The Benefits of General-Purpose On-NIC Memory

Boris Pismenny
Technion
Haifa, Israel

Liran Liss
NVIDIA
Yokne'am Illit, Israel

Adam Morrison
Tel Aviv University
Tel Aviv, Israel

Dan Tsafrir
Technion and VMware Research
Haifa, Israel

ABSTRACT

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

CCS CONCEPTS

• **Networks** → **Network adapters**.

KEYWORDS

NIC, operating system, hardware/software co-design

ACM Reference Format:
Boris Pismenny, Liran Liss, Adam Morrison, and Dan Tsafrir. 2022. The Benefits of General-Purpose On-NIC Memory. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22), February 28 – March 4, 2022, Lausanne, Switzerland*. ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3503222.3507711

1 INTRODUCTION

Network speeds grow quickly, with 100 [12, 53, 81] and 200 [13, 54, 82] Gigabit Ethernet (GbE) network interface controllers (NICs) already widely available, 400 GbE arriving in 2021 [10, 94], and 800 GbE expected in a few years [22]. The increased network volume introduces bottlenecks across all the relevant components within a host system: CPU [34, 44, 45, 50, 68, 88, 118], memory [41, 78, 80, 112, 113]; and the PCIe interconnect [67, 92, 109]. We broadly classify the approaches to addressing this challenge into two categories.

The first category offloads all packet processing activity onto the NIC. It includes full application workloads such as key-value stores [26, 33, 63, 70, 111], and various network functions [15, 71, 103, 118]. This approach eliminates much of the overheads. But migrating the networking stack away from the host system has serious drawbacks, hindering the ease and flexibility of general purpose programming, encumbering innovation in the network stack, and imposing undesirable security and maintenance overheads [89, 102].

The second category retains the network stack on the host. It attempts to, e.g., reduce the cost of system software abstractions with specialized network stacks that bypass the kernel, avoid copies, and leverage direct hardware access [4, 56, 107], and it includes latency hiding techniques [61], cross-layer program optimizations [34, 64, 99], and algorithmic client-server co-design [73]. The contribution of this paper is compatible with this category, focusing on reducing the overheads associated with data movement between the NIC and the CPU.[1]

More specifically, our goal is to propose an effective, previously unnoticed type of high-throughput networking optimization, which is different than the approaches discussed above. Our new optimization rests on three observations. Firstly, there exists a canonical class of applications that are tasked with moving messages around, from some source to some destination, by exclusively operating on the *metadata* of messages. In contrast, the associated data is not used by such applications except for the purpose of moving it, as is. We denote this type of applications as *data movers* (§3).

A notable example of data movers is found in network function virtualization (NFV) workloads such as network address translation (NAT) [96] and load balancing (LB) [32]. In accordance with our above definition of data movers, such network functions are characterized by the property that (1) they make decision based on packet headers and may additionally modify these headers, but (2) they neither modify nor use packet payloads save for delivering them to their destinations. Key-value stores such as Memcached [38] and Redis [66] constitute another notable example of a data mover family of applications. In this type of workloads, the key and value are the metadata and data, respectively.

The second observation that underlies our proposed optimization is that all major networking vendors, including Broadcom, Intel,

ASPLOS '22, February 28 – March 4, 2022, Lausanne, Switzerland

ACM ISBN 978-1-4503-9205-1/22/02…$15.00
https://doi.org/10.1145/3503222.3507711

[1] We expect our proposal to benefit system-on-chip SmartNICs as well, because high network speeds stress CPU, memory, and interconnect [11, 93] even more than server systems [74].

and NVIDIA, increasingly equip their NICs with a small, fast, internal memory [9, 12, 49, 53, 81, 82, 84, 94]. For example, the newest NVIDIA NIC (ConnectX6-Dx) is equipped with 4 MiB SRAM. Ordinarily, this memory is designated to be used by various offloading, acceleration, and transport functionalities that the NIC supports and that its users may employ.

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

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

We assign the name "nmNFV" (short for "nicmem NFV") to our system that optimizes for NFV data movers with the help of nicmem. In implementing it, we rely on the ability of existing NICs to split each incoming packet into two different buffers that store the packet's header and payload [14, 29, 94]. When arming the receive (Rx) ring with memory buffers that absorb the incoming traffic, the NIC's packet-splitting ability allows nmNFV software to use nicmem for storing payload buffers, simply by populating the relevant Rx ring fields with nicmem pointers. In parallel, nmNFV software uses pointers to regular host memory for header buffers. Consequently, when a packet arrives, its header is placed in host memory, but its payload remains on the NIC, thus reducing PCIe traffic, host memory traffic, and hence latency. For data mover network functions (NFs), the header provides all the information required, so the NF does not mind that the payload is remote. When the NF finishes operating on the header, it transmits the packet using the same payload (nicmem) pointer it received, thus further reducing PCIe and host memory traffic and latency.

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

In dealing with key-value stores (KVS), we use the name "nmKVS" (short for "nicmem KVS") to describe our system that optimizes KVS workloads with the help of nicmem. The nmKVS infrastructure accelerates KVS data mover applications by letting them store popular values in nicmem. When incoming requests target such values, the data mover induces smaller PCIe and hostmem traffic overheads

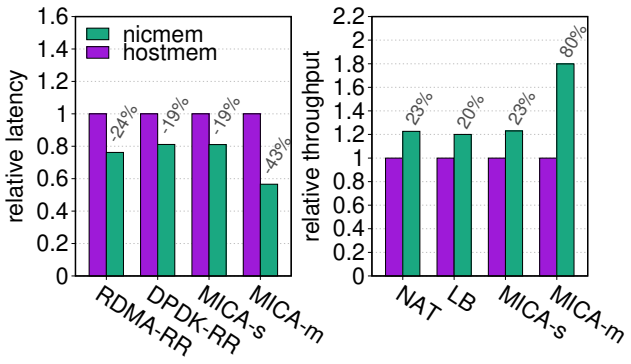

Figure 1: Preview of experimental results.

similarly to nmNFV, which likewise results in lower latency and higher throughput. KVS workloads are commonly skewed, exhibiting Zipf distributions [3, 6, 116]. Because nicmem is smaller than hostmem, such workloads are most suitable for nmKVS.

We describe our design of nmNFV and nmKVS (§4), and we explain how we implement a realistic prototype of the two using the NVIDIA ConnectX-5 NIC and its nicmem (§5). We experimentally evaluate our prototype using micro- and macro-benchmarks (§6). Figure 1 provides a preview of some of these results, using: two request-response ("RR") implementations [58, 106] that ping-pong a small message between them; the MICA [73] key-value store accelerated with nmKVS and serving a single ("s") or multiple ("m") clients; and the aforementioned NAT and LB network functions accelerated with nmNFV. As can be seen, our approach improves latency and throughput by up to 43% and 80%, respectively.

2 BACKGROUND

NICs commonly use two types of rings in each direction to store descriptors and completions. Descriptors point to software provided packet buffers, while completions mark descriptors processed by NIC hardware. We next describe how these rings are used to receive/transmit packets.

Receive flow (Rx). To receive packets, software pre-populates descriptors that point to buffers large enough to fit the maximum sized packet. Each time a packet arrives, the NIC hardware DMA reads a descriptor from the Rx ring, DMA writes the packet to the descriptor's payload buffer, and finally DMA writes a completion entry and possibly fires an interrupt. Then software uses the completion to find the corresponding descriptor, and replaces the payload buffer in the descriptor.

Transmit flow (Tx). Each time software sends a packet, the NIC driver writes a descriptor that points to the packet's payload, and rings the doorbell. Then, the NIC hardware DMA reads the descriptor to find the payload buffer. Next, it DMA reads the payload buffer, and finally it DMA writes a completion to allow software to release the payload buffer.

3 MOTIVATION

As noted, our goal is to demonstrate the benefit of nicmem for "data movers," network applications that move unchanged data to its destination exclusively based on the associated metadata. To this end, we focus on two types of data mover workloads: network function virtualization (NFV) applications and key-value stores (KVS). In this section, after we discuss these workloads in more detail (§3.1), we exemplify the latency cost that they pay due to superfluously moving data from NIC to host memory and back (§3.2). We then enumerate system bottlenecks triggered by this superfluous activity (§3.3) and highlight why direct data I/O (DDIO) caching technology cannot eliminate this problem (§3.4). Finally, we present the technological trends behind on-NIC memory (§3.5).

3.1 Workloads

NFV. A lot of effort has been put into studying NFV [2, 15, 27, 31, 32, 59, 71, 118]. In this class of applications (called "network functions" or NFs), flexible software and off-the-shelf hardware replace rigid proprietary network equipment. Common NFs include firewalls, virtual private networks (VPN), deep packet inspectors (DPI), routers and forwarders, network address translators (NAT), load balancers (LB), flow monitors, and rate limiters. Aside from VPN and DPI, all the above NFs are data movers, operating on metadata only (packet headers and per-flow state) before delivering the packets to their next destination. NAT and LB are two particularly important data movers: a study of data center NFs showed that up to ≈60% of total traffic goes through one or both [100]. As end-to-end encryption prevails, NF access to packet payload is diminished, making data movers more important [46, 91].

KVS. Key-value stores like Memcached [38] and Redis [66] underlie key cloud and data center infrastructures and drive much of their network traffic [51, 104]. Significant research effort has thus been dedicated to improving them, both in software [73, 77, 78] and in hardware [48, 70, 74]. "Get" KVS operations are data movers: clients send keys (metadata) and servers return the matching values (data). KVS access patterns are commonly highly skewed [3, 6, 116], so improving the performance of a small set of hot key-value pairs can improve overall performance significantly.

Importantly, in this work, we use the term KVS more broadly than typical, also associating it with such applications as web servers (like Apache [37]) when serving static files.

3.2 Latency Cost

In high-throughput workloads, when traffic approaches line rate, we show that superfluous data movement between NIC and host memory causes systems to bottleneck (§3.3). But superfluous movement is also disadvantageous in underloaded, non-bottlenecked conditions, because it increases latency.

To illustrate, Figure 2 (left) shows the latency breakdown of Data Plane Development Kit (DPDK) ping-pong [58], which sends 64B and 1500B (MTU) packets over the ICMP protocol back and forth between two machines. The first bar ("host") corresponds to the baseline system, which delivers entire packets to host memory, whereas the second bar ("nic") corresponds to storing payloads in nicmem. The third and fourth bars respectively add the header inlining ("inl") optimization, storing headers in NIC rings as outlined in §1. For 1500B packets, nicmem shortens latency by 8% and 15% without and with inlining. For 64B packets, latency is shortened by 19% due to inlining only (nicmem does not play a part as the entire packet is inlined).

