, because it does not increase NIC processing speed through parallelism. In fact, performance will degrade farther as additional rings will also contend over NIC memory.

2.6 Related Work

Packet inlining Splitting metadata (headers) from data (payload) is known to improve performance via better caching of CPU accessed metadata, and lower read amplification from prefetching and DMAing. This idea has been applied to log structured merge trees on SSDs [194], to sorting data [195], and with small request-response packets [196]. The contribution of this work is in combining these techniques with nicmem to efficiently accelerate data-mover applications.

Header-data split Previous work proposed storing NF packet payload on network switch memory [197]. Storing payload on nicmem is preferable because the NIC: (1) has more memory per host; (2) requires no coordination with switches on dropped packets; (3) allows for CPU offloading (e.g., checksum), which is impossible with switch parking; and (4) simplifies debugging as compared to switches.

NIC memory in RDMA NIC memory has been used exclusively for RDMA so far [198]. In RDMA, it improves the latency of atomic operations [199], and small message transfers as we demonstrated in §2.1.2.

Chapter 3

Autonomous NIC Offloads

CPUs routinely offload to NICs network-related processing tasks like packet segmentation and checksum. NIC offloads are advantageous because they free valuable CPU cycles. But their applicability is typically limited to layer<=4 protocols (TCP and lower), and they are inapplicable to layer-5 protocols (L5Ps) that are built on top of TCP. This limitation is caused by a misfeature we call "offload dependence," which dictates that L5P offloading additionally requires offloading the underlying layer<=4 protocols and related functionality: TCP, IP, firewall, etc. The dependence of L5P offloading hinders innovation, because it implies hard-wiring the complicated, ever-changing implementation of the lower-level protocols.

We propose "autonomous NIC offloads," which eliminate offload dependence. Autonomous offloads provide a lightweight software-device architecture that accelerates L5Ps without having to migrate the entire layer<=4 TCP/IP stack into the NIC. A main challenge that autonomous offloads address is coping with out-of-sequence packets. We implement autonomous offloads for two L5Ps: (i) NVMe-over-TCP zero-copy and CRC computation, and (ii) https authentication, encryption, and decryption. Our autonomous offloads increase throughput by up to 3.3x, and they deliver CPU consumption and latency that are as low as 2.4xB and 1.4xB, respectively. Their implementation is already upstreamed in the Linux kernel, and they will be supported in the next-generation of NVIDIA NICs.

3.1 L5P Acceleration

Considerable effort went into L5P acceleration. Here, we make the case for autonomous NIC offloads by categorizing existing approaches and showing that our proposal fills an important missing piece in the L5P acceleration design space. We focus on L5P over TCP since these are most challenging, but similar arguments could be made for lower layer network stack functionality.

Figure 3.1 depicts the categories. L5P acceleration has two flavors: software- (§3.1.1) and hardware-based. The latter occurs on- (§3.1.2) or off-CPU via specialized accelerators (§3.1.3) or NICs. Existing NIC acceleration requires implementing TCP and lower protocols on the NIC, which we thus call dependent NIC offloading (§3.1.4). In con-

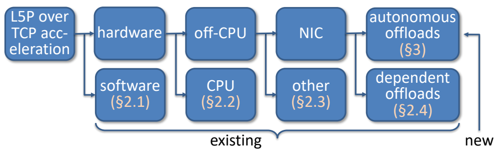


Figure 3.1: Categorizes of L5P acceleration. Autonomous NIC offloads do not require offloading the entire network stack.

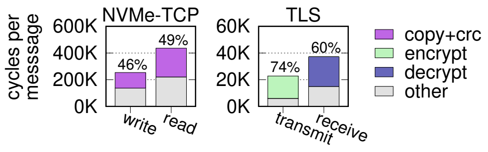


Figure 3.2: L5P overheads. NVMe-TCP and TLS use 256K and 16K messages, respectively. Labels show how many cycles out of the total are compute-bound and offloadable; see §3.5 for more details.

trast, our NIC provides autonomous offloading, which keeps all protocols in software, and which strictly improves performance per dollar (§1.1.2).

3.1.1 Software Acceleration

Software acceleration reduces overheads imposed on L5Ps by OS abstractions. It does not accelerate actual L5P computations, such as encryption, compression, and error detection.

In-kernel L5P implementations work around OS overheads. Examples include DRBD [200], SMB [201], NBD [202], iSCSI [203], and more recently TLS [74] and NVMe-TCP [73]. They reside alongside the OS's TCP/IP and improve performance through cross-layer data batching [72], fusing data manipulations [204], controlling flow group scheduling [205] and CPU scheduling. Notably, they eliminate system call overheads [206].

Instead of moving the entire L5P into the kernel, specialized software stacks bypass certain kernel overheads by: utilizing hardware queues exposed to userspace [207,208]; implementing the TCP/IP stack in userspace [209–213]; replacing the POSIX API abstractions [210,214,215]; and exploiting knowledge of L5P workloads [75,76].

In contrast to software, hardware acceleration targets data-intensive compute-bound processing, which accounts for much of the L5P cycles. Figure 3.2 measures these cycles in four in-kernel L5P workloads: (1) NVMe-TCP client write (compute: CRC of outgoing L5P message m); (2) NVMe-TCP client read (verify CRC of incoming m and copy its content to OS block layer); (3) TLS transmit (encrypt m); and (4) TLS receive (decrypt m). We see that the CPU spends 46%–74% of its cycles on the compute-bound part, even though these L5Ps are in-kernel. Such overhead can only be reduced via hardware acceleration. Thus, software and hardware accelerations are symbiotic, and L5Ps may benefit independently from both.

3.1.2 On-CPU Acceleration

On-CPU acceleration occurs via an in-core hardware implementation invoked by dedicated instructions. In the context of our above examples, it is available in commercial CPUs for AES encryption and decryption [78], and SHA and CRC32 digests [80,82,216, 217]. Dedicated instructions yield 2x–30x speedups [218,219] but might still account for a significant fraction of L5P processing cycles. For instance, the results in Figure 3.2

Table 3.1: Encryption bandwidth (MB/s) of AES-NI (on-CPU) vs. QAT (off-CPU) accelerators. Results for 16 KB blocks with 1 or 128 threads using a single core (2.40 GHz Intel Xeon E5-2620 v3 CPU).

cipher	*QAT 1*	*QAT 128*	*AES-NI 1*
AES-128-CBC-HMAC-SHA1	249	3144	695
AES-128-GCM	249	3109	3150

are obtained with on-CPU accelerators, and yet the accelerated computations take up to 49% and 74% of NVMe-TCP and TLS message processing cycles, respectively.

3.1.3 Dedicated Off-CPU Accelerators

Off-CPU acceleration *offloads* part of the L5P computation from the CPU to another device accessible via PCIe or the on-chip interconnect. Offloading aims at freeing CPU cycles that would otherwise be devoted to the computation, allowing the CPU to do other work instead. We distinguish dedicated accelerators, discussed here, from on-NIC offloads (§3.1.4).

Dedicated off-CPU accelerators exist for various computational operations, such as: (de)compression, (a)symmetric encryption, digest computation, and pattern matching [83, 220–222]. Such off-CPU acceleration still requires some CPU work for each computation, to invoke the accelerator and retrieve the results. This work incurs latency and overhead that depend on the amount of data moved, the interface of the accelerator, and its location in the non-uniform DMA topology [84,85,100]. As a result, off-CPU accelerators can struggle to outperform on-CPU accelerators [223]. Realizing benefits from off-CPU acceleration thus frequently requires re-engineering applications to eliminate blocking operations and/or using multiple threads, all so as to keep the CPU busy while waiting for the accelerator [84].

To illustrate, Table 3.1 compares the throughput (single core OpenSSL speed test) of Intel off-CPU QuickAssist Technology (QAT) [220] accelerated cryptographic operations to on-CPU AES-NI acceleration [78]. For QAT, we show single- and multi-threaded clients, where the latter uses threads to overlap waiting for QAT with useful work. When running AES128-CBC-HMAC-SHA1 (AES in cipher block chaining mode, authenticated by SHA1 hash-based message authentication code), AES-NI does not accelerate the SHA1 computation. Thus, single-threaded QAT throughput is 2.7x lower than AES-NI, but 128-thread QAT outperforms AES-NI by 4.5x. In contrast, for AES128-GCM (AES in Galois/counter mode), single-threaded QAT throughput is 12.5x lower than AES-NI, and 128 QAT threads sharing the core only yield comparable throughput to single-threaded AES-NI. An actual application might require substantial re-engineering to support this level of threading.

3.1.4 Dependent NIC Offloads

As opposed to dedicated off-CPU accelerators, NIC offloads impose neither additional data transfers, nor more CPU work. CPUs operate NICs in any case, and data flows through them in any case, making them ideally positioned for data offloading. The problem is that all existing L5P NIC offloads [14, 86, 87] are *dependent*. Namely, they

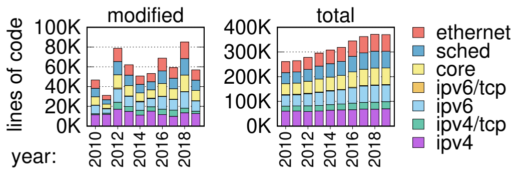


Figure 3.3: Per-year Linux kernel LoC of TCP/IP processing code.

require the NIC to handle the L5P, which then requires the NIC to also implement the underlying layer≤4 functionality in hardware, including TCP/IP and related subsystems like firewalls and tunneling.

Dependent offloading is thus undesirable. Whereas data functions (like CRC) are relatively simple and well-suited for hardware, TCP/IP stacks are complex, evolving, interact with many OS subsystems, and incur considerable maintenance costs. To illustrate, Figure 3.3 shows the yearly number of lines of code (LoC) in Linux's TCP/IP stack: modified and in total. The code is constantly changing, with 5–25% LoC modification in each component, each year, for the past decade. Having to additionally support NIC-based TCP/IP stacks would further increase this maintenance burden.

For these reasons, Linux kernel engineers resist supporting existing NIC TCP offload engines (TOEs) [93, 94]. Microsoft has deprecated TOE support for similar reasons [95]. And several operators recommend disabling TOEs due to performance issues and incompatibilities with OS interfaces [224–226].

Importantly, TOEs hinder innovation and are ill-suited for users who develop their network stack [227]. For instance, Netflix has made the following statement regarding TOEs [228]:

> *"TOEs are not a preferred solution for Netflix content delivery because we innovate in the protocol space [to improve] our customers' quality of experience (QOE). We have a team of people working on improvements to [the OS's] TCP and they have achieved significant QOE gains [...]. With the TCP stack sealed up in an ASIC, the opportunities for innovation [...] are quite limited. We also have concerns around potential security issues with TOE NICs."*

The security concerns arise as TCP/IP stacks are complex and might have security bugs [91,92], which can be easily and quickly fixed/hot-patched in software, but not in hardware.

We remark that we focus on TCP, as it is the most widely used protocol, and it handles reordering and loss in byte streams, which is *the* challenge for autonomous offloads. But simpler level-4 protocols are also in scope. For example, FlexNIC [229] dependently offloads with DCCP [230], implementing, e.g., DCCP's acks logic in the NIC; an autonomous offload would utilize the OS logic instead.

3.2 Autonomous Offloads

Autonomous offloads consist of a software/NIC architecture for moving data between L5P memory and TCP packets, optionally transforming or computing over the data. This architecture offloads data-intensive processing to the NIC, without having to migrate the entire TCP/IP stack into the NIC. Specific offload capabilities are cast into NIC silicon and are available for relevant L5P software as a NIC feature.

Table 3.2: Autonomous offload properties and associated limitations. (We are unaware of any non-constant size state computation or protocol.)

property	*limits the offload of (example)*
size-preserving on transmit	encapsulation and compression
incrementally computable	cipher block chaining (CBC)
constant size state	none
plaintext magic pattern	SSH's encrypted headers

Autonomous offloads target L5P software that can communicate directly with the NIC driver, e.g., in-kernel L5Ps or userspace TCP/IP stacks. High-performance L5P software already adopts this design (§3.1.1). The main idea of the L5P-NIC collaboration is to process L5P messages in the NIC transparently to the intermediating TCP/IP stack. The NIC performs the offload on in-sequence packets, with some help from L5P software on out-of-sequence packets.

Offloading is possible for operations and L5Ps that satisfy certain preconditions (summarized in Table 3.2). Offloadable operations must be *size-preserving* for seamless interoperation with software TCP/IP on transmit, while on receive we can work around this precondition in some cases (§3.2.1). Offloadable operations must be *incrementally computable* over any byte range of an L5P message, given only some *constant-size state* and access to packet data (§3.2.2). In particular, the offload cannot assume L5P message alignment to TCP packets. Offloadable L5P messages must contain *plaintext magic pattern and length* fields to identify and track messages speculatively on the wire. The offload will rely on these fields to recover after loss and reordering on receive (§3.2.3).

These preconditions are satisfied by most of the common data-intensive operations [71]: (1) copying, (2) encryption and decryption, (3) digesting and checksumming, and some (4) deserialization and (5) decompression methods. Most L5Ps meet our requirements as well. Examples include (1) HTTP/2 [66] (encryption/deserialization/decompression); (2) gRPC [69] and Thrift [68] (copy/deserialization); (3) iSCSI [203], NBD [202], and SMB [201] (copy/encryption/digest); (4) Memcached [48] and MongoDB [70] (copy).

The following presents the high-level ideas of autonomous offloading and its preconditions. We detail the design in §3.3.

3.2.1 Data Manipulation

An operation can be offloaded on one or both of the send/receive paths. To offload an operation when sending, L5P software "skips" performing the offloaded operation, thereby passing the "wrong" bytes down the stack to the NIC. The NIC performs the operation, resulting in a correct message being sent on the wire. For TCP-transparency, we require the offloaded operation to be *size-preserving*: it must never add or remove bytes from the stream. Were the operation to add/remove bytes from the stream, the NIC would have to handle these bytes' retransmission, acknowledgment, congestion control, etc., as the OS TCP stack is unaware of these bytes (Figure 3.4).

When receive offloading, the NIC parses incoming L5P messages within TCP packets, performs the offloaded operation, and passes packets pointing to partially pro-

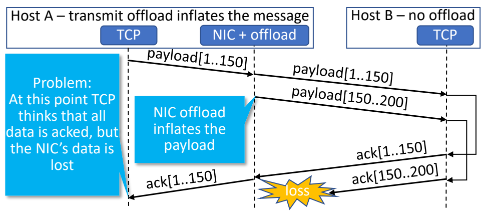

Host A – transmit offload inflates the message
TCP
NIC + offload
Host B – no offload
TCP
Problem: At this point TCP thinks that all data is acked, but the NIC's data is lost
NIC offload inflates the payload
payload[1..150]
payload[1..150]
payload[150..200]
ack[1..150]
ack[150..200]
ack[1..150]
loss

Figure 3.4: The problem with non-size-preserving offloads.

cessed L5P messages up the stack to L5P software. For TCP-transparency, we must preserve packet sizes observed by TCP, which is simple for size-preserving offloads. But, in contrast to the send side, receivers can offload non-size-preserving operations by DMAing offload results to pre-allocated L5P destination buffers while also DMAing the original unmodified data to the NIC driver receive ring. Consequently, receive-offload can be non-size-preserving, provided that L5P software can predict its output's size and prepare buffers for it.

3.2.2 In-Sequence Packet Processing

We require that the offloaded operation can be performed over any byte range of an L5P message (i.e., in-order TCP packets of any size), given only some constant-size state. The state is composed of *dynamic* and *static* components. The dynamic state is a function of (1) the previous bytes in the current message and (2) the number of previous messages. It is (conceptually) updated after each packet (byte) is processed. The static state is fixed per-request or per-connection, e.g., TLS session keys or NVMe-TCP host read response destination buffers.

The above properties allow the NIC to perform the offload without having to buffer packets until obtaining a full L5P message, which is impractical (e.g., because messages can be big, potentially exceeding the TCP receive window). Specifically, the NIC maintains per-flow contexts (for both outgoing and incoming flows) holding the state required to perform the operation on the next *in-sequence* TCP packet. Once that packet is handled, the NIC updates the flow's (dynamic) state.

Our requirements are satisfied by most L5P data-intensive operations, which typically process the current L5P message independently of the payload of previous messages [64, 65, 67–69, 231]. The requirements mainly preclude offloading of operations such as AES cipher block chaining (CBC), which operate on fixed blocks and not an arbitrary range. However, modern ciphers, such as AES-GCM and ChaCha20-Poly1305, satisfy our requirements.

3.2.3 Out-of-Sequence Packet Processing

Receive: The NIC cannot perform the offloaded operation on an out-of-sequence packet. It also cannot buffer it until the in-sequence packet arrives. Instead, we fall back on L5P software to perform the operation: an out-of-sequence packet is passed to the OS, and L5P software performs the operation on its bytes when it receives the message containing the packet. Such messages, in which the operation was offloaded on some/none of the bytes, are called *partially/fully un-offloaded*. To resume offloading, the NIC *resynchronizes* itself into knowing the next expected TCP packet and its associated operation state, as explained next.

Resync: We require that L5P message headers contain (1) at least one *magic pattern* plaintext field that identifies a message header on the wire and (2) a length field. Combined, these properties enable a *hardware-driven* NIC context resynchronization process, which begins when the NIC receives out-of-sequence data and loses track of the flow's state. To resynchronize, the NIC speculatively identifies an L5P message in the incoming byte stream by the magic pattern, and confirms this identification with the L5P software. While waiting for the L5P's reply, the NIC keeps track of incoming messages by using the L5P header length field to derive the TCP sequence number of the next expected message (where another magic pattern should appear). Once the L5P confirms the speculation, the NIC can resume offloading from the next L5P message, since the dynamic state at message boundary depends only on the number of previous messages, which is included in the L5P's confirmation (see §3.3.3).

Transmit: To perform the offloaded operation on a retransmitted outgoing packet, the NIC's dynamic state is *recovered* to the correct state for that packet by the NIC driver with the help of the L5P software. To this end, the L5P software must store the state for an L5P message until the TCP acknowledgment of all of its packets.

3.3 Design

This section describes the software and hardware designs that together form the autonomous NIC offloads architecture. We first describe the software and hardware interfaces (§3.3.1), followed by handling of transmitted (§3.3.2) and received (§3.3.3) packets, for both in-sequence and out-of-sequence (OoS) data.

3.3.1 Interfaces

The NIC maintains a per-flow *HW context*, which holds the state required to perform the offloaded computation for a specific packet of the flow (typically, the next in-sequence packet). Each HW context contains: (1) *tcpsn*, the TCP sequence number that this context can offload; (2) the L5P message type, length, and offset within the message at *tcpsn*; and (3) L5P state required to perform the offload, such as cipher keys. A context also stores a flow identifier, e.g., a TCP/IP 5-tuple.

L5P–NIC driver interface The NIC driver provides an interface to the L5P software for context creation, destruction, and recovery (Listing 3.1). After the L5 handshake is complete, the L5P calls *l5o_create* with the inputs required to process the next message in the stream (*l5_state*), and the TCP sequence number of the first byte in that message (*tcpsn*). To stop the offload, the L5P calls the *l5o_destroy* method. Offloading is typically terminated when the socket is destroyed.

```
l5o* l5o_create(sock*, l5_state*, u32 tcpsn)
void l5o_destroy(l5o*)
rr_state_id l5o_add_rr_state(l5o*, rr_state*)
void l5o_del_rr_state(l5o*, rr_state_id)
void l5o_resync_rx_resp(l5o*, u32 tcpsn, bool res)
```

Listing 3.1: Operations the NIC driver provides to the L5P.

```
l5_msg_state* l5o_get_tx_msgstate(sock*, u32 tcpsn)
void l5o_resync_rx_req(sock*, u32 tcpsn)
```

Listing 3.2: Operations the L5P provides to the NIC driver.

In Request-Response (RR) protocols, offloading the computation for an incoming message (a response) requires the NIC to associate the message with the request that triggered it. For example, an offload copying the response payload directly to an application buffer needs to know the buffer's address. To this end, the NIC can internally map incoming messages to the required offloading state, which is configured using the *l5o_add_rr_state* method. The L5P provides this state to the NIC before sending the request, and deletes it after the response is received using the *l5o_del_rr_state* method.

The *l5o_resync_rx_resp* method is used for receive-side context recovery from OoS packets (§3.3.3). The L5P also provides an interface to the driver (Listing 3.2) for context recovery, which we discuss in §3.3.2 and §3.3.3.

Driver–NIC interface Offload-related commands are passed to the NIC via special descriptors, which are placed into the flow's usual send ring to ensure ordering. The NIC passes information to the driver through descriptors in the flow's receive ring. Both rings are accessed through DMA.

3.3.2 Transmitted Packet Processing

To send application data, the L5P encapsulates it into L5P messages, preserving all fields of the message that appear on the wire, including fields that are filled by the offload (e.g., CRC). The L5P then hands the data for transmission to the next layer protocol. Typically, this protocol is TCP, but it can also be an L5P (e.g., TLS), as our offloads compose (§3.4.3).

When the NIC driver is handed a TCP packet for transmission, it must figure out if the packet is in- or out-of-sequence with respect to the NIC's flow context. To this end, it extracts the context ID from the packet's metadata; this ID is passed down from the L5P, which obtained it on context creation. The driver shadows the NIC's context in software, and so it can check the packet's TCP sequence against the context's expected TCP sequence to identify OoS packets. If the packet is OoS, the driver recovers the NIC's context, as described below. Next, the packet is posted to the NIC's send ring, tagged with the HW context ID (which saves the NIC from looking up the HW context based on the packet fields). Finally, the NIC performs the offloaded operation and sends the packet.

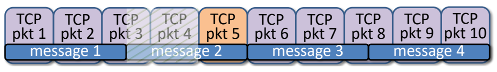


Figure 3.5: Packet 5 is retransmitted; the data required to offload it is marked with a diagonal pattern.

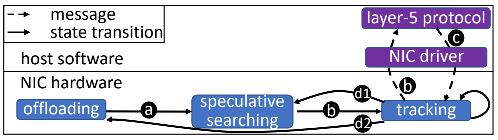


Figure 3.6: State-machine for L5P offload recovery from reordering.

Context recovery for OoS data To enable offloading of transmitted OoS data, the driver recovers the NIC's context to match the packet. As noted in §3.2, we assume that the state required to perform the offload can be obtained from the packet's L5P message (Figure 3.5). The driver obtains this state using the *l5o_get_tx_msgstate* upcall to the L5P, and then passes it to the NIC via a special descriptor. The driver also updates the context's expected TCP sequence (in both HW and its shadow) to match the packet's TCP sequence.

To answer *l5o_get_tx_msgstate* calls, the L5P software must maintain a map from TCP sequence numbers to their corresponding L5P messages (in our experience, this takes $\approx$ 200 LoC). The L5P holds a reference to the buffers which contain transmitted L5P message data, similarly to how TCP holds a reference to all unacknowledged data. The L5P releases its reference when the entire message is acknowledged.

3.3.3 Received Packet Processing

The NIC only performs the offload for in-sequence packets. OoS packets are handled by software. When a packet with a valid TCP/IP checksum arrives, the NIC looks up its flow's context.[1] If a context is found, the NIC performs the offloaded operation if the TCP sequence numbers of the context and the packet match. Both offloaded and un-offloaded packets are passed to the driver, with an indication (in their descriptors) of whether offloading was performed. The driver passes the packet and the offload result as metadata up to the network stack, which takes care not to coalesce packets with different offload results. The L5P software reads L5P messages handed to it by TCP packet-by-packet and skips computing the offloaded function if all packets are offloaded. Otherwise, the L5P must perform the relevant data manipulation itself.

Out-of-sequence packets The NIC never performs the offload on an OoS packet, but it processes such packets in an attempt to get back in sync with the TCP stream. The NIC cannot wait for the packet Q with the sequence number it expects to arrive, because that would require buffering all the flow's packets that arrive in the meantime. (Without such buffering, packets with sequence numbers higher than Q may reach the OS TCP stack while the NIC waits for Q, leaving it unaware of the next expected sequence number on the wire.) We thus need to resync without waiting for Q to arrive.

It follows from our requirement that offload state depend only on the previous bytes of a message and on the number of previous messages (§3.2.2) that the NIC can resync itself once it sees the *next* L5P message. Thus, when an OoS packet P arrives, the NIC

[1]Similar hardware functionality already exists for LRO and ARFS.

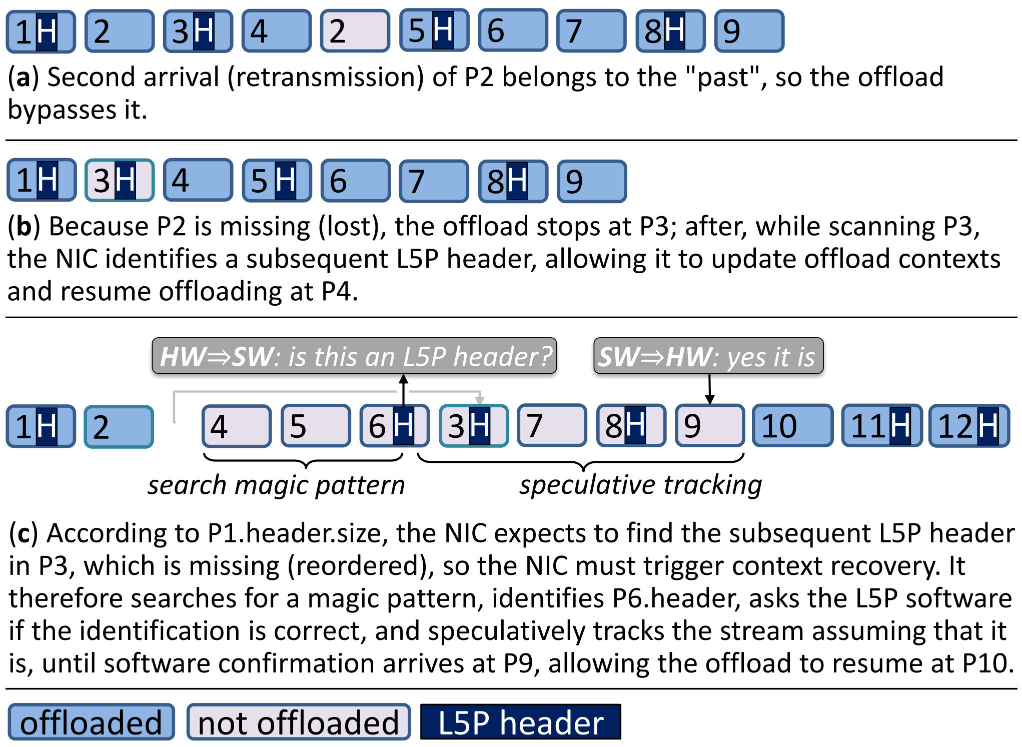
computes the TCP sequence number of the next L5P message *M* by using the length of the current message (which is stored in the context). If the TCP sequence number of *P*, *P.seq*, is before *M*'s sequence number, *M.seq*, then *P* is ignored. If *P* contains *M*'s header, the context is updated to *M*, so that the offloading can resume for the packet following *P*. Otherwise (*P.seq* is after *M.seq*), the NIC cannot resync, as it does not know which (if any) L5P messages appeared after *M*. In this case, the NIC begins a context recovery process in collaboration with the L5P software.

Context recovery A naive *software-driven* approach for context recovery is for L5P software to inform the NIC about the TCP sequences numbers of the messages it receives, thereby allowing the NIC to resync. However, such a scheme is inherently racy: by the time the NIC hears about a message, it may have already started receiving packets of subsequent messages. As a result, the NIC may never be able to successfully recover its context. To avoid this problem, our design employs a *hardware-driven* recovery process, in which the NIC speculatively identifies arriving messages and relies on software to confirm its speculation.

The recovery algorithm is depicted in Figure 3.6. Initially (transition ⓐ), the NIC enters a *speculative searching state*. In this state, whenever a valid TCP packet arrives, the NIC searches for the protocol's header magic pattern (§3.2) in the TCP payload. When found, the NIC requests the software L5P to acknowledge the detected message header TCP sequence number (*tcpsn*) via the NIC driver, which calls *l5o_resync_rx_req* to register the request with the L5P. The NIC also transitions to the *tracking state* (ⓑ). The L5P stores *tcpsn* and waits until the corresponding message is received from the OS TCP stack. The L5P then notifies the NIC whether the message's TCP sequence number matches the NIC's "guess," using the *l5o_resync_rx_resp* method (ⓒ).

Meanwhile, the NIC tracks received messages using the message header's length field, verifying that each subsequent message begins with the magic pattern. If an unexpected pattern is encountered or the L5P response indicates that the NIC misidentified a message header, then the NIC moves back to the speculative searching state (d1). If the L5P response indicates success while the NIC is in the tracking state, then the NIC can resume offloading from the next message (d2).

Example Figure 3.7 depicts the various cases of receive packet processing. If packets arrive in-sequence then all are offloaded. Otherwise, we have the following cases: (a) retransmitted packets (P2) bypass offload and do not affect NIC state; (b) L5P data reordering within the current message (P2). The NIC identifies the next L5P message header (P3), updates its context to expect P4 and continues processing from there. The packet containing the L5P header is not offloaded as it does not match the expected TCP sequence number, but the following packet (P4) does match it; (c) L5P header reordering (P3) causes the NIC to cease offloading. Then, it searches and finds an L5P message header magic pattern (P5) and it requests software to confirm its speculation. We note that it can identify patterns split between packets if they arrive in-sequence. Meanwhile, NIC HW tracks subsequent L5P headers (P8) using header length fields, and verifies their magic pattern. Eventually, L5P software receives the NIC HW request



(**a**) Second arrival (retransmission) of P2 belongs to the "past", so the offload bypasses it.

(**b**) Because P2 is missing (lost), the offload stops at P3; after, while scanning P3, the NIC identifies a subsequent L5P header, allowing it to update offload contexts and resume offloading at P4.

