ShRing: Networking with Shared Receive Rings

Boris Pismenny[†◇] Adam Morrison[‡] Dan Tsafrir[†±]

[†] *Technion – Israel Institute of Technology* [◇]*NVIDIA* [‡] *Tel Aviv University* [±] *VMware Research*

Abstract

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.

1 Introduction

Software systems drive Ethernet NICs through producer-consumer "rings." A ring is a logically circular memory array shared between software and NIC, such that each ring entry points to a buffer big enough to store an Ethernet packet. Software sends data by placing packet buffers in a transmission (Tx) ring, thereby handing them to the NIC to be sent. Software receives data by removing packet buffers from a receive (Rx) ring after they have been filled by the NIC, immediately replacing them with free buffers to be used in their stead for future incoming traffic. Thus, Rx rings are always fully populated with (free or filled) buffers, whereas Tx rings are commonly partially populated or empty. Consequently, Rx rings are more memory-consuming than Tx rings.

By default, a receive ring consists of ≥1Ki entries [11, 21, 60, 65, 86, 86, 92], each pointing to a 1500B buffer, Ethernet's maximum transmission unit (MTU) [36]. A typical Rx ring thus requires (1Ki $\times$ 1500B $\approx$) 1.5MiB. NICs support hundreds of such rings [12, 50, 66, 71], which software uses for synchronization-free parallelism, assigning different rings to different cores in both kernel [30, 33, 68, 77, 82, 84, 90] and user [2, 8, 24, 38, 52] network stacks. The combined size of Rx buffers across all cores—henceforth denoted as α—can therefore reach tens of MiBs, which might be bigger than the last-level cache (LLC). Notably, α constitutes a lower bound for the size of the NIC working set [25], 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. As a result, α exceeding LLC capacity can be problematic for high-throughput, low-latency workloads that sustain network traffic of up to hundreds of gigabits per second (Gbps).

The problem stems from these workloads relying on data direct I/O (DDIO) [20] technology or similar. DDIO allows NIC direct memory accesses (DMAs) to read and write packets to and from the LLC while avoiding high main memory access costs [15, 29, 63, 64, 78, 79, 87, 91]. But an α larger than the LLC undermines DDIO's effectiveness, as the NIC working set is too big to be cached. Consequently, CPU memory accesses become slower, contending with DMAs for insufficient cache capacity. Accesses are thus increasingly served by main memory, making the per-packet processing time longer. This overhead translates to degraded throughput and latency of networking workloads that experience the memory as a bottleneck resource.

We exemplify this problem in §2, using run-to-completion systems [6, 26, 35, 52, 58, 69, 73] common in microsecond-scale workloads like network function virtualization (NFV). In these systems, each thread of execution consists of a loop that iteratively polls an Rx ring, receives a packet from the wire, processes it to completion (without context switches or interrupts interfering), and then sends a response.

In §3, we consider addressing the problem by reducing the size of Rx rings [91]. We find that a size smaller than 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 α 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, in §4, we propose "shRing," a system that alleviates the above problem by sharing a 1Ki-sized Rx ring between a set of *N* cores. ShRing satisfies 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 α small.

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 [4, 10, 26, 59, 63, 73, 75, 76, 97], 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 [9, 22, 26, 55, 80, 90]. 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 [37]. 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 that RxArr shRing works as expected, improving NFV macrobenchmark throughput by up to 1.27x and latency by up to 38x. In §6, 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 §7 and conclude in §8.

2 Motivation

We begin by providing the necessary background (§2.1) and by characterizing the problem that shRing tackles, which is the increasing working set size of the NIC as compared to the LLC size (§2.2). We then experimentally demonstrate how this problem affects performance as well as shRing's ability to address its root cause (§2.3).

2.1 Background

Interacting with NICs Software and Ethernet NICs interact via logically cyclic producer-consumer queues called *rings*. The roles of producer and consumer depend on perspective: for received (Rx) traffic, the NIC can be viewed as producing incoming packets that software consumes; alternatively, software can be viewed as producing free buffers that the NIC consumes by filling them with incoming data. Transmitted (Tx) traffic can be viewed similarly. Software chooses the ring size and allocates it in main memory. The entries of a ring are architected *descriptor* structures consisting of several fields, one of which is a pointer to packet buffer.

Software pre-allocates packet buffers for all Rx descriptors. Each buffer can hold MTU bytes (≈1500 by default). When a packet arrives, the NIC DMA-writes it to the buffer pointed to by the *tail* descriptor of the Rx ring ("next free" index), incrementing the tail to point to the subsequent descriptor if the tail does not surpass the *head* ring descriptor. Symmetrically, software dequeues Rx packets for processing from the head descriptor ("next full" index), iteratively incrementing it so long as it does not surpass the tail; software replaces the current head's buffer, which the NIC has just filled, "reposting" a new free buffer instead and informing the NIC about this by "ringing the doorbell" (writing to a NIC register).

Tx traffic occurs similarly, with NIC and software flipping roles (NIC responds to software actions rather than the other way around). Thus, in contrast to the Rx case, Tx ring descriptors are initially empty and therefore consume less space.

A ring's head and tail are maintained as consumer- and producer-controlled registers, residing in NIC memory mapped I/O (MMIO) and holding ring indexes. Software may configure the NIC to trigger an interrupt when it updates a register, or it may instead poll the ring and observe changes.

The NIC distributes incoming traffic load between multiple Rx rings, and therefore between multiple cores, using receive side scaling (RSS [68], which computes a hash over the packet's header to produce a ring identifier) or accelerated receive flow steering (ARFS [90], which consults software-controlled packet steering tables).