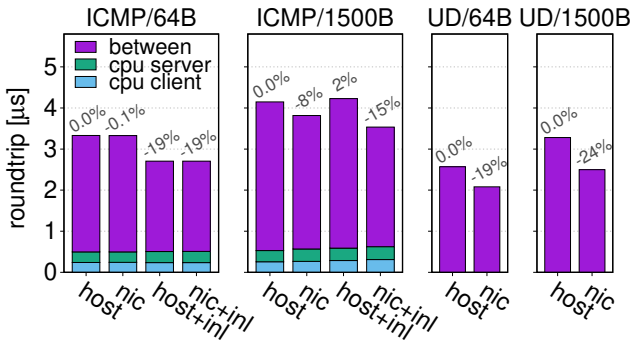

Figure 2: Superfluous data movement between NIC and host memory degrades performance even in underloaded conditions.

Observe that 64B latency is improved by our optimizations (19%) more than 1500B (15%), which is counterintuitive, as 64B benefits from only inlining, whereas 1500 also benefits from nicmem. We hypothesize that this happens because packet-splitting occurs only for 1500B, requiring software to process *two* ring entries when sending and receiving. We corroborate our hypothesis by repeating the ping-pong experiment using RDMA unreliable datagram (UD) [106], as RDMA rids software from having to handle headers. Figure 2 (right) shows that in this case the benefit for 1500B is indeed greater.

3.3 Bottlenecks

When high-throughput applications stress the network subsystem, the superfluous data movement we identify can bottleneck the three main components that are involved in accommodating this traffic: NIC, PCIe interconnect, and host memory. To exemplify, we run the DPDK Layer-3 forwarding benchmark (called l3fwd [57]) under three gradually intensifying setups configured to forward 1500B packets. The l3fwd server machine is connected back-to-back to a single client load generator machine running the Cisco T-Rex load generator [21]. Full details of our evaluation setup appear in §6.1.

NIC. The first experiment utilizes a single core driving a single 100 Gbps NIC. The average results are shown in Figure 3 (top), which depicts: (i) throughput; (ii) roundtrip latency; (iii) idle CPU cycles ("idleness"); (iv) PCIe bus traffic flowing from NIC to host-mem as observed by the NIC, expressed as percentage out of the maximal PCIe bandwidth available to the NIC, which is 125 Gbps ("PCIe out"); (v) PCIe traffic in the opposite direction ("PCIe in"); (vi) number of occupied Tx ring entries, as measured by the CPU whenever it enqueues packets, expressed as percentage of the ring size, which is 1024 ("Tx fullness"); and (vii) host DRAM bandwidth as measure by Intel pcm [55] ("mem bw"). We measure NIC PCIe utilization with NVIDIA NEO-Host [83].

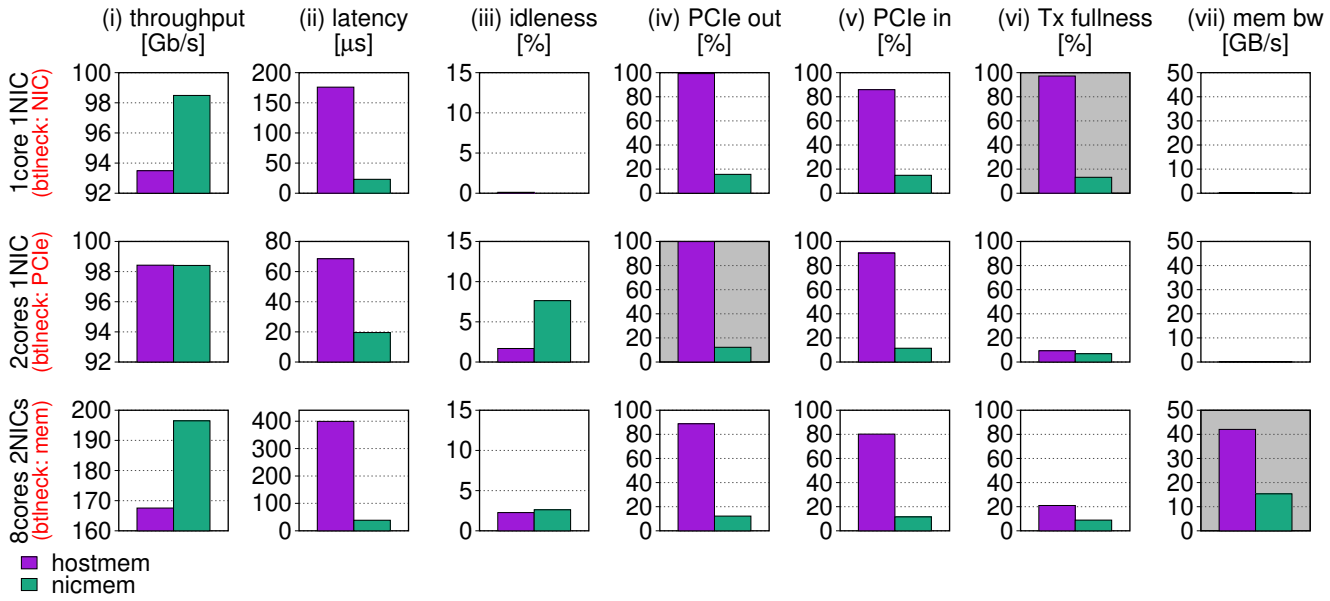

Figure 3: Bottlenecks triggered by superfluous traffic between NIC and host memory when running DPDK l3fwd.

Examining Figure 3 (top), we see that the baseline system is unable to achieve line rate (Fig. i), and that is suffers from high latency (Fig. ii) due to two bottlenecks: PCIe (Fig. iv) and Tx fullness (Fig. vi). We focus on the latter as it is unique to one core/ring processing and has been observed by others attempting to achieve one core/ring 100 Gbps [34, 35, 62]. We also remark that single ring bottlenecks are not unique to NICs as we also observe similar issues in NVMe SSDs.

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

The NIC's transmit engine gathers packets from Tx ring r over PCIe to stream them via the outgoing wire. PCIe is speedier (has higher throughput) than the wire, so r's packets accumulate in an internal NIC buffer b, until unavoidably b gets full. The NIC then reacts by de-scheduling transmission from r for a timeout duration of t, which, for reasons outside our scope, is set to be proportional to how long it takes to read another byte from r (≈PCIe roundtrip). The NIC assumes that other Tx rings will keep it busy during this timeout. But no such rings exist in our setup, and t is longer than b's drain time, so the NIC is left with no work, even though packets are awaiting in r. From the CPU perspective, the NIC appears to temporarily stop transmitting, causing r to fill up as observed. Nicmem does not suffer from this problem because for it b contains only headers, so the NIC has a lot more packets to send during t.

PCIe. The results of our second experiment are shown in the middle of Figure 3. Here, we use two cores (and hence two rings) instead of one. As expected, this eliminates the NIC bottleneck, allowing the baseline to achieve 100 Gbps. The bottleneck that remains is PCIe outbound, which is 99.8% saturated. NIC PCIe out operations are thus stalled and increase latency considerably (Fig. ii), and packets get discarded. We verified that PCIe out is indeed the bottleneck to blame by curbing the client to deliver 90 Gbps, which reduced server PCIe out bandwidth somewhat and resolved the problem.

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

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

Host Memory. Our third experiment resulted in the bottom row of Figure 3. Here, we use eight cores to handle double the throughput, utilizing two 100 Gbps NICs instead of one in both the client and the server. Additionally, to approximate a memory intensive NF, we configure l3fwd to perform 250 random memory reads per packet from a 8 MiB buffer. Although the baseline system offers an incoming load of 200 Gbps, the server is able to accommodate only ≈170 Gbps from it (Fig. i) with high latency (Fig. ii). This performance drop is caused by running out of DRAM bandwidth (Fig. vii), as the NIC reads and writes payload data that compete with the NF's memory activity, which prolongs its per-packet processing time. (Later, in Figure 7, we show that less than 10 per-packet memory reads are enough to bottleneck DRAM.) Nicmem does not suffer from these problems.

3.4 DDIO Limitations

The bottleneck resource of applications that exhibit poor memory locality is DRAM [20, 98]. As shown above, I/O-intensive applications might suffer from the same problem, because they involve high-throughput direct memory access (DMA) activity performed by I/O devices—an activity that contends for the same DRAM bandwidth resource [78, 109]. This issue also affects data movers like network functions, which consequently suffer from lower throughput, longer latency, and higher variability [30, 36, 40, 76, 113, 117]. The problem occurs even in "balanced" systems whereby, on paper, DRAM capacity exceeds I/O bandwidth. The reason is that, as memory utilization increases, access latency likewise increases: linearly at first, and then exponentially when nearing capacity [113].

Direct data I/O (DDIO) technology [24] can avoid or alleviate the problem, as it serves DMA reads from the last level cache (LLC) if the data is there, and it allows DMA writes to allocate up to two LLC ways by default, thereby bypassing DRAM. The effectiveness of DDIO, however, is inherently limited by LLC capacity dedicated to DMA writes [92, 97, 113]. Notably, at a fast enough rate, DDIO writes might evict still-unprocessed packets to DRAM (a.k.a. the "leaky DMA problem"), implying that for DDIO to be effective, the combined size of the buffers pointed by Rx rings should not exceed the LLC size dedicated to DDIO allocations [113].

Ideally, a handful of DDIO allocated cache lines would be enough to fit all receive buffers. However, multi-core packet processing requires a receive ring per core, and each ring entry must be large enough to store the maximum packet size (1500B). For example, an 1024-sized ring stores up to 2MiB of payload buffers which is as large as an entire LLC way on our system. To avoid exceeding DDIO capacity, one may consider to decrease ring sizes [113].

Unfortunately, one cannot arbitrarily reduce the size of rings to accommodate DDIO without negative implications. To illustrate, we employ the RFC2544 no drop rate (NDR) test [7] using single-core l3fwd with varying ring sizes, which finds the maximum throughput attainable without loss [117]. Figure 4 shows the result for 64B and 1500B packet sizes, which suggests a 1024-entries lower bound for 100 Gbps NICs (faster NICs may require more). Indeed, 1024 is the default ring size in all standard applications that come with the DPDK library as of 2018 [65]. Likewise, all major NIC drivers use 1024-sized rings or more [8, 79, 114]. We too use this size unless stated otherwise.