(**c**) According to P1.header.size, the NIC expects to find the subsequent L5P header in P3, which is missing (reordered), so the NIC must trigger context recovery. It therefore searches for a magic pattern, identifies P6.header, asks the L5P software if the identification is correct, and speculatively tracks the stream assuming that it is, until software confirmation arrives at P9, allowing the offload to resume at P10.

Figure 3.7: NIC processing of various OoS packets received from the wire: (a) retransmission, (b) L5P data reordering, and (c) L5P header reordering which triggers OoS recovery. ccc

and the corresponding packet after TCP processing, and then L5P software confirms NIC hardware speculation. Finally, offload resumes on the next packet boundary.

3.4 Implementation

Here, we describe case studies of autonomous offloads targeting in-kernel L5Ps in Linux: NVMe-TCP (§3.4.1), TLS (§3.4.2), and their composition (§3.4.3). The offloads described in this section are (or will be) available in NVIDIA ASIC NICs.

3.4.1 NVMe-Over-TCP Offload

NVMe-TCP [67] is a pipelined L5P which abstracts remote access to a storage controller, providing the host with the illusion of a local block device. In NVMe-TCP, each NVMe [232] submission and completion queue pair maps to a TCP socket. Read/write IO operations use request/response messages called *capsules*, whose header contains a (1) capsule type, (2) data offset, (3) data length, and (4) capsule identifier (CID). The CID field is crucial to correlate between requests and responses, as the controller can service requests in any order. Also, capsules can be protected by a trailing CRC.

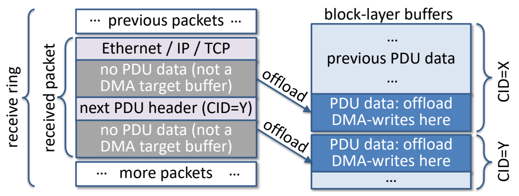


Figure 3.8: NVMe-TCP receive offload. The data is written directly to the block layer according to the CID, instead of to the receive ring.

Offloaded data manipulation We offload the two dominant data manipulation operations of the protocol: CRC32C [81] data digest computation/verification (on transmit/receive) and capsule data copy from TCP packets to block layer buffers (on receive). Note that this copy cannot be avoided with standard zero-copy techniques, as (1) the OS cannot make the NIC DMA directly into application or page cache buffers, since the OS does not know ahead of time which receive ring entry corresponds to which NVMe-TCP response; and (2) even if that were possible, packets contain capsule headers, which do not belong in block layer buffers.

Copy offload The NIC stores a map from CIDs to the corresponding block layer buffers. The map is updated by the NVMe-TCP before it sends a read request. When a response arrives, the NIC DMA writes the capsule payload to the block layer buffers for each offloaded packet, while placing the packet and capsule headers/trailers in the NIC's receive ring (see Figure 3.8). These receive packet descriptors provide all the information necessary to construct a socket buffer (SKB) that points to the received data, including the block layer buffer. When this SKB reaches the NVMe-TCP code responsible for copying capsule payload data to the block layer buffer, the copy is skipped, as the relevant `memcpy` source and destination addresses turn out to be equal. This means that partially- or un-offloaded capsules are handled transparently, with the `memcpy` performed as usual for the remaining un-offloaded parts.

CRC offload On transmit, NVMe-TCP prepares capsules with dummy CRC fields, which the offload fills based on the digest of capsule data of previous in-sequence packets. OoS packets are handled as described in §3.3.2. On receive, the NIC checks the CRC of all capsules in the TCP payload of in-sequence packets. It reports a single bit to the driver (along with the packet descriptor), which is set if and only if all capsules with the packet pass the CRC check. The driver sets a *crc_ok* bit in the SKB of the received packet[2] according to the NIC's indication, and hands the SKB to the network stack. When NVMe-TCP receives a complete capsule, it skips CRC verification if the *crc_ok* bits of all SKBs in the capsule are set. Otherwise, it falls back to software CRC verification. Partially- or un-offloaded capsules are thus handled easily.

Magic pattern For speculative searching, we rely on a number of fields from the NVMe-TCP capsule header and trailer to form the pattern and verify it: (1) PDU type: one of only eight valid values (1 byte); (2) header length: well known constant for each

[2]This requires adding a new bit to the SKB.

PDU type (1 byte); (3) header digest: optional CRC32 digest of the header (4 bytes); and (4) data digest: optional CRC32 digest of the data (4 bytes).

Implementation effort Our patches modifying Linux to support the NVMe-TCP offload are under review by the relevant maintainers. The changes in NVMe-TCP are 418 LoC, and another 1755 LoC in the NVIDIA NIC driver.

3.4.2 TLS Offload

The Transport Layer Security (TLS) protocol is an L5P that protects the confidentiality and integrity of TCP session data [64, 65]. A TLS session starts by exchanging keys via a handshake protocol, after which all data sent/received is protected with a symmetric cipher, such as AES-GCM [233].

Application typically use a library that implements TLS. We modify the popular OpenSSL library to use the Linux kernel's TLS (KTLS) data path, which can leverage our offload. OpenSSL's TLS handshake code remains unmodified.

TLS messages are called *records* and are at most 16 KiB in size. A TLS record consists of a header, data and a trailer. The header holds four fields of interest: (1) record type, (2) version, (3) record length, and (4) initialization vector (IV), used by the cipher. The trailer holds the integrity check value (ICV) of the entire record. For each socket send (receive), KTLS encapsulates (decapsulates) the data into records.

Our offload is motivated by TLS 1.3 [65], which support two symmetric ciphers: AES-GCM and Chacha20/Poly1305. We offload AES-GCM [233], as it is the most common TLS cipher [234–238].

Crypto offload On transmit, KTLS prepares plaintext records with dummy ICV fields, and the NIC replaces plaintext with ciphertext and fills the ICV. OoS packets are handled as described in §3.3.2. On receive, the NIC decrypts the payload of each offloaded received packet, and it checks all ICV values within the packet. It reports the result in a single bit to the driver (along with the packet descriptor). The driver sets a *decrypted* bit in the received packet's SKB according to the NIC's indication, and hands the SKB to the network stack. When a complete record is received by KTLS, it skips decryption and authentication if the *decrypted* bits of all SKBs in the record are set. Otherwise, KTLS falls back to software decryption and authentication.

Zero-copy sendfile support KTLS supports the `sendfile` system call. While `sendfile` is typically implemented without copying, KTLS cannot encrypt transmitted page cache buffers in-place, as that would corrupt their content. Instead, standard KTLS `sendfile` encrypts sent data in a separate buffer, allocated for this operation. Our offload *enables skipping this costly allocation*, as KTLS can hand the page cache buffers to the NIC, which encrypts them to the wire instead of in-place. As a result, KTLS with our offload can achieve performance comparable to plain TCP `sendfile` (see §3.5.3). However, the user becomes responsible for not changing files while they are transmitted.

Partial offload Conceptually, the software fallback for partially-offloaded records is to decrypt the non-offloaded packets and authenticate the record while reusing offload results. However, AES-GCM authentication is computed on the ciphertext data, and so performing authentication in software requires re-encrypting the packets decrypted by the NIC. Consequently, handling partial decryption is costlier than full decryption (see §3.5.4).

Magic pattern For speculative searching, we rely on a number of fields from the TLS record header to form the pattern and verify it: (1) record type (1 byte): one of only six valid values[3]; (2) record version (2 bytes): the version is constant after the TLS handshake; and (3) record length (2 bytes): this field must be less than 16 KiB.

Software implementation Our OpenSSL changes adding KTLS support consist of 1381 LoC. Offload support in KTLS is 2223 LoC. Offload support in the ConnectX6-Dx includes 2095 LoC. Our changes have been accepted for inclusion in OpenSSL [239–241] and Linux [242, 243], indicating the relevance of the offload. Others have added KTLS offload support to FreeBSD [244].

3.4.3 NVMe-TLS Offload

Combining NVMe-TCP and TLS offloads is simple, as the layering determines their ordering. NIC HW parsing starts from Ethernet, and proceeds to parse TLS then NVMe-TCP on transmit and receive. In-sequence packet processing remains the same, where each offload is processed independently: on transmit we do NVMe-TCP then TLS; and on receive vice versa. Transmit and receive OoS context recovery are performed independently for each protocol.

3.5 Evaluation

Using microbenchmarks, we measure the overhead of the data-intensive operations that our NIC autonomously offloads (§3.5.1). We then evaluate actual offload performance with macrobenchmarks (§3.5.3), and we quantify the effect of out-of-sequence TCP packets (§3.5.4). Finally, we examine the performance of autonomous NIC offload at scale (§3.5.5).

TLS results are obtained using *real* NVIDIA ConnectX6-Dx ASIC NICs. NVMe-TCP results are obtained via *emulation*, as this offload will only become available in NVIDIA's next-generation ConnectX-7 NICs. We validate the accuracy of our emulation methodology by comparing the performance of real and emulated TLS offloading (§3.5.2).

Our setup consists of a Dell PowerEdge R730 server and an R640 workload generator. The server has two 14-core 2.0 GHz Xeon E5-2660 v4 CPUs, and the generator has two 12-core 2.1 GHz Xeon Silver 4116 CPUs. Both have 128 GB (=4x16 GB) memory, and they run Ubuntu 16.04 (Linux 5.6.0) with hyperthreading and Turbo Boost off to

[3]HW can store an extensible list of these values.

avoid nondeterministic effects. For storage, the server utilizes an Optane DC P4800X NVMe SSD that resides remotely, on the generator.

The machines are connected back-to-back via 100 Gbps NVIDIA ConnectX6-Dx NICs that implement our TLS AES128-GCM crypto autonomous offload.

All the results presented in this section are trimmed means of ten runs; the minimum and maximum are discarded, and the standard deviation is below 3% unless specified otherwise.

3.5.1 Cycle Breakdown

NVMe-TCP When NVMe-TCP reads from a remote drive, recall that it accesses the received bytes twice: (1) when copying them from the network buffers to their designated memory locations; and (2) when computing the incoming capsule's CRC. Figure 3.9 shows how long these two operations take out of the total of an individual I/O request. We use fio [245] to generate random read requests of different sizes (title of subfigures) and to vary the number of outstanding requests (I/O depth along the x axis). The left and right y axis labels show per-request duration in cycles and the relative cost of the copy+CRC overheads out of the total, respectively. The system is limited to using a single core for all of its activity.

We can see that smaller requests have a potential improvement of 2%–8%, and bigger requests have a potential improvement of 25% (lower parallelism, up to depth=64) to 55% (higher parallelism, as of depth=128). In the latter case, the 32 MiB LLC becomes too small to hold the working set (128 requests times 256 KiB per request = 32 MiB). From this point onward, copying becomes the dominant overhead, as every memory access is served by DRAM.

TLS We similarly measure TLS's offloadable overheads: encryption, decryption, and authentication, which we collectively denote as "crypto" operations. For this purpose, we use iperf [246], which measures the maximal TCP bandwidth between two machines, and which we modified to support OpenSSL. We use 256 KiB messages at the sender and ensure that the server's core always operates at 100% CPU. Recall that each message consists of a sequence of TLS records, which can be as big as 16 KiB.

Figure 3.10 shows the results. Unsurprisingly, bigger TLS records reduce the weight of network stack processing relative to the crypto operations, making the potential offload benefit more pronounced at the right. This outcome is consistent with the fio results. Typically, network stacks operate more efficiently when sending than when receiving, because batching is easier; the receive side has to work harder. Consequently, the potential benefit of offloading is higher for transmitting ($\leq$74%) than for receiving ($\leq$60%).

With real TLS offloading, we find that iperf's single core throughput improves by 3.3x and 2.2x upon transmit and receive, respectively, relative to the non-offloaded baseline. When saturating the NIC with multiple iperf instances, CPU utilization respectively improves by up to 2.4x and 1.7x.

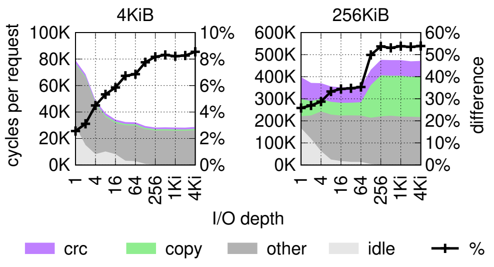


Figure 3.9: NVMe-TCP/fio cycles per random read on the server (drive resides on the generator); "%" shows copy+crc out of the total.

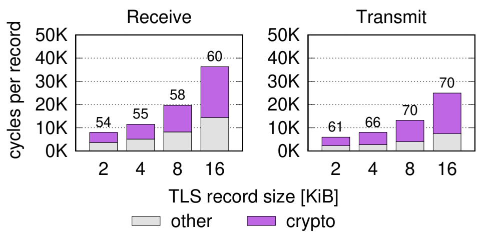


Figure 3.10: Kernel-TLS/iperf per-record cycles when using AES-GCM crypto operations (encryption, decryption, and authentication using AES-NI); standard deviation is between 0%–8.2%.

3.5.2 NVMe-TCP Offload Emulation

We hypothesize that commenting out the software functionalities to be offloaded (without really implementing them in the NIC) yields similar performance to real offloading. We verify this hypothesis with TLS offloading. We find that the "other" component in Figure 3.10 is an accurate performance predictor for our new TLS offload capability: at most 7% difference between the real and predicted improvements in all cases.

We use this finding to emulate NVMe-TCP offloading by: (1) setting the value of all stored data to be a repetitive sequence of an 8-byte "magic" word (0xCC...CC); (2) modifying NVMe-TCP receive-side to *refrain* from copying incoming "magic capsules" (that start with the magic word) to their target buffers, and also; (3) skipping CRC computation and verification for magic capsules. Clearly, a block device driver that fails to copy device content to the designated target buffers seems problematic. We next describe how the integrity of our experiments is preserved despite this problematic behavior.

Nginx The subsequent evaluation uses two macrobenchmarks. The first is the nginx http web server [247], configured to serve files from an ext4 filesystem mounted on our NVMe-TCP block device. (Recall that the drive resides remotely, on the workload generator machine.) We pre-populate ext4 with "magic files", which exclusively contain magic word sequences. We set the size of magic files to be an integral multiple of 4 KiB, and we configure nginx clients to only request these files. We also set ext4 read-ahead to the file size, such that there are no block requests that exceed this size.

Neither the kernel of the server machine, nor nginx and its clients care about the content of the files that they send/receive. Ext4 does not collocate metadata within the 4KiB blocks of magic files, so it is indifferent to whether their content is copied to the server's page cache; nginx, which sends this page cache content to its clients, is likewise indifferent to the content; and the clients do not actually use the content either.

Redis-on-Flash The second macrobenchmark we use is Redis-on-Flash (RoF), a key-value store [248, 249] that uses RocksDB [250] as storage backend. RocksDB is incompatible with our emulation. It runs with RoF on the server and uses the NVMe-TCP block device to read and write its internal data structures, which interleave metadata

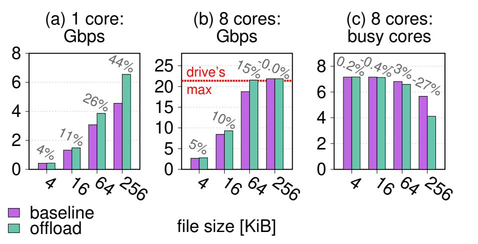


Figure 3.11: Nginx improvements with the NVMe-TCP offload. None of the files reside in the server's page cache (configuration C_1), so the throughput is bounded by the drive's maximal bandwidth. Bar labels show offload improvement over the baseline.

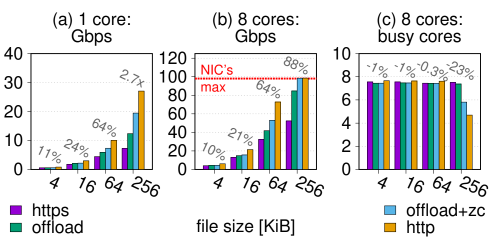


Figure 3.12: Nginx improvements with two TLS offload variants compared to https (baseline encryption) and http (no encryption). All files reside in the page cache (C_2), so throughput is bounded by the NIC line rate. Bar labels show offload+zc improvement over https.

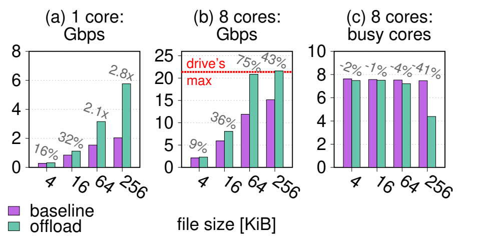


Figure 3.13: Nginx improvements obtained with the NVMe-TLS combined offload.

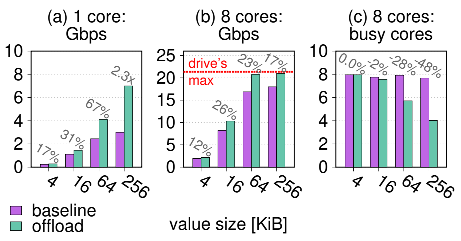


Figure 3.14: Redis-on-Flash improvements obtained with the NVMe-TLS combined offload.

and data in nontrivial ways. Consequently, magic capsules do not exist in this setup, even if the values we store exclusively consist of magic words.

To overcome this problem, with some help from Redis Labs [251] engineers, we implemented OffloadDB, a simple alternative storage backend for RoF, which does separate between keys, values, and metadata (568 LoC). Coupling RoF with an OffloadDB storage backend makes our emulation approach applicable to RoF as well.

3.5.3 Macrobenchmarks

As noted in §3.5.2, we use the nginx and RoF macrobenchmarks to evaluate the performance of our two autonomous NVMe-TCP and TLS offloads, individually and together. We begin with nginx and drive it with the wrk [252] http benchmarking tool. Wrk connects to nginx using 16 threads, which together maintain 1024 open connections that repeatedly request files of a specified size and then wait for a response. We utilize two configurations: C_1 and C_2. In C_1, none of the drive's relevant data is found in the server's page cache. C_1 stresses NVMe-TCP offloading, with a maximal possible rate of the drive's optimal read bandwidth: 2.67 GB/s (≈21.38 Gbps). In C_2, all of the drive's relevant data already resides in the server's page cache, and so it is not read from the remote drive while nginx is operational. C_2 stresses TLS offloading, with a maximal possible rate of 100 Gbps, our NIC's line rate.

Individual Offloads Figure 3.11a shows the http throughput of nginx in C_1, with and without the NVMe-TCP offload. We replicate the microbenchmark methodology and limit system activity at the server to one core (which becomes 100% busy as a result). The outcome turns out qualitatively similar: throughput improvements range between 4%–44% and are correlated with the size of the requested files. In Figure 3.11b and Figure 3.11c, we allow server activity to utilize up to eight cores, which is enough computational power for nginx to be able to fully utilize the remote drive's bandwidth. When maximal bandwidth is reached, NVMe-TCP offload improvements manifest in up to 27% reduced CPU consumption.

We repeat the above experiment in C_2 (data in page cache) using four different setups: (1) "https" employs the baseline, KTLS sendfile with AES-NI crypto operations without any offloads; (2) "offload" improves the baseline by adding TLS offload; (3) "offload+zc" further improves it by instructing TLS to refrain from making copies and instead send files directly from the page cache in a zero-copy ("zc") manner (making it the responsibility of users to avoid changing files while transmitted); and (4) "http" sends unencrypted text and thus serves as an upper bound on improvements.

Figure 3.12 shows the results. With one core, offload and offload+zc deliver 7%–70% and 11%–2.7x higher throughput as compared to https, respectively. With eight cores, they reduce CPU consumption by 0%–2% and 0%–23%, respectively. Offload+zc delivers 88% higher throughput when reaching the NIC line rate. Offload+zc throughput is within 25%–28% of http throughput with one core, and it consumes 23% more CPU cycles with eight cores when using 256 KiB files. Interestingly, for smaller files, offload+zc consumes 3% *less* CPU than http. This happens due to TCP batching effects, which cause http to utilize more, smaller packets for sending.

Overall, offloading eliminates the per-byte cost of the data manipulation, leaving only per-packet costs. This can be seen in the smaller files (size between 128B–1024B), where per-byte costs are small and so offloading yields only a small improvement of 0%–10% and 0%–4% in throughput and CPU consumption, respectively.

Combined Offloads To combine the NVMe-TCP and TLS offloads (together denoted "NVMe-TLS"), we add TLS support in NVMe-TCP Linux subsystem (210 LoC, not yet upstreamed). We evaluate nginx and RoF in the C_1 configuration. In the RoF experiment, we run one RoF instance per core and use the memtier [253] "get" workload to drive the instances with 8 concurrent request-response connections per instance.

Figure 3.13 shows the outcome for nginx. It is qualitatively consistent with the previous results that were bounded by the drive's bandwidth (Figure 3.11, which was dedicated to the NVMe-TCP offload). But the quantitative improvement of the offload combination is more substantial. For example, with a single core and an I/O size of 256 KiB, the improvement provided by the NVMe-TLS offload is 4.0x bigger than that of the NVMe-TCP offload (44% vs. 180%$\equiv$2.0x).

Figure 3.14 shows the benefit of NVMe-TLS offloading for RoF. When comparing it to the corresponding single-offload RoF experiment (not shown), we find that the improvement is 4.6x bigger (28% vs. 130%$\equiv$2.3x), similarly to the aforementioned nginx ratio.

Table 3.3: Average latency in μsec for a single, synchronous request when cumulatively adding our L5P autonomous offloads. Values in parentheses show relative latency as compared to the baseline. Values to the right of the $\pm$ sign show standard deviation in percentages.

size	*base*	*+TLS*	*+copy*	*+CRC*
4K	169 $_{\pm 0.6}$	167 $_{\pm 0.4}$ (0.99)	165 $_{\pm 0.3}$ (0.98)	165 $_{\pm 0.5}$ (0.98)
16K	221 $_{\pm 0.5}$	210 $_{\pm 0.5}$ (0.95)	204 $_{\pm 0.3}$ (0.92)	200 $_{\pm 0.4}$ (0.90)
64K	466 $_{\pm 0.5}$	396 $_{\pm 0.5}$ (0.85)	376 $_{\pm 0.3}$ (0.81)	365 $_{\pm 0.2}$ (0.78)
256K	1321 $_{\pm 5.1}$	1056 $_{\pm 0.5}$ (0.80)	980 $_{\pm 0.0}$ (0.74)	941 $_{\pm 0.4}$ (0.71)

So far, our workloads have been throughput-oriented. In Table 3.3, we show the average latency of a single http GET request (single connection) for multiple offload combinations. In particular, we cumulatively add to the baseline configuration the TLS offload, then the NVMe-TCP copy offload, and then the NVMe-TCP CRC offload. TLS symmetric crypto is much costlier than copying and CRC-ing, so the corresponding offload unsurprisingly achieves the majority of the benefit: a 1%–19% latency reduction. The NVMe-TCP offloads then reduces the latency further by 1%–9% percentage points. As before, bigger requests benefit more.

3.5.4 Reordering and Loss

Out-of-sequence TCP packets (caused by reordering and loss) make our autonomous offloads less effective and imply that NICs and/or CPUs must work harder. Figure 3.15a depicts the effect of gradually increasing packet loss rate on a single sender core transmitting through 128 iperf streams at 100% CPU utilization. We use loss rates between 0%–5% because on the internet, loss rate is typically $\leq$2% [254] and reordering is likewise $\leq$2% [255]. (In datacenters, loss can largely be avoided with DCTCP [256]). On transmit, we can see that TLS offload performance is close to regular TCP performance, delivering throughput that is within 8%–11% of plain TCP. The benefit of offloading compared to software TLS becomes smaller as loss increases, but it nevertheless remains non-negligible, with a minimal 33% improvement at 5% loss. Figure 3.15b reports the internal interconnect bandwidth that the NIC consumes when reading data from memory to reconstruct its contexts (in percents out of the total gen3 x16 PCIe available bandwidth). The figure shows that even with 5% loss, context recovery costs no more than 2.5% of total PCIe bandwidth.

Figure 3.16a and Figure 3.17a show the results of conducting the same experiment but with a receiver using loss and reordering, respectively. Loss and reordering are much costlier at the receiving end when offloading, which is why the corresponding curves rapidly get closer to the software TLS curve. Recall that each out-of-sequence (reordered or retransmitted) packet implies that the encapsulating TLS record will not be offloaded. Figure 3.16b and Figure 3.17b classify TLS records into three: offloaded (no packet in the record was out-of-sequence), partially offloaded (some were out-of-sequence), and not offloaded. Even with 5% loss, we see that more than half of the records are fully offloaded, which highlights the effectiveness of the NIC's context recovery. Figure 3.16a indeed shows that offloading still yields a non-negligible throughput improvement of 19% with the highest packet loss rate. With reordering of even 2%

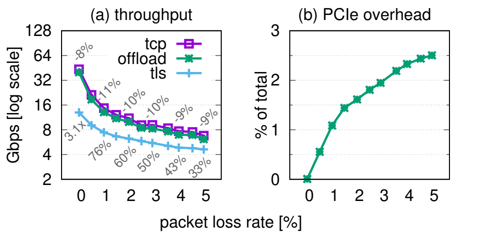


Figure 3.15: Loss effect at sender (top and bottom labels show how offload relates to no encryption and software TLS, respectively). Throughput standard deviation is below 1.6%.

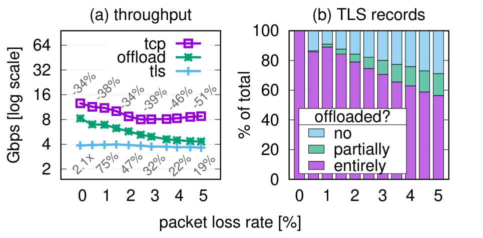


Figure 3.16: Loss effect at receiver (top and bottom labels show how offload relates to no encryption and software TLS, respectively). Throughput standard deviation is below 2.61%.

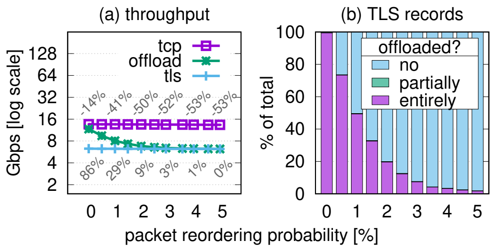


Figure 3.17: Reordering effect at receiver (top and bottom labels show how offload relates to no encryption and software TLS, respectively).

, unlike loss, we see that only 24% of the records are fully offloaded, and with 5% almost no TLS record is offloaded($\leq$ 2%). Nevertheless, Figure 3.17a shows that offloading yields a 9% improvement in throughput with 2% of packet reordering, and in the worst case (5%), performance is still as good as software tls.

3.5.5 Scalability

Autonomous NIC offloads use per-flow state stored in NIC caches to perform well. But NIC caches are inherently limited. They can be exhausted when serving a few thousands of flows, triggering flow state eviction into main memory, which later incurs costly DMA operations over PCIe upon state reuse. For this reason, previous studies indicated that RDMA (which also uses per-flow state) does not scale well [257, 258]. The question is: do autonomous offloads suffer from the same problem as the number of connections exceeds the capacity of the NIC caches?

To answer this question, we add another generator machine and connect it to the server using its ConnectX6-Dx second port. (As it happens, using two ports allows the throughput to exceed 100 Gbps somewhat.) We repeat the nginx experiment involving TLS offloading with eight cores and 256 KiB files in C_2 (data in page cache). But this time, we increase the number of connections, exponentially, from 64 to 128 K. With 4 MiB of on-NIC memory and 208 B per-flow state, the NIC can store at most 20 K flows, ignoring memory used for other resources such as packet queues.

Figure 3.18 shows the results. As the number of connections increases, CPU utilization likewise increases until the CPU becomes the bottleneck. Contributing to the

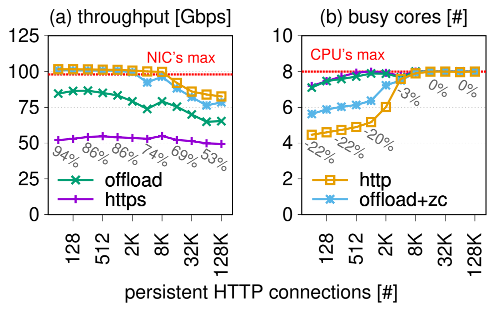


Figure 3.18: Nginx scalability with two TLS offload variants compared to https (baseline encryption) and http (no encryption). All files reside in the page cache (C_2). The labels show offload+zc improvement over https.

increased utilization is the fact that TCP packet batching becomes less effective with more connections: from 48 packets per batch with 128 connections, to only 8 packets per batch with 128 K connections.

Observe that for low connection counts, https and offload performance is not visibly bottlenecked on neither the CPU nor the NIC. This anomaly happens due to imbalanced request spreading that results in some underutilized cores.

In all measurements, offload+zc throughput is within 10% of http throughput, and it consumes at most 1.25x more CPU; offload and offload+zc deliver 32%–63% and 53%–94% higher throughput as compared to https, respectively.

Overall, our workload scales reasonably despite the inherent cache contention problem at the NIC caused by the growing aggregated size of per-flow state. These scalability results disagree with that of certain previous studies [257, 258]. We find that the reason for this disagreement is packet batching, which is dominant in our workloads (at least 8 packets per batch), as they involve bigger message sizes. In contract, the cited previous studies focus on smaller messages. More specifically, as the flow state size exceeds the NIC's cache capacity, each newly serviced packet might in principle trigger a cache miss and a costly memory access. But only the first packet in the batch incurs this cost, whereas subsequent packets do enjoy temporal locality. (We remark that batching might not be dominant if nginx is made to serve only small files. In this case, however, the workload ceases to be data-intensive and is thus outside the scope of our work.)

When comparing our work to the aforementioned previous studies [257, 258], it should also be noted that we use more recent NICs and thus benefit from their improved cache management and increased parallelism, which, similarly to batching, help hide cache miss latencies as demonstrated by a more recent NIC scalability study [259].