NFs In this work, we mostly focus on improving the performance of network function (NF) workloads. NFs are packet-processing applications that were once implemented using rigid proprietary hardware middleboxes and are now increasingly implemented with software on off-the-shelf servers [4, 23, 26, 27, 53, 54, 58, 74]. Common NF examples include switches, routers, firewalls, virtual private networks (VPN), deep packet inspectors (DPI), network address translators (NAT), and load balancers (LB). Evidence suggests that nearly 60% of all data center network traffic relies on NFs [74].

To attain high throughput and low latency, NFs commonly employ a packet processing model based on kernel bypass and direct NIC access [4, 23, 27, 53, 58] as provided by, e.g., the data plane development kit (DPDK) [51]. To improve efficiency and minimize overheads, this model typically foregoes abstractions like blocking I/O, context switching, and multitasking. Instead, it is designed as a simple run-to-completion, polling system, which does away with costly device interrupts as means of driving networking activity. Thus, each NF thread T gets its own dedicated core and rings. T continuously polls its Rx ring, and when a packet arrives, T processes the packet, generates a response, sends the response by placing it in its Tx ring, and resumes its Rx polling.

DDIO High-throughput, low-latency apps like NFs benefit from Intel's direct data I/O (DDIO) technology [20] (other processor vendors support similar technologies [5, 93]). When possible, DDIO satisfies DMA operations from the LLC rather than main memory, which is faster/cheaper and may thus 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.

2.2 The Problem: I/O Working Sets

Let the *I/O working set* be the memory area that an I/O device (e.g., NIC) reads/writes via DMA in a given time interval. For NFs, this set should preferably fit in the LLC due to DDIO. An I/O-intensive workload whose I/O working set size exceeds (or even approaches) LLC capacity implies: that I/O-related data likely competes for cache capacity; that DMAs are thus increasingly served by main memory instead of the LLC; and that LLC contention and memory bandwidth bottlenecks might occur as a result [15, 29, 63, 64, 78, 79, 87, 91].

Rx ring size is a key factor in determining the I/O working set size. Recall that all Rx descriptors are pre-populated with MTU (1500B) packet buffers upon startup. Subsequently, whenever software replenishes the ring's head descriptor with a free buffer B, the head-tail protocol (§2.1) dictates that the NIC will DMA-write a new packet to B only after the associated Rx ring tail wraps around back to B's position. Thus, the aggregate Rx size (denoted α) serves as a lower bound for the I/O working set. If software utilizes r Rx rings of size s, then this lower bound is $\alpha = r \times s \times 1500B$.

year	*Intel NIC*	*gen. (GbE)*	*max ring num. (r_m)*	*default size (s)*	*Xeon CPU*	*LLC*	*cores*
2001	[40]	1	1	256	[41]	256 KiB	1
2007	[42]	10	64	512	[43]	12 MiB	4
2014	[46]	40	1536	512	[45]	38 MiB	15
2020	[49]	100	2048	2048	[48]	77 MiB	56

Table 1: 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.

The problem that motivates our work is that α grows faster than the LLC and nowadays routinely exceeds it, with Rx rings increasing both in number (r) and size (s). Underlying this phenomenon are, notably, the following technology trends. NIC throughput has been growing faster than CPU packet processing speed for over a decade [32, 87]. The higher bandwidth increases variability and necessitates bigger network queues [28, 47, 94]. Moreover, the ever-growing traffic volume implies that the days when a single CPU core was able to drive an Ethernet NIC to its full capacity are long gone [31]. Thus, modern systems must employ multicore parallelism [9, 22, 26, 80, 90]. NICs have therefore evolved to offer multiple Rx/Tx rings, allowing each core to interact with the NIC through its own private ring instances in isolation. We refer to this architecture as *privRing*.

To demonstrate the rapidly increasing α phenomenon (along with the underlying technology trends), we collected the ring maximal number (r_m) and default size (s) from the datasheet and driver, respectively, of every Intel NIC model released during 2000–2022. Table 1 shows a representative summary; to conserve space, we only include the first NIC of each Ethernet generation with increasing throughput. 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.[1] 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 [60].

The right side of Table 1 matches each NIC with an Intel CPU model launched at that year, whose LLC was the largest

[1]It makes sense for r to be much bigger than CPU core number in order to support, e.g.: per-application rings [38, 95]; per-container rings [2, 24]; a ring for every SRIOV [44] instance of every virtual CPU of every virtual machine that runs on the host machine [82, 84]; and a hypervisor ring per VM ring for fallback when flow rule offloading is not yet configured [33, 77].

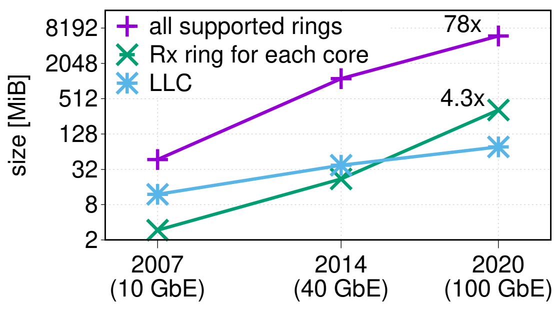

Figure 1: Aggregate Rx size ($\alpha = r \times s \times MTU$) grows faster than LLC size and has already exceeded it in even the most minimalist configuration (based on data from Table 1).

at the time. Using this data, Figure 1 plots the size of the LLC and the minimal and maximal α of the associated NIC. We see that the maximal aggregate Rx size (assuming all r_m 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 [13, 14].

2.3 Implications

Assume that the I/O working set size of some NF exceeds the LLC capacity and/or the LLC space it needs for satisfying DMA-writes of incoming data (constrained by DDIO to only two ways per LLC set by default) is insufficient. In this case, we claim that the overhead is significant to the point that it may be preferable to abandon dedicated private rings (privRings) in favor of shared rings (shRings), despite the synchronization cost associated with the latter.