To accommodate high I/O rates, in addition to increasing ring sizes, researchers proposed to increase the number of LLC ways available for DDIO DMA writes [35, 117]. In both cases, the problem is that I/O and CPU potentially contend for the same LLC resource. Using Nicmem alleviates this problem.

3.5 On-NIC Memory Today

Our proposal hinges on several technological trends: (1) On-NIC memory *already* exists to support various NIC functionalities, it just is not available for software use; (2) the NIC's demand for on-NIC memory is limited; and (3) on-NIC memory size can be increased to support data mover usage.

Many commercial ASIC NICs contain on-NIC SRAM [9, 12, 49, 53, 81, 82, 84, 94] to support various optional NIC offloading, acceleration, and transport functionalities. For instance, on-NIC memory is used to cache packet steering rules [43, 62]. Due to its current target use case, on-NIC memory is relatively small. For example, the latest NIC model of NVIDIA (ConnectX-6Dx) is equipped with 4 MiB of on-NIC SRAM memory. But this size is dictated by the current use cases, not by technological constraints. NVIDIA architects acknowledge that it is feasible and cost-effective to increase on-memory NIC size to several MiBs—roughly, the equivalent of a CPU LLC size—given a compelling use case.

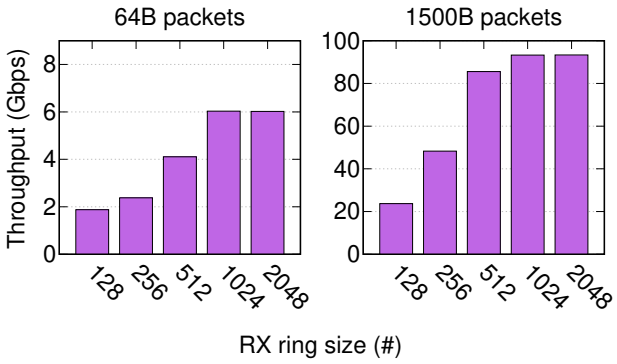

Figure 4: maximal attainable throughput without loss.

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

When adding SRAM we increase NIC die size, each bit cell spans 0.3 μm^2 of silicon [18] which translates to 0.21 $ per MB at estimated 7 nm process wafer prices [25, 28]. For 16$ (2% of the price of the cheapest 100 GbE NIC, 795$) [17], we can obtain 80 MBs which exceeds the LLC of the most powerful Intel 3rd gen scalable processor [52], and can sustain 37 NIC queues with 1024 entries each. Furthermore, we speculate that SRAM die size and price will decrease as 3D stacking technology unlocks hundreds of MBs of SRAM [101].

Because it is cost-effective and it simplifies our implementation, we assume nicmem is as large as CPU LLC. In our design, we show that it is possible to overcome this limitation (§4.1). In our evaluation, we show that even small amounts of nicmem are useful (§6.4).

4 DESIGN

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

We describe the required NIC hardware changes in §4.1. We then demonstrate nicmem's utility by designing nicmem-based systems for accelerating NFV and KVS applications (§4.2).

4.1 Nicmem Hardware

At a high-level, the nicmem design consists of providing large on-NIC SRAM, which is partitioned into two regions. Most of the

SRAM is called *nicmem* and is exposed to software, allowing data mover applications to use it for data storage and thereby accelerate data transfer. The rest of the SRAM is not exposed to software and is used by NIC hardware to support various functionalities, as is the case today.

NIC hardware supports identifying packet descriptors (either Rx or Tx) whose payload is located in nicmem. When writing (Rx) or reading (Tx) such a descriptor's payload, the NIC directly accesses its SRAM instead of going through the PCIe interconnect. This mechanism allows data movers to reap nicmem benefits via pure software techniques, by allocating their payloads on the nicmem and thus avoiding hostmem and PCIe traffic for data transfers (as we demonstrate in §4.2).

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

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

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

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

Receiving traffic into nicmem. Typical NIC receive flow (§2) makes it challenging to use nicmem for Rx traffic. Since nicmem size is limited, at high networking rates it might not suffice to hold a burst of incoming traffic. As a result, an Rx ring containing only nicmem buffers may become empty during such a burst, leading to packet drops.

To address this problem, we propose to employ a *split rings* mechanism, which is inspired by network page faults [69]. In this design, NIC receive rings are split in two: primary and secondary receive rings. When a packets arrives, the NIC tries to consume a buffer from the primary ring, which holds nicmem buffers. If the primary ring is empty, the NIC proceeds to consume a buffer from the secondary queue (Figure 5). Completion entries are stored in a single queue, as before, but with the entries indicating the order and location of received packets, i.e., primary or secondary ring. Packet buffers are subsequently returned by software to their original ring.

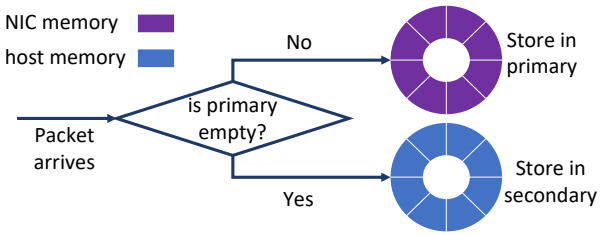

Figure 5: The split rings approach.

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

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

4.2 Leveraging Nicmem in Data Movers

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

In the following, we describe designs that use the above idea to accelerate NFV (§4.2.1) and KVS (§4.2.2) applications. Our designs assume that the application can safely manipulate NIC Rx/Tx rings directly and does not require OS intervention to send/receive traffic. This is the case, for example, in applications using DPDK which offers a packet processing programming model that is based on kernel bypass and direct hardware access for efficiency.

4.2.1 NFV Acceleration (NmNFV). Our design for accelerating NFV data movers with nicmem is named *nmNFV*. NmNFV mitigates memory bandwidth, DDIO and LLC contention caused by copying of packet payloads into hostmem for NF operations, as shown in Figure 6(a). Without nicmem, each incoming packet is (1) DMAed to hostmem by the NIC, (2) operated on by the NF; and finally (3) transmitted, which requires the NIC to read the header and

payload from hostmem with DMA again. Crucially, however, packet payloads are completely ignored by most NFs. The waste of copying payloads into hostmem is compounded by the fact that payloads are typically an order of magnitude larger than headers: network traffic characteristics studies show that packet sizes in data centers, universities, and on the Internet follow a bimodal clustering pattern around small ≈ 200 B and large ≈ 1400 B packets [5, 16, 42, 60, 108].

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

First, we rely on the pre-existing capability of the NIC to write an incoming packet's header and payload into different buffers [14, 29, 94]. NmNFV uses this packet-splitting functionality to configure the Rx ring with Rx descriptors that instruct the NIC to write headers into hostmem and payloads into nicmem (Figure 6(b)). Consequently, when a packet arrives, its payload remains on the NIC. Only the header is written to hostmem, which suffices for the NF to perform its operation. Finally, on transmit, the NIC already has the packet's payload in nicmem.

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

To address this overhead and to further optimize the NF flow, we propose to store a packet's header in its descriptor instead of in an independent hostmem buffer (Figure 6(c)). We call this optimization *header inlining*. It leverages pre-existing NIC inlining functionality by which descriptor flags can instruct the NIC to read/write a small range of packet data from/to the descriptor. Header inlining reduces the number of scatter-gather entries required to represent a packet back to one (for the payload). More importantly, it enables the NIC to fetch only the descriptor when sending/receiving data. This optimization thus reduces both the amount of data fetched from hostmem as well as the number of PCIe roundtrips required to do so, because in the non-optimized case the NIC must first read the descriptor in order to obtain the header's address in hostmem.

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

4.2.2 KVS Acceleration (NmKVS). In theory, a KVS could leverage nicmem by serving its item set from nicmem. The KVS would store the values (data) associated with the keys (metadata) in nicmem and each read request would be answered with a response whose payload is in the relevant item's nicmem. This approach is not viable in general, however, because the size of KVS item sets are as large as host DRAM [1, 110], which dwarfs the size of the multi-MiB nicmem.

We address this problem by leveraging the property that KVS workloads are commonly skewed, exhibiting a Zipf distribution [3, 6, 116]. We therefore propose *nmKVS*, which is a KVS design that stores a subset of *hot* items on nicmem and serves them from it.

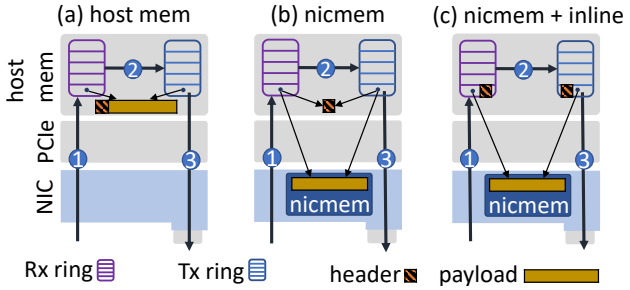

Figure 6: Host memory based packet transmission compared to data on nicmem with and without header inlining.

Our design focuses on the mechanism of serving hot items out of nicmem and not on identifying hot items in the first place. That is, we assume that a KVS can efficiently identify the hottest items—e.g., using a heavy hitters algorithm [19, 23, 87]—and move them to nicmem, while evicting "colder" items back to hostmem.

NmKVS relies on header-data splitting (§4.2.1) to perform zero-copy sends of values residing on nicmem. The basic idea is straightforward, but it creates a concurrency challenge. Suppose that a response containing an item is posted to the NIC's Tx queue, but has not been transmitted yet. In the meantime, the KVS receives an update operation of that value and the CPU starts overwriting the old value. Because the value is updated in place, if the NIC now begins transmission of the queued response packet, it might read (and transmit) an inconsistent mix of the old and new values, since it reads the value concurrently to the CPU updating it.

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

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

5 IMPLEMENTATION

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

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

DPDK API. To expose nicmem to DPDK applications, we introduce a new API for DPDK NICs (Listing 1). Applications allocate NIC memory using `alloc_nicmem` and free it using `dealloc_nicmem`. Applications manage this memory using standard DPDK memory allocator APIs such as packet memory buffer pools.

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

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

NmNFV. We implement nmNFV in the DPDK l3fwd [57] application and in the modular FastClick [4, 90] NF composition framework. Our implementation closely follows the nmNFV design. After allocating and mapping nicmem, the NF creates a packet buffer pool on top of nicmem. Next, it configures receive rings to split packets at a 64B offset into header and data buffers residing in hostmem and nicmem buffer pools, respectively, and to inline the headers. Split packets consist of two DPDK `mbuf` structures chained together: one that holds the header and another that points to the data which is either in hostmem or in nicmem.