3.6 Applicability

Next, we further discuss the applicability of autonomous offloads to additional computations and protocols.

Decompression and deserialization As discussed in §3.2.1, non-size-preserving operations preclude offloading when sending, but not when receiving. A non-sized-preserving operation can be performed on receive by having the NIC write the offloaded operation's results to pre-allocated buffers (set up by the L5P) while also writing the original packet data as-is to the driver's receive ring. The driver will pass packets with offload results as metadata up to the network stack. Later, L5P software will skip performing the offloaded operation if all the packets in the message were offloaded; otherwise, it will fall back to software using the original packet data.

As mentioned above, we require pre-allocated buffers to offload non-size-preserving computation. To pre-allocate these buffers, we need to have either (1) predetermined response sizes, as in the NVMe-TCP protocol, or (2) maximum message size limits enforced by the implementations, such as in HTTP servers that limit request headers to 16 KB.

We note that in contrast to copy, encryption, and digest offloads, which pass packet data through PCIe and memory only once, non-size-preserving offloads will pass packet data through PCIe and memory twice: (1) offload results and (2) original packet data that is needed only for software fallback processing. Nevertheless, this is still better than off-CPU accelerators that pass data three times: (1) from the network; (2) to the off-CPU accelerator; and (3) from the off-CPU accelerator.

Pattern matching Deep packet inspection (DPI) software looks for known patterns in packet payloads using either fixed-length string pattern matching or regular expression matching. Patterns are matched only within L5P messages and never across messages. Thus, these computations fit our offload properties and we can autonomously offloaded them as follows: for each packet, check if some pattern match completes within it using the per-flow context to track pattern matches across packets. If yes, report the match with metadata to indicate the pattern; otherwise, report that the packet contains no match. Later, DPI software inspects packets in-order and if all packets of an L5P message are marked by the NIC, then report results according to offload metadata. Otherwise, if some packet bypassed NIC offload, perform DPI in software.

Not restricted to TCP This work focuses on L5Ps built on top of TCP. But autonomous offloading is, in fact, orthogonal to the specific layer-4 protocol that is being used. Namely, an L5P is autonomously offloadable if it has the properties defined in §3.3, regardless of the specific underlying layer-4 that it is built upon. The reason we have chosen to focus on TCP (in addition to its popularity) is because its properties make it the most challenging to autonomously offload. All the other layer-4 protocols that we are aware of can be either similarly offloaded or are easier to offload.

Consider, e.g., a simple L5P that is built on top of UDP and directly mirrors its properties. A message of this L5P is therefore a datagram that is entirely contained in a UDP packet; the message might get lost or be handed to the receiving end out-of-order. For example, DTLS (Datagram Transport Layer Security [260]) is such an L5P, as it only encrypts and decrypts UDP packets. Autonomously offloading this type of L5Ps is trivial and does not merit an academic publication (we indeed do not consider it part of our contribution). Because the NIC operates on individual, self-contained datagrams, it never has to worry about such issues as losing and having to reconstruct its position in the sequence due to packet reordering and loss. Falling back on L5P software processing is likewise never needed: the NIC always knows what to do next, since all the information required for acceleration is encapsulated inside the currently-processed incoming or outgoing datagram.

The main contribution of this work is coming up with a way to autonomously offload a more sophisticated type of protocols—those that provide some *stream abstraction* for their users. The challenging aspect in autonomously offloading such protocols is that an L5P message can be spread across multiple packets in the stream with no alignment between L5P messages and packets, making it challenging for the NIC to identify L5P message boundaries in the face of packet reordering and loss.

SCTP (Stream Control Transmission Protocol [261]) can be viewed as an L5P that uses UDP to provide reliable, in-sequence delivery of a stream of messages with congestion control. SCTP divides messages into "chunks," such that each chunk is entirely contained in a UDP packet along with its own header. A chunk header indicates, in particular, whether the associated data starts a new SCTP message. Therefore, autonomously offloading SCTP is similar to, but easier than TCP-based offloads, because the NIC can identify message beginnings within packets in a deterministic manner, ridding it from the need to speculate using magic patterns.

QUIC [262] is an emerging protocol that provides a stream abstraction. It is capable of multiplexing multiple byte streams on top of encrypted UDP packets. Each packet contains one or more "frames" that corresponds to some byte stream. A QUIC autonomous offload must be able to encrypt and decrypt the packets. (Simpler than TLS offloading, as it is done per UDP packet.) Then, given access to the frames' content, all autonomously offloadable operations become relevant: copy to avoid L5P message reassembly, decompression (e.g., QPACK [263]), pattern matching, etc.

Chapter 4

The I/O Working Set Problem

4.1 Background: The NIC-CPU Interface Today

Software interacts with NICs via per-core logically cyclic arrays called receive (Rx) and transmit (Tx) rings. We focus on Rx rings as they dictate the I/O working set (see §4.2). The NIC spreads incoming traffic among cores using receive side scaling (RSS [24]). With RSS, when a packet arrives from the network, the NIC selects its destination Rx ring according to a hash computed over packet header fields.

Rx rings combine two producer-consumer functionalities: (1) software producing empty buffers for the NIC to consume by storing incoming packets (memory allocation) and (2) the NIC producing incoming packets for software to consume (packet reception).

Software initially chooses the Rx rings' size and allocates them in main memory. Ring entries are architectural *descriptor* structures with several fields, one of which is a packet buffer pointer. Software prepopulates all Rx descriptors with MTU-sized buffers. When a packet targeting the ring arrives, the NIC DMA-writes it to the buffer pointed to by the *head* ("next empty") descriptor index, incrementing it to point to the subsequent descriptor if it does not surpass the *tail* ("next full") descriptor index. Software consumes packets from the Rx ring in the same cyclic order. It iteratively swaps the tail descriptor's buffer (containing a newly delivered packet) with a new empty buffer and then increments the tail (if it does not surpass the head).

Software informs the NIC about new free buffers (ring tail advances) by means of an MMIO write to a NIC register, known as "ringing a doorbell." In contrast, software does not poll the ring head to detect new packets, as such polling would result in cache line bounces between the NIC and the CPU (if the head were stored in memory) or expensive MMIO reads (if it were stored in a NIC register). Instead, the NIC informs software of produced packets by means of a cache-friendly memory-based protocol, described next.

Completion Rings Modern NICs notify software of delivered packets via per-core, in-memory *completion ring (CR)* structures [144,232,264,265]. CRs, like descriptor rings, are circular buffers. Each CR is associated with one or more descriptor rings. CR entries indicate which Rx ring descriptors are ready for software processing by providing their ring and index.

To optimize cache coherence traffic, the NIC only writes CR entries and software only reads them. The NIC and software coordinate CR access with a sense reverse indication mechanism [266]. Each CR entry contains a "done" bit, indicating to software that the entry is ready for processing. The flag value used for this indication alternates on each pass through the ring: it is 1 on odd passes and 0 on even passes. Both the NIC and software internally maintain a "generation" bit for the current value. Software checks for new packets by polling the head CR entry, comparing its "done" bit with the "generation" bit.

Figure 4.1 depicts packet reception with CRs. Initially (Figure 4.1a), three packets arrive for a core whose Rx and completion rings are empty. Packet delivery (Figure 4.1b) consists of (1) the NIC using RSS to find the Rx and completion ring matching the packet; (2) writing the packet to an Rx descriptor, and (3) writing a CR entry, indicating the index of the Rx descriptor that holds the packet. These entries have "done" set to 1, because the CR's generation is 1. Packet processing (Figure 4.1c) occurs when the core, polling the head CR entry, notices its "done" flag has changed. It starts traversing "done" CR entries, processing packets pointed to by the Rx descriptors indicated by them, replenishing these buffers, and advancing both rings' tails. Once the core reaches a CR entry with "done" set to 0, it stops and resumes polling the CR.

We remark that a separate CR is used instead of piggybacking this protocol on Rx descriptors to avoid having both the NIC and software writing to the same cache line concurrently (by updating different descriptors in the same cache line), which creates cache contention and hazards [267].

4.2 The I/O Working Set Problem

Network-intensive applications depend on DDIO [43] and similar technologies to keep up with network rates of hundreds of Gbps as well as for low-latency packet processing. DDIO enables DMAs by I/O devices to go directly to/from the CPU's LLC, if possible, instead of main memory. With DDIO, DMA reads are fulfilled by the LLC if the associated bytes are there and DMA writes overwrite bytes that already reside in the LLC. DMA writes can also allocate new cache lines in up to two LLC ways, evicting other data in the process.

DDIO's effectiveness depends on the size of the *I/O working set*, which is defined as the memory areas that are DMAed by an I/O device during some time interval [268]. The I/O working set of an I/O intensive workload exceeding the LLC's capacity leads to the "leaky DMA" problem [97, 164], wherein new packets written by the NIC evict not-yet-processed packets from LLC. Consequently, CPU accesses to packet data slow down due to being served from main memory, which may even become a bottleneck resource [75, 97–101, 172, 269–272]. In particular, if the slower packet processing makes a core unable to keep up with the packet arrival rate, its Rx ring will fill up, causing latency to become linear in the ring size.

Ideally, the I/O working set would depend only on software processing time, i.e., buffers could be reused by the NIC immediately when released by software. However,

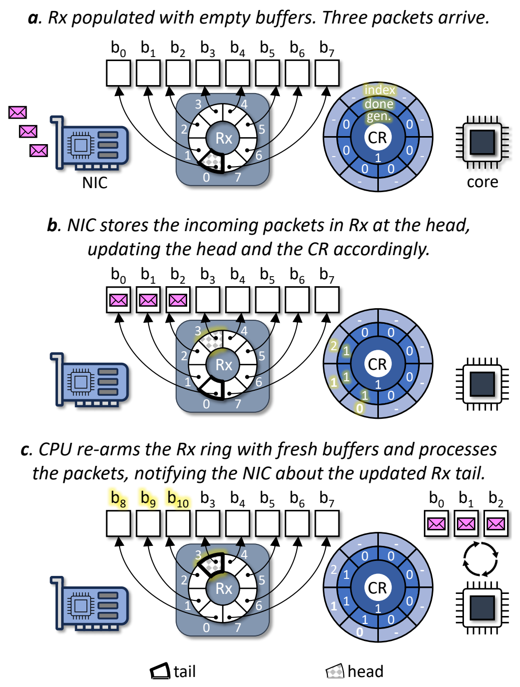

a. Rx populated with empty buffers. Three packets arrive.
b0 b1 b2 b3 b4 b5 b6 b7
NIC
Rx
CR
index
done
gen.
core
b. NIC stores the incoming packets in Rx at the head, updating the head and the CR accordingly.
c. CPU re-arms the Rx ring with fresh buffers and processes the packets, notifying the NIC about the updated Rx tail.
b8 b9 b10 b3 b4 b5 b6 b7
b0 b1 b2
tail
head

Figure 4.1: PrivRing packet reception using a single private receive ring with its completion ring. Highlighted regions indicates changes relative to the previous stage. For simplicity, we depict only the Rx ring's head and tail positions, assuming identical head/tail positions in the CR (which is not necessary the case).

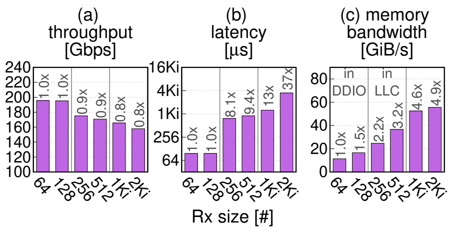


Figure 4.2: Large I/O working set causes high memory bandwidth (c) which degrades throughput (a) and latency (b). Top labels compare to Rx size of 64. Lines demarcate where the I/O working set size fits in DDIO's portion of the LLC, in the LLC, or exceeds the LLC.

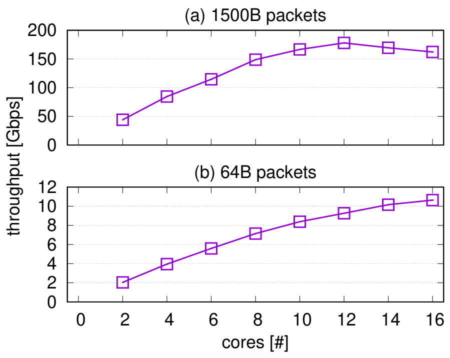


Figure 4.3: The LB NF scales with 64 B packets, but scalability suffers with large 1500 B packets due to the increased I/O working set size.

the NIC interface creates a dependency on the number (N) and size (R) of Rx rings. Rx rings are prepopulated with buffers, and as ring descriptors are cyclically accessed, an Rx buffer b can be reused only after the NIC uses all other buffers in the ring—even if software has released b earlier. In contrast, Tx rings contain only in-flight packets, so they are usually empty or partially full. Thus, the I/O working set size is at least the union of all Rx buffers, which is of size $|Rx| = N \times R \times 1500$ B.

The growing gap between stagnant CPU speed and ever-increasing NIC bandwidth results in $|Rx|$ growing with hardware advances, thus exceeding LLC capacity [100, 268, 273]. The reason is that this gap requires increasing both the size R and number N of the Rx rings, because: (1) packet bursts experienced by individual cores become bigger and should be absorbed to avoid packet loss [274–276], and (2) packet processing requires additional cycles, disallowing any single core from driving the NIC to its full capacity [170, 271].

4.2.1 Implications

We demonstrate the I/O working set problem by evaluating the impact of increasing the Rx ring size on a stateful load balancer (LB) network function (NF). In each experiment, LB uses all cores of a 16-core CPU, which has a 22 MiB LLC and two 100 Gbps NVIDIA ConnectX-5 NICs, and processes 1500 B packets. (§6.3 details the full experimental setup.) Figure 4.2 shows that enlarging the I/O working set worsens throughput by up to 0.8×, latency by up to 37× (due to rings filling, as explained above), and memory bandwidth by up to 4.9×. Line rate throughput is achieved when the I/O working set fits in the two LLC ways used by DDIO (ring size $R \leq 128$). Results degrade in two steps: when the I/O working set exceeds the DDIO ways but fits in the LLC ($128 < R < 1024$) and when it exceeds the LLC ($R \geq 1024$). Other NF applications (not shown) behave similarly.

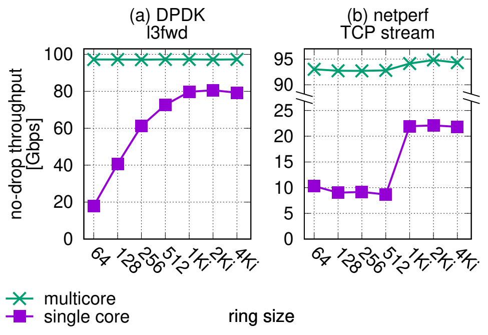


Figure 4.4: DPDK l3fwd no drop throughput and Netperf TCP stream throughput. Small rings work well when traffic is spread across multiple cores but cause drops otherwise.

One may wonder why NFs should use all cores, if the result is an excessive I/O working set. The answer is that systems often do not have fixed workloads, and all cores are necessary to maximize throughput in certain workloads. Figure 4.3 demonstrates this issue by showing LB throughput under maximal rates of either 1500 B or 64 B packets, as the number of LB cores varies. For 1500 B packets, maximal throughput is reached at 12 cores. But for 64 B packets, throughput peaks at 16 cores, as the I/O working remains small. Thus, *our goal is to address the I/O working set problem in the most demanding cases*—for the benefit of all workloads.

4.2.2 Shortcomings of Existing Solutions

We discuss various approaches for shrinking the I/O working set to fit in the LLC and explain why they are unsatisfactory.

Few Dispatchers This approach, showcased by Shinjuku [269] and Shenango [172], uses a few "dispatcher" cores, each with a large Rx ring, to distribute packets among the remaining worker cores. While such systems can saturate ≈40 GbE links, the dispatcher cores become the bottleneck as link speeds increase to 100 Gbps and beyond [170].

Small Private Rings A privRing system can employ smaller per-core rings, such that the I/O working set fits in the LLC. However, decreasing the Rx ring size below the default (1 Ki) makes rings unable to absorb packet bursts, resulting in packet loss that degrades the performance of loss sensitive protocols such as TCP. We show this in two experiments: DPDK layer-3 forwarding (l3fwd), which we load using MTU-sized packets with the RFC2544 no-drop rate benchmark [138], and TCP netperf stream [40]. We test both with all traffic directed at a single receiver core ("single core") and then with traffic spread across 8 receiver cores ("multicore").

The results in Figure 4.4 demonstrate that small rings work well for multicore, because load and queuing are spread evenly between the individual cores and rings. But in the single core case, representing an individual core's capacity, rings smaller than 1 Ki result in packets drops and throughput loss.

Chapter 5

ShRing: Networking with Shared Receive Rings

Multicore systems parallelize to accommodate incoming Ethernet traffic, allocating one receive (Rx) ring with >=1Ki entries per core by default. This ring size is sufficient to absorb packet bursts of single-core workloads. But the combined size of all Rx buffers (pointed to by all Rx rings) can exceed the size of the last-level cache. We observe that, in this case, NIC and CPU memory accesses are increasingly served by main memory, which might incur nonnegligible overheads when scaling to hundreds of incoming gigabits per second.

To alleviate this problem, we propose "shRing," which shares each Rx ring among several cores when networking memory bandwidth consumption is high. ShRing thus adds software synchronization costs, but this overhead is offset by the smaller memory footprint. We show that, consequently, shRing increases the throughput of NFV workloads by up to 1.27x, and that it reduces their latency by up to 38x. The substantial latency reduction occurs when shRing shortens the per-packet processing time to a value smaller than the packet interarrival time, thereby preventing overload conditions.

5.1 ShRing's Design and Implementation

ShRing is an architecture for driving high bandwidth NICs. Instead of using private per-core default-sized Rx rings, it shares each default-sized Rx ring between a set of cores. (ShRing leaves the Tx path unmodified.) ShRing can improve throughput, latency, or both, depending on the workload (§5.1.1).

Sharing a receive ring among cores requires us to synchronize the ring accesses of the CPU (using locks or atomic instructions), which incurs overhead compared to the synchronization-free privRing. ShRing curbs this overhead by limiting the number of cores sharing a ring to *N*; we use *N*=8, but other values may work better for other setups. Also, shRing reduces synchronization overhead by leveraging per-core completion rings (CRs) with which the NIC spreads incoming packets between cores [144], ridding them from having to compete for newly arriving packets (§5.1.2). As a result, shRing's benefits outweigh its synchronization costs for workloads that suffer from ineffective DDIO use.

We propose two shRing designs that represent the ring as an array (RxArr, §5.1.3) or a linked list (RxList, §5.1.4). Both can be implemented with recent NVIDIA NICs. RxArr's synchronization is costlier, but RxList's interferes with the NIC's Rx entry prefetching, so we rule it out (but propose a modest NIC ASIC modification that will fix this problem).

ShRing dynamically turns itself on/off depending on whether or not the workload is benefiting from it (§5.1.5). We describe the implementation details in §5.1.6.

5.1.1 Benefits and Constraints

ShRing can improve throughput and/or latency, depending on the workload. Next, we define the workload properties necessary for shRing to be advantageous, and we explain the expected benefits of shRing and how it provides them. When shRing is counterproductive (necessary properties are absent), it dynamically disables itself.

ShRing is relevant only for workloads that avoid *pathological core overload*, where a subset of the sharing cores are continuously overloaded while their peers are underloaded. Pathological conditions may occur due to continuous, highly skewed per-packet processing time differences, or because of chronic incoming traffic imbalance. For reasons detailed later on (§5.1.5), when cores share a ring under pathological conditions, the fact that only some of them are overloaded implies that the packets of the overloaded cores increasingly and disproportionately accumulate within the ring, to the point that no room is left for packets of underloaded cores. This pathology causes new packets directed at underloaded cores to get dropped despite there being available processing capacity.

We term these conditions "pathological" because (1) they are suboptimal and may indicate the system is misconfigured, and (2) they are atypical when measuring NFV performance, as many NFV studies [99,132–137] and IETF benchmarking methodology [138] generate packet headers using randomization, balancing load across cores with hash-based packet spreading (e.g., RSS).

Throughput ShRing improves a workload's throughput if (1) its I/O working set with privRing exceeds the LLC DDIO capacity and (2) the penalty of the resulting cache misses is non-negligible compared to the overall packet processing time. Relative to privRing, shRing multiplicatively decreases the number of rings by a factor equal to the number of cores sharing each Rx ring (N=8 in our case). This decrease results in a corresponding $1/N$ reduction of the I/O working set, possibly to below the LLC DDIO capacity. ShRing therefore mitigates and possibly eliminates the I/O-related cache miss penalty and thus enables more effective packet processing.

Latency ShRing improves a workload's latency if the associated cores are saturated because packet service rate (number of packets processed per second, denoted μ) is smaller than packet arrival rate (number of packets arriving per second, denoted λ). Latency is linear in the ring size s in this case, as queuing theory dictates that $\mu <$

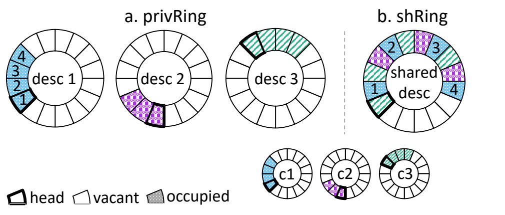


Figure 5.1: PrivRing (private Rx rings) vs. shRing (shared Rx ring) with $N = 3$ completion rings.

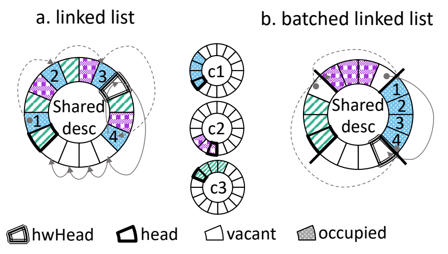


Figure 5.2: RxList vs. Batched RxList designs for one shared ring with three completion rings. In (b), the batch size is 4.

λ implies fully occupied Rx rings, which means every newly arriving packet waits for $s - 1$ preceding packets to be processed. But in contrast to privRing, where each core has its own default-sized ring, shRing shares each such ring between N cores, so the "effective" ring capacity that each core experiences is s/N, which means the latency proportionally becomes $1/N$ smaller (recall that we assume no pathological core imbalance).

Moreover, whenever shRing improves throughput, it also improves latency, as this throughput improvement stems from making the per-packet processing time (P_t) shorter. Notably, if shRing's shorter P_t transforms the overall service rate from slower than arrival rate (under privRing) to faster ($\mu > \lambda$ instead of $\mu < \lambda$), queuing theory says that Rx ring occupancy drops from fully to barely occupied. Namely, latency drops sharply, essentially becoming $O(P_t)$ with shRing instead of $O(P_t \times s)$ with privRing. This shRing property underlies Figure 5.7g.

5.1.2 Synchronization with Completion Rings

In principle, N cores may share a receive ring by synchronously accessing the ring's head. But this approach creates a synchronization bottleneck [26,132,141–143]. ShRing sidesteps this problem by reusing RSS to spread incoming packets between different sharing *cores* (in addition to spreading them between different *rings*, which is the usual role of RSS). So when the NIC stores incoming packets in a shared ring, it communicates to each of the N sharing cores which packets belong to that core via a per-core *completion ring* (CR), as depicted in Figure 5.1.

A CR is a circular array in host memory. There are N CRs associated with each shared ring R: one for each core C that shares R. The CR stores indexes of R's packet descriptors, specifying which descriptors are ready to be processed by C. Similarly to descriptor rings, a CR has head/tail entries whose indexes reside in NIC memory. When the NIC stores in R an incoming packet P that is mapped to core C, it writes the index of P's descriptor to the tail of C's CR and advances this tail. To receive packets, C polls its CR head awaiting notification about the next available packet in R. When C removes this packet from R, it advances its CR head.

Thus, per-core CRs allow cores to poll without synchronizing with their peers. CRs negligibly increase the I/O working set size, as a CR entry occupies only a single cacheline (for storing metadata about the associated packet, such as size and header offsets).

Nonetheless, CRs do not obviate the need for synchronization when a core reposts a descriptor for the NIC to consume. RxList and RxArr address this synchronization problem in different ways.

NIC Support Recent NVIDIA NICs already support associating multiple CRs with a shared Rx ring as part of a shared receive queue (SRQ) buffers feature [144, 277]. The motivation for this feature is reducing DRAM pinning for RDMA (see §5.4), as opposed to shRing's goal of improving throughput and latency for Ethernet.

We expect support for Ethernet Rx ring sharing among CRs to become widely available in the future, because it is included in the infrastructure datapath function (IDPF) specification [278] and the Open Compute Project NIC specification [279], which are proposed industry standards for network device interfaces.

5.1.3 Array Ring Sharing (RxArr)

In the baseline privRing, each core C processes and reposts descriptors of its private ring in array order, one after the other. Namely, after C processes a descriptor D_i, it reposts D_i by advancing the head of the ring past D_i to D_{i+1}, thereby indicating that D_i can be reused by the NIC to store some other incoming packet in the future.

In contrast, RxArr shRing implements a ring array that is shared between N cores. It therefore cannot automatically advance the ring's head in this way, as D_i might become ready for reuse before its k preceding descriptors $\{D_j\}_{j=i-k}^{j=i-1}$. For example, if they were assigned to cores different than C and require a longer processing time as compared to D_i. Or if RSS happened to assign all of them to some other core C', which must now work harder than C to catch up.

RxArr must thus guarantee that the NIC is notified that D_i can be reused only when all preceding descriptors are also ready for reuse. For this purpose, RxArr maintains a bitmap with a bit per descriptor, tracking which ring descriptors between head and tail have been processed and made available for reuse. After core C consumes D_i and re-arms it with a new empty buffer, C (1) atomically sets bit i in this bitmap, (2) consults the bitmap to find the maximal contiguous sequence of descriptors available for reuse beginning at the head $\{D_j\}_{j=head}^{j=maxContig}$, and (3) atomically clears the corresponding bits and advances the head past them.

The drawback of RxArr is its synchronization overhead, as its bitmap is a shared and frequently updated data structure that requires core coordination. Also, RxArr is suboptimal in that it delays the reuse of descriptors made ready by some cores, if prior descriptors have not yet been processed by other cores. Conceivably, packet loss might occur under RxArr despite available CPU and buffer capacity. In the privRing baseline, in contrast, ready descriptors reside in different rings and so the NIC can reuse them as they become available.

Listing 5.1 shows the RxArr receive function, which dequeues a batch of packets for processing. It receives a shared descriptor ring (`sd_ring`), the calling core's CR (`c_ring` completion ring), and an output array of packet pointers (`pkts`) of length `len`. It returns the number of received packets. Lines 10–15 poll the CR to find the location of a ready descriptor assigned to the calling core and store the descriptor's buffer in the output array, replacing this buffer with a new one. Lines 16–22 mark received descriptors in

```
#define BIT(x)  (1 << ((x) & 63))
#define WORD(x) ((x) >> 6)
#define ISSET(bmp, x) \
        (bmp[WORD(x & (bmp->size - 1))] & BIT(x))
int shRing(sd_ring *sdr, c_ring *cr,
           void **pkts, int len) {
  int rcvd = 0, lidx = -1;
  uint_64t lbits = 0
  while (rcvd < len) {
    c_ring_ent *cre = get_cre(cr);
    if (cre == NULL)
      break;
    int idx = cre->idx;
    pkts[rcvd++] = sdr->desc[idx].buf;
    sdr->desc[idx].buf = alloc_buf();
    if (lidx == -1) lidx = WORD(idx);
    else if (lidx == WORD(idx)) {
      atomic_or(&sdr->bitmap[lidx], lbits);
      lidx = WORD(idx);
      lbits = 0;
    }
    lbits |= BIT(idx);
  }
  if (rcvd == 0) return 0;
  if (lbits != 0)
    atomic_or(&sdr->bitmap[lidx], lbits);
  cr->ci += rcvd;
  *cr->doorbell = cq->ci;
  lock(sdr->lock);
  while (ISSET(sdr->bitmap, sdr->ci) != 0) {
    setb = ffs(~sdr->bitmap[WORD(sdr->ci)]);
    atomic_clear(&sdr->bitmap[WORD(sdr->ci)],
                 setb - 1);
    sdr->ci += setb - 1;
  }
  *sdr->doorbell = sdr->ci;
  unlock(sdr->lock);
  return rcvd;
}
```

Listing 5.1: RxArr shared ring receive code.

the shared bitmap (`sdr->bitmap`) while batching updates within 64-bit words. This is done using atomic instructions, as other cores may be concurrently setting/clearing other bits in the bitmap. Line 24 handles the corner case of an empty CR. Lines 25–26 handle the remaining accumulated bitmap updates after exiting the loop. Lines 27–28 ring the CR's doorbell.

Lines 29–37 identify the maximal contiguous sequence of descriptors beginning at the ring head that is available for reuse, notifying the NIC about them. These operations are performed under a lock to guarantee the atomicity of (1) inspecting and modifying the bitmap and of (2) notifying the NIC. Line 31 uses the find-first-set instruction to identify the contiguous set bits. Lines 32–33 atomically clear them. Finally, Line 34 advances the ring's head (consumer index, `sdr->ci`) accordingly, and Line 36 writes the updated head to the shared ring's doorbell.

5.1.4 Linked List Ring Sharing (RxList)

RxList is a shRing design that alleviates RxArr's bitmap coordination problem, eliminating the requirement to repost descriptors in array order. To this end, RxList represents the empty packet buffer descriptor queue as a linked list. The NIC correspondingly follows list order when storing incoming packets. The list itself is overlaid on the Rx descriptor array, with each descriptor holding a "next" field pointing to the next list item. (Linked list functionality is part of the SRQ feature [280].) Initially, each descriptor points to the subsequent descriptor in the array. But as packet processing occurs and cores process and repost descriptors out of array order, the descriptor order in the list changes. We denote the first and last descriptors in the empty descriptor list as *hwHead* and *hwTail*, respectively, to distinguish them from the "head" and "tail" used in the rest of the work to describe the first and last descriptors holding packets.