To demonstrate, we use a synthetic FastClick NF microbenchmark configured to iteratively receive a packet, access an array, perform routing, and send the packet out [8]. The NF uses all (16) cores of our 2.1 GHz CPU, experiencing a theoretical incoming load of 200 Gbps of MTU packets (line rate), which in practice is 195.6 Gbps (due to 34B Ethernet overhead for each 1500B MTU packet). We execute this experiment using the baseline privRing, as well as three shRing variants that unify the rings of 2, 4, and 8 cores, respectively denoted as shRing/2, shRing/4, and shRing/8. (The full details of shRing are specified in §4, and the full details of the experiment are specified in §5.)

Figure 2a 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 2b, is approximately inversely proportional to cycles-per-packet (Figure 2a) as long as the CPU constitutes a bottleneck resource 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 2a) 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 2c (memory bandwidth) and 2d (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 2e). 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 2f). 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 2g).

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.

3 Fewer or Smaller Private Rings

Conceivably, we can reduce the I/O working set size without ring sharing in two straightforward ways. One can use much smaller per-core Rx rings, or one can employ a single core (using a bigger Rx ring) as the system's centralized "dispatcher" for all incoming traffic. Here, we briefly explain why neither is satisfactory for high-bandwidth networking applications.

The single-core, single-ring centralized dispatcher approach is used by such systems as Shinjuku [57] and Shenango [72]. It can be an effective way to reduce I/O memory consumption, and it has been shown to work well for NIC bandwidth of up to 40 Gbps. But more powerful NICs might not be served well by this approach, as the dispatcher's

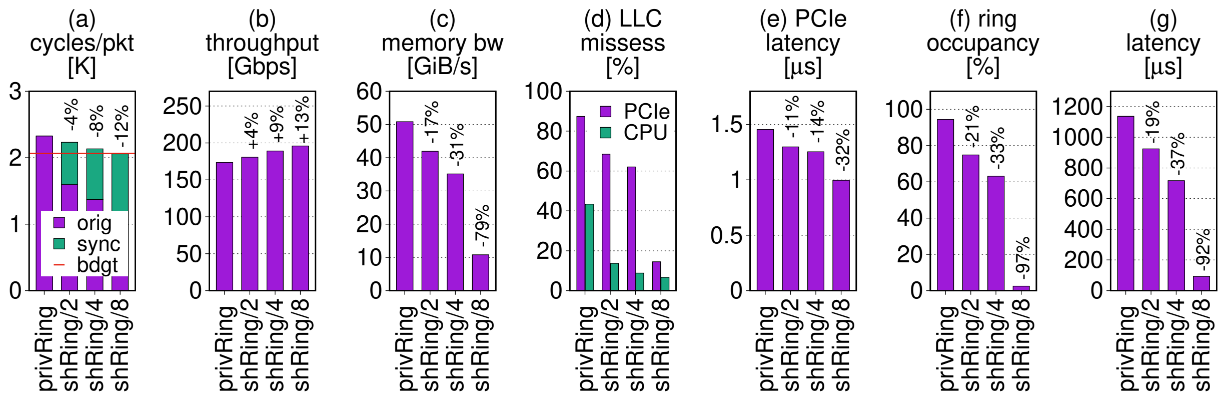

Figure 2: ShRing's synchronization costs are significant but are nevertheless worthwhile, as they are cheaper than the overheads associated with privRing's larger I/O working set. When shRing's cycles-per-packet meet the line rate budget (a), its packet processing rate exceeds the packet arrival rate, generating low occupancy in the ring (f) and thus substantially reducing the latency (g).

limited compute capacity becomes a bottleneck [31].

The other potential approach, of reducing the size of all rings while retaining the ring-per-core design, is compatible with multicore parallelism. But we contend that the existing ring size is necessary and that reducing it has negative repercussions. To illustrate, we run the standard RFC2544 no-drop rate (NDR) test [10] with DPDK Layer-3 MTU packet forwarding (l3fwd) on 8 cores. This test finds the maximum throughput attainable without loss. We run it once with traffic evenly spread across the cores ("multicore") and again with traffic directed at one of them ("single core").

Figure 3a 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 3b 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.

4 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 (§4.1).

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

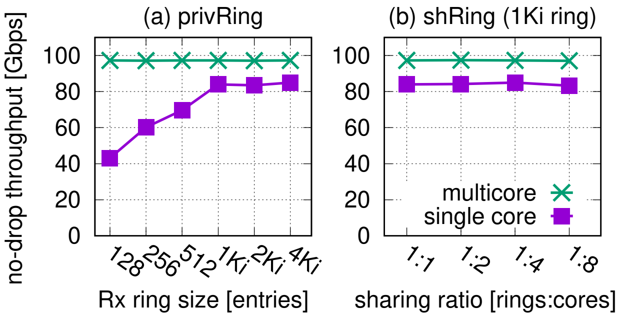

Figure 3: DPDK l3fwd no-drop rate. Small privRings work well when traffic is evenly spread across cores but cause drops otherwise. ShRings work well in both cases at a fraction of the buffer size.

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 [37], ridding them from having to compete for newly arriving packets (§4.2). 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, §4.3) or a linked list (RxList, §4.4). 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 (§4.5). We describe the implementation details in §4.6.

4.1 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 (§4.5), 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 [4,26,63,73,75,76,97] and IETF benchmarking methodology [10] 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 $\mu < \lambda$ 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 2g.