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

NmKVS. We implement nmKVS on top of MICA [73], a highly optimized DPDK-based KVS that is built to achieve the highest performance on CPUs [72]. However, MICA get operations do copy item data twice: once from the KVS table to the stack and again from the stack to the response packet. We speculate that the reason behind this implementation is that copy semantics greatly simplify the design and implementation of the system and/or that it was forced by missing DPDK features, such as a callback upon completion of a packet transmission. Our implementation extends DPDK to support these features.

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

Listing 1: routines to control NIC memory

Our nmKVS implementation modifies MICA to serve a set of hot items directly from nicmem with the zero-copy design described in §4.2.2. We allocate stable buffers for hot items according to available nicmem and use `memcpy` to overwrite values on nicmem as needed. We additionally introduce a DPDK callback on transmit completion to decrement the stable buffer's reference count. Such a callback was not available in DPDK before, and so we modify DPDK and NVIDIA drivers to support it.

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

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

6 EVALUATION

We use microbenchmarks and macrobenchmarks to evaluate nicmem performance for KVS and NFV workloads. After introducing our methodology (§6.1), we evaluate NF performance with a syntactic microbenchmark (§6.2) and then real NF applications: network address translation (NAT) and load-balancer (LB). Based on these, we quantify the number of cores required to saturate 200 Gbps and measure the impact of various packet and NIC receive ring sizes and DDIO way allocations (§6.3). We then measure the impact of split-ring spilling to hostmem by varying the nicmem available in NAT and LB NFs (§6.4). We next measure the cost of accessing nicmem from the CPU (§6.5) and quantify KVS performance using nicmem (§6.6).

6.1 Methodology

Our setup consists of a pair of Dell PowerEdge R640 servers, one of which is the system under test and the other is the load generator. Both have 16-core 2.1 GHz Xeon Silver 4216 CPUs, 128 GiB (=4x16 GiB) 2933 MHz DDR4 memory, 22 MiB LLC split across 11

ways. They run Ubuntu 18.04 (Linux 5.6.0) with hyperthreading and Turbo Boost off.

The machines are connected back-to-back via two 100 GbE NVIDIA ConnectX-5 NICs [81]. All the results presented are trimmed means of ten runs; the minimum and maximum are discarded. The standard deviation is always below 5%.

NF Benchmarking. On the load generator machine, we run the stateless Cisco T-Rex packet generator [21], which we modify to improve latency measurement accuracy from 10-100μs to 1μs (similarly to Primorac [105]). Unless stated otherwise, we send packets at 200 Gbps using our two NICs.

For macrobenchmarking, on the server, we run FastClick [4] based NAT and LB using FastClick's DPDK mbuf pool to avoid unnecessary packet metadata copies. Unless stated otherwise: we disable pause frames; use 1024 Rx and Tx ring descriptors (default); two DDIO LLC ways (default); and 14 cores (as our experience with NAT and LB shows that 14 cores are needed to process 200 Gbps; see Figure 8). To maximize CPU efficiency and reach line rate speeds, we spread load equally among all cores using a different flow per packet. We use large 1500B UDP packets unless stated otherwise, as this is a common use case (§4.2.1), and because it helps us sustain 200 Gbps processing on our setup, which generates the highest load on PCIe, DDIO, and memory bandwidth.

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

As in §3.3, we measure NIC PCIe utilization using NVIDIA NEO-Host [83] and CPU core and unncore counters using Intel pcm [55].

KVS Benchmarking. We evaluate the performance of nmKVS using MICA [73] executing on 4 cores. MICA's client is the load generator, using 800K large key-value pairs (128B keys and 1024B values), which we access uniformly at random. As noted, large values ease CPU processing and are common in real workloads [1, 3]. We evaluate two server configurations: (C1) 256 KiB hot area cache that corresponds to the size of nicmem available on our NICs, and (C2) 64 MiB hot area corresponding to a future device that we emulate (§5).

6.2 NF Microbenchmarks

We use a synthetic NF to explore how memory subsystem contention affects CPU efficiency and, in turn, the throughput and latency of NFs with different attributes. To control NF memory intensity we run layer-2 forwarding followed by the WorkPackage FastClick element, which performs a number of random memory reads from preallocated buffers. We perform 480 runs, covering the space of the following parameters: Rx ring size: 256, 512, 1024, or 2048; accessed memory buffer size: 1, 2, 4, 8, 16, or 32; memory reads per packet: 2, 4, 6, 8, or 10; and DDIO ways: 0, 2, 8, or 11. For each NF processing configuration, we plot the missing throughput (200Gbps - measured) and latency of all 480 runs in a scatter-plot in Figure 7.

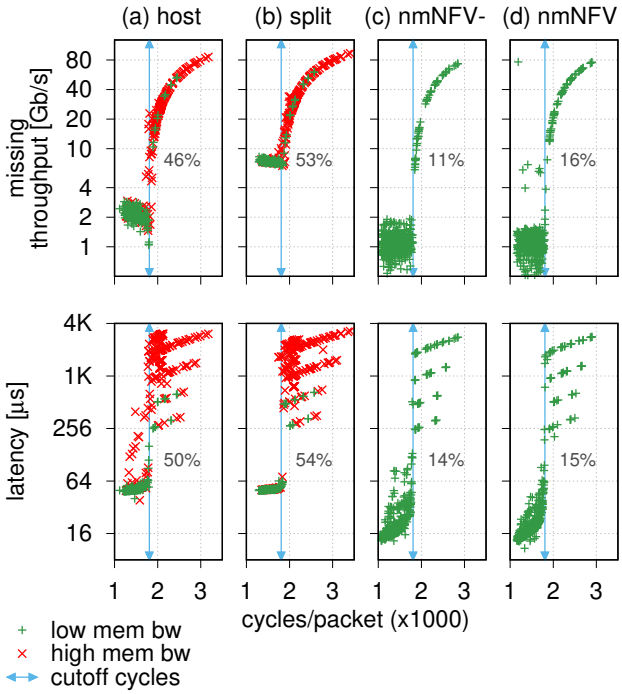

Figure 7: Synthetic NF performance. The points are runs with varying Rx ring size, NF memory intensity, and DDIO ways. Percentages show the number of runs past the cutoff.

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

We mark runs with less than 30 GB/s memory bandwidth with "+" and the rest with "x". We observe that both nmNFV variants eliminate memory bandwidth contention (all are below 30 GB/s), while base and split suffer from the leaky DMA problem and high memory bandwidth contention: at least 60% of runs have more than 30 GB/s memory bandwidth, and in fact at least 31% exceed 40 GB/s. Consequently, both nmNFV variants have as much as 42% more runs within the cutoff budget. Pleasingly, the majority of both nmNFV variants results below the cutoff also have better throughput and latency.

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

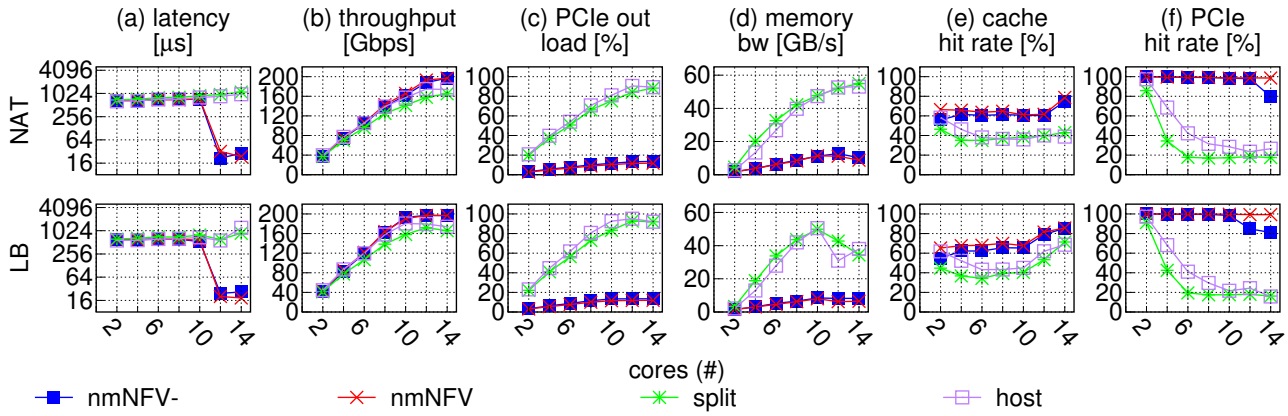

Figure 8: To handle 200 Gbps loads NAT and LB need (1) at least 12 cores and (2) to reduce memory and PCIe load.

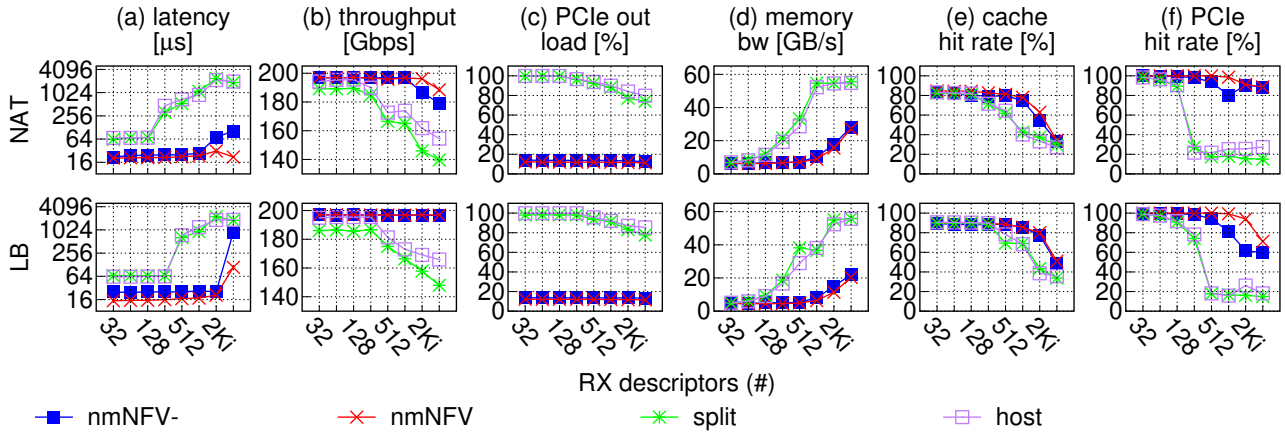

Figure 9: Small receive ring size can alleviate memory bandwidth bottlenecks, increasing throughput. But, these are susceptible to packet loss during bursts.