Figure 5.2a depicts RxList's structure using three cores sharing a single Rx ring. Observe that RxList's descriptor ring entries are not contiguous: there are multiple non-vacant descriptors in the array between *hwHead* and its successor vacant descriptor in the list, which is impossible in an array-based design. The figure also shows dashed links between non-vacant descriptors. These represent the order in which these descriptors were filled by the NIC, i.e., their order in the list when they were vacant.

We now detail RxList's receive flow, whose code is shown in Listing 5.2. The function's inputs and outputs are the same as RxArr's receive function. Lines 5–10 batch packets for processing exactly as in RxArr: the completion ring is polled to find the location of ready descriptors, each such descriptor's buffer is stored in the packet output array, and the descriptor's buffer is replaced with a new buffer. Lines 11–13 are unique to RxList: they link dequeued descriptors one after the other, creating a linked list that will eventually be appended to the tail of the empty descriptor list. Lines 15–17 are again standard functionality. First, the case of an empty completion ring is checked, and then the core's completion ring head (denoted `ci`, or consumer index) is updated, including a notification to the NIC via a doorbell MMIO write. Lines 18–24 are again new to RxList. They lock the shared descriptor ring to atomically (1) append the new list created in lines 11–13 after the tail of the list and (2) notify the NIC, via a doorbell write, of the number of descriptors with empty buffers that are appended to the list. Finally, line 25 returns the number of received packets.

```
int ll_recv(sd_ring *sdr, c_ring *cr,
            void **pkts, int len) {
  int idx, rcvd = 0, myhead, *iptr = NULL;
  while (rcvd < len) {
    c_ring_ent *cre = get_cre(cr);
    if (cre == NULL)
      break;
    idx = cre->idx;
    pkts[rcvd++] = sdr->desc[idx].buf;
    sdr->desc[idx].buf = alloc_buf();
    if (iptr == NULL) myhead = idx;
    else iptr->next = idx;
    iptr = &sdr->desc[idx];
  }
  if (rcvd == 0) return 0;
  cr->ci += rcvd;
  *cr->doorbell = cq->ci;
  lock(sdr->lock);
  int prevtail = sdr->hwTail;
  sdr->desc[prevtail].next = myhead;
  sdr->hwTail = idx;
  sdr->ci += rcvd;
  *sdr->doorbell = sdr->ci;
  unlock(sdr->lock);
  return rcvd;
}
```

Listing 5.2: RxList (linked list) shared ring receive code.

Prefetching Problem We find that RxList neutralizes descriptor prefetching, an important NIC performance optimization. Because descriptor rings are typically stored contiguously, the NIC reads sequences of contiguous descriptors in a single PCIe read transaction and caches valid descriptors in NIC memory to improve throughput and reduce latency for subsequent packets. When descriptors are linked out of array order, the NIC fails to find the next descriptor on the list in its on-NIC cache, resulting in more descriptor DMA reads being required.

Effective descriptor prefetching is critical for high PCIe-based NIC performance [171], and even more crucial for shRing. In privRing, a descriptor cache miss on some ring

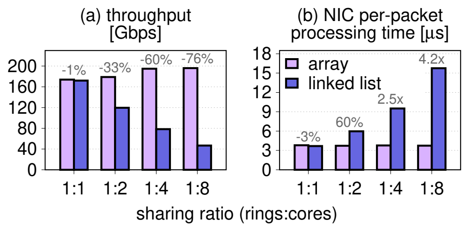


Figure 5.3: Although conceptually more suitable for sharing, RxList interferes with descriptors' contiguity, hampering their prefetching and thus degrading performance. (Labels show List to Arr ratio.)

does not stall incoming traffic destined to other rings, but with shRing there are fewer rings and so more traffic is stalled.

To demonstrate this effect, we evaluate the performance of various descriptor ring to core sharing ratios. We compare RxList to RxArr, in which the NIC follows descriptor array order when storing packets. We run the synthetic NF (from §4.2.1) on all cores and try to process traffic at line rate.

Figure 5.3a shows the throughput achieved by both designs. When there is no sharing, then RxList, RxArr, and privRing (not shown) perform similarly ($\approx$ 2%). This is expected since in this case, all approaches maintain ordering within the single descriptor ring. However, as we decrease the ring to core ratio, linked list descriptors become reordered and RxList's throughput declines sharply as sharing increases: 33% for 1:2 sharing ratio and 76% for 1:8 sharing ratio.

Figure 5.3b shows how costly out-of-order descriptors are, motivating RxArr. Specifically, we report the NIC's internal packet processing time, and see that for linked lists this time grows as more cores share a descriptor ring: from 3.7 μs at 1 core per ring to 16.3 μs at 8. In contrast, RxArr performance remains the same regardless of the sharing ratio.

Prefetching Solution We propose *batched RxList*, a shRing design that obtains RxList's resiliency against pathological core overload conditions without damaging the NIC's performance. Batched RxList amortizes the cost of locking and descriptor reordering in RxList by batching packets to descriptors. In this design, depicted in Figure 5.2b, each RxList descriptor points to a buffer that can hold multiple packets. For each RxList, the NIC stores new packets destined to a core via the same descriptor used to store previous packets for that core, provided that room remains in the descriptor's packet buffer. Only once this descriptor "fills up" will the NIC consume a new descriptor from the list and start storing incoming packets for that core in the new descriptor's buffer. To perform this batching, the NIC caches the last Rx descriptor used for each CR associated with the RxList. The NIC thus effectively maintains per-core "mini hwHeads" pointing to each core's current descriptor.

The benefit of the batched RxList design is twofold. From the NIC's perspective, batching packets in descriptors and caching the descriptors reduces the importance of descriptor prefetching, as packets destined to a core experience a single cache miss per batch. From the cores' perspective, batching reduces RxList synchronization, as locking the RxList to repost a descriptor is now guaranteed to occur only once per batch, instead of potentially once per packet.

Although recent NICs support batching multiple packets in a single large descriptor buffer [281], batched RxList requires NIC ASIC modifications to support a list consisting of such descriptors. Therefore, we cannot evaluate batched RxList. We present this design to underscore that RxList's tradeoffs are likely not fundamental and are caused by current NIC ASIC limitations, which can be fixed.

5.1.5 Dynamic ShRing

We propose a dynamic approach that switches between privRing and shRing during run time, depending on which architecture is more beneficial at the moment. Our goal is to disable shRing if the workload experiences pathological core overload or if it is not bottlenecked on I/O-related cache misses. We describe the heuristic we currently use to identify these conditions. We leave improving the precision and robustness of the heuristic for production use to future work.

Pathological Overload Pathological overloaded conditions can make overloaded cores monopolize ring descriptors. If continuous, high per-packet processing time differences are such that the packet service rate of overloaded cores is smaller than their packet arrival rate, queuing theory dictates that the Rx ring eventually becomes fully occupied with their packets. If incoming traffic is chronically imbalanced, large batches of packets destined to overloaded cores can arrive and occupy most if not all the descriptors.

In both of the above scenarios, overloaded cores invoke their ring's receive function less frequently than underloaded cores. This is clearly the case for cores overloaded due to high per-packet processing time, but also happens if overload is due to incoming traffic imbalance. In this case, an overloaded core's receive call produces a large batch of packets, which takes the core longer to process before returning to the ring to dequeue more packets. We detect overloaded cores based on this behavior, as explained below.

I/O-Related Cache Miss Significance Recall that under non-pathological conditions, a workload will benefit from shRing if (1) its I/O working set with privRing exceeds the LLC DDIO capacity and (2) the penalty of the resulting cache misses is non-negligible (§5.1.1). We associate (1) with high memory bandwidth utilization and (2) with high networking throughput.

Heuristic We measure throughput, memory bandwidth, and time between subsequent calls to the receive function and record the results in a sliding window of 16 entries. When more than half of throughput and memory bandwidth measurements exceed a predefined threshold while no core is overloaded (calls receive infrequently compared to other cores), we switch from privRing rings to shRing rings. To switch back from shRing to privRing, we wait until $\frac{7}{8}$ of measurements are below the threshold

To switch between privRing and shRing, we pre-program two sets of RSS tables, which are NIC data structures used to steer incoming packets to descriptor and completion rings based on packet headers. Each RSS table set points to its own set of rings, i.e., privRing and shRing. Then, based on the heuristic's decision, we update NIC steering rules to redirect packets to the appropriate RSS table set. After switching, before we begin polling the new rings for packets, we drain remaining packets from the previous ring set.

5.1.6 Implementation

Our implementation of RxArr and RxList targets 100 GbE NVIDIA NICs with unmodified ASICs. We initially relied on firmware patches to expose ring sharing mechanisms, originally aimed for InfiniBand RDMA (see §5.4), for Ethernet use. However, NVIDIA NIC firmware now makes these mechanisms generally available.

We implement our designs with 2039 lines of code (LOC) in the NVIDIA DPDK driver and only 137 LOC in DPDK's core. We leverage DPDK's command line driver options to enable the desired ring sharing mechanism and to specify how many cores share each ring. This approach enables unmodified DPDK-based applications to benefit from shRing.

Dynamic shRing is implemented in a dedicated thread that runs every 10 ms on a separate core which polls Intel PCM [161] counters for PCIe generated memory bandwidth and NIC byte and packet counters. We expose PCM counters through a library that we link with DPDK; the library is 116 LOC and the code using it in DPDK is 330 LOC. As the threshold for switching from privRing to shRing, we use throughput greater than 170 Gbps, memory bandwidth greater than 25 GiB/s, and the standard deviation between calls to Rx functions being at most 32x larger than the median (where 32 is the maximum packet batch that shRing's Rx functions can return). We experimentally find that these values provide good results for the NFs we tested.

5.2 Evaluation

We evaluate shRing's effectiveness using synthetic microbenchmarks as well as NAT and LB macrobenchmarks. We measure the gains obtained with shRing's efficient I/O working set utilization in both non-pathological and pathological conditions (§5.1.1) under 200 GbE load.

5.2.1 Methodology

Experimental Setup Our setup consists of two Dell PowerEdge R640 servers, connected back-to-back via two pairs of 100 GbE NVIDIA ConnectX-5 NICs with pause frames disabled. One server is the evaluated system and the other is the load generator. Both servers have 16-core 2.1 GHz Xeon Silver 4216 CPUs, 128 GiB (=4x16 GiB) 2933 MHz DDR4 memory, and a 22 MiB LLC that consists of 11 ways. They run Ubuntu 18.04 (Linux 5.4.0) with hyperthreading and Turbo Boost disabled. The kernel is configured to isolate CPUs from the OS scheduler, use 1 GiB hugepages, disable power saving states, and disable microarchitectural side channel mitigations.

On the load generator machine, we run the stateless Cisco T-Rex packet generator [160], which we modify to improve latency measurement accuracy from 10–100μs to 1μs [192]. Unless specified otherwise, we use default application settings: 1024 descriptor Rx and Tx rings and 2 DDIO LLC ways, and we run application logic on all 16 of the available CPU cores—8 cores per NIC. All the results presented are trimmed means of ten runs; the minimum and maximum are discarded. The standard deviation is always below 5%.

Measurement Tools We measure cycles per packet by modifying applications to record cycle counters, cache hit rate using Linux perf, Tx ring occupancy by comparing com-

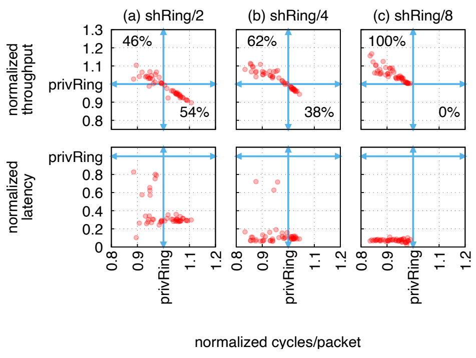


Figure 5.4: Normalized performance of shRing to privRing for NFs with varying memory intensity: shRing/8 improves performance in all cases. (Labels show percentage of NFs in quadrant.)

pletion ring producer and consumer indexes, PCIe latency using NVIDIA Mellanox Neo-host [162], and memory bandwidth and PCIe hit rate using Intel PCM [161].

Ring Mechanisms We compare between privRing; non-dynamic array ring sharing (RxArr) between 8 cores—the maximum possible on a CPU with 16 cores and 2 NICs—which we denote "shRing/8;" and a small privRing configuration whose aggregate descriptor count equals that of shRing/8, i.e., 128 entries per ring when shRing/8 uses 1024 entries per RxArr. We remark that small privRing is impractical since it imposes loss when traffic is bursty, as shown in §4.2.2. We show it for a thorough comparison between privRing and shRing.

5.2.2 Non-Pathological Conditions

We show the benefits of using shRing under high load without pathological core overload conditions. Specifically, we evaluate (1) synthetic NFs with varying memory intensity and cache pressure; (2) synthetic NF CPU cycle breakdown; (3) l3fwd no-drop performance; (4) NAT and LB performance; and (5) MICA key-value store performance.

For NFs, we use large 1500B UDP packets sent at 200 Gbps to stress the I/O working set, and select packet 5-tuples at random to spread the load across cores.

Memory Intensity To explore shRing performance with NFs of various memory intensity, we run FastClick's synthetic WorkPackage module [116] which receives a packet, performs routing, followed by a number of random memory reads from a buffer, and then sends the packet out. We modify WorkPackage to optionally read or overwrite packet payload.

We test 60 configurations: randomly reading 1, 2, 4, 8, or 12 times from a 1MiB, 10MiB, 20MiB, or 40MiB buffer (corresponding to L1, L2, LLC, and larger than LLC sizes), while packet payload is either untouched, read, or overwritten.

For each configuration, we plot shRing throughput, latency, and cycles per packet normalized to privRing; Figure 5.4 shows the results. We find that throughput and

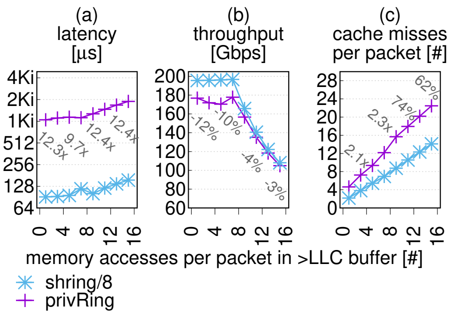


Figure 5.5: High cache pressure decreases performance for shRing and privRing. (Labels show privRing to shRing ratio.)

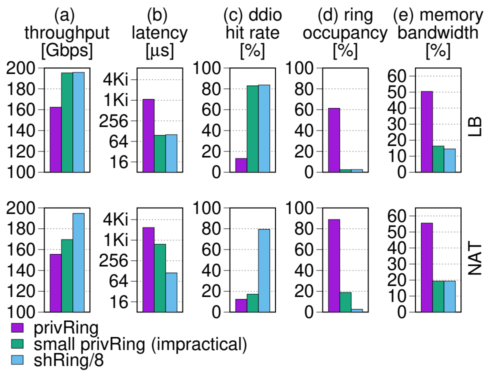


Figure 5.6: LB and NAT performance at 200Gbps load.

latency improve with descriptor sharing ratio: shRing/8 obtains the best throughput and latency followed by shRing/4 and then shRing/2. Moreover, shRing/8 always outperforms privRing (all are above the horizontal line), while shRing/4 and shRing/2 underperform privRing for 54% and 38% of the most memory intensive configurations, respectively. Exploring the configurations where shRing/2 and shRing/4 are less successful than privRing, we find that they consist of 3/16 and 11/16 NFs that read packet payload, and 5/16 and 6/16 configurations that overwrite payload, for shRing/2 and shRing/4, respectively.

Workload Cache Footprint We explore shRing effectiveness as the workload's cache footprint grows. We use the aforementioned synthetic NF with 1–16 random memory accesses per packet in a 40 MiB array. Figure 5.5 shows the results. ShRing mitigates I/O working set induced cache misses, improving application cache hit rates by up to 2.1x, which translates to up to 13% higher throughput and up to 13.1x lower latency. As the workload's cache footprint grows, so does CPU processing time per packet, so eventually cores exceed the CPU cycle budget needed for line rate processing. Both throughput and latency degrade as a result. As the number of processed packets thus decreases, the I/O working set induced cache stress decreases too, and so the gap between cache misses per packet in privRing and shRing shrinks.

Cycle Breakdown We analyse the trade-off between synchronization and cache contention by breaking down the CPU cycles of the sythetic NF described above with 2 random memory accesses per packet in a 40 MiB array runing on all cores.

Figure 5.7a distills our case. It shows the average number of cycles it takes to handle one packet, breaking it down to synchronization overhead ("sync") vs. actual processing time ("orig"). While synchronization overheads are substantial and increase with the level of sharing, we see that it is nevertheless advantageous to pay the cost, as cycles-per-packet improves by about 4% each time we halve the I/O working set size.

The NF throughput, shown in Figure 5.7b, is approximately inversely proportional to cycles-per-packet (Figure 5.7a) as long as the CPU constitutes a bottleneck resource

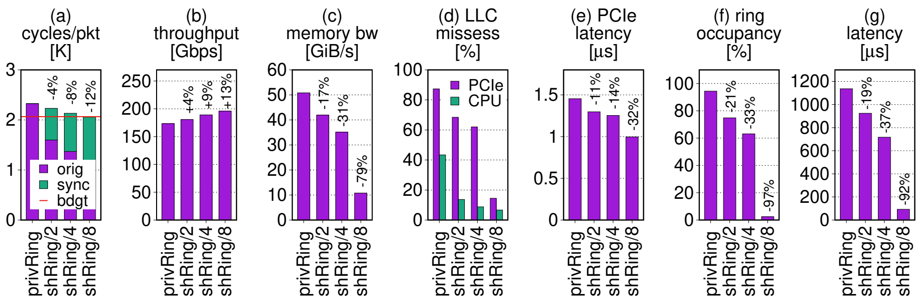


Figure 5.7: ShRing's synchronization costs are significant but are nevertheless worthwhile, as they are cheaper than the overheads associated with privRing's larger I/O working set. When shRing's cycles-per-packet meet the line rate budget (a), its packet processing rate exceeds the packet arrival rate, generating low occupancy in the ring (f) and thus substantially reducing the latency (g).

and line rate is not yet attained. Specifically, let C denote the average number of cycles required to process one packet, let hz (=2.1 GHz) denote the cycles-per-second clock speed of the CPU, and let n (=16) denote the number of running CPU cores, then $n \times \frac{hz}{C}$ is the number of packets that the CPU handles per second, and so Gbps(C) = 1500B $\times$ 8bit $\times$ $n \times \frac{hz}{C}$ is the throughput.

Using this equation, we can compute C_{bdgt}, the budget of per-packet cycles that the system must meet to achieve the 195.6 Gbps line rate (denoted "bdgt" in Figure 5.7a) as follows: C_{bdgt} = 1500B $\times$ 8bit $\times$ $n \times hz$ / 195.6 Gbps = 2061 cycles per packet. Only shRing/8 meets the budget here.

We have argued that the reason underlying shRing's improved performance is its smaller I/O working set, which curbs memory bandwidth consumption by increasing cache efficiency. This argument is directly supported by Figures 5.7c (memory bandwidth) and 5.7d (LLC misses as experienced by both CPU and NIC). In the latter figure, we see that privRing's NIC PCIe miss rate is as high as 85%, which is why privRing's average NIC PCIe read latency grows to 1.45 µs (Figure 5.7e). Such a long PCIe latency is enough to saturate the DMA engines within the NIC (designed to hide PCIe latency with parallelism), and so it hampers the NIC's ability to quickly process rings, which in turn generates high ring occupancy of 94% on average (Figure 5.7f). The implication is that, on average, each privRing packet P must wait for 966 packets (=94% of ring size) to be processed before P is finally processed itself, which explains privRing's high latency (Figure 5.7g).

In contrast, shRing/8's occupancy is small, as it meets the C_{bdgt} budget and so its processing rate (μ) is larger than the arrival rate (λ). Because $\mu > \lambda$, latency is much lower. Even when shRing does not meet the C_{bdgt} budget (the /2 and /4 variants), it improves latency, as its per-packet processing time is lower than in privRing.

No-Drop We study shRing performance with the RFC2544 no-drop rate (NDR) test [138] repeating the Layer-3 MTU packet forwarding (l3fwd) on 8 cores experiment of §4.2.2. This test finds the maximum throughput attainable without loss. We run it once with

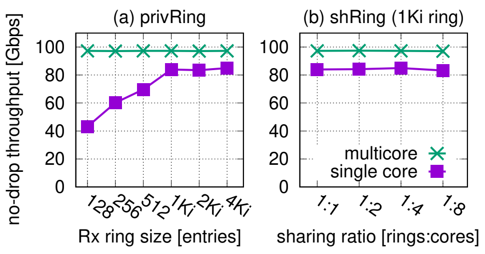


Figure 5.8: DPDK l3fwd no-drop rate. Small privRings work well when traffic is evenly spread across cores but cause drops otherwise. ShRings work well in both cases at a fraction of the buffer size.

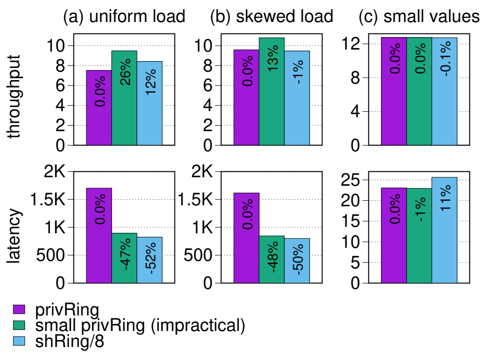


Figure 5.9: ShRing benefits the MICA key-value store with large I/O working sets, non-pathological load imbalance, and high load.

traffic evenly spread across the cores (“multicore”) and again with traffic directed at one of them (“single core”).

Figure 5.8a shows that small rings work well for multiple cores if traffic is evenly spread between them, curbing the load and bursts that each core/ring experiences, which allows the fewer Rx buffers to cope. But small rings cease to deliver when traffic is uneven: the overloaded (“single”) core’s ring overflows and causes packet drops if it is smaller than 1Ki. In contrast, Figure 5.8b shows that one shared 1Ki-ring is enough to sustain optimal NDR of either 8 competing cores (each using 128 entries on average) or just one overloaded core, as shRing allows more loaded cores to use more Rx entries at the expense of their less loaded peers that are adequately served by fewer entries at that particular time.

NAT and LB We use two stateful FastClick NFs as macrobenchmarks: NAT and LB, which cache up to 10M flows using per-core cuckoo hash tables. NAT consistently remaps and rewrites incoming and outgoing packet IP packet headers. LB matches each flow with one of 32 destination servers, maintaining the match for each flow and making new matches with a round-robin policy. NAT is more memory intensive than LB, as it uses two cache entries per flow (one for each direction) while LB uses only one

We show results with a load of 200 Gbps. Results with speeds greater than 170 Gbps are similar, while lower speeds show no difference in throughput and less than 5 µs in latency in favor of privRing due to the synchronization overhead of shRing. The results we show are for the default Rx ring size (i.e, 1024), results for other ring sizes are similar in nature.

Figure 5.6 depicts the resulting (a) throughput, (b) latency, (c) ring occupancy, (d) PCIe (DDIO) miss rate, and (e) memory bandwidth. The results show that shRing/8 outperforms privRing in throughput and latency, which is consistent with previously presented microbenchmarks. This happens because at high offered load the I/O working set starts contending with the CPU for LLC space and memory bandwidth, which slows CPU packet processing. CPU slowdown, in turn, causes ring occupancy to grow, which increases latency (as explained in §4.2.1).

We expect small privRing to perform similarly to shRing/8, and indeed this is the case for LB, but surprisingly small privRing NAT performance is worse than shRing. For NAT, small privRing has a notably lower DDIO hit rate and higher ring occupancy. We speculate that the root cause is that shRing reposts buffers slower as it waits for other cores to make progress, and therefore its working set is slightly smaller because less buffers are exposed to I/O.

ShRing achieves high performance because it shrinks the I/O working set size to fit in the default DDIO portion of the LLC (i.e., two LLC cache ways). When disabling DDIO, namely forbidding NIC DMA writes from allocating ways within the LLC, all ring types achieve only 150 Gbps throughput and 1.3 µs latency, which is 3% and 27% lower than privRing and shRing/8 with default DDIO (not shown in the figure). When assigning all LLC ways to DDIO, privRing performance matches shRing for LB, but it is insufficient for the more memory intensive NAT application, which uses twice as much state and whose throughout improves by less than 5% (also not shown).

Key-Value Store We use the MICA key-value store [63] to show that shRing is applicable beyond NFs and to highlight how workload conditions impact shRing's effectiveness. We run MICA on 8 cores using a single 100GbE NIC, with 128 B keys and 1KiB values.

Figure 5.9a shows the results of a workload with 95% set operations, uniformly distributed among all cores, at the highest possible request rate. This workload satisfies the conditions that make shRing beneficial (§5.1.1)—i.e., (1) no pathological core overload, (2) a large I/O working set, and (3) non-negligible penalty of I/O-related cache misses. ShRing improves MICA throughput by 12% and reduces latency by 52% in this workload; small privRing shows the potential throughput gain from reducing the I/O working set, without shRing's synchronization cost.

Figure 5.9b changes the workload's traffic spread, making it imbalanced (Zipf distribution of skewness 0.99). Consequently, shRing reduces throughput by 1% over privRing but still improves latency by 50%. Figure 5.9c shows the initial workload but with 128B values, which makes the I/O working set small. ShRing makes no throughput improvement and increases latency by 11%. We obtain similar results when lowering the request rate of Figure 5.9a's workload (not shown). In both these cases, shRing adds synchronization overhead which is not offset by I/O working set related improvements, either because the I/O working set was small to begin with (Figure 5.9c) or because the penalty of I/O-related cache misses is negligible (low load).

5.2.3 Pathological Conditions

This section demonstrates shRing's sensitivity to pathological core overload, where one of the shared ring's cores is continuously overloaded compared to the rest. We evaluate shRing/8, referred to as "shRing" here, as well as dynamic shRing/8 (denoted "dshRing") and its ability to gracefully fall back to privRing in pathological conditions. We evaluate two causes for pathological conditions: variability in processing and variability in incoming packet distribution among cores. We also evaluate NAT

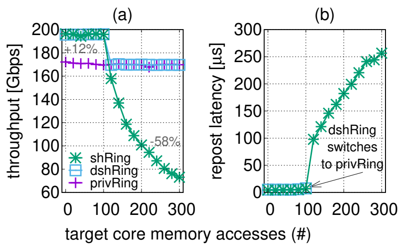


Figure 5.10: When incoming packet rate is fixed, processing variability in one core (e.g., due to increased number of memory accesses) might degrade shRing's throughput and delay descriptor reposting in peer cores. Dynamic shRing falls back on privRing when this happens.

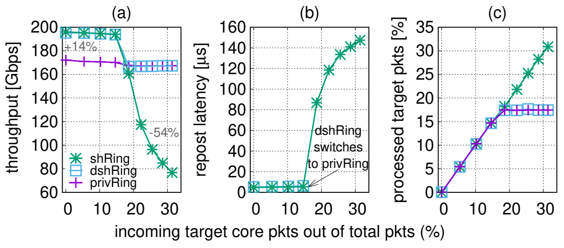


Figure 5.11: When variability manifests as increased rate of packets targeting one specific core (x axis), at some point, it prolongs the latency of peer core descriptor reposting (b); at this point, performance degrades (a) as the target core processing can no longer match the volume of incoming traffic (c).

and LB throughput when offered load switches from non-pathological to pathological over time.

Processing Variability In this experiment, we choose a target core per NIC and control its processing speed by varying the number of memory accesses it performs per packet while all other cores run the synthetic workload described in §4.2.1.

Figure 5.10a depicts the resulting throughput. When the target core's packet processing is fast, shRing and dshRing throughput is 12% higher than privRing, but as the core's processing slows down, shRing throughput declines to 58% lower than privRing. In contrast, dshRing notices that one core is slowing down shRing and switches to privRing, thereby avoiding performance degradation.

Figure 5.10b explains the observed throughput, by showing the time shRing Rx descriptors wait for co-sharing core bitmap updates before being handed back to the NIC. We present only shRing and dshRing, because privRing does not have such delays. In shRing, slow processing on the target core can delay co-sharing cores from making their processed Rx descriptors available for NIC reuse. This effect is negligible when the target core makes less than 100 memory accesses per packet, but subsequently, descriptor wait time increases dramatically (up to 257 µs) and throughput decreases.

Traffic Variability Here, we choose a target core per NIC and vary the percentage of packets directed to it up to 30%. All cores run the synthetic workload. We direct 64 B packets at the target core and 1500 B packets at the others, so that even when receiving 30% of the packets, the target core's incoming traffic is $<$ 3% of total incoming throughput. This means that in principle, the target core's behavior should have negligible effect on overall throughput.

Figure 5.11a shows the throughput in practice. When the packet load on the target core is less than 15%, shRing outperforms privRing and dshRing's heuristic correctly enables shRing. But as load exceeds 15%, the targeted core becomes overloaded and so

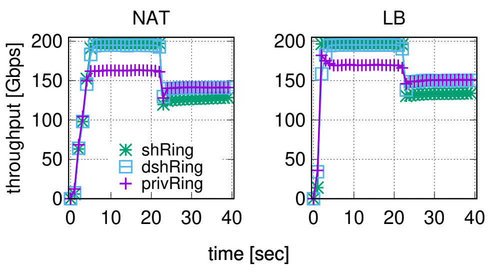


Figure 5.12: NAT and LB throughput when switching from a non-pathological to a pathological workload.