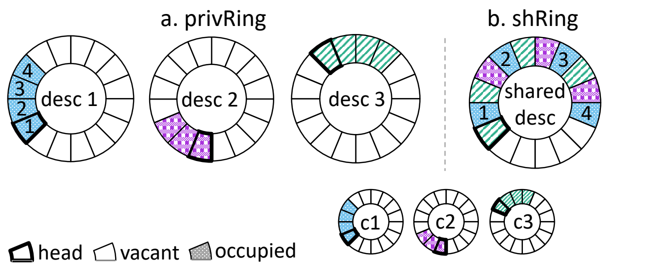

Figure 4: PrivRing (private Rx rings) vs. shRing (shared Rx ring) with $N = 3$ completion rings.

4.2 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 [9,22,26,80,90]. 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 4.

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 [37, 61]. The motivation for this feature is reducing DRAM pinning for RDMA (see §7), 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 [17] and the Open Compute Project NIC specification [18], which are proposed industry standards for network device interfaces.

4.3 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 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 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.

```
 1 #define BIT(x)   (1 << ((x) & 63))
 2 #define WORD(x)  ((x) >> 6)
 3 #define ISSET(bmp, x) \
 4         (bmp[WORD(x & (bmp->size - 1))] & BIT(x))
 5 int shRing(sd_ring *sdr, c_ring *cr,
 6            void **pkts, int len) {
 7   int rcvd = 0, lidx = -1;
 8   uint_64t lbits = 0
 9   while (rcvd < len) {
10     c_ring_ent *cre = get_cre(cr);
11     if (cre == NULL)
12       break;
13     int idx = cre->idx;
14     pkts[rcvd++] = sdr->desc[idx].buf;
15     sdr->desc[idx].buf = alloc_buf();
16     if (lidx == -1) lidx = WORD(idx);
17     else if (lidx == WORD(idx)) {
18       atomic_or(&sdr->bitmap[lidx], lbits);
19       lidx = WORD(idx);
20       lbits = 0;
21     }
22     lbits |= BIT(idx);
23   }
24   if (rcvd == 0) return 0;
25   if (lbits != 0)
26     atomic_or(&sdr->bitmap[lidx], lbits);
27   cr->ci += rcvd;
28   *cr->doorbell = cq->ci;
29   lock(sdr->lock);
30   while (ISSET(sdr->bitmap, sdr->ci) != 0) {
31     setb = ffs(~sdr->bitmap[WORD(sdr->ci)]);
32     atomic_clear(&sdr->bitmap[WORD(sdr->ci)],
33                  setb - 1);
34     sdr->ci += setb - 1;
35   }
36   *sdr->doorbell = sdr->ci;
37   unlock(sdr->lock);
38   return rcvd;
39 }
```

Listing 1: RxArr shared ring receive code.

4.4 Linked List Ring Sharing (RxList)

RxList is a shRing design that alleviates RxArr's bitmap coordination problem, eliminating the requirement to repost

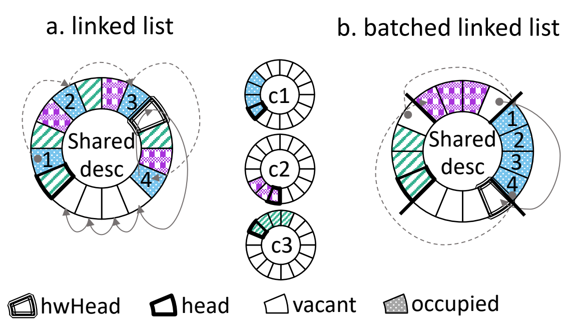

Figure 5: RxList vs. Batched RxList designs for one shared ring with three completion rings. In (b), the batch size is 4.

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 [7].) 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 paper to describe the first and last descriptors holding packets.

Figure 5a 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 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.

```
 1 int ll_recv(sd_ring *sdr, c_ring *cr,
 2             void **pkts, int len) {
 3   int idx, rcvd = 0, myhead, *iptr = NULL;
 4   while (rcvd < len) {
 5     c_ring_ent *cre = get_cre(cr);
 6     if (cre == NULL)
 7       break;
 8     idx = cre->idx;
 9     pkts[rcvd++] = sdr->desc[idx].buf;
10     sdr->desc[idx].buf = alloc_buf();
11     if (iptr == NULL) myhead = idx;
12     else iptr->next = idx;
13     iptr = &sdr->desc[idx];
14   }
15   if (rcvd == 0) return 0;
16   cr->ci += rcvd;
17   *cr->doorbell = cq->ci;
18   lock(sdr->lock);
19   int prevtail = sdr->hwTail;
20   sdr->desc[prevtail].next = myhead;
21   sdr->hwTail = idx;
22   sdr->ci += rcvd;
23   *sdr->doorbell = sdr->ci;
24   unlock(sdr->lock);
25   return rcvd;
26 }
```

Listing 2: RxList (linked list) shared ring receive code.

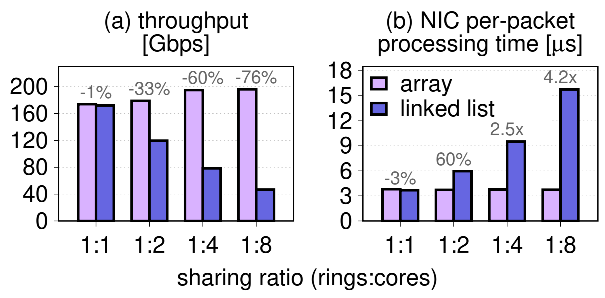

Figure 6: Although conceptually more suitable for sharing, RxList interferes with descriptors' contiguity, hampering their prefetching and thus degrading performance. (Labels show List to Arr ratio.)