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

6.3 NF Macrobenchmarks

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

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

Cores. Figure 8 shows the results for LB and NAT scalability from 2 to 14 cores. Host and split fall short of reaching line rate throughput and as a result their latency increases with the number of cores. The reason is DDIO thrashing of the LLC due to the leaky DMA problem. The DDIO hit rate declines and memory bandwidth increases as we increase the core count.

Both nmNFV variants, in contrast, achieve line-rate throughput at 12 and 14 cores for LB and NAT, respectively. When approaching line-rate, improvements manifest in reduced latency. As expected, both variants improve PCIe hit rate, PCIe outbound utilization, memory bandwidth, and CPU cache hit rate. We remark that split

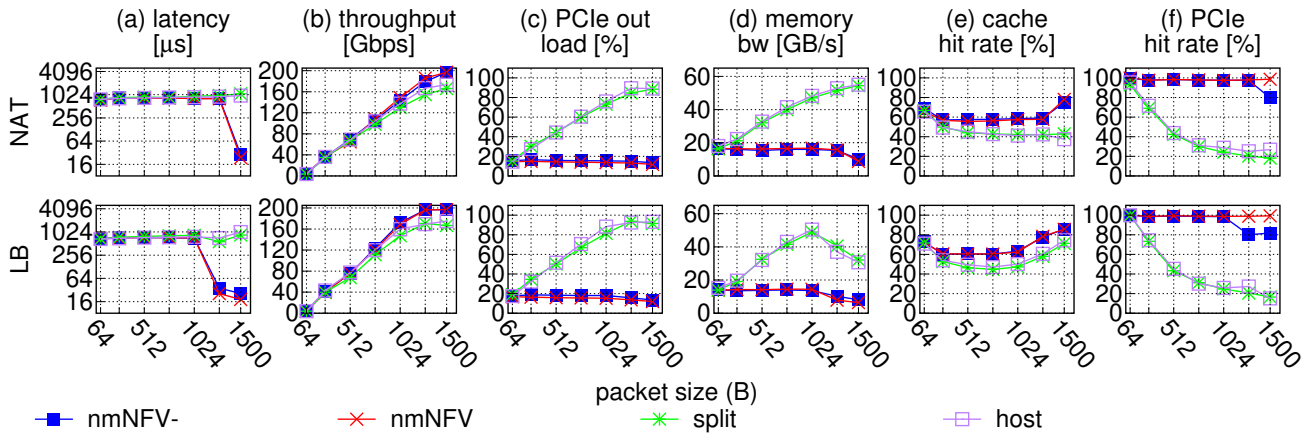

Figure 10: Our approach enables efficient 200 Gbps processing for large packets. Small packet workloads are always CPU bound.

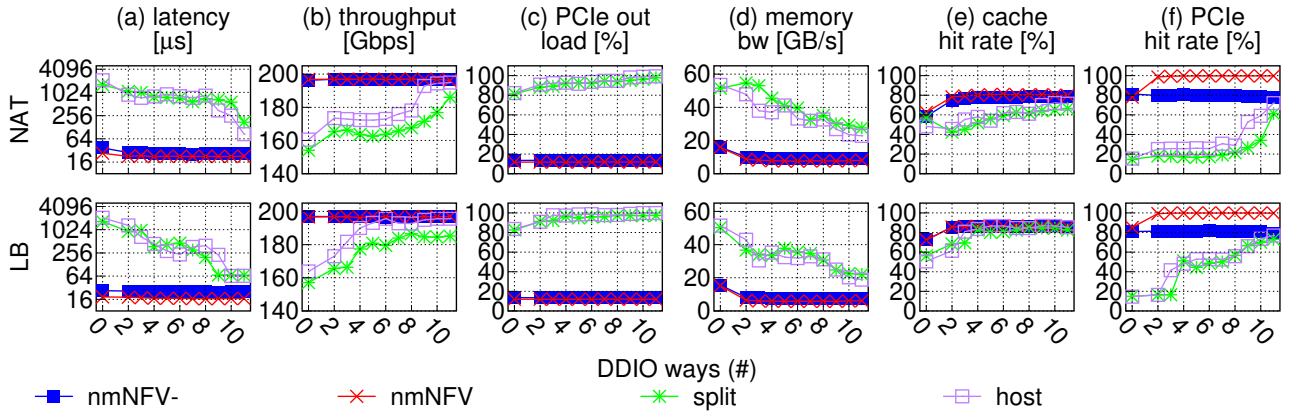

Figure 11: A system with DDIO disabled and nicmem enabled outperforms the same system with maximum DDIO and no nicmem.

and nmNFV- use two scatter-gather entries compared to one for nmNFV and host, and as result their performance is lower.

Real trace. We repeated the experiment above with the first million packets from a 2019 real-world CAIDA packet trace form the Equinix NYC monitor [16]. The trace we used contains 43261 unique source IPs and 58533 unique destination IPs with an average packet size of 916 bytes (small and large packet clusters). Figure 12 presents the results. Due to limitations in our load generator (T-Rex), we cannot measure latency so we focus on throughput. Both nmNFV- variants outperform base by up to 28%. The results are similar to Figure 8 with lower throughput for all as small packets increase the load on the CPU without benefiting from nicmem.

Rx Descriptors. To examine the performance impact of growing Rx ring sizes, which are necessary to handle packet bursts (see §3.4), we measure the performance with Rx ring sizes between 32 and 4096 (Figure 9). We observe that increasing ring size decreases throughput by up to 15% and 20% for LB and NAT, respectively.

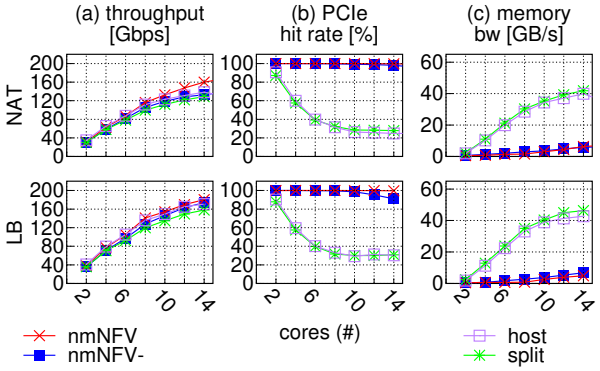

Figure 12: Performance with real packet trace from CAIDA.

Latency grows exponentially as LB and NAT fail to meet the offered

load at 256 and 128 Rx descriptors, respectively. This is preceded by a sharp decline in PCIe hit rate, as the total Rx ring buffers exceed available LLC space for DDIO: $256 \times 14 \times 1500 \approx 5\,MiB > 4\,MiB$ available to DDIO. Interestingly, host and split NAT performance diminishes before exceeding DDIO LLC capacity. We observe that NAT's higher LLC access rate and occupancy are responsible, as NAT uses two cache entries per flow, i.e., one for each direction and LB uses only one. Base and split application cache hit rate plummets from 83% to 27% and memory bandwidth grows from 5 GB/s to 55 GB/s, in correlation with PCIe hit rate, which reaffirms the importance of LLC locality and low memory bandwidth to NF performance.

Packet Size. Figure 10 shows the performance with packet sizes between 64B and 1500B. We observe that for both nmNFV variants throughput and latency is similar or better than host and split for all packet sizes. Both variants achieve better throughput for packets larger than 1024 B. Both variants also improve memory bandwidth, PCIe utilization, and PCIe hit rate for all packet sizes.

DDIO. Figure 11 shows performance with various DDIO cache way allocations. To control DDIO cache ways, we use the DDIOTune fastclick element developed by Farshin et al. [35]. The results show that a system with DDIO disabled and nicmem enabled outperforms the same system with maximum DDIO assigned LLC ways and no nicmem in latency (22 µs vs. 84 µs) and throughput (197 Gbps vs. 195 Gbps).

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

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

6.4 Insufficient NIC Memory Capacity

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

Figure 13 presents the results. We observe that a single nicmem queue (out of 7 in total per NIC) drastically improves latency and throughput as it eliminates the PCIe bottleneck discussed in §3.3. Meanwhile, increasing nicmem queues further reduces memory bandwidth, DDIO contention, and improves application LLC hit rate (not shown).

6.5 Cost of Accessing NIC Memory

In this section, we compare the cost of CPU access to nicmem in comparison to hostmem. Figure 14 compares the copy rate within

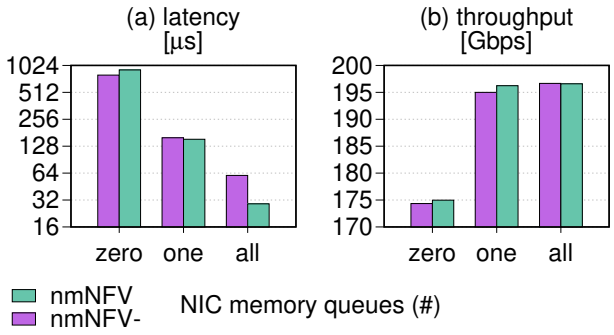

Figure 13: NFV performance improves with 1/7 nicmem queues.

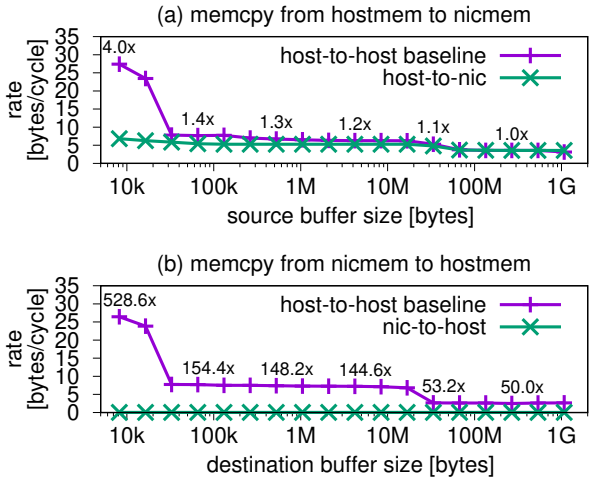

Figure 14: Cost of copy between hostmem and nicmem.

hostmem with the copy rate from hostmem to nicmem, and vice versa. Our experiment measures the throughput of 100 copy iterations within hostmem to the same copy loop with source/destination in nicmem.