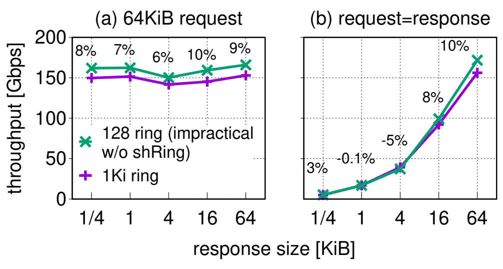


Figure 5.13: Netperf TCP_RR throughput with (a) 64KiB requests for various response sizes and (b) equal request and response sizes. Small rings work better as they reduce the I/O working set.

shRing throughput declines by up to 54%. In contrast, privRing throughput declines by only 3%, since other cores are not affected. DshRing's heuristic identifies when the achieved throughput is too low and that it will not be improved by shRing, and thus switches to privRing.

Figure 5.11b shows that as with processing variability, shRing's throughput decreases because the unloaded cores' Rx descriptor reposting is delayed by the overloaded core.

Figure 5.11c presents the ratio of packets successfully processed by the target core out of all packets. While shRing maintains the target core's ratio of outgoing to incoming packets, the cost is that as more packets target this core, shRing delays receiving on other cores. This results in drops of the 1500 B packets when the target core is overloaded, and thus throughput declines. In contrast, privRing drops excess packets that exceed the target core's processing capacity, and as a result it has at most 17% outgoing packets on the target core.

Handling Variability with Dynamic ShRing We run an experiment where the incoming load switches from non-pathological to pathological after 20 seconds. Figure 5.12 shows NAT and LB throughput sampled every second. DshRing initially uses privRing, but as load increases, it identifies high throughput and memory bandwidth with no overloaded cores and switches to shRing. At 20 seconds, we reconfigure the load generator to send a pathological load, which overloads cores and decreases throughput. DshRing identifies the drop in throughput and switches back to privRing. Consequently, dshRing achieves good performance in both.

5.3 Kernel-Based TCP Sockets

Our implementation and evaluation focus on NFV workloads, which typically bypass the operating system (OS) networking stack and the socket abstraction. This section explores the potential benefit to socket-based TCP applications from deploying shRing in the Linux networking stack.

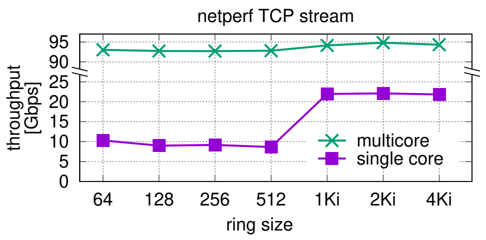


Figure 5.14: Netperf TCP stream throughput. Small rings work well when traffic is spread across multiple cores but cause drops otherwise.

Concerns about the effectiveness of a shRing-based NIC OS driver are that (1) application working sets may be too large for shRing's improved DDIO utilization to matter and (2) even if not, small private rings might not lead to packet loss in the Linux kernel, as opposed to with DPDK.

Because our shRing prototype is DPDK-based, we cannot directly evaluate shRing in the Linux kernel. We therefore use "small privRing" as a proxy, to show the benefit of reducing the I/O working set in the Linux kernel. We run Netperf [40] microbenchmarks to show that: (1) smaller I/O working sets can improve performance of a socket-based application and (2) 1Ki-sized rings are necessary to handle burstiness in the kernel.

Pros of Smaller I/O Working Sets We measure Netperf TCP request-response throughput (sum of Rx and Tx). We use 16 cores and two NICs with two threads per core (one per NIC). For symmetry, we use the same ring size on both sides. In all experiments, the CPU is not the bottleneck.

Figure 5.13a shows the throughput obtained for 64KiB requests and various response sizes. In this setting, small rings outperform large rings by up to 10%. But when the size of the request and the response are equal (Figure 5.13b), the results become less conclusive, e.g., for 1KiB messages throughput is almost the same for both ring sizes, and for 4KiB messages, the small ring's throughput is 5% less than the default.

Cons of Small Private Rings We measure Netperf TCP stream throughput for various private ring sizes, with traffic either directed at a single core or evenly spread among 8 cores. Figure 5.14 (similarly to Figure 5.8) demonstrates that small rings work well for multicore TCP traffic, as the spread of load curbs the bursts each individual core/ring experiences. However, a single ring smaller than 1Ki overflows and causes drops, which cause TCP to back off and thus degrade throughput.

5.4 Related Work

Efficient LLC Utilization DDIO enabled platforms allow NICs to access data faster via the relatively small LLC. Many previous works, unrelated to ring sharing, proposed techniques to improve DDIO efficiency: (1) using small private rings to reduce the I/O working set [97]; (2) placing packets in LLC slices closest to the target processing CPU core [98]; (3) eliminating interference between applications and I/O devices when partitioning the LLC [169]; (4) placing only packet headers in the LLC to reduce LLC contention [197, 271, 282]; and (5) modifying CPUs to prefetch DDIO-written data into mid-level caches and to invalidate data without writeback when possible to conserve memory bandwidth [283]. We show that small private rings are insufficient and propose a ring sharing mechanism that is symbiotic with the last four techniques.

Sharing Within a Core in Software Linux `io_uring` "automatic buffer selection" [284] lets applications pre-register buffers and later consume these via `read/recv` system calls for different file descriptors. Similarly, buffers posted to shRing are pre-registered and later assigned to cores at packet arrival time. But unlike `io_uring`, shRing operates between software and hardware.

Sharing Within a Core in Ethernet NICs When a single core and privilege level have multiple NIC rings, sharing their buffers and CRs to conserve resources is desirable. For example, SRIOV NICs expose a ring per VM on the hypervisor to receive packets missing hardware virtual switching rules, allowing the hypervisor to install matching rules [113, 114]. As the number of VMs exceeds the number of cores, multiple such rings must share a core. To optimize this, NVIDIA NICs recently started sharing ring buffers and CRs within each core [277] via the same firmware changes that we used, which are now publicly available. ShRing, in contrast, shares rings between cores.

Sharing Between Cores in RDMA RDMA applications typically employ queue pairs (QPs) with dedicated buffers to connect between endpoints—consuming GiBs of DRAM [285, 286]. Shared Receive Queues (SRQ), like shRing, decrease memory use by sharing buffers. Whereas SRQ helps RDMA applicability by fitting I/O buffers in server DRAM, shRing improves performance by fitting I/O buffers in server LLC.

Sharing Between Cores in Integrated NICs Nebula [287] is an on-chip integrated NIC design optimized for RPC workloads. Nebula, like shRing, fits the I/O working set within the LLC. Whereas Nebula is applicable only for RDMA-like hardware-terminated protocols, shRing is applicable to typical general purpose Ethernet software network stacks.

Chapter 6

Disentangling the Dual Role of NIC Receive Rings

CPUs parallelize packet processing by assigning each core with its own receive ("Rx") ring. The default size of each Rx ring is >=1Ki entries, to absorb packet bursts. Consequently, the associated I/O working set—all packet buffers pointed to by all Rx rings—can easily exceed the LLC capacity, resulting in decreased performance due to high memory bandwidth. We contend that the I/O working set size is needlessly large because Rx rings have two orthogonal "producer-consumer" functionalities that are unnecessarily entangled: (1) memory allocation, whereby the core "produces" empty buffers that the NIC "consumes" for storing packets; and (2) packet delivery, whereby the NIC "produces" incoming packets that the core "consumes" (receives). We propose rxBisect, a new way for the CPU and NIC to interface, which disentangles these functionalities. RxBisect substitutes each individual Rx ring with two rings that correspond to the two functionalities, such that memory allocation is done independently of packet reception. RxBisect's I/O working set can thus be smaller, as empty buffers are no longer tied to their origin cores, so fewer of them are needed. We implement RxBisect using software emulation. RxBisect improves throughput by up to 20% and reduces latency by up to 11x. The significant latency gains occur when rxBisect meets line rate load whereas the baseline fails to do so.

6.1 RxBisect

RxBisect is a new NIC-CPU interface designed to address the I/O working set problem. RxBisect disentangles the traditional Rx ring's empty buffer allocation and packet reception functionalities, allowing them to be managed independently. RxBisect supports two types of rings: *allocation (Ax) rings*, where a core produces empty buffers for the NIC, and *bisected reception (Bx) rings*, where the NIC produces incoming packets, stored in buffers it consumes from allocation rings.

The crux of rxBisect is that each bisected reception ring *r* can be associated with *several* allocation rings (of different cores), enabling the NIC to deliver packets to *r* as long as *some* allocation ring is not empty. This association is not exclusive—multiple

Bx rings can be associated with overlapping (or identical) sets of Ax rings. In this way, rxBisect turns the union of the set of empty buffers produced by each core into a *globally shared* resource, co-managed by the NIC and software in a lockless manner. This design allows each core to maintain a large bisected reception ring (for absorbing bursts) *without* having to independently maintain a large set of empty buffers—it only requires a small allocation ring. Thus, rxBisect ensures that the I/O working set size is kept below LLC capacity.

In the following, we detail rxBisect's receive-side processing in the NIC (§6.1.1) and by software (§6.1.2). RxBisect does not modify the Tx side and so we do not discuss it. We then pinpoint how rxBisect addresses the shortcomings of the privRing and shRing designs (§6.1.3). Finally, we discuss NIC hardware implementation issues (§6.1.4).

6.1.1 NIC Side

An rxBisect NIC supports two types of receive-side rings: allocation (Ax) and bisected reception (Bx) rings. Each Ax ring entry points to an empty buffer for the NIC to consume in order to store an arriving packet. Each Bx ring entry holds a descriptor through which the NIC notifies software of packet delivery and/or consumption of an empty buffer. Each Bx ring entry holds a pointer to a received packet and the index of the Ax entry and Ax ring that produced the packet's buffer. Bx entries also hold a "done" flag and a corresponding sense-reverse indication flag, whose purpose is the same as in the CRs of the existing NIC interface described in §4.1.

Figure 6.1 depicts rxBisect's flow. Initially (Figure 6.1a), software allocates memory for allocation and bisected reception rings. Allocation rings are filled with entries pointing to MTU-sized buffers to receive packets and bisected reception ring entries are left empty, as the NIC will overwrite them. Crucially, the number of allocated buffers in each Ax ring can be smaller than the size of the Bx rings, which is the key to reducing the I/O working set size (§6.1.3).

Software then *associates* each Bx ring *r* with several Ax rings, indicating to the NIC that buffers from these allocation rings can be used to store packets destined to *r*. Software also *links* each Ax ring *a* with some Bx ring *r*, indicating to the NIC that notifications about buffers consumed from *a* should be delivered through *r*. For simplicity, assume for now that software (1) allocates per-core Ax and Bx rings, (2) links a core's Ax ring to its Bx ring, and (3) associates every Bx ring in a NUMA domain with all the Ax rings residing in that domain. (We discuss another software usage model in §6.1.2.)

When packets arrive (Figure 6.1b), an rxBisect NIC maps each incoming packet to a reception ring exactly as a packet is mapped to an Rx ring in today's NICs, e.g., using RSS. For each packet, the NIC chooses an allocation ring with available buffers according to some policy (e.g., the linked Ax ring or a random non-empty Ax ring if it is empty), and consumes a buffer from that Ax ring to store the packet. In the depicted scenario, a burst of five packets arrives for core 0. The first four packets exhaust its Ax ring, and thus the fifth packet is placed in a buffer allocated from core 1's Ax ring. To deliver each packet, the NIC first DMA-reads a packet buffer address and size from the Ax head descriptor entry, and then DMA-writes the packet's data to the packet buffer. The NIC uses internal synchronization to prevent race conditions, wherein parallel pro-

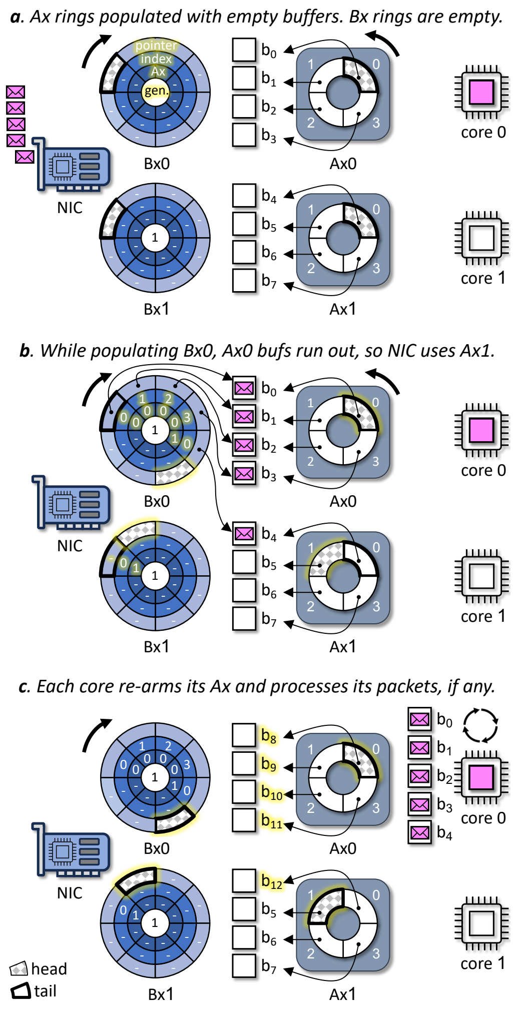

a. Ax rings populated with empty buffers. Bx rings are empty.
pointer
index
Ax
gen.
Bx0
Ax0
Bx1
Ax1
NIC
core 0
core 1
b. While populating Bx0, Ax0 bufs run out, so NIC uses Ax1.
c. Each core re-arms its Ax and processes its packets, if any.
head
tail

Figure 6.1: RxBisect packet reception using two allocation (Ax) and two bisected reception (Bx) rings. Ax ring buffers are shared by both Bx rings through NIC hardware. Highlighted text indicates changes relative to the previous stage. The Ax head advances counter-clockwise and the Bx head advances clockwise.

cessing units consume the same packet buffer when delivering different packets; we expand on this aspect in §6.1.4.

The NIC notifies the receiving core about a delivered packet by DMA-writing the packet buffer's address to the target Bx ring's head descriptor entry. The NIC notifies the allocating core about the consumed buffer by DMA-writing the buffer's Ax ring and entry to the head descriptor of the Bx ring linked to it. In the common case, in which the receiving core is also the allocating core, these notifications are combined into a single descriptor, as shown in Figure 6.1b for buffers $b_0, \dots, b_3$. Finally, the NIC updates the Ax and Bx ring head indexes, optionally raising an interrupt for both. Importantly, the critical path to deliver a packet in terms of DMAs is equivalent in privRing, shRing, and rxBisect (§6.1.4).

Finally (Figure 6.1c), each core processes notifications in its Bx ring (either by polling or due to receiving the aforementioned interrupt). It processes delivered packets and/or replenishes buffers consumed from its Ax ring with empty buffers that it allocates, including notifying the NIC (by means of updating the Ax ring's tail) that new buffers are available. After processing a packet, its buffer is freed back to the system allocator. We expand on software-side processing in §6.1.2.

6.1.2 Software Side

Our discussion relates to software that directly interacts with the NIC, i.e., kernel-bypass applications or in-kernel drivers. Software has flexibility in how it leverages rxBisect to minimize the I/O working set by configuring Ax rings such that the aggregated size of their buffers does not exceed LLC capacity. For example, software can use small (e.g., 128-entry) per-core Ax rings. Or, software can employ a small number of large Ax rings, which are served by a few dedicated allocation cores while the remaining cores focus on packet reception. The discussion below does not assume a specific configuration. In any case, the buffer-free bisected receive rings should remain large, to absorb bursts.

Allocation Mechanism The only constraint rxBisect makes on the software architecture is that it support allocation and freeing of a buffer by different threads/cores. This scenario can occur when a buffer allocated from one core's Ax ring is used to hold a packet destined to a different core, which will then have to free this buffer after processing the packet. Fortunately, many modern multi-core allocators support this allocator capability [288]. In particular, both the Linux kernel and DPDK already use such allocators [289, 290]. At a high level, these allocators employ a two-level design consisting of a shared buffer pool with a per-core caching level, which reduces contention on the shared pool. Caches are filled from the shared pool when they run out of buffers and caches drain excess buffers to the shared pool when they grow beyond some threshold of the cache's expected size (e.g., $1.5\times$ in DPDK). The two-level allocator design is important for amortizing the cost of buffer transfer between cores. Due to it, we observe a difference of at most 15 cycle in average allocator call latency between rxBisect and privRing (where buffers never move across cores).

```
 1 int RxBisect(Ring *ax, Ring *bx,
 2              void **pkts, int len) {
 3   uint32_t idx, npkts = 0, nalloc = 0;
 4   BXEntry *bxe;
 5   while (npkts < len && bxe = consumeBXE(bx)) {
 6     if (bxe->buf != NULL)
 7       pkts[npkts++] = bxe->buf;
 8     if (bxe->idx == ax->idx) {
 9       ax->desc[bxe->idx].buf = alloc_buf();
10       nalloc++;
11     }
12   }
13   if (nalloc > 0) {
14     ax->tail += nalloc;
15     *ax->doorbell = ax->tail;
16   }
17   return npkts;
18 }
```

Listing 6.1: RxBisect disentangled ring receive code.

Receive Flow Listing 6.1 shows the rxBisect receive function, which dequeues a batch of packets for processing as well as handles notifications about allocated buffers that require replenishing. The function receives an allocation ring (`ax`), a bisected reception ring (`bx`), and an output array of packet pointers (`pkts`) of length `len`. It returns the number of received packets. Multiple cores run this code in parallel with different ring arguments, which are all interlinked by NIC hardware to share Rx buffers.

Lines 5–12 check the Bx ring for new entries. The Bx ring check in `consumeBXE(bx)` uses the sense reverse technique (§4.1) to identify ready entries without writing to Bx descriptors. When no Bx entry is ready, this function returns `NULL`; otherwise, it returns the first ready entry and updates the Bx ring's tail. (We omit `consumeBXE`'s code.)

Each returned Bx entry indicates both the pointer of a received buffer, if there is one, and an Ax ring and entry index of a buffer consumed by the NIC. If the Bx entry contains a packet buffer, the packet is stored for processing (lines 6–7). If the Bx entry describes a buffer originating from the core's Ax ring, then a new Ax buffer is allocated in its stead (lines 8–10). Lines 13–16 check if allocation requests were handled and advance the Ax ring's tail index if necessary. It is correct to advance the tail because the NIC consumes buffers in ring order and enqueues all the related notifications to this Bx ring in the same order. The code does not limit the number of allocations, to replenish as many buffers as possible. Nevertheless, the number of allocation iterations is bounded by the Ax ring size, because once it becomes empty, only this function can replenish it.

6.1.3 Comparison to PrivRing and ShRing

By disentangling buffer allocation from reception, rxBisect obtains three advantages: (1) the total number of allocated buffers in Ax rings can be smaller than the total size of the Bx rings, which reduces the I/O working set size; (2) the NIC can use buffers from any Ax ring to populate any Bx ring, so buffer sharing is achieved similarly to shRing;

but (3) in contrast to shRing, sharing buffers is achieved without sharing the cores' packet reception capacity or software synchronization.

Figure 6.2 depicts these differences. The figure compares the minimal I/O working set of two cores running privRing, shRing, and rxBisect. In privRing (Figure 6.2a), each core can work independently, without synchronization or other dependencies on other cores' behavior. However, the existing NIC ring interface necessitates that an Rx ring able to absorb packet bursts must also hold many empty buffers, resulting in an excessive I/O working set.

ShRing (Figure 6.2b) solves privRing's I/O working set problem by sharing a default-sized Rx ring, which can absorb packet bursts, between the cores, but it requires synchronization (with locks) to serialize reposting of buffers to the shared ring and advancing the ring's tail. Crucially, shRing always incurs this latency-increasing per-packet overhead, even if the workload does not suffer from the I/O working set problem (e.g., because the packet rate allows a packet's processing to complete before its eviction from the LLC).

In addition, with shRing, traffic destined to an overloaded core can monopolize the shared ring, preventing other cores from receiving packets. E.g., in Figure 6.2c, because the overloaded core 2 cannot sustain its packet rate, packets destined to it start queueing, eventually filling the ring. Thus, packets to core 1 get dropped, despite it being able to process them.

RxBisect (Figure 6.2d) combines the advantages of privRing and shRing without inheriting their disadvantages. Disentangling the existing Rx ring functionality into bisected reception and allocation rings with independent sizes allows rxBisect to realize a shared buffer pool, based on small per-core allocation rings—but with cores still working independently, without software synchronization or inter-core dependence. Synchronization is only required when moving freed buffers between cores, but as explained in §6.1.2, this synchronization is infrequent and its cost is low.

Handling of Pathological Overload Like shRing, RxBisect reduces the minimal I/O working set by relying on a shared buffer pool, which avoids the over-provisioning of Rx buffers that occurs in privRing. It is therefore natural to ask how rxBisect responds to pathological overload conditions that, in shRing, cause the ring to be monopolized by one or more overloaded cores. As discussed next, thanks to rxBisect's disentanglement of packet reception from buffer allocation, overloaded cores can only "hog" packet buffers, leading to more buffers being allocated, but they cannot interfere with packet reception by other cores.

In rxBisect, per-core Ax rings guarantee the availability of vacant receive buffers for the NIC to consume as long as there is at least one underloaded core with an Ax ring and the buffer allocator can satisfy allocation requests. For allocators preconfigured with a fixed number of buffers, their buffer pool must be large enough to keep satisfying allocation requests even when Bx rings of overloaded cores are full. Consider, for example, the scenario in Figure 6.2d, in which core 2's Ax ring has been exhausted and its Bx ring is nearly full. Suppose that core 2's Bx ring becomes full (which necessarily requires allocation of buffers from core 1's Ax ring) and core 2 slows to a standstill. Core 1's Bx ring indicates that its Ax ring buffers should be repopulated, and because core

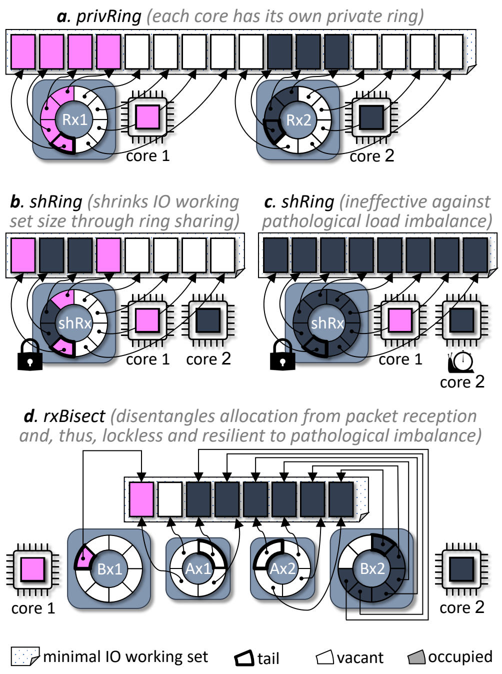

a. privRing (each core has its own private ring)
Rx1
core 1
Rx2
core 2
b. shRing (shrinks IO working set size through ring sharing)
shRx
core 1
core 2
c. shRing (ineffective against pathological load imbalance)
shRx
core 1
core 2
d. rxBisect (disentangles allocation from packet reception and, thus, lockless and resilient to pathological imbalance)
Bx1
Ax1
Ax2
Bx2
core 1
core 2
minimal IO working set
tail
vacant
occupied

Figure 6.2: Summary of the minimal I/O working set with privRing, shRing, and rxBisect. Completion rings are removed for clarity. RxBisect and shRing reduce the I/O working set size compared to privRing. RxBisect, unlike shRing, shares buffers without locking.

1 is not overloaded, it will process these notifications and replenish the in-use buffers. Once these Ax buffers are replenished, core 1's Bx rings can continue to receive packets, using buffers allocated from its Ax ring. Meanwhile, the lack of empty Bx entries on core 2 will cause new packets for this core to be dropped by the NIC. The important property here is that *cores with available cycles can receive packets*, which is not true of shRing (Figure 6.2c). However, the I/O working set might grow due to the additional buffer allocations, beyond the "minimal I/O working set" depicted in the figure.

6.1.4 Hardware Design Considerations

RxBisect provides a new NIC-CPU interface, so its full implementation requires NIC ASIC modifications that are beyond our scope. Instead, we discuss the necessary changes and show that they are compatible with existing NIC mechanisms.

Packet Delivery Mechanically, rxBisect's packet delivery algorithm is analogous to that of existing NICs, which already access two rings (Rx and CR) on packet delivery (§4.1). Consequently, the critical path of DMAs performed for packet delivery in rxBisect is identical to that of the current NICs used by privRing and shRing. In both hardware designs, the NIC DMA-reads (and can prefetch) buffers populated by software in ring structures (an Rx ring in current NICs and an Ax ring in rxBisect). RxBisect NICs can read from multiple Ax rings in parallel, to avoid increasing the critical path. Subsequently, both types of NICs DMA-write packet data followed by DMA-writing a notification in a descriptor of a target ring (a CR in current hardware and a Bx ring in rxBisect). When rxBisect needs to notify different Bx rings about packet delivery and buffer consumption, it performs these DMA-writes in parallel, without increasing the critical path length.

While such parallel writes might increase PCIe bandwidth consumption, this issue can be mitigated by batching notifications. The NIC will delay writing a Bx entry that describes only a buffer consumption until a packet for that Bx ring arrives (or a timeout). Because the waiting Bx entry and the new Bx entry (for packet arrival) are adjacent, they can typically be written with the same PCIe transaction. Similar "completion compression" mechanisms already exist in NVIDIA NICs [291], indicating that this technique is practical.

Ax Ring Access Synchronization High speed NICs process packets destined to different reception rings (Rx rings today and Bx rings in rxBisect) in parallel, via multiple processing units (PUs) [292]. When PUs read buffers from Ax rings, the NIC prefetches them in batches to hide PCIe latency. As a result, an rxBisect NIC needs some form of synchronized PU access to Ax rings, to prevent a race condition in which two PUs consume the same empty buffer when delivering packets to different Bx rings. A possible concern about rxBisect is that internal NIC synchronization will impose similar overheads on the hardware to those imposed on software by shRing's shared ring synchronization, and therefore degrade the NIC's throughput.

Fortunately, this is not the case, as internal NIC synchronization is more efficient than CPU multi-core synchronization. Indeed, existing NICs are *already capable* of performing this type of ring synchronization while delivering traffic at line rate. ShRing, which uses off-the-shelf NICs, associates multiple per-core CRs with the same Rx ring. Thus, with shRing, NIC PUs delivering packets to different CRs associated with the same Rx ring must already prefetch and consume buffers from a single (Rx) ring, which involves internal synchronization. This NIC synchronization is for consuming buffers and is necessary independently of shRing's software synchronization, which synchronizes buffer production. Yet despite internal synchronization, the NIC remains able to deliver traffic at 100 Gbps under shRing [268]. It follows that an rxBisect NIC can reuse existing NIC synchronization mechanisms without degrading NIC throughput, as the NIC synchronization required in shRing and rxBisect are analogous (Bx/Ax rings act similarly to completion/Rx rings, respectively).

6.2 Prototype Implementation

Because an rxBisect implementation requires changing the NIC ASIC, we evaluate a prototype based on software emulation. We implement a "software NIC" framework in DPDK which runs unmodified DPDK applications.

The emulator dedicates a single core that emulates the NIC by consuming packets from the real NIC and producing them to packet-processing "worker" cores. The emulator core resides on a separate NUMA node from the one housing the worker cores, buffers, and the hardware NIC. Because the emulator core only reads and writes ring entries and never touches packet data, processing cores experience DDIO effects as with a real NIC, including I/O working set effects.

The emulator core executes an infinite loop that (1) reads a batch of packet pointers from a hardware Rx ring; (2) dispatches the received packets to the appropriate worker Bx rings according to their RSS hash, without touching their data; and (3) replenishes the hardware ring with buffers it consumes from worker Ax rings. Both hardware Rx ring and worker Bx rings are sized to the sum of all Ax ring sizes.

When the emulated NIC consumes a buffer from an Ax ring, it is not used to store a packet (as a hypothetical rxBisect NIC would), but instead replenishes a buffer in the hardware Rx ring. The buffer will house a packet only after the hardware NIC fills every buffer after it in the hardware ring. The I/O working set is thus larger with emulation compared to a real rxBisect NIC, as the hardware ring's buffers are added to it.

6.2.1 Emulation Fidelity

Performance under emulation should underestimate the performance achievable with a real NIC. To substantiate this claim (without building an rxBisect ASIC), we extend the emulator to support privRing and evaluate the fidelity of its emulation.

PrivRing emulation employs one hardware Rx ring per worker, but still with a single core to poll the hardware rings. When dispatching a packet, the emulator replaces the hardware Rx buffer with a buffer from the receiving worker's (emulated) Rx ring.

We compare native and emulated throughput and latency obtained by network address translation (NAT) and LB NFs with different privRing configurations: default (1 Ki-entry) rings and small 128-entry rings (see §6.3 for the evaluation setup). We

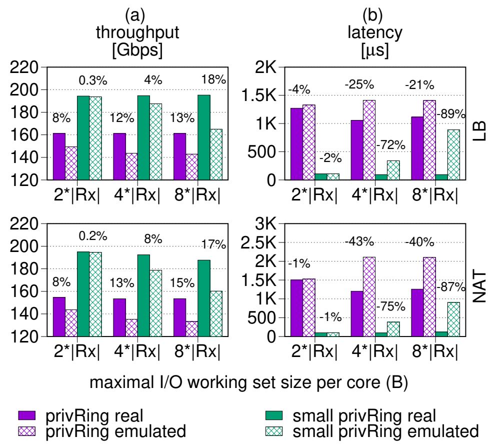


Figure 6.3: PrivRing and small privRing compared to their emulation. Labels show native to emulated ratio.

perform this comparison while varying the allocator's buffer pool size, which is the maximal I/O working set (i.e., maximal number of allocatable buffers). We report the maximal I/O working set as a function of $|Rx|$, which is the minimal I/O working set's size in buffers (i.e., aggregated number of entries in worker Rx rings). Because DPDK's allocator requires this bound to be a power of two, we evaluate with $2|Rx|$, $4|Rx|$, and $8|Rx|$ buffers. (We cannot use $|Rx|$, as then *all* buffers would reside in the initial rings, making replenishing a buffer impossible.)