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 [70], and even more crucial for shRing. In privRing, a descriptor cache miss on some ring 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 §2.3) on all cores and try to process traffic at line rate.

Figure 6a 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 6b 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 5b, 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 [3], 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.

4.5 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 (§4.1). 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.

4.6 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 §7), 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 [39] 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 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 (§4.1) under 200 GbE load.

5.1 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 [16], which we modify to improve latency measurement accuracy from 10–100μs to 1μs [81]. 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%.

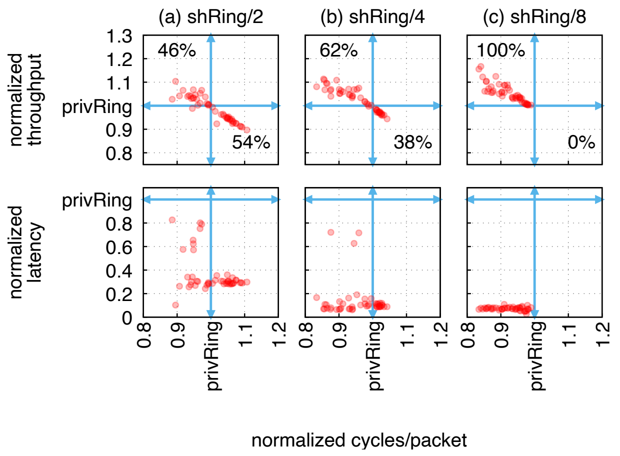

Figure 7: Normalized performance of shRing to privRing for NFs with varying memory intensity: shRing/8 improves performance in all cases. (Labels show percentage of NFs in quadrant.)

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 completion ring producer and consumer indexes, PCIe latency using NVIDIA Mellanox Neo-host [67], and memory bandwidth and PCIe hit rate using Intel PCM [39].

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 §3. We show it for a thorough comparison between privRing and shRing.

5.2 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) NAT and LB performance; and (3) 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 [8] which receives a packet, performs routing, followed by a number of random memory reads from

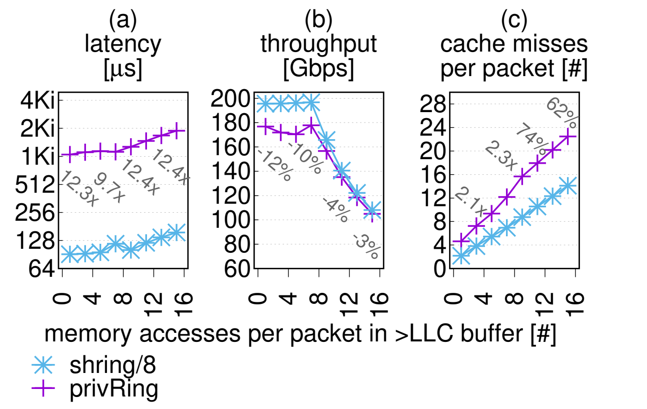

Figure 8: High cache pressure decreases performance for shRing and privRing. (Labels show privRing to shRing ratio.)

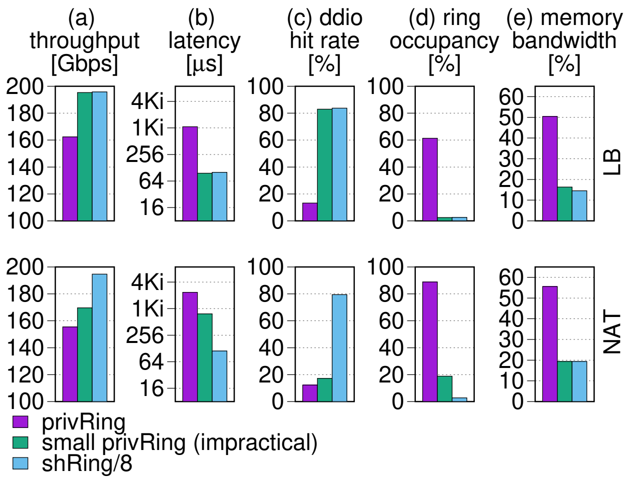

Figure 9: LB and NAT performance at 200Gbps load.

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 7 shows the results. We find that throughput and 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 8 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.

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 9 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 §2.3).

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 through-

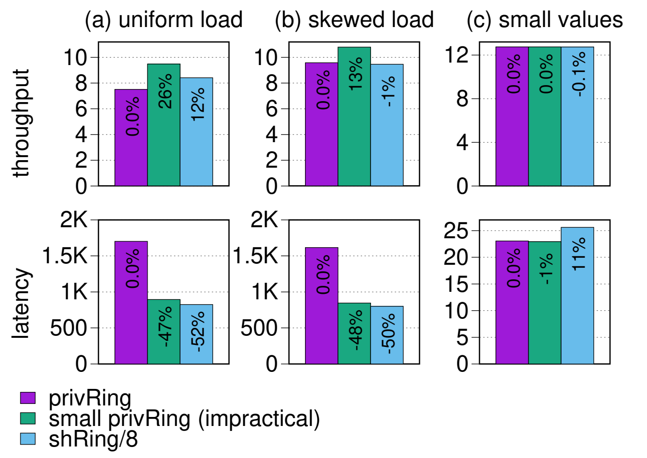

Figure 10: ShRing benefits the MICA key-value store with large I/O working sets, non-pathological load imbalance, and high load.

put 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 [62] 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 1 KiB values.

Figure 10a 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 (§4.1)—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 10b 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 10c 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 10a'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 10c) or because the penalty of I/O-related cache misses is negligible (low load).

5.3 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 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 §2.3.

Figure 11a 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 11b 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 12a 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 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 12b shows that as with processing variability, shRing's throughput decreases because the unloaded cores' Rx descriptor reposting is delayed by the overloaded core.