We observe that the results differ greatly between the two. On the one hand, the rate of copy into nicmem decreases as the source buffer grows in size, from 4.0x for buffers in L1 (32 KiB) to 1.0x for non-cached data. On the other hand, the rate of copy from nicmem to hostmem incurs between 528x and 50x overhead. This is because nicmem is marked for write-combined caching, which permits the caching of writes, but prevents the caching of reads.

6.6 Key-Value Store

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

100% Get Workload. Figure 15 shows nmKVS performance with 100% get load, varying the load directed at hot items. This is the best-case scenario, as nicmem is never accessed directly by the CPU

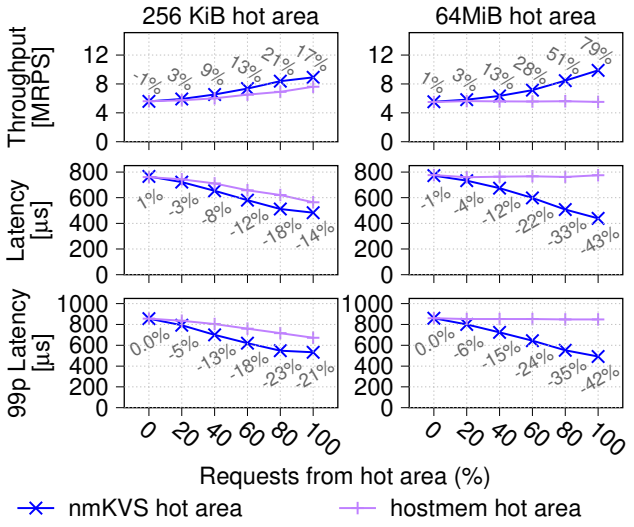

Figure 15: MICA 100% get throughput and latency. Labels show improvement over hostmem.

when processing get requests (§4.2.2); response packet descriptors only reference data in nicmem that the NIC fetches when sending packets to the wire.

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

We remark that nmKVS improves throughput more than nmNFV while using nicmem only on transmit. The reason is that MICA must copy data to avoid zero-copy races (§4.2.2), an overhead we avoid in nmKVS. Meanwhile, in NFV systems, the baseline performs no copies and therefore the gap is smaller.

Mixed Workload. Figure 16 shows the throughput of nmKVS under various get/set request ratios. Recall that nmKVS sets are more costly as they need to write data in both hostmem and nicmem to avoid zero-copy races, therefore 100% sets is the worst case scenario for nmKVS. To show this scenario, we direct all sets to the hot area. We then consider two types of workloads, one in which all gets are served from the hot area (best case), denoted "allhit", and another where all gets go to non-hot area (worst case), denoted "nohit". Then, we compare (C1) and (C2) as above.

We observe that the nmKVS is no more than 5% worse in both (C1) and (C2) indicating that most set operations write into non-cached memory, which we confirm by observing 70% cache misses using 100% sets. In the best case, throughput improves by up to 23%

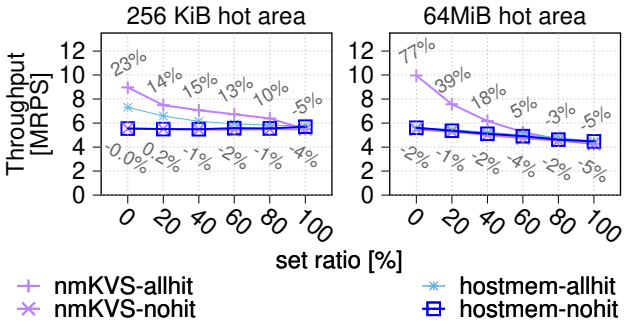

Figure 16: MICA set+get throughput using 4 cores. Labels show nmKVS relative to the corresponding baseline.

and 77% for (C1) and (C2), respectively. In (C1), serving gets from hot hostmem area improves throughput by up to 31%, in contrast to (C2) which performs the same regardless of whether gets are served from the hot area. This is due to the larger than LLC hot area in (C2).

7 NIC MEMORY AND NFV ACCELERATION

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

Today, in NFV acceleration, NIC memory stores per-flow state, such as packet steering rules used for NFV acceleration. In particular, products such as NVIDIA ASAP2 [85] will group packets to flows, apply actions such as count, modify, encapsulate, and decapsulae packet headers, and then send packets out (i.e., hairpin); all in ASIC without software involvement. This approach works best when all per-flow states fit inside NIC memory, but performance degrades as the number of flows grows. In contrast, nmNFV NIC memory utilization is independent of the number of flows, and it scales as well as baseline CPU based NFs while improving their performance.

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

Figure 17 shows the resulting throughput, latency, CPU utilization, and NIC cache misses. The figure shows that accelNFV is idle even when processing 100Gbps, as NIC ASIC processes packets without interfering with the CPU. We also observe that increasing the number of flows beyond on-NIC memory capacity, increases the time to process packets as the number of NIC context misses requires fetching and also evicting contexts to hostmem. When packets are processed too slowly, the Rx ring overflows causing significant packet loss and increased latency. Increasing the number of rings would not mitigate this problem, because it does not increase NIC processing speed through parallelism. In fact, performance

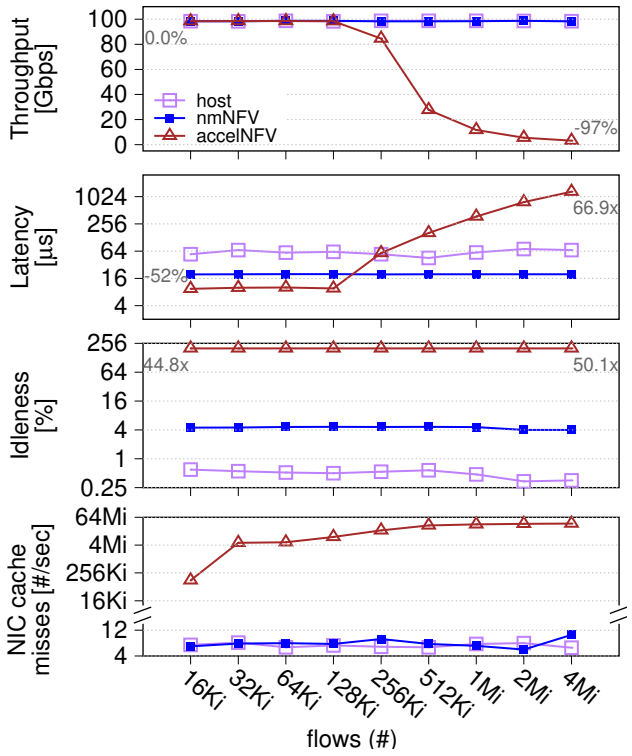

Figure 17: NFV scalability to large numbers of flows. The labels show the difference between nmNFV and accelNFV.

will degrade farther as additional rings will also contend over NIC memory.

8 RELATED WORK

Packet inlining. Splitting metadata (headers) from data (payload) is known to improve performance via better caching of CPU accessed metadata, and lower read amplification from prefetching and DMAing. This idea has been applied to log structured merge trees on SSDs [75], to sorting data [95], and with small request-response packets [39]. The contribution of this paper is in combining these techniques with nicmem to efficiently accelerate data-mover applications.

Header-data split. Previous work proposed storing NF packet payload on network switch memory [47]. Storing payload on nicmem is preferable because the NIC: (1) has more memory per host; (2) requires no coordination with switches on dropped packets; (3) allows for CPU offloading (e.g., checksum), which is impossible with switch parking; and (4) simplifies debugging as compared to switches.

NIC memory in RDMA. NIC memory has been used exclusively for RDMA so far [86]. In RDMA, it improves the latency of atomic operations [115], and small message transfers as we demonstrated in §3.2.

9 CONCLUSIONS

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

ACKNOWLEDGMENTS

This research was funded in part by the Israel Science Foundation (grant 2005/17), and the Blavatnik Family Foundation.

A ARTIFACT APPENDIX

A.1 Abstract

This artifact provides source code for NVIDIA ConnectX-5 NIC memory (nicmem) support in DPDK, which we then use in Fastclick to implement nmNFV and in MICA to implement nmKVS. This artifact also provides experiments that stress nmNFV and nmKVS. nmNFV experiments use an network address translator and a load balancer with 200GbE traffic generated by the T-Rex traffic generator. nmKVS uses MICA's load generator. In principle, our design stores packet payload on nicmem and passes packet headers to the CPU for processing. This reduces DDIO, LLC, memory and PCIe bandwidth.

A.2 Artifact check-list (meta-information)

- **Run-time environment:** DPDK, Fastclick.
- **Hardware:** ConnectX-5.
- **Metrics:** DDIO, LLC, Memory bandwidth, PCIe bandwidth.
- **How much time is needed to prepare workflow?:** 5min.
- **How much time is needed to complete experiments (approximately)?:** 1 run in 1.5–8hrs, 10 runs in 15–80hrs.

A.3 Description

A.3.1 Hardware dependencies. This artifact expects two Intel Skylake machines connected back-to-back using two NVIDIA ConnectX-5 NICs.

A.3.2 Software dependencies. OS: Ubuntu 18.04, Linux kernel 5.4; libraries: DPDK 20.08; applications: Fastclick and MICA; traffic generator: T-Rex

A.4 Installation

Our artifact is available on github: https://github.com/BorisPis/nicmem-asplos22-artifact

(1) Clone the artifact to the same path on both machines.
(2) Only on the server, setup a kernel with hugepages. Fastclick experiments use 1G hugepages and MICA experiments use 2M hugepages.
(3) On both client and server, import useful environment variables: `source ./scripts/env.sh`
(4) On the client machine, prepare all git modules: `./scripts/make-client.sh`
(5) On the server machine, prepare all git modules: `./scripts/make-server.sh`

The last two steps clone and compile everything you need. Given all was successful, then all is ready to run benchmarks. Benchmarks are executed from the server machine which operates the load generator remotely via ssh. Make sure password-less ssh is configured between the server and the client.

We remark that our scripts use hard-coded MAC and IP addresses. Different configurations will require updating our scripts accordingly.

A.5 Experiment workflow

Enter the directory corresponding to the desired figure (e.g., fig7) in the paper and follow instructions in the README. In all cases, one file runs the experiment, and another file (`plot.sh`) analyses results and produces the corresponding figure.

A.6 Evaluation and expected results

Running the experiments 10 times (`REPEAT=10`) should produce Figures 7, 8, 9, 11, and 12 from the paper.

REFERENCES

[1] Atul Adya, Daniel Myers, Henry Qin, and Robert Grandl. 2019. Fast key-value stores: An idea whose time has come and gone (HotOS'19 talk slides). https://ai.google/research/pubs/pub48030. (Accessed: Aug 2021).