Figure 6.3 shows that native execution indeed outperforms emulation. The difference is up to 10% when the maximal I/O working set is $2|Rx|$, but as the maximal I/O working set grows, emulation performance declines more sharply than native performance (whose decline can be hard to see). We observe, however, that with a $2|Rx|$-buffer pool, native privRing sometime fails due to running out of buffers. This happens because the NF does not release a buffer until it is transmitted, and as privRing consumes packets quickly, the buffer pool can get exhausted before packet transmissions complete. Emulated privRing avoids this problem, because its packet consumption rate is lower. We manually remove said failed runs from the data shown in Figure 6.3, and avoid using a $2|Rx|$-buffer pool for native execution in our evaluation (see §6.3).

6.3 Evaluation

We evaluate rxBisect using network function benchmarks and the MICA key-value store [63], under ordinary workloads (§6.3.1) and pathological overload conditions (§6.3.2).

System Setup We use two Dell PowerEdge R640 servers. One server is the system under test and the other is the load generator. The load generator runs the stateless Cisco T-Rex packet generator [160], modified to improve latency measurement accuracy [192]. Each server has dual 2.1 GHz Xeon Silver 4216 CPUs, each with 16 cores and an 11-way 22 MiB LLC, and 128 GiB (=4 × 16 GiB) 2933 MHz DDR4 memory. The servers are connected back-to-back via two pairs of 100 Gbps NVIDIA ConnectX-5 Ethernet NICs [29], and configured following NVIDIA's DPDK best performance guidelines [293].

All experiments use the default system and application settings. We report trimmed means of ten runs, i.e., with the minimum and maximum discarded. Standard deviation is always below 5%. Throughput results reflect the traffic rate sent back to the load generator after processing, i.e., they discount packets dropped by the server.

Applications For NFs, we use network address translation (NAT) and load balancing (LB), implemented using FastClick [116]. NAT remaps network addresses and rewrites packet IP headers accordingly. LB assigns flows to one of 32 servers and rewrites packet IP headers accordingly. Both use a 10 M-entry hash table to maintain their state. As a macrobenchmarks, we use the MICA key-value store [63].

Applications run on one CPU of the test server (because for rxBisect, the emulator must run on the other CPU). Unless noted otherwise, we dedicate 8 cores to each NIC.

Evaluated Architectures We compare four packet reception architectures: (1) "rxBisect," with per-core 128-entry Ax rings and 1 Ki-entry Bx rings (emulated); (2) "shRing," with two shared default-sized (1 Ki-entry) Rx rings, one per each NIC shared by 8 cores, implemented with the RxArr variant [268]; (3) "privRing," which uses per-core 1 Ki-entry Rx rings; and (4) "small privRing," which decreases the per-core ring size by $8\times$ to 128 to obtain the same minimal I/O working set size as rxBisect and shRing. However, small privRing is not a practical approach, as its rings cannot absorb packets bursts. We include it as a yardstick.

Comparison Methodology We compare native execution of privRing and shRing to our emulated rxBisect, which underestimates the performance rxBisect would yield if implemented in hardware. Bars showing emulation results contain a diagonal line weave pattern. We size DPDK's buffer pool at $2|Rx|$ in emulated executions and $4|Rx|$ in native executions, where $|Rx|$ is the minimal I/O working set (in buffers). Based on the results of §6.2.1, we expect a $2|Rx|$-sized pool for emulation to yield results comparable to those of real hardware, whereas a $4|Rx|$-sized pool for native execution yields similar performance to a $2|Rx|$-sized pool, without suffering from buffer pool exhaustion.

6.3.1 Ordinary Workloads

No-Drop Throughput Figure 6.4 shows the DPDK l3fwd no-drop throughput with 1500 B packets (§4.2.2) of an individual core ("single core") and of all cores combined ("multicore"). Buffer sharing in rxBisect and shRing enables them to match privRing's no-drop throughput both with multiple cores, each of which handles 1/8 of the traffic, and with a single core, which absorbs bursts by fully utilizing all available buffers.

NAT and LB We run NAT and LB on all 16 cores and load them with 200 Gbps using 1500 B packets. Figure 6.5 shows (a) throughput, (b) latency, (c) ring occupancy,

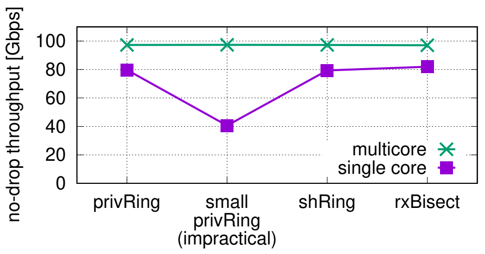


Figure 6.4: Comparison of DPDK l3fwd no-drop throughput for the evaluated architectures, utilizing 8 cores and a single 100 Gbps NIC. Buffer sharing in rxBisect and shRing enables the same throughput as 1 Ki privRing with an $\frac{1}{8}$th of its I/O working set.

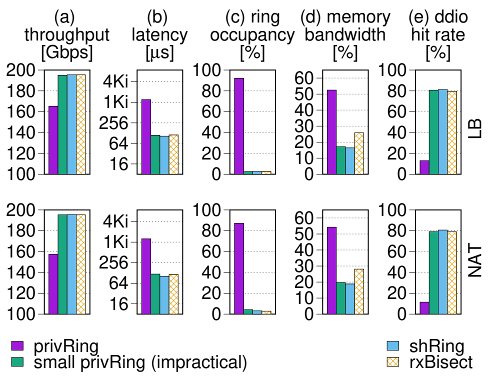


Figure 6.5: LB and NAT performance.

(d) DDIO hit rate, and (e) memory bandwidth. The latter two are measured using Intel PCM [161].

At this load, architectures with a small I/O working set achieve line rate throughput and comparable average latency (100 µs–119 µs). This is due to their effective use of DDIO's default configuration (Figure 6.5e), with 2 LLC ways assigned to it.

Due to its large I/O working set, privRing throughput collapses by up to 20% compared to small privRing. As a result of failing to sustain line rate, privRing's Rx rings fill up, causing latency to increase by 11× due to the additional queueing time. PrivRing can achieve line rate in LB if DDIO's LLC portion is increased to 8 LLC ways, but fails to achieve line rate for NAT even if all the LLC is assigned to DDIO. (Of course, exposing more LLC ways to DDIO is a double-edged sword, as I/O and application memory accesses compete [169].)

Key-Value Store We use the MICA key-value store [63] to evaluate the effect of rxBisect beyond NFs. We run MICA on 8 cores (using a single NIC) with 128 B keys and 1024 B values. MICA maps incoming requests to processing cores by hashing the target key. We use workloads with 95% `PUT` requests at the highest possible rate.

Figure 6.6 shows throughput and latency obtained when the key distribution is (a) uniform or (b) skewed (Zipf with parameter 0.99). The throughput of rxBisect (emulated) is higher than privRing and shRing, by up to 19%. RxBisect underperforms (by up to 12%) only the yardstick small privRing, which is not a practical architecture. In terms of latency, rxBisect is better than privRing but worse than shRing by up to 12%. Given that emulation increases the time spent processing each packet and thus latency, we believe that these results are encouraging.

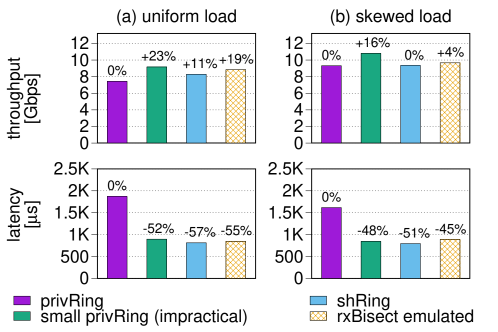


Figure 6.6: Performance of the MICA key-value store with uniform and skewed key distributions. Top labels compare to privRing.

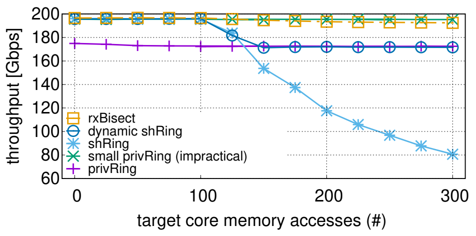


Figure 6.7: Processing variability caused by increased memory accesses has no effect on rxBisect throughput despite ring sharing. In contrast, shRing's approach to ring sharing degrades throughput.

6.3.2 Pathological Overload Conditions

This section shows that rxBisect remains effective under conditions which render shRing ineffective. ShRing is ineffective at low traffic rates, because then the benefit of reducing the I/O working set does not outweigh its synchronization overhead, and under pathological core overload conditions of continuous highly skewed per-packet processing time variability or incoming traffic imbalance.

To address its limitations, ShRing proposed a "dynamic shRing" variant, which switches between shRing and privRing during run time, according to heuristics for determining which is currently more beneficial [268]. We therefore compare to this variant as well.

Processing Variability We evaluate a synthetic NF workload running on all 16 cores and processing 1500 B packets. Packet processing consists of accessing two random addresses in a 40 MiB buffer, performing a routing table lookup (similarly to the l3fwd NF), and sending the packet out. We designate one core per each NIC as the "target core." We modify the load generator to send only 1 Gbps of traffic to the target core while the rest is spread between the other cores. We also tweak the target core's packet processing routine to access memory a configurable number of times per packet (other cores' processing remains unchanged).

Figure 6.7 presents the resulting throughput. RxBisect and small privRing attain close to line rate throughput. However, shRing's throughput degrades by up to 60% as the target core's processing slows down, resulting in its traffic monopolizing the shared ring and blocking other cores from receiving packets (which thus get dropped). Dynamic shRing only offers the best of shRing and privRing, and thus it declines from shRing's near line rate speed to privRing's throughput (never above 178 Gbps, due to its I/O working set) when the target core's processing exceeds 100 memory accesses per packet. Consequently, rxBisect outperforms dynamic shRing by up to 12%.

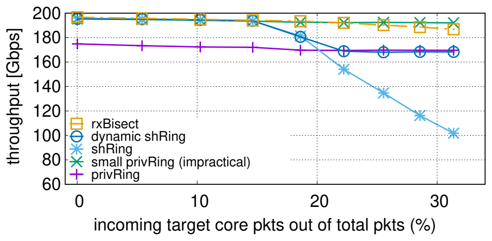


Figure 6.8: Traffic variability caused by increased packet arrival rate on a single core has no effect on rxBisect throughput despite ring sharing. In contrast, shRing's approach to ring sharing degrades throughput.

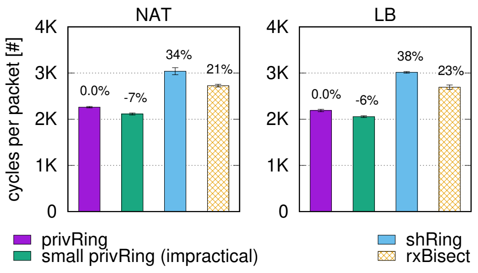


Figure 6.9: Under low load shRing yields no benefits while showing its synchronization overhead. In contrast, rxBisect overhead is minimal. Top labels compare emulated bars to emulated privRing.

Traffic Variability We use the same synthetic NF as above. Only now, all cores run the same packet processing logic, but we modify the load generator configuration to vary the percentage of packets directed at the target core. We use 64 B packets for traffic directed at the target core and 1500 B packets for traffic directed at all other cores, to minimize the target core's impact on overall throughput. For example, even if as much as a third of all packets go to the target core, its traffic (in Gbps) will be less than 3% of all incoming traffic.

Figure 6.8 shows the resulting throughput. RxBisect and small privRing attain near line rate throughput. RxBisect experiences a 5% throughput degradation when the target core's packet rate exceeds 25%, as a result of the emulator core becoming a bottleneck.[1] ShRing throughput declines by up to 49% as the target core's incoming packet rate increases, again due to its traffic monopolizing the shared ring. PrivRing throughput never exceeds 175 Gbps, and so dynamic shRing is outperformed by rxBisect.

Low Traffic Rate Figure 6.9 shows the per-packet processing time (in CPU cycles) for NAT and LB when arriving traffic consists of 1 Gbps of 1500 B packets, which are spread evenly among all cores. ShRing's synchronization overhead results in it spending up to 34% more cycles per packet than privRing and up to 46% more than small privRing. In contrast, rxBisect (which runs under emulation) requires $\approx$ 10% fewer cycles to process a packet than shRing (which runs natively). We can safely conclude that with rxBisect hardware, rxBisect will leverage Rx buffer sharing for a small I/O working set with faster packet processing than in shRing. Based on rxBisect's design, which is similar to privRing, we hypothesize that native rxBisect packet processing efficiency will be comparable to privRing's.

[1]We validate this claim by parallelizing the emulator core's work using two cores, which implement a pipelined version of the emulator's processing. We do not use this version elsewhere because this pipelining necessitates an additional queue between emulator cores, which inflates the I/O working set.

6.4 Related Work

Leaky DMA NICs write directly to two LLC ways with DDIO. Poor I/O working set management causes incoming packets to unnecessarily evict LLC data including other packets being processed; this is also called the "leaky DMA" problem [97]. Several works, complementary to ours, tackled the leaky DMA problem by partitioning the LLC between I/O and applications to avoid interference [169] and by placing only packets headers in the LLC [197, 271, 282].

Host-NIC Interfaces The earliest host-NIC interface similar to rxBisect is described in the context of the Osiris ATM board [294]. This interface consisted of two rings for receiving packets: one for the host to supply empty buffers to the NIC and another for the NIC to store pointers to received packets to be processed by the host. Unlike rxBisect, this interface did not consider sharing receive buffers or multi core CPUs. Other host-NIC interfaces that were described in the past have either used programmed I/O rather than DMA [295] or used linked-lists rather than circular arrays to pass descriptors [296]. The latter was shown to degrade performance due to PCIe latency in shRing [268] and the former was shown to incur high CPU overhead in modern NICs [271]. To address PCIe bottlenecks in high-speed parallelized NIC interfaces, streaming interfaces for NICs have recently been proposed in academia (Ensō [22]) and by industry (NVIDIA MPRQ [21]). Our work is complementary to these interfaces.

Sharing Receive Buffers ShRing [268] is most closely related to rxBisect as it also tackles the I/O working set size problem by sharing buffers between cores. But unlike rxBisect, shRing suffers from performance collapse with imbalanced workloads.

DPDK recently enabled sharing Rx buffers in Ethernet when multiple Rx rings share the same core [277], which is common when SRIOV virtual function Rx rings have corresponding representor Rx rings in the hypervisor [113, 114]. In contrast, rxBisect shares Rx buffers between cores.

RDMA NICs support sharing buffers between cores using Shared Receive Queues (SRQ) [144]. SRQs were created to shrink RDMA queue memory consumption from exceeding DRAM capacities in high performance computing clusters [285, 286]. In contrast, rxBisect shrinks Rx buffer memory consumption to fit in LLC and even inside the smaller DDIO's default portion of the LLC.

Chapter 7

Conclusion

7.1 Autonomous NIC Offloads

Autonomous NIC offloading is a new way to accelerate layer-5 protocols. The approach is appealing because it is nonintrusive, allowing system designers to keep the existing TCP/IP stack intact. We speculate that the non-intrusiveness of the design would give rise to additional layer-5 offloads in the future and perhaps even influence protocol design.

7.2 The Benefits of General-Purpose On-NIC Memory

There is an important class of network applications that move data of messages exclusively based on the associated metadata. For these, the act of transferring the data from the NIC to host memory and back is superfluous and hampers performance. On-NIC memory is now prevalent and, if exported to software, can eliminate the problem.

7.3 ShRing: Networking with Shared Receive Rings

Multicore systems with per-core Ethernet rings use too many receive rings, creating memory pressure that hampers performance. We show that shared receive rings alleviates this problem despite the associated synchronization costs.

7.4 Disentangling the Dual Role of NIC Receive Rings

Today's NIC interface needlessly overcrowds the LLC with I/O buffers to absorb worst-case packet bursts, degrading the performance of high-throughput applications. RxBisect absorbs bursts with much fewer buffers by untangling the dual role of Rx rings.

Bibliography

[1] Mellanox, "Mellanox NIC pricing list effective March 2020." https://store.mellanox.com/, 2020. Accessed: 2020-03-24.

[2] Intel, "Intel ARK: Product Specifications." http://ark.intel.com/, 2017. Accessed: January 2019.

[3] AMD, "AMD EPYC 7000 Series: Product Specifications." https://www.amd.com/en/products/epyc-7000-series, 2019. Accessed: January 2020.

[4] T. P. Morgan, "Intel to challenge AMD with 48 core "Cascade Lake" Xeon AP." https://www.nextplatform.com/2018/11/05/intel-to-challenge-amd-with-48-core-cascade-lake-xeon-ap/, 2018. Accessed: January 2019.

[5] E. Alliance, "The 2018 Ethernet Roadmap." https://ethernetalliance.org/the-2018-ethernet-roadmap/, 2018. Accessed: January 2019.

[6] Mellanox, "Press release: Introduction of ConnectX-3 40GbE." http://www.mellanox.com/page/press_release_item?id=1009, 2013. Accessed: January 2019.

[7] Mellanox, "Press release: Introduction of ConnectX-4 100GbE." http://www.mellanox.com/page/press_release_item?id=1416, 2014. Accessed: January 2019.

[8] Mellanox, "Product brief: ConnectX-6 200Gb/s." www.mellanox.com/related-docs/prod_silicon/PB_ConnectX-6_EN_IC.pdf, 2018. Accessed: January 2019.

[9] C. R. Inc., "400GbE to drive the majority of data center ethernet switch bandwidth within five years." http://www.crehanresearch.com/wp-content/uploads/2018/01/CREHAN-Data-Center-Networking-January-2018.pdf, 2018. Accessed: January 2019.

[10] B. Tester, "Amazon Israel: Tnuvot: Amazon to open the AWS Israel (Tel Aviv) Region by 2023." https://baxtel.com/news/amazon-to-open-the-aws-israel-tel-aviv-region-by-2023, 2021. Accessed: 2023-10-05.

[11] M. Intelligence, "North america structured cabling market size & share analysis - growth trends & forecasts (2023–2028)." https://www.mordorintelligence.com/industry-reports/north-america-structured-cabling-market, 2022. Accessed: 2023-10-05.

[12] Y. Kuperman, E. Moscovici, J. Nider, R. Ladelsky, A. Gordon, and D. Tsafrir, "Paravirtual remote I/O," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 49–65, 2016. `http://dx.doi.org/10.1145/2872362.2872378`.

[13] Y. Moon, S. Lee, M. A. Jamshed, and K. Park, "AccelTCP: Accelerating network applications with stateful TCP offloading," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 77–92, 2020. `https://www.usenix.org/conference/nsdi20/presentation/moon`.

[14] H. Eran, L. Zeno, M. Tork, G. Malka, and M. Silberstein, "NICA: An infrastructure for inline acceleration of network applications," in *USENIX Annual Technical Conference (ATC)*, pp. 345–362, 2019. `https://www.usenix.org/conference/atc19/presentation/eran`.

[15] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg, "Azure accelerated networking: SmartNICs in the public cloud," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 51–66, 2018. `https://www.usenix.org/conference/nsdi18/presentation/firestone`.

[16] S. Han, K. Jang, K. Park, and S. Moon, "PacketShader: A gpu-accelerated software router," in *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*, pp. 195–206, 2010. `https://doi.org/10.1145/1851182.1851207`.

[17] Mellanox, "Connectx®-3 pro product brief." `https://www.mellanox.com/sites/default/files/related-docs/prod_adapter_cards/PB_ConnectX-3_Pro_Card_EN.pdf`, 2011. Accessed: 2019-08-06.

[18] E. Cree, "Checksum offloads." `https://www.kernel.org/doc/html/latest/networking/checksum-offloads.html`, 2016. Accessed: 2020-03-24.

[19] A. Duyck, "Segmentation offloads." `https://www.kernel.org/doc/html/latest/networking/segmentation-offloads.html`, 2016. Accessed: 2020-03-24.

[20] Mellanox, "Connectx®-4 en card product brief." `https://www.mellanox.com/sites/default/files/related-docs/prod_adapter_cards/PB_ConnectX-4_EN_Card.pdf`, 2014. Accessed: 2019-08-06.

[21] T. Tokun, "Bottlenecks in increasing network speeds." `https://netdevconf.info/0x13/session.html?talk-bottlenecks`, 2019. Accessed: 2020-07-14.

[22] H. Sadok, N. Atre, Z. Zhao, D. S. Berger, J. C. Hoe, A. Panda, J. Sherry, and R. Wang, "Ensō: A streaming interface for NIC-Application communication," in *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pp. 1005–1025, 2023. `https://www.usenix.org/conference/osdi23/presentation/sadok`.

[23] Microsoft, "Overview of receive segment coalescing." https://docs.microsoft.com/en-us/windows-hardware/drivers/network/overview-of-receive-segment-coalescing, 2017. Accessed: January 2020.

[24] Microsoft, "Introduction to receive side scaling." https://docs.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling, 2017. Accessed: January 2020.

[25] "VLAN offload tests." https://dpdk-test-plans.readthedocs.io/en/latest/vlan_test_plan.html, 2014. Accessed: 2019-08-30.

[26] H. Tom and d. B. Willem, "Scaling in the linux networking stack." https://www.kernel.org/doc/Documentation/networking/scaling.txt, 2011. Accessed: 2020-03-05.

[27] I. Lesokhin, H. Eran, S. Raindel, G. Shapiro, S. Grimberg, L. Liss, M. Ben-Yehuda, N. Amit, and D. Tsafrir, "Page fault support for network controllers," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 449–466, 2017. https://doi.org/10.1145/3037697.3037710.

[28] T. Oved, "T10-DIF offload." https://www.openfabrics.org/images/2018workshop/presentations/307_TOved_T10-DIFOffload.pdf, 2018. OpenFabrics alliance workshop. Accessed: 2020-05-23.

[29] Mellanox, "Connectx®-5 en card product brief." https://www.mellanox.com/sites/default/files/related-docs/prod_adapter_cards/PB_ConnectX-5_EN_Card.pdf, 2017. Accessed: 2019-08-06.

[30] I. Mark, D. Majd, and T. Anita, "Deployable OVS hardware offloading for 5G telco clouds." https://www.openvswitch.org/support/ovscon2019/day2/1125-dibbiny-tragler-iskra-shern-efraim.pdf, 2019. Accessed: 2020-05-23.

[31] R. Mittal, A. Shpiner, A. Panda, E. Zahavi, A. Krishnamurthy, S. Ratnasamy, and S. Shenker, "Revisiting network support for RDMA," in *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*, pp. 313–326, 2018. https://doi.org/10.1145/3230543.3230557.

[32] H. Li, M. Hao, S. Novakovic, V. Gogte, S. Govindan, D. R. Ports, I. Zhang, R. Bianchini, H. S. Gunawi, and A. Badam, "Leapio: Efficient and portable virtual nvme storage on arm socs," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 591–605, 2020. https://doi.org/10.1145/3373376.3378531.

[33] D. Kim, A. Memaripour, A. Badam, Y. Zhu, H. H. Liu, J. Padhye, S. Raindel, S. Swanson, V. Sekar, and S. Seshan, "Hyperloop: group-based NIC-offloading to accelerate replicated transactions in multi-tenant storage systems," in *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*, pp. 297–312, 2018. https://doi.org/10.1145/3230543.3230572.

[34] W. P. Marts, M. G. F. Dosanjh, W. Schonbein, R. E. Grant, and P. G. Bridges, "Mpi tag matching performance on connectx and arm," in *Proceedings of the 26th European MPI Users' Group Meeting*, EuroMPI '19, 2019. https://doi.org/10.1145/3343211.3343224.

[35] P. Boris and K. Yossi, "UDP GSO offload." https://netdevconf.info/0x12/session.html?udp-segmentation-offload, 2018. Accessed: 2020-05-23.

[36] Mellanox, "Connectx®-6 en card product brief." https://www.mellanox.com/sites/default/files/related-docs/prod_adapter_cards/PB_ConnectX-6_EN_Card.pdf, 2018. Accessed: 2019-08-06.

[37] B. Idan, "Enabling remote persistent memory." https://www.snia.org/educational-library/enabling-remote-persistent-memory-2019, 2019. Persistent memory summit. Accessed: 2020-05-23.

[38] Mellanox, "Mellanox company timeline." https://www.mellanox.com/company/timeline, 2020. Accessed: 2019-08-06.

[39] E. Frachtenberg, "Holistic datacenter design in the open compute project," *Computer*, vol. 45, no. 7, pp. 83–85, 2012. https://doi.ieeecomputersociety.org/10.1109/MC.2012.235.

[40] R. A. Jones, "Netperf: A network performance benchmark (Revision 2.0)." http://www.netperf.org/netperf/training/Netperf.html, 1995. Accessed: August, 2016.

[41] "Congestion control architecture for host congestion." https://netdevconf.info/0x17/sessions/talk/congestion-control-architecture-for-host-congestion.html, 2023. (Accessed: Nov 2023.).

[42] V. Viswanathan, K. Kumar, T. Willhalm, and S. Sakthivelu, "Intel® Memory Latency Checker v3.10." https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html. Accessed: 2023-10-06.

[43] I. Corporation, "Intel data direct I/O technology (intel DDIO): A primer." https://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/data-direct-i-o-technology-brief.pdf, 2012. Accessed: 2020-07-18.

[44] ARM, "ARM cache stashing." https://developer.arm.com/documentation/102407/0100/Cache-stashing, 2017. Accessed: 2022-12-10.

[45] S. Van Doren, "Compute express link," in *IEEE Symposium on High Performance Interconnects (HOTI)*, 2019. https://doi.org/10.1109/HOTI.2019.00017.

[46] V. A. Olteanu, F. Huici, and C. Raiciu, "Lost in network address translation: Lessons from scaling the world's simplest middlebox," in *ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization (HotMiddlebox)*, pp. 19—24, 2015. https://doi.org/10.1145/2785989.2785994.

[47] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein, "Maglev: A fast and reliable software network load balancer," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 523–535, 2016.

[48] B. Fitzpatrick, "Distributed caching with memcached," *Linux Journal*, vol. 2004, p. 5, Aug 2004. `http://dl.acm.org/citation.cfm?id=1012889.1012894`.

[49] R. Labs, "Redis." `https://redis.io/`, 2009. Accessed: 2021-08-06.

[50] I. E. P. Group, "Intel ethernet controller x710/xxv710/xl710." `https://www.intel.com/content/dam/www/public/us/en/documents/release-notes/xl710-ethernet-controller-feature-matrix.pdf`, 2021. Accessed: 2021-08-10.

[51] Intel, "Intel ethernet network adapter e810-2cqda2." `https://ark.intel.com/content/www/us/en/ark/products/192561/intel-ethernet-network-adapter-e810-cqda1.html`, 2021. Accessed: 2021-08-10.

[52] NVIDIA, "ConnectX®-7 Card Product Brief." `https://www.nvidia.com/content/dam/en-zz/Solutions/networking/ethernet-adapters/connectx-7-datasheet-Final.pdf`, 2021. Accessed: 2021-04-16.

[53] Mellanox, "ConnectX®-6 Dx En Card Product Brief." `https://www.mellanox.com/sites/default/files/related-docs/prod_adapter_cards/PB_ConnectX-6_Dx_EN_Card.pdf`, 2020. Accessed: 2020-07-06.

[54] Broadcom, "NetXtreme BCXM57XX User Guide." `https://docs.broadcom.com/doc/INGSRV170-CDUM100-R`, 2015. Accessed: 2021-04-16.

[55] Broadcom, "BCM957504-N1100G." `https://docs.broadcom.com/doc/957504-N1100G-DS`, 2020. Accessed: 2021-04-16.

[56] Broadcom, "NetXtreme-E User Guide." `https://docs.broadcom.com/doc/netxtreme-e-user-guide`, 2021. Accessed: 2021-08-10.

[57] I. E. N. Division, "Ethernet controller e810 datasheet." `https://cdrdv2.intel.com/v1/dl/getContent/331520?wapkw=Intel%2082599%2010%20GbE%20Controller%20Datasheet`, 2019. Accessed: 2021-08-10.

[58] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pp. 53—-64, 2012. `https://doi.org/10.1145/2254756.2254766`.

[59] P. Bodik, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson, "Characterizing, modeling, and generating workload spikes for stateful services," in *Symposium on Cloud Computing (SoCC)*, pp. 241—252, 2010. `https://doi.org/10.1145/1807128.1807166`.

[60] J. Yang, Y. Yue, and K. V. Rashmi, "A large scale analysis of hundreds of in-memory cache clusters at twitter," in *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pp. 191–208, 2020. https://www.usenix.org/conference/osdi20/presentation/yang.

[61] Intel Corporation, "DPDK ping-pong." https://github.com/zylan29/dpdk-pingpong, 2020. (Accessed: May 2021).

[62] "RDMA Core Userspace Libraries and Daemons." https://github.com/linux-rdma/rdma-core, 2005. (Accessed: May 2021.).

[63] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "MICA: A holistic approach to fast in-memory key-value storage," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 429–444, 2014. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/lim.

[64] T. Dierks and E. Rescorla, "The transport layer security (TLS) protocol version 1.2," RFC, Internet Engineering Task Force, 2008. https://rfc-editor.org/rfc/rfc5246.txt.

[65] E. Rescorla, "The transport layer security (TLS) protocol version 1.3," RFC, Internet Engineering Task Force, 2018. https://rfc-editor.org/rfc/rfc8446.txt.

[66] E. Rescorla, "HTTP over TLS," RFC 2818, Internet Engineering Task Force, May 2000. http://www.rfc-editor.org/rfc/rfc2818.txt.

[67] NVM Express Workgroup, "NVMe/TCP transport binding specification." https://nvmexpress.org/wp-content/uploads/NVM-Express-over-Fabrics-1.0-Ratified-TPs.zip, Nov 2018. Accessed: Jan 2020.