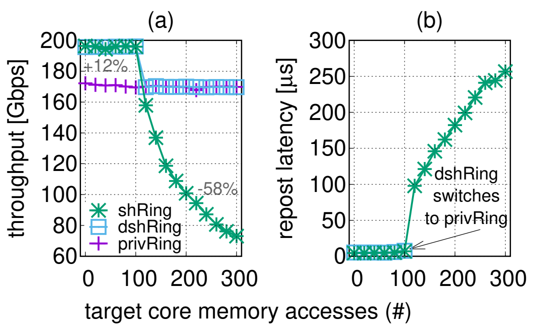

Figure 11: When incoming packet rate is fixed, processing variability in one core (e.g., due to increased number of memory accesses) might degrade shRing's throughput and delay descriptor reposting in peer cores. Dynamic shRing falls back on privRing when this happens.

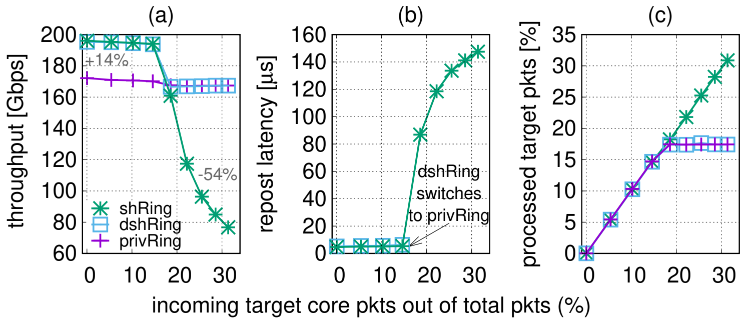

Figure 12: When variability manifests as increased rate of packets targeting one specific core (x axis), at some point, it prolongs the latency of peer core descriptor reposting (b); at this point, performance degrades (a) as the target core processing can no longer match the volume of incoming traffic (c).

Figure 12c 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 13 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.

6 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.

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 [56] 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 14a 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 14b), 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 15 (similarly to Figure 3) 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.

7 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 [91]; (2) placing packets in LLC slices closest to the target processing CPU core [29]; (3) eliminating interference between applications and I/O devices when partitioning the LLC [96]; (4) placing only packet

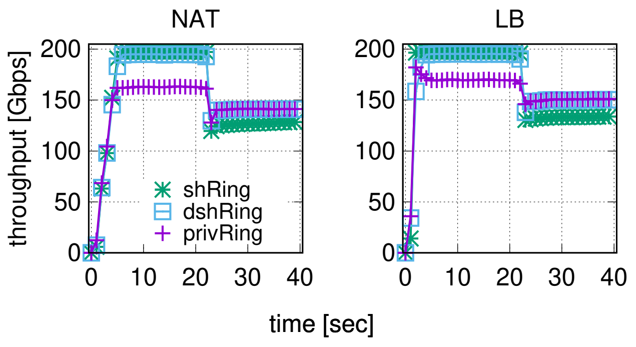

Figure 13: NAT and LB throughput when switching from a non-pathological to a pathological workload.

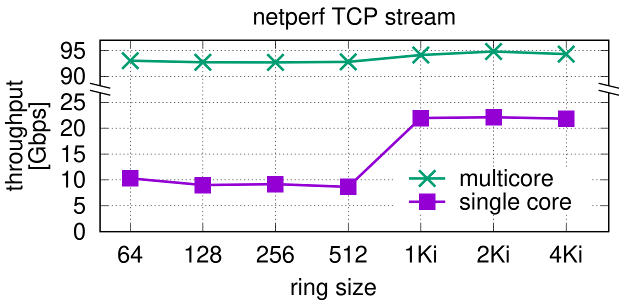

Figure 15: Netperf TCP stream throughput. Small rings work well when traffic is spread across multiple cores but cause drops otherwise.

headers in the LLC to reduce LLC contention [34, 79, 83]; 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 [1]. 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" [19] 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 [33, 77]. As the number of VMs exceeds the number of cores, multiple such rings must share a core. To optimize this, NVIDIA NICs

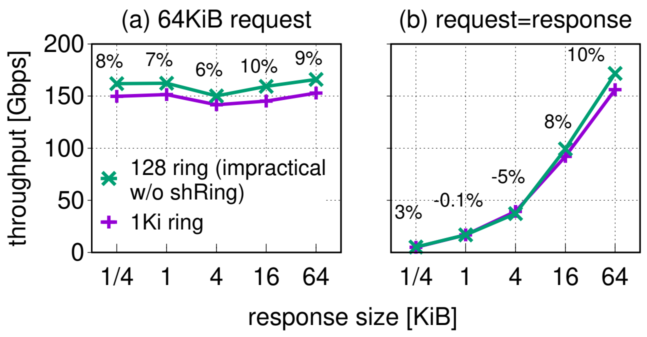

Figure 14: Netperf TCP_RR throughput with (a) 64KiB requests for various response sizes and (b) equal request and response sizes. Small rings work better as they reduce the I/O working set.

recently started sharing ring buffers and CRs within each core [61] 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 [85, 88]. 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 [89] 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.

8 Conclusions

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.

Acknowledgments

We thank the paper's shepherd, Adam Belay, and the anonymous reviewers for their valuable feedback.

References

[1] Mohammad Alian, Siddharth Agarwal, Jongmin Shin, Neel Patel, Yifan Yuan, Daehoon Kim, Ren Wang, and Nam Sung Kim. IDIO: Network-driven, inbound network data orchestration on server processors. In *IEEE/ACM International Symposium on*

Microarchitecture (MICRO), pages 480–493, 2022. https://doi.org/10.1109/MICRO56248.2022.00042.