[2] Fabien André, Stéphane Gouache, Nicolas Le Scouarnec, and Antoine Monsifrot. 2018. Don't share, Don't lock: Large-scale Software Connection Tracking with Krononat. In *USENIX Annual Technical Conference (ATC)*. 453–466. https://www.usenix.org/conference/atc18/presentation/andre.

[3] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload Analysis of a Large-Scale Key-Value Store. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. 53—-64. https://doi.org/10.1145/2254756.2254766.

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

[5] Theophilus Benson, Aditya Akella, and David A. Maltz. 2010. Network Traffic Characteristics of Data Centers in the Wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*. 267––280. https://doi.org/10.1145/1879141.1879175.

[6] Peter Bodik, Armando Fox, Michael J. Franklin, Michael I. Jordan, and David A. Patterson. 2010. Characterizing, Modeling, and Generating Workload Spikes for Stateful Services. In *Symposium on Cloud Computing (SoCC)*. 241––252. https://doi.org/10.1145/1807128.1807166.

[7] S. Bradner and J. McQuaid. 1999. *Benchmarking Methodology for Network Interconnect Devices*. RFC 2544. Internet Engineering Task Force. 31 pages.

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

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

[10] Broadcom. 2019. BCM957508-P2200G. https://docs.broadcom.com/doc/957508-P2200G-DS. Accessed: 2021-04-16.

[11] Broadcom. 2020. BCM5880X SmartNIC Adapters. https://docs.broadcom.com/docs/5880X-UG30X. Accessed: 2021-04-16.

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

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

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

[15] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. 2020. hXDP: Efficient Software Packet Processing on FPGA NICs. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*. 973–990. https://www.usenix.org/conference/osdi20/presentation/brunella.

[16] Caida 2019. CAIDA dataset. https://www.caida.org/catalog/datasets/trace_stats/. (Accessed: May 2021.).

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

[18] Jonathan Chang, Yen-Huei Chen, Wei-Min Chan, Sahil Preet Singh, Hank Cheng, Hidehiro Fujiwara, Jih-Yu Lin, Kao-Cheng Lin, John Hung, Robin Lee, Hung-Jen Liao, Jhon-Jhy Liaw, Quincy Li, Chih-Yung Lin, Mu-Chi Chiang, and Shien-Yang Wu. 2017. A 7nm 256Mb SRAM in high-k metal-gate FinFET technology with write-assist circuitry for low-V_{MIN} applications. In *IEEE International Solid-State Circuits Conference (ISSCC)*. 206–207. https://doi.org/10.1109/ISSCC.2017.7870333.

[19] Moses Charikar, Kevin Chen, and Martin Farach-Colton. 2002. Finding frequent items in data streams. In *International Colloquium on Automata, Languages, and Programming*. 693–703. https://doi.org/10.14778/1454159.1454225.

[20] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. 2019. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509* (2019).

[21] Cisco. 2015. TRex: Realistic Traffic Generator. https://trex-tgn.cisco.com/. (Accessed: May 2021.).

[22] Ehernet Technology Consortium. 2020. 800G specification. https://ethernettechnologyconsortium.org/wp-content/uploads/2020/03/800G-Specification_r1.0.pdf. Accessed: 2021-08-09.

[23] Graham Cormode and Shan Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* (2005), 58–75.

[24] Intel Corporation. 2012. 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. Accessed: 2020-07-18.

[25] CSET. 2020. AI Chips: What They Are and Why They Matter. https://cset.georgetown.edu/publication/ai-chips-what-they-are-and-why-they-matter/. Accessed: 2021-08-28.

[26] Alexandros Daglis, Mark Sutherland, and Babak Falsafi. 2019. RPCValet: NI-Driven Tail-Aware Balancing of µs-Scale RPCs. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 35–48. https://doi.org/10.1145/3297858.3304070.

[27] 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. 2018. Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 373–387. https://www.usenix.org/conference/nsdi18/presentation/dalton.

[28] Daniel Nenni. 2020. 7nm price is about right. https://semiwiki.com/forum/index.php?threads/5nm-wafer-cost-very-high.13101/#post-44127. SemiWiki forum discussion of CSET wafer prices report. Accessed: 2021-08-28.

[29] Intel Ethernet Networking Division. 2019. Intel 82599 10 GbE Controller Datasheet. https://www.intel.com/content/www/us/en/products/details/ethernet/500-controllers/82599-10-controllers/docs.html?s=Newest. Accessed: 2021-08-10.

[30] Mihai Dobrescu, Katerina Argyraki, and Sylvia Ratnasamy. 2012. Toward Predictable Performance in Software Packet-Processing Platforms. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 141–154. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/dobrescu.

[31] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. 2009. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *ACM Symposium on Operating Systems Principles (SOSP)*. 15—-28. https://doi.org/10.1145/1629575.1629578.

[32] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. 2016. Maglev: A Fast and Reliable Software Network Load Balancer. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 523–535. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/eisenbud

[33] Haggai Eran, Lior Zeno, Maroun Tork, Gabi Malka, and Mark Silberstein. 2019. NICA: An Infrastructure for Inline Acceleration of Network Applications. In *USENIX Annual Technical Conference (ATC)*. 345–362. https://www.usenix.org/conference/atc19/presentation/eran.

[34] Alireza Farshin, Tom Barbette, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. 2021. PacketMill: Toward per-Core 100-Gbps Networking. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 1—-17.

https://doi.org/10.1145/3445814.3446724.
[35] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. 2020. eexamining Direct Cache Access to Optimize I/O Intensive Applications for Multi-hundred-gigabit Networks. In *USENIX Annual Technical Conference (ATC)*. 673–689. https://www.usenix.org/conference/atc20/presentation/farshin.
[36] Alireza Farshin, Amir Roozbeh, Gerald Q Maguire Jr, and Dejan Kostić. 2019. Make the most out of last level cache in intel processors. In *ACM Eurosys*. 1–17.
[37] Roy T. Fielding and Gail Kaiser. 1997. The Apache HTTP Server Project. *IEEE Internet Computing* 1, 4 (Jul 1997), 88–90. http://dx.doi.org/10.1109/4236.612229.
[38] Brad Fitzpatrick. 2004. Distributed Caching with Memcached. *Linux Journal* 2004, 124 (Aug 2004), 5. http://dl.acm.org/citation.cfm?id=1012889.1012894.
[39] Mario Flajslik and Mendel Rosenblum. 2013. Network Interface Design for Low Latency Request-Response Protocols. In *USENIX Annual Technical Conference (ATC)*. 333–346. https://www.usenix.org/conference/atc13/technical-sessions/presentation/flajslik.
[40] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. 2020. Caladan: Mitigating interference at microsecond timescales. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*. 281–297.
[41] Drew Gallatin. 2017. Serving 100 Gbps from an Open Connect Appliance. https://netflixtechblog.com/serving-100-gbps-from-an-open-connect-appliance-cdb51dda3b99. Accessed: 2020-09-09.
[42] Johan Garcia, Topi Korhonen, Ricky Andersson, and Filip Västlund. 2018. Towards Video Flow Classification at a Million Encrypted Flows Per Second. In *2018 IEEE 32nd International Conference on Advanced Information Networking and Applications (AINA)*. 358–365. https://doi.org/10.1109/AINA.2018.00061.
[43] Massimo Girondi, Marco Chiesa, and Tom Barbette. 2021. High-speed Connection Tracking in Modern Servers. In *IEEE International Conference on High Performance Switching and Routing (HPSR)*. 1–8. https://doi.org/10.1109/HPSR52026.2021.9481841.
[44] Younghwan Go, Muhammad Asim Jamshed, YoungGyoun Moon, Changho Hwang, and KyoungSoo Park. 2017. APUNet: Revitalizing GPU as Packet Processing Accelerator. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 83–96. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/go.
[45] Hossein Golestani, Amirhossein Mirhosseini, and Thomas F. Wenisch. 2019. Software Data Planes: You Can't Always Spin to Win. In *Symposium on Cloud Computing (SoCC)*. 337–350. https://doi.org/10.1145/3357223.3362737.
[46] Google. 2021. HTTPS encryption on the web. https://transparencyreport.google.com/https/overview. Accessed: 2021-08-05.
[47] Swati Goswami, Nodir Kodirov, Craig Mustard, Ivan Beschastnikh, and Margo Seltzer. 2020. Parking Packet Payload with P4. In *ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT)*. 274—-281. https://doi.org/10.1145/3386367.3431295.
[48] Stewart Grant, Anil Yelam, Maxwell Bland, and Alex C. Snoeren. 2020. SmartNIC Performance Isolation with FairNIC: Programmable Networking for the Cloud. In *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*. 681––693. https://doi.org/10.1145/3387514.3405895.
[49] Intel Ethernet Product Group. 2021. Intel Ethernet Controller X710/XXV710/XL710. https://www.intel.com/content/dam/www/public/us/en/documents/release-notes/xl710-ethernet-controller-feature-matrix.pdf. Accessed: 2021-08-10.
[50] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. 2010. PacketShader: A GPU-Accelerated Software Router. In *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*. 195–206. https://doi.org/10.1145/1851182.1851207.
[51] Thulara N. Hewage, Malka N. Halgamuge, Ali Syed, and Gullu Ekici. 2018. Review: Big data techniques of google, Amazon, Facebook and Twitter. *Oxford University Press Journal of Communications* 13, 2 (Feb 2018), 94–100. https://doi.org/10.12720/jcm.13.2.94-100.
[52] Intel. 2021. 3rd Generation Intel® Xeon® Scalable Processors. https://ark.intel.com/content/www/us/en/ark/products/series/204098/3rd-generation-intel-xeon-scalable-processors.html. Accessed: 2021-08-31.
[53] Intel. 2021. Intel Ethernet Network Adapter E810-2CQDA2. https://ark.intel.com/content/www/us/en/ark/products/192561/intel-ethernet-network-adapter-e810-cqda1.html. Accessed: 2021-08-10.
[54] Intel. 2021. Intel Ethernet Network Adapter E810-2CQDA2. https://ark.intel.com/content/www/us/en/ark/products/210969/intel-ethernet-network-adapter-e810-2cqda2.html. Accessed: 2021-08-10.
[55] Intel. 2022. Processor Counter Monitor (PCM). https://github.com/opcm/pcm. Accessed: 2021-02-05.
[56] Intel Corporation. 2010. DPDK: Data Plane Development Kit. http://dpdk.org. (Accessed: May 2016).
[57] Intel Corporation. 2012. L3 Forwarding Sample Application. https://doc.dpdk.org/guides/sample_app_ug/l3_forward.html. (Accessed: May 2021).
[58] Intel Corporation. 2020. DPDK Ping-Pong. https://github.com/zylan29/dpdk-pingpong. (Accessed: May 2021).
[59] 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. 2013. B4: Experience with a Globally-Deployed Software Defined Wan. In *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*. 3––14. https://doi.org/10.1145/2486001.2486019.
[60] Piotr Jurkiewicz, Grzegorz Rzym, and Piotr Boryło. 2021. Flow length and size distributions in campus Internet traffic. *Computer Communications* 167 (2021), 15–30. https://www.sciencedirect.com/science/article/pii/S0140366420320223.
[61] Anuj Kalia, Dong Zhou, Michael Kaminsky, and David G. Andersen. 2015. Raising the Bar for Using GPUs in Software Packet Processing. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 409–423. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/kalia.
[62] Georgios P. Katsikas, Tom Barbette, Dejan Kostić, Rebecca Steinert, and Gerald Q. Maguire Jr. 2018. Metron: NFV Service Chains at the True Speed of the Underlying Hardware. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 171–186. https://www.usenix.org/conference/nsdi18/presentation/katsikas.
[63] Antoine Kaufmann, SImon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. 2016. High Performance Packet Processing with FlexNIC. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 67–81. http://dx.doi.org/10.1145/2872362.2872367.
[64] Eddie Kohler, Robert Morris, and Benjie Chen. 2002. Programming Language Optimizations for Modular Router Configurations. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 251—-263. https://doi.org/10.1145/605397.605424.
[65] Kevin Laatz. 2018. [dpdk-dev] [PATCH v2 0/3] Increase default RX/TX ring sizes. https://mails.dpdk.org/archives/dev/2018-January/086889.html. Intel DPDK. Accessed: June 2021.
[66] Redis Labs. 2009. Redis. https://redis.io/. Accessed: 2021-08-06.
[67] Nikita Lazarev, Shaojie Xiang, Neil Adit, Zhiru Zhang, and Christina Delimitrou. 2021. Dagger: Efficient and Fast RPCs in Cloud Microservices with near-Memory Reconfigurable NICs. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 36—-51. https://doi.org/10.1145/3445814.3446696.
[68] Baptiste Lepers, Vivien Quema, and Alexandra Fedorova. 2015. Thread and Memory Placement on NUMA Systems: Asymmetry Matters. In *USENIX Annual Technical Conference (ATC)*. 277–289. https://www.usenix.org/conference/atc15/technical-session/presentation/lepers.
[69] Ilya Lesokhin, Haggai Eran, Shachar Raindel, Guy Shapiro, Sagi Grimberg, Liran Liss, Muli Ben-Yehuda, Nadav Amit, and Dan Tsafrir. 2017. Page fault support for network controllers. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 449–466. https://doi.org/10.1145/3037697.3037710.
[70] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. 2017. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *ACM Symposium on Operating Systems Principles (SOSP)*. 137–152. https://doi.org/10.1145/3132747.3132756.
[71] Bojie Li, Kun Tan, Layong (Larry) Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. 2016. ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*. 1—-14. https://doi.org/10.1145/2934872.2934897.
[72] Sheng Li, Hyeontaek Lim, Victor W. Lee, Jung Ho Ahn, Anuj Kalia, Michael Kaminsky, David G. Andersen, Seongil O, Sukhan Lee, and Pradeep Dubey. 2015. Architecting to achieve a billion requests per second throughput on a single key-value store server platform. In *ACM International Symposium on Computer Architecture (ISCA)*. 476–488. https://doi.org/10.1145/2749469.2750416.
[73] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 429–444. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/lim.
[74] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. 2019. Offloading Distributed Applications onto SmartNICs Using IPipe. In *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*. 318–333. https://doi.org/10.1145/3341302.3342079.
[75] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *USENIX Conference on File and Storage Technologies (FAST)*. 133–148. https://www.usenix.org/conference/fast16/technical-sessions/presentation/lu.