[68] M. Slee, A. Agarwal, and M. Kwiatkowski, "Thrift: Scalable cross-language services implementation," *Facebook White Paper*, vol. 5, no. 8, 2007. https://thrift.apache.org/static/files/thrift-20070401.pdf.

[69] Google, "gRPC: a high-performance, open source universal RPC framework." https://grpc.io/, 2015. Accessed: 2020-03-05.

[70] R. A. Jones, "MongoDB: The database for modern applications." https://www.mongodb.com/, 2009. Accessed: August, 2020.

[71] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," in *ACM International Symposium on Computer Architecture (ISCA)*, pp. 158—–169, 2015. https://doi.org/10.1145/2872887.2750392.

[72] J. Hwang, Q. Cai, A. Tang, and R. Agarwal, "TCP $\approx$ RDMA: CPU-efficient remote storage access with i10," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 127–140, 2020. https://www.usenix.org/conference/nsdi20/presentation/hwang.

[73] S. Grimberg, "TCP transport binding for NVMe over Fabrics." https://lwn.net/Articles/772556/, 2018. Accessed: 2020-03-24.

[74] W. Dave, P. Boris, L. Ilya, and Y. Aviad, "Kernel TLS." https://lwn.net/Articles/725721/, 2017. Accessed: 2020-03-24.

[75] I. Marinos, R. N. Watson, M. Handley, and R. R. Stewart, "Disk | Crypt | Net: Rethinking the stack for high-performance video streaming," in *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*, pp. 211–224, 2017. https://doi.org/10.1145/3098822.3098844.

[76] I. Marinos, R. N. Watson, and M. Handley, "Network stack specialization for performance," in *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*, pp. 175–186, 2014. http://doi.acm.org/10.1145/2619239.2626311.

[77] J. Daemen and V. Rijmen, *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media, 2013.

[78] S. Gueron, "Intel advanced encryption standard instructions (AES-NI)," *Intel White Paper*, 2010. https://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf.

[79] D. E. 3rd and P. Jones, "US secure hash algorithm 1 (SHA1)," RFC 3174, Internet Engineering Task Force, Sept. 2001. http://www.rfc-editor.org/rfc/rfc3174.txt.

[80] S. Gulley, V. Gopal, K. Yap, W. Feghali, J. Guilford, and G. Wolrich, "Intel sha extensions," *Intel White Paper*, 2013. https://software.intel.com/content/dam/develop/external/us/en/documents/intel-sha-extensions-white-paper-402097.pdf.

[81] D. Sheinwald, J. Satran, P. Thaler, and V. Cavanna, "Internet protocol small computer system interface (iSCSI) cyclic redundancy check (CRC)/Checksum considerations," RFC 3385, Internet Engineering Task Force, Sept. 2002. http://www.rfc-editor.org/rfc/rfc3385.txt.

[82] V. Gopal, J. Guilford, E. Ozturk, G. Wolrich, W. Feghali, J. Dixon, and D. Karakoyunlu, "Fast CRC computation for iSCSI polynomial using CRC32 instruction," *Intel Corporation*, 2011. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/crc-iscsi-polynomial-crc32-instruction-paper.pdf.

[83] Y. Go, M. A. Jamshed, Y. Moon, C. Hwang, and K. Park, "APUNet: Revitalizing GPU as packet processing accelerator," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 83–96, 2017. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/go.

[84] X. Hu, C. Wei, J. Li, B. Will, P. Yu, L. Gong, and H. Guan, "QTLS: High-performance TLS asynchronous offload framework with intel quickassist technology," in *ACM Symposium on Principals and Practice of Parallel Programming (PPoPP)*, pp. 158–172, 2019. http://doi.acm.org/10.1145/3293883.3295705.

[85] M. S. B. Altaf and D. A. Wood, "Logca: A high-level performance model for hardware accelerators," in *ACM International Symposium on Computer Architecture (ISCA)*, pp. 375–388, 2017. https://doi.org/10.1145/3079856.3080216.

[86] Chelsio Communications, "Chelsio cryptographic offload and acceleration solution overview." https://www.chelsio.com/crypto-solution/, 2018. Accessed: 2018-12-13.

[87] Z. István, D. Sidler, G. Alonso, and M. Vukolic, "Consensus in a box: Inexpensive coordination in hardware," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 425–438, 2016. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/istvan.

[88] Z. István, D. Sidler, and G. Alonso, "Caribou: Intelligent distributed storage," *Proceedings of the VLDB Endowment*, pp. 1202–1213, 2017. https://doi.org/10.14778/3137628.3137632.

[89] J. C. Mogul, "TCP offload is a dumb idea whose time has come," in *USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, pp. 25–30, 2003. https://www.usenix.org/conference/hotos-ix/tcp-offload-dumb-idea-whose-time-has-come.

[90] S. Pope and D. Riddoch, "10Gb/s Ethernet performance and retrospective," *ACM SIGCOMM Computer Communication Review (CCR)*, vol. 37, p. 89–92, Mar. 2007. https://doi.org/10.1145/1232919.1232930.

[91] RedHat, "SegmentSmack and FragmentSmack: IP fragments and TCP segments with random offsets may cause a remote denial of service." https://access.redhat.com/articles/3553061, 2019. Accessed: 2020-08-07.

[92] JSOF research lab, "Ripple20: 19 zero-day vulnerabilities amplified by the supply chain." https://www.jsof-tech.com/ripple20/, 2019. Accessed: 2020-08-07.

[93] L. Foundation, "Why Linux engineers currently feel that TOE has little merit." https://wiki.linuxfoundation.org/networking/toe, 2016. Accessed: 2018-11-06.

[94] "Linux and TCP offload engines." https://lwn.net/Articles/148697/, 2005. Accessed: 2018-11-06.

[95] Microsoft, "Why are we deprecating network performance features (kb4014193)?." https://techcommunity.microsoft.com/t5/Core-Infrastructure-and-Security/Why-Are-We-Deprecating-Network-Performance-Features-KB4014193/ba-p/259053, 2017. Accessed: 2019-08-30.

[96] P. J. Denning, "The working set model for program behavior," *Communications of the ACM (CACM)*, vol. 11, pp. 323—333, May 1968. https://doi.org/10.1145/363095.363141.

[97] A. Tootoonchian, A. Panda, C. Lan, M. Walls, K. Argyraki, S. Ratnasamy, and S. Shenker, "Resq: Enabling slos in network function virtualization," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 283—-297, 2018. `https://www.usenix.org/conference/nsdi18/presentation/tootoonchian`.

[98] A. Farshin, A. Roozbeh, G. Q. Maguire Jr, and D. Kostić, "Make the most out of last level cache in intel processors," in *ACM Eurosys*, pp. 1–17, 2019. `https://doi.org/10.1145/3302424.3303977`.

[99] A. Manousis, R. A. Sharma, V. Sekar, and J. Sherry, "Contention-aware performance prediction for virtualized network functions," in *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*, pp. 270—-282, 2020. `https://doi.org/10.1145/3387514.3405868`.

[100] I. Smolyar, A. Markuze, B. Pismenny, H. Eran, G. Zellweger, A. Bolen, L. Liss, A. Morrison, and D. Tsafrir, "Ioctopus: Outsmarting nonuniform dma," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 101–115, 2020. `https://doi.org/10.1145/3373376.3378509`.

[101] Q. Cai, S. Chaudhary, M. Vuppalapati, J. Hwang, and R. Agarwal, "Understanding host network stack overheads," in *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*, pp. 65—-77, 2021. `https://doi.org/10.1145/3452296.3472888`.

[102] T. Toukan, "[PATCH net-next 08/10] net/mlx4_en: Increase default TX ring size." `https://www.mail-archive.com/netdev@vger.kernel.org/msg173779.html`, 2017. Mellanox. Accessed: June 2021.

[103] Marvell, "FastLinQ 41000 Series Adapters." `https://www.marvell.com/content/dam/marvell/en/public-collateral/ethernet-adaptersandcontrollers/marvell-ethernet-adapters-fastlinq-41000-series-user-guide.pdf`, 2020. Accessed: June 2021.

[104] K. Laatz, "[dpdk-dev] [PATCH v2 0/3] Increase default RX/TX ring sizes." `https://mails.dpdk.org/archives/dev/2018-January/086889.html`, 2018. Intel DPDK. Accessed: June 2021.

[105] J. Brandeburg, "ice: change default number of receive descriptors." `https://marc.info/?l=linux-netdev&m=156771568024262&w=2`, 2019. Intel. Accessed: June 2021.

[106] S. Shuler, "[dpdk-dev] [patch v2 2/2] net/mlx5: add rx and tx tuning parameters." `https://mails.dpdk.org/archives/dev/2018-May/099834.html`, 2018. Mellanox. Accessed: Nov. 2022.

[107] N. Dabilpuram, "[dpdk-dev] [patch 00/44] marvell CNXK ethdev driver." `https://inbox.dpdk.org/dev/20210306153404.10781-4-ndabilpuram@marvell.com/T`, 2021. Marvell. Accessed: 2022-11-28.

[108] "802.3-105 – IEEE standard for Ethernet." https://doi.org/10.1109/IEEESTD.2016.7428776, 2016.

[109] Broadcom, "NetXtreme E-Series PCIe NIC Ethernet Adapters Specification Sheet." https://docs.broadcom.com/doc/netxtreme-e-series-pcie-nic-ethernet-adapters-specification-sheet, 2021. Accessed: 2021-08-10.

[110] FreeBSD, "Network RSS." https://wiki.freebsd.org/NetworkRSS, 2014. Accessed: January 2017.

[111] S. Rixner, "Network virtualization: Breaking the performance barrier: Shared I/O in virtualization platforms has come a long way, but performance concerns remain," *ACM Queue*, vol. 6, pp. 36—44, January 2008. https://doi.org/10.1145/1348583.1348592.

[112] J. Shafer, D. Carr, A. Menon, S. Rixner, A. Cox, W. Zwaenepoel, and P. Willman, "Concurrent direct network access for virtual machine monitors," in *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 306–317, 01 2007. https://doi.org/10.1109/HPCA.2007.346208.

[113] J. Pirko and S. Feldman, "Ethernet switch device driver model (switchdev)." https://www.kernel.org/doc/Documentation/networking/switchdev.txt, 2015. Accessed: 2022-10-10.

[114] O. Gerlitz, H. Hen-Zion, A. Vadai, and R. Efraim, "Introduction to switchdev SR-IOV offloads." https://legacy.netdevconf.info/1.2/slides/oct6/04_gerlitz_efraim_introduction_to_switchdev_sriov_offloads.pdf, 2016. Accessed: 2022-10-10.

[115] Intel Corporation, "DPDK programmer's guide: Poll mode driver." https://doc.dpdk.org/guides/prog_guide/poll_mode_drv.html, 2014. (Accessed: Dec 2022).

[116] T. Barbette, C. Soldani, and L. Mathy, "Fast userspace packet processing," in *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pp. 5—16, 2015. https://doi.org/10.1109/ANCS.2015.7110116.

[117] N. Amritha, S. Sridhar, and P. Kiran, "Hardware acceleration of container networking interfaces." https://legacy.netdevconf.info/0x14/session.html?talk-hardware-acceleration-of-container-networking-interfaces, 2020. Accessed: 2022-10-10.

[118] D. Dan, "Introduction infrastructure programming." https://ipdk.io/documentation/IPDK-io%20-%20Recipes.pdf, 2021. Accessed: 2022-10-10.

[119] Intel, "Application Device Queues." https://www.intel.com/content/www/us/en/architecture-and-technology/ethernet/adq-resource-center.html. Accessed: 2022-09-29.

[120] Z. Yang, B. Walker, J. R. Harris, Y. Li, and G. Cao, "Optimal use of the tcp/ip stack in user-space storage applications with ADQ feature in NIC," in *2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS)*, pp. 363–371, 2020. `https://doi.org/10.1109/ICPADS51040.2020.00056`.

[121] Intel, "PCI-SIG SR-IOV primer: An introduction to SR-IOV technology." `http://www.intel.com/content/www/us/en/pci-express/pci-sig-sr-iov-primer-sr-iov-technology-paper.html`, Jan 2011.

[122] Intel, "Intel® 82544ei gigabit ethernet controller." `https://ark.intel.com/content/www/us/en/ark/products/2276/intel-82544ei-gigabit-ethernet-controller.html`, 2001. Accessed: 2022-10-10.

[123] Intel, "Intel® Xeon processors reach 2 gigahertz for workstations." `https://www.intel.com/pressroom/archive/releases/2001/20010925comp.htm`, 2001. Accessed: 2022-10-10.

[124] Intel, "Intel® 82598eb 10 gigabit ethernet controller." `https://ark.intel.com/content/www/us/en/ark/products/36918/intel-82598eb-10-gigabit-ethernet-controller.html`, 2007. Accessed: 2022-10-10.

[125] Intel, "Intel® Xeon® processor x5482." `https://ark.intel.com/content/www/us/en/ark/products/33088/intel-xeon-processor-x5482-12m-cache-3-20-ghz-1600-mhz-fsb.html`, 2007. Accessed: 2022-10-10.

[126] Intel, "X710-am2." `https://ark.intel.com/content/www/us/en/ark/products/82944/intel-ethernet-controller-x710am2.html`, 2014. Accessed: 2022-10-10.

[127] Intel, "Intel® Xeon® processor e7-2880 v2." `https://ark.intel.com/content/www/us/en/ark/products/75241/intel-xeon-processor-e72880-v2-37-5m-cache-2-50-ghz.html`, 2014. Accessed: 2022-10-10.

[128] Intel, "E810-cam1." `https://ark.intel.com/content/www/us/en/ark/products/187409/intel-ethernet-controller-e810cam1.html`, 2020. Accessed: 2022-10-10.

[129] Intel, "Intel® Xeon® platinum 9282 processor." `https://ark.intel.com/content/www/us/en/ark/products/194146/intel-xeon-platinum-9282-processor-77m-cache-2-60-ghz.html`, 2019. Accessed: 2022-10-10.

[130] B. Burres, D. Daly, M. Debbage, E. Louzoun, C. Severns-Williams, N. Sundar, N. Turbovich, B. Wolford, and Y. Li, "Intel's hyperscale-ready infrastructure processing unit (IPU)," in *Hot Chips*, 2021. `https://doi.org/10.1109/HCS52781.2021.9567455`.

[131] I. Burstein, "NVIDIA data center processing unit (DPU) architecture," in *Hot Chips*, 2021. `https://doi.org/10.1109/HCS52781.2021.9567066`.

[132] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy, "Routebricks: Exploiting parallelism to scale software routers," in *ACM Symposium on Operating Systems Principles (SOSP)*, pp. 15—–28, 2009. `https://doi.org/10.1145/1629575.1629578`.

[133] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker, "NetBricks: Taking the V out of NFV," in *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pp. 203–216, 2016. `https://www.usenix.org/conference/osdi16/technical-sessions/presentation/panda`.

[134] A. Zaostrovnykh, S. Pirelli, L. Pedrosa, K. Argyraki, and G. Candea, "A formally verified NAT," in *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*, pp. 141—154, 2017. `https://doi.org/10.1145/3098822.3098833`.

[135] F. André, S. Gouache, N. L. Scouarnec, and A. Monsifrot, "Don't share, don't lock: Large-scale software connection tracking with krononat," in *USENIX Annual Technical Conference (ATC)*, pp. 453–466, 2018. `https://www.usenix.org/conference/atc18/presentation/andre`.

[136] S. Pirelli and G. Candea, "A simpler and faster NIC driver model for network functions," in *USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2020. `https://www.usenix.org/conference/osdi20/presentation/pirelli`.

[137] S. Pirelli, A. Valentukonytė, K. Argyraki, and G. Candea, "Automated verification of network function binaries," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 585–600, 2022. `https://www.usenix.org/conference/nsdi22/presentation/pirelli`.

[138] S. Bradner and J. McQuaid, "Benchmarking methodology for network interconnect devices," RFC 2544, Internet Engineering Task Force, Mar. 1999. `http://www.rfc-editor.org/rfc/rfc2544.txt`.

[139] A. Kherbouche, "Scaleway natasha performance test." `https://github.com/scaleway/natasha/tree/master/test/perf`, 2018. Accessed: 2022-11-28.

[140] Z. Jia, Z. Liang, and Y. Dai, "Scalability evaluation and optimization of multi-core SIP proxy server," in *International Conference on Parallel Processing (ICPP)*, pp. 43–50, 2008. `10.1109/ICPP.2008.30`.

[141] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich, "An analysis of linux scalability to many cores," in *USENIX Symposium on Operating System Design and Implementation (OSDI)*, (Vancouver, BC), USENIX Association, Oct. 2010. `https://www.usenix.org/conference/osdi10/analysis-linux-scalability-many-cores`.

[142] G. Prekas, M. Kogias, and E. Bugnion, "Zygos: Achieving low tail latency for microsecond-scale networked tasks," in *ACM Symposium on Operating Systems Principles (SOSP)*, pp. 325—–341, 2017. `https://doi.org/10.1145/3132747.3132780`.

[143] A. Daglis, M. Sutherland, and B. Falsafi, "Rpcvalet: Ni-driven tail-aware balancing of µs-scale rpcs," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 35–48, 2019. `https://doi.org/10.1145/3297858.3304070`.

[144] InfiniBand Trade Association (IBTA), "What is InfiniBand." `https://www.infinibandta.org/ibta-specification/`. (Accessed: Dec 2021).

[145] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat, "B4: Experience with a globally-deployed software defined wan," in *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*, pp. 3—14, 2013. `https://doi.org/10.1145/2486001.2486019`.

[146] M. Dalton, D. Schultz, J. Adriaens, A. Arefin, A. Gupta, B. Fahs, D. Rubinstein, E. C. Zermeno, E. Rubow, J. A. Docauer, J. Alpert, J. Ai, J. Olson, K. DeCabooter, M. de Kruijf, N. Hua, N. Lewis, N. Kasinadhuni, R. Crepaldi, S. Krishnan, S. Venkata, Y. Richter, U. Naik, and A. Vahdat, "Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 373–387, 2018. `https://www.usenix.org/conference/nsdi18/presentation/dalton`.

[147] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, P. Cheng, and E. Chen, "Clicknp: Highly flexible and high performance network processing with reconfigurable hardware," in *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*, pp. 1—-14, 2016. `https://doi.org/10.1145/2934872.2934897`.

[148] M. S. Brunella, G. Belocchi, M. Bonola, S. Pontarelli, G. Siracusano, G. Bianchi, A. Cammarano, A. Palumbo, L. Petrucci, and R. Bifulco, "hxdp: Efficient software packet processing on FPGA NICs," in *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pp. 973–990, 2020. `https://www.usenix.org/conference/osdi20/presentation/brunella`.

[149] Z. Zhao, H. Sadok, N. Atre, J. C. Hoe, V. Sekar, and J. Sherry, "Achieving 100Gbps intrusion prevention on a single server," in *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pp. 1083–1100, 2020. `https://www.usenix.org/conference/osdi20/presentation/zhao-zhipeng`.

[150] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, C. Kim, and N. Karri, "Ananta: Cloud scale load balancing," in *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*, pp. 207—-218, 2013. `https://doi.org/10.1145/2486001.2486026`.

[151] Google, "Https encryption on the web." `https://transparencyreport.google.com/https/overview`. Accessed: 2021-08-05.

[152] D. Naylor, A. Finamore, I. Leontiadis, Y. Grunenberger, M. Mellia, M. Munafò, K. Papagiannaki, and P. Steenkiste, "The cost of the "s" in https," in *ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, pp. 133—140, 2014. `https://doi.org/10.1145/2674005.2674991`.

[153] S. Pouyanfar, Y. Yang, S.-C. Chen, M.-L. Shyu, and S. S. Iyengar, "Multimedia big data analytics: A survey," *ACM Compututing Surveys (CSUR)*, vol. 51, p. Article No. 10, Apr 2018. `https://doi.org/10.1145/3150226`.

[154] T. N. Hewage, M. N. Halgamuge, A. Syed, and G. Ekici, "Review: Big data techniques of google, Amazon, Facebook and Twitter," *Oxford University Press Journal of Communications*, vol. 13, pp. 94–100, Feb 2018. `https://doi.org/10.12720/jcm.13.2.94-100`.

[155] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang, "KV-Direct: High-performance in-memory key-value store with programmable NIC," in *ACM Symposium on Operating Systems Principles (SOSP)*, pp. 137–152, 2017. `https://doi.org/10.1145/3132747.3132756`.

[156] S. Grant, A. Yelam, M. Bland, and A. C. Snoeren, "Smartnic performance isolation with fairnic: Programmable networking for the cloud," in *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*, pp. 681—693, 2020. `https://doi.org/10.1145/3387514.3405895`.

[157] M. Liu, T. Cui, H. Schuh, A. Krishnamurthy, S. Peter, and K. Gupta, "Offloading distributed applications onto SmartNICs using ipipe," in *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*, pp. 318–333, 2019. `https://doi.org/10.1145/3341302.3342079`.

[158] R. T. Fielding and G. Kaiser, "The Apache HTTP server project," *IEEE Internet Computing*, vol. 1, pp. 88–90, Jul 1997. `http://dx.doi.org/10.1109/4236.612229`.

[159] Intel Corporation, "L3 forwarding sample application." `https://doc.dpdk.org/guides/sample_app_ug/l3_forward.html`, 2012. (Accessed: May 2021).

[160] Cisco, "TRex: Realistic Traffic Generator." `https://trex-tgn.cisco.com/`. (Accessed: May 2021.).

[161] Intel, "Processor Counter Monitor (PCM)." `https://github.com/opcm/pcm`. Accessed: 2021-02-05.

[162] Mellanox, "Mellanox NEO-Host." `https://www.mellanox.com/sites/default/files/related-docs/prod_management_software/PB_Mellanox_NEO_Host.pdf`, 2018. Accessed: 2021-04-16.

[163] A. Farshin, T. Barbette, A. Roozbeh, G. Q. Maguire Jr., and D. Kostić, "Packetmill: Toward per-core 100-Gbps networking," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 1—-17, 2021. `https://doi.org/10.1145/3445814.3446724`.

[164] A. Farshin, A. Roozbeh, G. Q. M. Jr., and D. Kostić, "Reexamining direct cache access to optimize I/O intensive applications for multi-hundred-gigabit networks," in *USENIX Annual Technical Conference (ATC)*, pp. 673–689, 2020. https://www.usenix.org/conference/atc20/presentation/farshin.

[165] G. P. Katsikas, T. Barbette, D. Kostić, R. Steinert, and G. Q. M. Jr., "Metron: NFV service chains at the true speed of the underlying hardware," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 171–186, 2018. https://www.usenix.org/conference/nsdi18/presentation/katsikas.

[166] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web.," tech. rep., Stanford InfoLab, 1999.

[167] R. Child, S. Gray, A. Radford, and I. Sutskever, "Generating long sequences with sparse transformers," *arXiv preprint arXiv:1904.10509*, 2019.

[168] M. Dobrescu, K. Argyraki, and S. Ratnasamy, "Toward predictable performance in software packet-processing platforms," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 141–154, 2012. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/dobrescu.

[169] Y. Yuan, M. Alian, Y. Wang, R. Wang, I. Kurakin, C. Tai, and N. S. Kim, "Don't forget the I/O when allocating your LLC," in *ACM International Symposium on Computer Architecture (ISCA)*, pp. 112–125, 2021. https://doi.org/10.1109/ISCA52012.2021.00018.

[170] J. Fried, Z. Ruan, A. Ousterhout, and A. Belay, "Caladan: Mitigating interference at microsecond timescales," in *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pp. 281–297, 2020. https://www.usenix.org/conference/osdi20/presentation/fried.

[171] R. Neugebauer, G. Antichi, J. F. Zazo, Y. Audzevich, S. López-Buedo, and A. W. Moore, "Understanding pcie performance for end host networking," in *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*, pp. 327—–341, 2018. https://doi.org/10.1145/3230543.3230560.

[172] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan, "Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 361–378, 2019. https://www.usenix.org/conference/nsdi19/presentation/ousterhout.

[173] M. Girondi, M. Chiesa, and T. Barbette, "High-speed connection tracking in modern servers," in *IEEE International Conference on High Performance Switching and Routing (HPSR)*, pp. 1–8, 2021. https://doi.org/10.1109/HPSR52026.2021.9481841.

[174] J. Chang, Y.-H. Chen, W.-M. Chan, S. P. Singh, H. Cheng, H. Fujiwara, J.-Y. Lin, K.-C. Lin, J. Hung, R. Lee, H.-J. Liao, J.-J. Liaw, Q. Li, C.-Y. Lin, M.-C. Chiang,

and S.-Y. Wu, "A 7nm 256mb SRAM in high-k metal-gate FinFET technology with write-assist circuitry for low-v_{MIN} applications," in *IEEE International Solid-State Circuits Conference (ISSCC)*, pp. 206–207, 2017. https://doi.org/10.1109/ISSCC.2017.7870333.

[175] CSET, "AI chips: What they are and why they matter." https://cset.georgetown.edu/publication/ai-chips-what-they-are-and-why-they-matter/. Accessed: 2021-08-28.

[176] Daniel Nenni, "7nm price is about right." https://semiwiki.com/forum/index.php?threads/5nm-wafer-cost-very-high.13101/#post-44127. SemiWiki forum discussion of CSET wafer prices report. Accessed: 2021-08-28.

[177] CDW, "100GbE adapter prices." https://www.cdw.com/search/networking/network-adapters/ethernet-adapters/?w=RB1&ln=0&filter=af_networking_data_link_protocol_rb1_ss%3a(%22100+Gigabit+Ethernet%22)%2caf_networking_form_factor_rb1_ss%3a(%22Plug-in+ard%22)&SortBy=PriceAsc. Accessed: 2021-08-31.

[178] Intel, "3rd Generation Intel® Xeon® Scalable Processors." https://ark.intel.com/content/www/us/en/ark/products/series/204098/3rd-generation-intel-xeon-scalable-processors.html. Accessed: 2021-08-31.

[179] Paul Alcorn, "AMD Shows New 3D V-Cache Ryzen Chiplets, up to 192MB of L3 Cache, 15% Gaming Improvement (Updated)." https://www.tomshardware.com/uk/news/amd-shows-new-3d-v-cache-ryzen-chiplets-up-to-192mb-of-l3-cache-per-chip-15-gaming-improvement. Accessed: 2021-08-28.

[180] P. Jurkiewicz, G. Rzym, and P. Boryło, "Flow length and size distributions in campus internet traffic," *Computer Communications*, vol. 167, pp. 15–30, 2021. https://www.sciencedirect.com/science/article/pii/S0140366420320223.

[181] J. Garcia, T. Korhonen, R. Andersson, and F. Västlund, "Towards video flow classification at a million encrypted flows per second," in *2018 IEEE 32nd International Conference on Advanced Information Networking and Applications (AINA)*, pp. 358–365, 2018. https://doi.org/10.1109/AINA.2018.00061.

[182] "CAIDA dataset." https://www.caida.org/catalog/datasets/trace_stats/. (Accessed: May 2021.).

[183] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," in *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*, pp. 123—137, 2015. https://doi.org/10.1145/2785956.2787472.

[184] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, pp. 267—280, 2010. https://doi.org/10.1145/1879141.1879175.

[185] A. Adya, D. Myers, H. Qin, and R. Grandl, "Fast key-value stores: An idea whose time has come and gone (HotOS'19 talk slides)." https://ai.google/research/pubs/pub48030. (Accessed: Aug 2019).

[186] P. Stuedi, A. Trivedi, and B. Metzler, "Wimpy nodes with 10gbe: Leveraging one-sided operations in soft-rdma to boost memcached," in *USENIX Annual Technical Conference (ATC)*, pp. 347–353, 2012. https://www.usenix.org/conference/atc12/technical-sessions/presentation/stuedi.

[187] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, pp. 58–75, 2005.

[188] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," in *International Colloquium on Automata, Languages, and Programming*, pp. 693–703, 2002. https://doi.org/10.14778/1454159.1454225.

[189] A. Metwally, D. Agrawal, and A. El Abbadi, "Efficient computation of frequent and top-k elements in data streams," in *International conference on database theory*, pp. 398–412, 2005. https://doi.org/10.1007/978-3-540-30570-5_27.

[190] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek, "The click modular router," in *ACM Symposium on Operating Systems Principles (SOSP)*, pp. 217—–231, 1999. https://doi.org/10.1145/319151.319166.

[191] S. Li, H. Lim, V. W. Lee, J. H. Ahn, A. Kalia, M. Kaminsky, D. G. Andersen, S. O, S. Lee, and P. Dubey, "Architecting to achieve a billion requests per second throughput on a single key-value store server platform," in *ACM International Symposium on Computer Architecture (ISCA)*, pp. 476–488, 2015. https://doi.org/10.1145/2749469.2750416.

[192] M. Primorac, E. Bugnion, and K. Argyraki, "How to measure the killer microsecond," in *Proceedings of the Workshop on Kernel-Bypass Networks*, pp. 37—–42, 2017. https://doi.org/10.1145/3098583.3098590.

[193] Mellanox, "Mellanox ASAP2." https://www.mellanox.com/files/doc-2020/sb-asap2.pdf, 2020. Accessed: 2022-01-05.

[194] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Wisckey: Separating keys from values in ssd-conscious storage," in *USENIX Conference on File and Storage Technologies (FAST)*, pp. 133–148, 2016. https://www.usenix.org/conference/fast16/technical-sessions/presentation/lu.

[195] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet, "Alphasort: A risc machine sort," in *ACM SIGMOD International Conference on Management of Data*, pp. 233—-242, 1994. https://doi.org/10.1145/191839.191884.

[196] M. Flajslik and M. Rosenblum, "Network interface design for low latency request-response protocols," in *USENIX Annual Technical Conference (ATC)*, pp. 333–346, 2013. https://www.usenix.org/conference/atc13/technical-sessions/presentation/flajslik.