[2] Nambiar Amritha, Samudrala Sridhar, and Patil 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.

[3] Amir Ancel, Tariq Tokun, and Saeed 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.

[4] Fabien André, Stéphane Gouache, Nicolas Le Scouarnec, and Antoine Monsifrot. Don't share, don't lock: Large-scale software connection tracking with krononat. In *USENIX Annual Technical Conference (ATC)*, pages 453–466, 2018. https://www.usenix.org/conference/atc18/presentation/andre.

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

[6] Dave Barach. VPP/software architecture. https://wiki.fd.io/view/VPP/Software_Architecture, 2018. Accessed: 2022-11-28.

[7] Dotan Barak. ibv_post_srq_recv. https://www.rdmamojo.com/2013/02/08/ibv_post_srq_recv/. Accessed: 2022-09-26.

[8] Tom Barbette, Cyril Soldani, and Laurent Mathy. Fast userspace packet processing. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 5—-16, 2015. https://doi.org/10.1109/ANCS.2015.7110116.

[9] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An analysis of linux scalability to many cores. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, Vancouver, BC, October 2010. USENIX Association. https://www.usenix.org/conference/osdi10/analysis-linux-scalability-many-cores.

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

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

[12] 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.

[13] Brad Burres, Dan Daly, Mark Debbage, Eliel Louzoun, Christine Severns-Williams, Naru Sundar, Nadav Turbovich, Barry Wolford, and Yadong Li. Intel's hyperscale-ready infrastructure processing unit (IPU). In *Hot Chips*, 2021. https://doi.org/10.1109/HCS52781.2021.9567455.

[14] Idan Burstein. NVIDIA data center processing unit (DPU) architecture. In *Hot Chips*, 2021. https://doi.org/10.1109/HCS52781.2021.9567066.

[15] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. Understanding host network stack overheads. In *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*, pages 65—-77, 2021. https://doi.org/10.1145/3452296.3472888.

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

[17] OASIS IDPF Technical 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.

[18] OASIS IDPF Technical Committee. OCP Server NIC SW Specification: Core Features. https://docs.google.com/document/d/1FaVPGYipZ1sPhnYg7KItAS7ivL_svvZP8ZVJeFJezc0, 2023. Accessed: 2023-05-13.

[19] Jonathan Corbet. Automatic buffer selection for io_uring. https://lwn.net/Articles/815491/, 2020. Accessed: 2023-04-13.

[20] Intel 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.

[21] Nithin 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.

[22] Alexandros Daglis, Mark Sutherland, and Babak 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)*, pages 35–48, 2019. https://doi.org/10.1145/3297858.3304070.

[23] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, Jesse Alpert, Jing Ai, Jon Olson, Kevin DeCabooter, Marc de Kruijf, Nan Hua, Nathan Lewis, Nikhil Kasinadhuni, Riccardo Crepaldi, Srinivas Krishnan, Subbaiah Venkata, Yossi Richter, Uday Naik, and Amin Vahdat. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 373–387, 2018. https://www.usenix.org/conference/nsdi18/presentation/dalton.

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

[25] Peter J. Denning. The working set model for program behavior. *Communications of the ACM (CACM)*, 11(5):323—333, May 1968. https://doi.org/10.1145/363095.363141.

[26] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. Routebricks: Exploiting parallelism to scale software routers. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 15—–28, 2009. https://doi.org/10.1145/1629575.1629578.

[27] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 523–535, 2016.

[28] Matt 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.

[29] Alireza Farshin, Amir Roozbeh, Gerald Q Maguire Jr, and Dejan Kostić. Make the most out of last level cache in intel processors. In *ACM Eurosys*, pages 1–17, 2019. https://doi.org/10.1145/3302424.3303977.

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

[31] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 281–297, 2020. https://www.usenix.org/conference/osdi20/presentation/fried.

[32] Fritz Kruger. CPU bandwidth - the worrisome 2020 trend. https://blog.westerndigital.com/cpu-bandwidth-the-worrisome-2020-trend/, 2020. Accessed: 2021-06-09.

[33] Or Gerlitz, Hadar Hen-Zion, Amir Vadai, and Rony 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.

[34] Swati Goswami, Nodir Kodirov, Craig Mustard, Ivan Beschastnikh, and Margo Seltzer. Parking packet payload with p4. In *ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, pages 274—-281, 2020. https://doi.org/10.1145/3386367.3431295.

[35] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. SoftNIC: A software NIC to augment hardware. Technical Report UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-155.html.

[36] 802.3-105 – IEEE standard for Ethernet. https://doi.org/10.1109/IEEESTD.2016.7428776, 2016.

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

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

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

[40] 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.

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

[42] 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.

[43] 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.

[44] 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.

[45] 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.

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

[47] 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.

[48] 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.

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

[50] 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.

[51] Intel Corporation. DPDK: Data plane development kit. http://dpdk.org, 2010. (Accessed: May 2016).

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

[53] Rishabh Iyer, Katerina Argyraki, and George Candea. Performance interfaces for network functions. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 567–584, 2022. https://www.usenix.org/conference/nsdi22/presentation/iyer.

[54] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a globally-deployed software defined wan. In *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*, pages 3—14, 2013. https://doi.org/10.1145/2486001.2486019.

[55] Zou Jia, Zhiyong Liang, and Yiqi Dai. Scalability evaluation and optimization of multi-core SIP proxy server. In *International Conference on Parallel Processing (ICPP)*, pages 43–50, 2008. 10.1109/ICPP.2008.30.

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

[57] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 345–360, 2019. https://www.usenix.org/conference/nsdi19/presentation/kaffes.