[76] Antonis Manousis, Rahul Anand Sharma, Vyas Sekar, and Justine Sherry. 2020. Contention-Aware Performance Prediction For Virtualized Network Functions. In *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*. 270—-282. https://doi.org/10.1145/3387514.3405868.

[77] Ilias Marinos, Robert N.M. Watson, and Mark Handley. 2014. Network Stack Specialization for Performance. In *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*. 175–186. http://doi.acm.org/10.1145/2619239.2626311.

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

[79] Marvell. 2020. 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. Accessed: June 2021.

[80] John D. McCalpin. 2016. Memory Bandwidth and System Balance in HPC Systems. In *ACM/IEEE Supercomputing (SC)*. https://sites.utexas.edu/jdm4372/2016/11/22/sc16-invited-talk-memory-bandwidth-and-system-balance-in-hpc-systems/.

[81] Mellanox. 2017. ConnectX®-5 En Card Product Brief. https://www.mellanox.com/sites/default/files/related-docs/prod_adapter_cards/PB_ConnectX-5_EN_Card.pdf. Accessed: 2019-08-06.

[82] Mellanox. 2018. ConnectX®-6 En Card Product Brief. https://www.mellanox.com/sites/default/files/related-docs/prod_adapter_cards/PB_ConnectX-6_EN_Card.pdf. Accessed: 2019-08-06.

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

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

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

[86] Mellanox. 2021. Device Memory Programming Model. https://docs.mellanox.com/display/OFEDv502180/Programming#Programming-DeviceMemoryProgramming. Accessed: 2021-11-20.

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

[88] Amirhossein Mirhosseini, Hossein Golestani, and Thomas F. Wenisch. 2020. HyperPlane: A Scalable Low-Latency Notification Accelerator for Software Data Planes. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 852–867. https://doi.org/10.1109/MICRO50266.2020.00074.

[89] Jeffrey C Mogul. 2003. TCP Offload Is a Dumb Idea Whose Time Has Come. In *USENIX Workshop on Hot Topics in Operating Systems (HotOS)*. 25–30. https://www.usenix.org/conference/hotos-ix/tcp-offload-dumb-idea-whose-time-has-come.

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

[91] David Naylor, Alessandro Finamore, Ilias Leontiadis, Yan Grunenberger, Marco Mellia, Maurizio Munafò, Konstantina Papagiannaki, and Peter Steenkiste. 2014. The Cost of the "S" in HTTPS. In *ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT)*. 133—-140. https://doi.org/10.1145/2674005.2674991.

[92] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. 2018. Understanding PCIe Performance for End Host Networking. In *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*. 327—-341. https://doi.org/10.1145/3230543.3230560.

[93] NVIDIA. 2021. Bluefield-2 DPU. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-2-dpu.pdf. Accessed: 2021-04-16.

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

[95] Chris Nyberg, Tom Barclay, Zarka Cvetanovic, Jim Gray, and Dave Lomet. 1994. AlphaSort: A RISC Machine Sort. In *ACM SIGMOD International Conference on Management of Data*. 233—-242. https://doi.org/10.1145/191839.191884.

[96] Vladimir Andrei Olteanu, Felipe Huici, and Costin Raiciu. 2015. Lost in Network Address Translation: Lessons from Scaling the World's Simplest Middlebox. In *ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization (HotMiddlebox)*. 19—-24. https://doi.org/10.1145/2785989.2785994.

[97] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 361–378. https://www.usenix.org/conference/nsdi19/presentation/ousterhout.

[98] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank citation ranking: Bringing order to the web.* Technical Report. Stanford InfoLab.

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

[100] 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. 2013. Ananta: Cloud Scale Load Balancing. In *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*. 207—-218. https://doi.org/10.1145/2486001.2486026.

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

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

[103] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, Felipe Huici, and Giuseppe Siracusano. 2019. FlowBlaze: Stateful Packet Processing in Hardware. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 531–548. https://www.usenix.org/conference/nsdi19/presentation/pontarelli.

[104] Samira Pouyanfar, Yimin Yang, Shu-Ching Chen, Mei-Ling Shyu, and S. S. Iyengar. 2018. Multimedia Big Data Analytics: A Survey. *ACM Compututing Surveys (CSUR)* 51, 1 (Apr 2018), Article No. 10. https://doi.org/10.1145/3150226.

[105] Mia Primorac, Edouard Bugnion, and Katerina Argyraki. 2017. How to Measure the Killer Microsecond. In *Proceedings of the Workshop on Kernel-Bypass Networks*. 37—-42. https://doi.org/10.1145/3098583.3098590.

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

[107] Luigi Rizzo. 2012. Netmap: A Novel Framework for Fast Packet I/O. In *USENIX Annual Technical Conference (ATC)*. https://www.usenix.org/conference/atc12/technical-sessions/presentation/rizzo.

[108] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. 2015. Inside the Social Network's (Datacenter) Network. In *ACM SIGCOMM Conference on Applications Technologies Architecture and Protocols for Computer Communications*. 123—-137. https://doi.org/10.1145/2785956.2787472.

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

[110] Patrick Stuedi, Animesh Trivedi, and Bernard Metzler. 2012. Wimpy Nodes with 10GbE: Leveraging One-Sided Operations in Soft-RDMA to Boost Memcached. In *USENIX Annual Technical Conference (ATC)*. 347–353. https://www.usenix.org/conference/atc12/technical-sessions/presentation/stuedi.

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

[112] Shelby Thomas, Geoffrey M. Voelker, and George Porter. 2018. CacheCloud: Towards Speed-of-light Datacenter Communication. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*. https://www.usenix.org/conference/hotcloud18/presentation/thomas.

[113] Amin Tootoonchian, Aurojit Panda, Chang Lan, Melvin Walls, Katerina Argyraki, Sylvia Ratnasamy, and Scott Shenker. 2018. ResQ: Enabling SLOs in Network Function Virtualization. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 283—-297. https://www.usenix.org/conference/nsdi18/presentation/tootoonchian.

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

[115] Qing Want, Youyou Lu, and Jiwu Shu. 2022. Sherman: A Write-Optimized Distributed B+Tree Index on Disaggregated Memory. In *To appear in ACM*

SIGMOD International Conference on Management of Data. https://arxiv.org/abs/2112.07320.

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

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

[118] Zhipeng Zhao, Hugo Sadok, Nirav Atre, James C. Hoe, Vyas Sekar, and Justine Sherry. 2020. Achieving 100Gbps Intrusion Prevention on a Single Server. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*. 1083–1100. https://www.usenix.org/conference/osdi20/presentation/zhao-zhipeng.