[197] S. Goswami, N. Kodirov, C. Mustard, I. Beschastnikh, and M. Seltzer, "Parking packet payload with p4," in *ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, pp. 274—–281, 2020. `https://doi.org/10.1145/3386367.3431295`.

[198] Mellanox, "Device memory programming model." `https://docs.mellanox.com/display/OFEDv502180/Programming#Programming-DeviceMemoryProgramming`. Accessed: 2021-11-20.

[199] Q. Want, Y. Lu, and J. Shu, "Sherman: A write-optimized distributed B+Tree index on disaggregated memory," in *To appear in ACM SIGMOD International Conference on Management of Data*, 2022. `https://arxiv.org/abs/2112.07320`.

[200] P. Reisner, "The distributed replicated block device (drbd) driver." `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=b411b3637fa71fce9cf2acf0639009500f5892fe`, 2009. Accessed: 2020-03-24.

[201] S. French, "Cifs vfs - advanced common internet file system for linux." `https://linux-cifs.samba.org/`, 2007. Accessed: 2020-03-24.

[202] P. Breuer, A. Marín-López, and A. Ares, "The network block device," *Linux Journal*, vol. 73, 05 2000. `https://www.linuxjournal.com/article/3778`.

[203] A. Alex and Y. Dmitry, "Open-iscsi high-performance initiator for linux." `https://lwn.net/Articles/126530/`, 2005. Accessed: 2020-03-24.

[204] D. D. Clark and D. L. Tennenhouse, "Architectural considerations for a new generation of protocols," in *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*, pp. 200–208, 1990. `https://doi.org/10.1145/99508.99553`.

[205] M. Chowdhury and I. Stoica, "Coflow: A networking abstraction for cluster applications," in *ACM Workshop on Hot Topics in Networks (HotNets)*, pp. 31—-36, 2012. `https://doi.org/10.1145/2390231.2390237`.

[206] A. Raza, P. Sohal, J. Cadden, J. Appavoo, U. Drepper, R. Jones, O. Krieger, R. Mancuso, and L. Woodman, "Unikernels: The next stage of linux's dominance," in *USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, pp. 7—13, 2019. `https://doi.org/10.1145/3317550.3321445`.

[207] Intel Corporation, "DPDK: Data plane development kit." `http://dpdk.org`, 2010. (Accessed: May 2016).

[208] L. Rizzo, "Netmap: A novel framework for fast packet I/O," in *USENIX Annual Technical Conference (ATC)*, 2012. `https://www.usenix.org/conference/atc12/technical-sessions/presentation/rizzo`.

[209] A. Kaufmann, T. Stamler, S. Peter, N. K. Sharma, A. Krishnamurthy, and T. Anderson, "Tas: TCP acceleration as an OS service," in *ACM Eurosys*, pp. 1–16, 2019. `https://doi.org/10.1145/3302424.3303985`.

[210] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe, "Arrakis: The operating system is the control plane," in *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pp. 1–16, 2014. https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-peter_simon.pdf.

[211] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, "mtcp: a highly scalable user-level TCP stack for multicore systems," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 489–502, USENIX Association, 2014. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/jeong.

[212] S. Pope and D. Riddoch, "Introduction to OpenOnload," tech. rep., Solarflare Communication, 2011. https://www.openonload.org/.

[213] Mellanox, "Messaging accelerator (vma)." https://www.mellanox.com/products/software/accelerator-software/vma, 2013. Accessed: 2020-02-05.

[214] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion, "IX: A protected dataplane operating system for high throughput and low latency," in *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pp. 49–65, 2014. https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-belay.pdf.

[215] S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy, "MegaPipe: A new programming interface for scalable network I/O," in *USENIX Symposium on Operating System Design and Implementation (OSDI)*, (Hollywood, CA), pp. 135–148, USENIX, 2012. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/han.

[216] A. Limited, "ARMv8 instruction set overview," tech. rep., ARM, 2011. https://www.element14.com/community/servlet/JiveServlet/previewBody/41836-102-1-229511/ARM.Reference_Manual.pdf.

[217] J. Stuecheli, "Power8," in *Hot Chips*, pp. 1–20, 2013. https://doi.org/10.1109/HOTCHIPS.2013.7478303.

[218] WolfSSL, "WolfSSL ARMv8 support." https://www.wolfssl.com/wolfssl-on-armv8-lemaker-2/, 2018. Accessed: 2020-04-05.

[219] V. Gopal, S. Gulley, W. Feghali, D. Zimmerman, and I. Albrekht, "Improving openssl performance," tech. rep., Intel Corporation, 2015. https://software.intel.com/en-us/articles/improving-openssl-performance.

[220] Intel, "Intel quickassist adapter 8950 product brief." https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/quickassist-adapter-8950-brief.pdf, 2015. Accessed: 2018-12-13.

[221] H. Franke, J. Xenidis, C. Basso, B. M. Bass, S. S. Woodward, J. D. Brown, and C. L. Johnson, "Introduction to the wire-speed processor and architecture," *IBM*

Journal of Research and Development, vol. 54, no. 1, pp. 3:1–3:11, 2010. https://doi.org/10.1147/JRD.2009.2036980.

[222] M. Shah, R. Golla, G. Grohoski, P. Jordan, J. Barreh, J. Brooks, M. Greenberg, G. Levinsky, M. Luttrell, C. Olson, *et al.*, "Sparc t4: A dynamically threaded server-on-a-chip," *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, vol. 32, no. 2, pp. 8–19, 2012. https://doi.org/10.1109/MM.2012.1.

[223] WolfSSL, "Wolfssl/wolfcrypt async with intel quickassist." https://www.wolfssl.com/docs/intel-quickassist/, 2016. Accessed: 2020-02-05.

[224] V. Support, "How to disable TCP chimney, TCP/IP offload engine and/or TCP segmentation offload." https://www.veritas.com/content/support/en_US/article.100031033, 2015. Accessed: 2020-03-24.

[225] E. S. Network, "NIC Tuning - Chelsio 10Gig NIC, Linux and FreeBSD." https://fasterdata.es.net/host-tuning/nic-tuning/, 2012. Accessed: 2020-03-24.

[226] T. Network, "TCP offload's promises and limitations for enterprise networks." https://searchdatacenter.techtarget.com/tip/TCP-offloads-promises-and-limitations-for-enterprise-networks, 2013. Accessed: 2020-03-24.

[227] A. Singhvi, A. Akella, D. Gibson, T. F. Wenisch, M. Wong-Chan, S. Clark, M. M. K. Martin, M. McLaren, P. Chandra, R. Cauble, H. M. G. Wassel, B. Montazeri, S. L. Sabato, J. Scherpelz, and A. Vahdat, "1RMA: Re-envisioning remote memory access for multi-tenant datacenters," in *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*, pp. 708—–721, 2020. https://doi.org/10.1145/3387514.3405897.

[228] D. Gallatin, "Netflix view on TOE," Jan 2020. Private email communication with a Netflix engineer; quote approved by Netflix and used with permission.

[229] A. Kaufmann, S. Peter, N. K. Sharma, T. Anderson, and A. Krishnamurthy, "High performance packet processing with FlexNIC," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 67–81, 2016. http://dx.doi.org/10.1145/2872362.2872367.

[230] E. Kohler, M. Handley, and S. Floyd, "Datagram Congestion Control Protocol (DCCP)," RFC, Internet Engineering Task Force, 2006. https://rfc-editor.org/rfc/rfc4340.txt.

[231] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner, "Internet small computer systems interface (iSCSI)," RFC 3720, Internet Engineering Task Force, Apr. 2004. http://www.rfc-editor.org/rfc/rfc3720.txt.

[232] NVM Express Workgroup, "NVM Express (NVMe) specification – Revision 1.2." http://www.nvmexpress.org/wp-content/uploads/NVM-Express-1_2-Gold-20141209.pdf, Nov 2014. Accessed: Jan 2015.

[233] A. C. J. Salowey and D. McGrew, "AES Galois Counter Mode (GCM) Cipher Suites for TLS," RFC, Internet Engineering Task Force, 2008. https://tools.ietf.org/html/rfc5288.

[234] V. Krasnov, "It takes two to ChaCha (Poly)." https://blog.cloudflare.com/it-takes-two-to-chacha-poly/, 2016. The Cloudflare Blog. Accessed: 2020-03-05.

[235] N. Sullivan, "Padding oracles and the decline of CBC-mode cipher suites." https://blog.cloudflare.com/padding-oracles-and-the-decline-of-cbc-mode-ciphersuites/, 2016. The Cloudflare Blog. Accessed: 2020-03-05.

[236] N. Sullivan, "Do the ChaCha: better mobile performance with cryptography." https://blog.cloudflare.com/do-the-chacha-better-mobile-performance-with-cryptography/, 2015. The Cloudflare Blog. Accessed: 2020-03-05.

[237] V. Krasnov, "How "expensive" is crypto anyway?." https://blog.cloudflare.com/how-expensive-is-crypto-anyway/, 2017. The Cloudflare Blog. Accessed: 2020-03-05.

[238] F. Kiefer, "Improving AES-GCM performance." https://blog.mozilla.org/security/2017/09/29/improving-aes-gcm-performance/, 2017. Mozilla Security Blog. Accessed: 2020-03-05.

[239] B. Pismenny, "Kernel TLS socket API." https://github.com/openssl/openssl/pull/5253, 2018. Accessed: 2019-08-27.

[240] B. Pismenny, "Kernel TLS receive side." https://github.com/openssl/openssl/pull/7848, 2018. Accessed: 2019-08-27.

[241] B. Pismenny, "KTLS sendfile." https://github.com/openssl/openssl/pull/8727, 2019. Accessed: 2019-08-27.

[242] L. Ilya, P. Boris, and Y. Aviad, "tls: Add generic NIC offload infrastructure." https://lwn.net/Articles/738847/, 2017. Accessed: 2019-08-27.

[243] P. Boris and L. Ilya, "TLS offload rx, netdev & mlx5." https://lwn.net/Articles/759052/, 2018. Accessed: 2019-08-27.

[244] J. Baldwin, "TLS in the kernel." https://reviews.freebsd.org/D21277, 2019. FreeBSD Kernel patches. Accessed: 2020-03-24.

[245] J. Axboe, "Fio - Flexible I/O tester." https://fio.readthedocs.io/en/latest/fio_doc.html, 2014.

[246] A. Tirumala, F. Qin, J. Dugan, J. Ferguson, and K. Gibbs, "Iperf: The tcp/udp bandwidth measurement tool," *dast. nlanr. net/Projects*, p. 38, 2005. https://iperf.fr/.

[247] W. Reese, "Nginx: The high-performance web server and reverse proxy," *Linux J.*, vol. 2008, Sept. 2008.

[248] K. Shu, "Optimize redis with nextgen nvm." https://www.snia.org/sites/default/files/SDC/2018/presentations/PM/Shu_Kevin_Optimize_Redis_with_NextGen_NVM.pdf, 2018. Intel. Accessed: 2019-08-06.

[249] Intel, "Accelerating redis with intel dc persistent memory." https://ci.spdk.io/download/2019-summit-prc/02_Presentation_13_Accelerating_Redis_with_Intel_Optane_DC_Persistent_Memory_Dennis.pdf, 2019. Accessed: 2019-08-06.

[250] "RocksDB: A persistent key-value store." https://rocksdb.org, 2012. (Accessed: May 2020.).

[251] "Redis labs." https://redislabs.com, 2011. (Accessed: May 2020.).

[252] W. Glozer, "Wrk - a HTTP benchmarking tool." https://github.com/wg/wrk.git, 2012. Accessed: 2019-08-06.

[253] R. Labs, "Memtier benchmark." https://github.com/RedisLabs/memtier_benchmark, 2013. Accessed: 2029-03-05.

[254] N. Dukkipati, T. Refice, Y. Cheng, J. Chu, T. Herbert, A. Agarwal, A. Jain, and N. Sutin, "An argument for increasing TCP's initial congestion window," in *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*, pp. 26–33, 2010. https://doi.org/10.1145/1823844.1823848.

[255] V. Paxson, "End-to-end internet packet dynamics," in *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*, pp. 139–152, 1997. https://doi.org/10.1145/263109.263155.

[256] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center TCP (DCTCP)," in *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*, pp. 63–74, 2010. https://doi.org/10.1145/1851275.1851192.

[257] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, "FaRM: Fast remote memory," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 401–414, 2014. https://www.usenix.org/conference/nsdi14/technical-sessions/dragojevi{ć}.

[258] A. Kalia, M. Kaminsky, and D. G. Andersen, "FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram rpcs," in *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pp. 185–201, 2016. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/kalia.

[259] S. Novakovic, Y. Shan, A. Kolli, M. Cui, Y. Zhang, H. Eran, B. Pismenny, L. Liss, M. Wei, D. Tsafrir, and M. Aguilera, "Storm: A fast transactional dataplane for remote data structures," in *ACM International Systems and Storage Conference (SYSTOR)*, (New York, NY, USA), pp. 97—-108, Association for Computing Machinery, 2019. https://doi.org/10.1145/3319647.3325827.

[260] E. Rescorla and N. Modadugu, "Datagram Transport Layer Security Version 1.2," RFC, Internet Engineering Task Force, 2012. `https://rfc-editor.org/rfc/rfc6347.txt`.

[261] R. J. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, and M. Kalla, "Stream control transmission protocol," RFC 2960, Internet Engineering Task Force, Oct. 2000. `http://www.rfc-editor.org/rfc/rfc2960.txt`.

[262] J. Iyengar and M. Thomson, "QUIC: A UDP-based multiplexed and secure transport," RFC draft, Internet Engineering Task Force, 2020. `https://tools.ietf.org/html/draft-ietf-quic-transport-34`.

[263] C. Krasic, M. Bishop, and E. A. Frindell, "QPACK: Header compression for HTTP/3," RFC draft, IETF, 2020. `https://tools.ietf.org/html/draft-ietf-quic-qpack-20`.

[264] J. Guo, "[patch 01/13] net/idpf/base: introduce base code." `https://yhbt.net/lore/dpdk-dev/LV2PR11MB5997FEBB6B1EC4B900EF6CFFF7379@LV2PR11MB5997.namprd11.prod.outlook.com/T/`, 2022. Intel IDPF. Accessed: 2022-11-28.

[265] B. Forrest, "[patch net-next 00/16] gve: Introduce DQO descriptor format." `https://lore.kernel.org/netdev/20210624180632.3659809-1-bcf@google.com/`, 2021. Google Virtual Ethernet NIC. Accessed: 2022-11-28.

[266] S. S. Mukherjee, B. Falsafi, M. D. Hill, and D. A. Wood, "Coherent network interfaces for fine-grain communication," in *ACM International Symposium on Computer Architecture (ISCA)*, pp. 247—-258, 1996. `https://doi.org/10.1145/232973.232999`.

[267] Anonymized, "NIC-CPU ring contention," Aug 2022. Private communication with a major NIC vendor.

[268] B. Pismenny, A. Morrison, and D. Tsafrir, "ShRing: Networking with shared receive rings," in *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pp. 949–968, 2023. `https://www.usenix.org/conference/osdi23/presentation/pismenny`.

[269] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis, "Shinjuku: Preemptive scheduling for μsecond-scale tail latency," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 345–360, 2019. `https://www.usenix.org/conference/nsdi19/presentation/kaffes`.

[270] B. Pismenny, H. Eran, A. Yehezkel, L. Liss, A. Morrison, and D. Tsafrir, "Autonomous NIC offloads," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 18—-35, 2021. `https://doi.org/10.1145/3445814.3446732`.

[271] B. Pismenny, L. Liss, A. Morrison, and D. Tsafrir, "The benefits of general purpose on-NIC memory," in *ACM International Conference on Architectural Support for*

Programming Languages and Operating Systems (ASPLOS), pp. 1130—-1147, 2022. https://doi.org/10.1145/3503222.3507711.

[272] S. Agarwal, R. Agarwal, B. Montazeri, M. Moshref, K. Elmeleegy, L. Rizzo, M. A. de Kruijf, G. Kumar, S. Ratnasamy, D. Culler, and A. Vahdat, "Understanding host interconnect congestion," in *ACM Workshop on Hot Topics in Networks (HotNets)*, 2022. https://doi.org/10.1145/3563766.3564110.

[273] Fritz Kruger, "CPU bandwidth - the worrisome 2020 trend." https://blog.westerndigital.com/cpu-bandwidth-the-worrisome-2020-trend/, 2020. Accessed: 2021-06-09.

[274] VMware, "Large packet loss in the guest os using vmxnet3 in esxi (2039495)." https://kb.vmware.com/s/article/2039495, 2021. Accessed: June 2021.

[275] M. Faraclas, "Received packets have been dropped by NIC." https://indeni.com/blog/cross-vendor-alert-of-the-week-some-received-packets-have-been-dropped-by-nic/, 2014. Accessed: June 2021.

[276] Intel, "Tuning the buffers: a practical guide to reduce or avoid packet loss in dpdk applications." https://indeni.com/blog/cross-vendor-alert-of-the-week-some-received-packets-have-been-dropped-by-nic/, 2017. Accessed: June 2021.

[277] X. Li, "[dpdk-dev] [patch v11 0/7] ethdev: introduce shared rx queue." https://lore.kernel.org/all/20211020075319.2397551-1-xuemingl@nvidia.com/, 2021. Accessed: 2023-04-13.

[278] O. I. T. Committee, "IDPF (Infrastructure Data Path Function)." https://www.oasis-open.org/committees/download.php/70738/IDPF%20Spec_v0_9.pdf, 2023. Accessed: 2023-05-13.

[279] O. I. T. Committee, "OCP Server NIC SW Specification: Core Features." https://docs.google.com/document/d/1FaVPGYipZ1sPhnYg7KItAS7ivL_svvZP8ZVJeFJezc0, 2023. Accessed: 2023-05-13.

[280] D. Barak, "ibv_post_srq_recv." https://www.rdmamojo.com/2013/02/08/ibv_post_srq_recv/. Accessed: 2022-09-26.

[281] A. Ancel, T. Tokun, and S. Mahameed, "Rx and Tx bulking/batching." https://legacy.netdevconf.info/2.1/slides/apr6/network-performance/04-amir-RX_and_TX_bulking_v2.pdf, 2017. Accessed: 2022-10-10.

[282] M. Scazzariello, T. Caiazzi, H. Ghasemirahni, T. Barbette, D. Kostić, and M. Chiesa, "A High-Speed stateful packet processing approach for tbps programmable switches," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 1237–1255, 2023. https://www.usenix.org/conference/nsdi23/presentation/scazzariello.

[283] M. Alian, S. Agarwal, J. Shin, N. Patel, Y. Yuan, D. Kim, R. Wang, and N. S. Kim, "IDIO: Network-driven, inbound network data orchestration on server processors," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 480–493, 2022. `https://doi.org/10.1109/MICRO56248.2022.00042`.

[284] J. Corbet, "Automatic buffer selection for io_uring." `https://lwn.net/Articles/815491/`, 2020. Accessed: 2023-04-13.

[285] G. M. Shipman, T. S. Woodall, R. L. Graham, A. B. Maccabe, and P. G. Bridges, "Infiniband scalability in open MPI," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2006. `https://doi.org/10.1109/IPDPS.2006.1639335`.

[286] S. Sur, L. Chai, H.-W. Jin, and D. Panda, "Shared receive queue based scalable MPI design for InfiniBand clusters," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2006. `https://doi.org/10.1109/IPDPS.2006.1639336`.

[287] M. Sutherland, S. Gupta, B. Falsafi, V. Marathe, D. Pnevmatikatos, and A. Daglis, "The nebula rpc-optimized architecture," in *ACM International Symposium on Computer Architecture (ISCA)*, pp. 199–212, 2020. `https://doi.org/10.1109/ISCA45697.2020.00027`.

[288] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson, "Hoard: A scalable memory allocator for multithreaded applications," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 117–128, 2000.

[289] I. Apalodimas, "Page pool API." `https://www.kernel.org/doc/html/latest/networking/page_pool.html`, 2019. Accessed: 2020-07-14.

[290] "DPDK memory pool library." `https://doc.dpdk.org/guides/prog_guide/mempool_lib.html`, 2014. Accessed: 2023-08-30.

[291] Mellanox, "Mellanox adapters programmer's reference manual (PRM)." `https://network.nvidia.com/files/doc-2020/ethernet-adapters-programming-manual.pdf`, 2016. Accessed: Nov. 2022.

[292] T. Hoefler, S. Di Girolamo, K. Taranov, R. E. Grant, and R. Brightwell, "Spin: High-performance streaming processing in the network," in *ACM/IEEE Supercomputing (SC)*, 2017. `https://doi.org/10.1145/3126908.3126970`.

[293] NVIDIA, "NVIDIA NICs performance report with DPDK 23.03." `https://fast.dpdk.org/doc/perf/DPDK_23_03_NVIDIA_NIC_performance_report.pdf`, 2023. Accessed: 2023-08-08.

[294] P. Druschel, L. L. Peterson, and B. S. Davie, "Experiences with a high-speed network adaptor: a software perspective," in *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*, pp. 2––13, 1994. `https://doi.org/10.1145/190809.190315`.

[295] I. Leslie and D. McAuley, “Fairisle: An atm network for the local area,” in *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*, 1991. `https://doi.org/10.1145/115992.116022`.

[296] Z. D. Dittia, G. M. Parulkar, and J. R. Cox Jr, “The APIC approach to high performance network interface design: Protected DMA and other techniques,” in *IEEE International Conference on Computer Communications (INFOCOM)*, 1997.

פורמת את הפונקציונליות הזאת ע״י החלפת הטבעת בשני טבעות שמתאיות לשני הפונקציונליות כך שהקצאת זכרון נעשת באופן בלתי תלוי מקבלת חבילות מהרשת. לפיכך, בטבעת מפוצלת קבוצת העבודה של הקלט/פלט יכולה להיות קטנה יותר, כי חוצצים ריקים לא חייבים להיות משויכים לליבה שהקצתה אותם, ולכן צריך פחות מהם בסה״כ. מימשנו את טבעת מפוצלת באמולציה בתוכנה. הניסויים שלנו מראים כי טבעת מפוצלת משפרת את התפוקה בעד כ-20% ומקטינה את זמן הגישה פי לכל היותר 11.

תקציר

קיצו של חוק מור והגידול המהיר במהירותם של רשתות תקשורת מביאים להתפתחותם של צווארי בקבוק בכל הרכיבים בשרתים, וספציפית במעבד ובזכרון. כרטיסי רשת ממוקמים אידאלית בשביל לשחרר עומס מהמעבד ומהזכרון מכיוון שכל תקשורת הרשת זורמת דרך כרטיס הרשת בכל מקרה. בנוסף, מגמות במחירי חומרה מראות שיותר זול לשדרג כרטיסי רשת או להוסיף להם פונקציונליות מאשר לשדרג את המעבד. מטרת התזה הזאת היא לשחרר את צווארי הבקבוק הנ"ל ע"י תכנון כרטיס הרשת ביחד עם התוכנה שרצה על השרת תוך כדי שימור השליטה על מימוש פרוטוקולי התקשורת בתוכנה.

<u>שימוש בזכרון על כרטיס הרשת.</u> אנו מזהים שקיימת מחלקה של אפליקציות שמעבדות חבילות רשת נכנסות על בסיס הכותרות בחבילות בלבד—ולא לפי התוכן של החבילות—והעברת כל החבילה מהרשת לזיכרון גורמת לצוואר בקבוק בזכרון, בחיבור בין כרטיס הרשת למעבד, ואף בכרטיס הרשת עצמו. במקרה זה, העברת התוכן של החבילות אינו הכרחי, ולכן אנו מציעים לחשוף את הזכרון הזמין על כרטיס הרשת (זכרון-כרטיס-רשת), לשימוש ישיר ע"י האפליקציה, למשל בשביל לשמור את התוכן של חבילות נכנסות. זכרון זה משמש את הכרטיס להאצות מתוחכמות שאינן בשימוש במקרה שלנו כמו Remote Direct Memory Access (RDMA). אנו מדגימים את היתרונות בזכרון-כרטיס-רשת בשני אפליקציות: פונקציות רשת (network functions) ומפות (key-value stores). בפונקציות רשת אנו מפצלים את התוכן מהכותרת של חבילות נכנסות, ומתי שאפשר אנו שומרים את התוכן בזכרון-כרטיס-רשת ואת הכותרת מעבירים אל הזכרון הראשי, ובזאת אנו משחררים עומס מהזכרון הראשי, מהחיבור בין הכרטיס למעבד, ומהכרטיס עצמו. גישה זאת משפרת את התפוקה (throughput) של פונקציות רשת בלכל היותר 19% ואת זמן הגישה (latency) בלכל היותר 23%. באופן דומה, בעבור מפות, מכיוון שאופן הגישה למפות הוא לעיתים קרובות מוטה כלפי מספר ערכים נפוצים במיוחד, אנו מאפשרים לאפליקציות לשמור את הערכים הנפוצים ביותר בזכרון-כרטיס-רשת. גישה זאת משפרת את התפוקה של מפות בלכל היותר 80% ואת זמן הגישה בלכל היותר 43%.

<u>העברת פונקציונליות לכרטיס באופן עצמאי.</u> אנו פונים לעסוק בבעיה של העברת פונקציונליות ברמה-5 של פרוטוקולי תקשורת (כמו למשל הצפנה) שבנויים מעל TCP/IP לכרטיס רשת ללא העברה של כל הפונקציונליות של רמות-4 ומטה לכרטיס. האתגר העיקרי שאיתו אנו מתמודדים הוא איך להתגבר על חבילות שמגיעות שלא לפי הסדר. אנחנו מממשים את הפתרון שלנו עבור שני פרוטוקולים ושני סוגי פונקצינליות: העתקות וחישוב CRC בפקוטוקול NVMe-TCP, והצפנה ופיענוח וחתימה בפרוטוקול HTTPS. הפתרון שלנו משפר את התפוקה פי לכל היותר 3.3, מוריד את צריכת המעבד בלכל היותר 60% ומשפר את זמן התגובה בלכל היותר 30%. המימוש שלנו כבר זמין בקרנל של לינוקס והוא נתמך בכרטיס הרשת החדשים של NVIDIA.

<u>בעית קבוצת העבודה לקלט/פלט.</u> במעבדים מרובוי ליבות בדרך כלל ממקבלים את קבלת החבילות מהרשת ע"י הקצאת טבעת אחת לקבלה בגודל של לפחות 1024 כניסות לכל ליבה. הגודל של כל טבעת צריך להיות גדול מספיק בשביל שליבה אחת תוכל לספוג פרץ של חבילות מהרשת. הבעיה היא שסך כל החוצצים ששייכים לכל טבעות הקבלה יכולים להיות גדולים יותר מגודלו של המטמון במעבד, ובמקרה הזה הגישות של המעבד וכרטיס הרשת לחבילות שמגיעות מהרשת יטופלו בעיקר בזכרון הראשי ולא במטמון מה שיוביל לביצועים נמוכים כאשר מדובר במאות גיגה ביט בשניה. בשביל לפתור את הבעיה הזאת אנו מציעים לשתף חוצצים בין הליבות, ואנחנו חוקרים שתי גישות לשיתוף חוצצים: (1) טבעת-משותפת ו-(2) טבעת-מפוצלת.

<u>טבעת משותפת.</u> בטבעת משותפת אנו משלבים בין טבעת אחת משותפת לקבלת חבילות בין כמה ליבות שלכל אחת מהן טבעת ייעודית שהכניסות בה מסמנות אינדקסים בטבעת המשותפת של חבילות שמיודעות לאותה ליבה. הטבעות הייעודיות מאפשרות לליבות השונות לקבל חבילות ללא סינכרון, אבל נדרש סנכרון כאשר מעדכנים את הטבעת המשותפת, כמו למשל כאשר מחליפים חוצצים שמכילים חבילות בחוצצים חדשים וכשמקדמים את ראש הטבעת. לפיכך, אנו מפעילים את מנגנון הטבעת המשותפת רק כאשר זה משתלם, כלומר רק כאשר יש שימוש רב בזכרון ע״י כרטיס הרשת וכאשר התעבורה מפוזרת היטב בין הליבות שמשתפות חוצצים. אנו מראים שבגישה זאת ניתן לשפר את התפוקה של פונקציות רשת בלכל היותר 27% ואת זמן הגישה פי לכל היותר 38.

<u>טבעת מפוצלת.</u> בטבעת מפוצלת אנו מאפשרים שיתוף חוצצים בין ליבות ללא הצורך בסנכרון יקר. אנו מבחינים כי הסנכרון בטבעת משותפת נובע משזירה של שני מנגנוני ייצור-צריכה בטבעות של כרטיסי רשת: (1) הקצאת זכרון, כשהמעבד ״מייצר״ חוצצים לכרטיס הרשת ״שצורך״ אותם בשביל לשמור חבילות ו-(2) קבלת חבילות, כשכרטיס הרשת ״מייצר״ חבילות בשביל המעבד ״שצורך״ אותם. הגישה של טבעת מפוצלת

המחקר נעשה בהנחייתם של פרופ׳ אדם מוריסון מאוניברסיטת תל-אביב ופרופ׳ דן צפריר מהטכניון בפקולטה למדעי המחשב, אני אסור תודה על תמיכתם, הדרכתם, סבלנותם, וזמנם.
אני מודה לטכנון, ולחברת נבידיה על התמיכה הכספית הנדיבה בהשתלמותי.

שחרור צווארי בקבוק של המעבד והזכרון ברשתות תקשורת של מאה ג'יגה ביט

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר
דוקטור לפילוסופיה

בוריס פיסמניי

הוגש לסנט הטכניון – מכון טכנולוגי לישראל

כסלו התשפ״ד   חיפה   דצמבר 2023

שחרור צוואַרי בקבוק של המעבד והזכרון ברשתות תקשורת של מאה ג'יגה ביט

בוריס פיסמניי