[58] Georgios P. Katsikas, Tom Barbette, Dejan Kostić, Rebecca Steinert, and Gerald Q. Maguire Jr. Metron: NFV service chains at the true speed of the underlying hardware. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 171–186, 2018. https://www.usenix.org/conference/nsdi18/presentation/katsikas.

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

[60] Kevin 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.

[61] Xueming 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.

[62] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 429–444, 2014. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/lim.

[63] Antonis Manousis, Rahul Anand Sharma, Vyas Sekar, and Justine Sherry. Contention-aware performance prediction for virtualized network functions. In *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*, pages 270—-282, 2020. https://doi.org/10.1145/3387514.3405868.

[64] Ilias Marinos, Robert N.M. Watson, Mark Handley, and Randall R. Stewart. Disk|Crypt|Net: Rethinking the stack for high-performance video streaming. In *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*, pages 211–224, 2017. https://doi.org/10.1145/3098822.3098844.

[65] 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.

[66] 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.

[67] 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.

[68] 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.

[69] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek. The click modular router. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 217—-231, 1999. https://doi.org/10.1145/319151.319166.

[70] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. Understanding pcie performance for end host networking. In *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*, pages 327—-341, 2018. https://doi.org/10.1145/3230543.3230560.

[71] 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.

[72] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 361–378, 2019. https://www.usenix.org/conference/nsdi19/presentation/ousterhout.

[73] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. NetBricks: Taking the V out of NFV. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 203–216, 2016. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/panda.

[74] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A. Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, Changhoon Kim, and Naveen Karri. Ananta: Cloud scale load balancing. In *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*, pages 207—-218, 2013. https://doi.org/10.1145/2486001.2486026.

[75] Solal Pirelli and George 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.

[76] Solal Pirelli, Akvilė Valentukonytė, Katerina Argyraki, and George Candea. Automated verification of network function binaries. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 585–600, 2022. https://www.usenix.org/conference/nsdi22/presentation/pirelli.

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

[78] 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.

[79] 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.

[80] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 325—-341, 2017. https://doi.org/10.1145/3132747.3132780.

[81] Mia Primorac, Edouard Bugnion, and Katerina Argyraki. How to measure the killer microsecond. In *Proceedings of the Workshop on Kernel-Bypass Networks*, pages 37—42, 2017. https://doi.org/10.1145/3098583.3098590.

[82] Scott Rixner. Network virtualization: Breaking the performance barrier: Shared I/O in virtualization platforms has come a long way, but performance concerns remain. *ACM Queue*, 6(1):36—44, January 2008. https://doi.org/10.1145/1348583.1348592.

[83] Mariano Scazzariello, Tommaso Caiazzi, Hamid Ghasemirahni, Tom Barbette, Dejan Kostić, and Marco Chiesa. A High-Speed stateful packet processing approach for tbps programmable switches. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 1237–1255, 2023. https://www.usenix.org/conference/nsdi23/presentation/scazzariello.

[84] Jeff Shafer, David Carr, Aravind Menon, Scott Rixner, Alan Cox, Willy Zwaenepoel, and Paul Willman. Concurrent direct network access for virtual machine monitors. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 306–317, 01 2007. https://doi.org/10.1109/HPCA.2007.346208.

[85] Galen M Shipman, Timothy S Woodall, Richard L Graham, Arthur B Maccabe, and Patrick 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.

[86] Shahaf 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.

[87] Igor Smolyar, Alex Markuze, Boris Pismenny, Haggai Eran, Gerd Zellweger, Austin Bolen, Liran Liss, Adam Morrison, and Dan Tsafrir. Ioctopus: Outsmarting nonuniform dma. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 101–115, 2020. https://doi.org/10.1145/3373376.3378509.

[88] S. Sur, Lei Chai, Hyun-Wook Jin, and D.K. 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.

[89] Mark Sutherland, Siddharth Gupta, Babak Falsafi, Virendra Marathe, Dionisios Pnevmatikatos, and Alexandres Daglis. The nebula rpc-optimized architecture. In *ACM International Symposium on Computer Architecture (ISCA)*, pages 199–212, 2020. https://doi.org/10.1109/ISCA45697.2020.00027.

[90] Herbert Tom and de Bruijn Willem. Scaling in the linux networking stack. https://www.kernel.org/doc/Documentation/networking/scaling.txt, 2011. Accessed: 2020-03-05.

[91] Amin Tootoonchian, Aurojit Panda, Chang Lan, Melvin Walls, Katerina Argyraki, Sylvia Ratnasamy, and Scott Shenker. Resq: Enabling slos in network function virtualization. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 283—-297, 2018. https://www.usenix.org/conference/nsdi18/presentation/tootoonchian.

[92] Tariq 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.

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

[94] VMware. Large packet loss in the guest os using vmxnet3 in esxi (2039495). https://kb.vmware.com/s/article/2039495, 2021. Accessed: June 2021.

[95] Ziye Yang, Ben Walker, James R Harris, Yadong Li, and Gang 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)*, pages 363–371, 2020. https://doi.org/10.1109/ICPADS51040.2020.00056.

[96] Yifan Yuan, Mohammad Alian, Yipeng Wang, Ren Wang, Ilia Kurakin, Charlie Tai, and Nam Sung Kim. Don't forget the I/O when allocating your LLC. In *ACM International Symposium on Computer Architecture (ISCA)*, pages 112–125, 2021. https://doi.org/10.1109/ISCA52012.2021.00018.

[97] Arseniy Zaostrovnykh, Solal Pirelli, Luis Pedrosa, Katerina Argyraki, and George Candea. A formally verified NAT. In *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*, pages 141—154, 2017. https://doi.org/10.1145/3098822.3098833.