SmartNS: Enabling Line-rate and Flexible Network Stack with SmartNIC

Anonymous Author(s)
Submission Id: 260

Abstract

As the gap between network and CPU speeds rapidly increases, the CPU-centric network stack proves inadequate due to excessive CPU and memory overheads. Though hardware-offloaded network stacks alleviate these issues, they suffer from limited flexibility in both control and data planes. It seems promising to offload network stacks to SmartNICs to provide high flexibility. However, naive offloading leads to low throughput due to the inherent architectural limitations of widespread off-path SmartNICs. Even simple operations on staged network traffic would overwhelm the limited SmartNIC memory bandwidth. To this end, we design SmartNS, a SmartNIC-centric network stack with software transport programmability and line-rate packet processing capabilities. To tackle the limitations of SmartNIC-induced challenges, we propose a header-only offloading TX path and an unlimited-working-set in-cache processing RX path to minimize memory traffic to fit the wimpy SmartNIC memory bandwidth. To fully utilize the SmartNIC computing resources, we propose a programmable offloading engine to enable cloud providers to offload customized tasks along with the network stack processing. We prototype SmartNS using the widespread Nvidia BlueField-3 SmartNIC, and implement RoCEv2 and Solar transport protocols by leveraging SmartNS's software programmability. SmartNS achieves 2.2× higher throughput than the microkernel-based baseline in block storage disaggregation and 1.3× higher throughput than the hardware-offloaded baseline in KVCache transfer.

1 Introduction

In the modern cloud, the demand for network speed grows quickly, with 200/400 Gigabit Ethernet (GbE) network interface controllers (NICs) widely deployed [53, 54] and 800 GbE expected in the near future [55]. Consequently, the network stack must deliver elevated processing capacity. Moreover, as deployments vary broadly (e.g., single/multi-tenant, lossless/lossy fabric [46, 58, 94]), and upper-layer applications expand from HPC and disaggregated storage [21, 30, 56] to GPU communication [18, 70], enhanced flexibility and programmability within network stack become indispensable.

CPU-centric network stacks, represented by monolithic kernel-based [88, 90, 97, 108] and microkernel-based [17, 22, 31, 35, 36, 52, 60, 65, 69] designs, offer certain programmability via high-level software language (i.e., C/C++). However, more portions of the CPU resources and memory bandwidth have to be occupied to handle increasing network bandwidth [6, 7] due to the slowing down of Moore's law. Besides, co-locating with other CPU workloads induces cache and memory interference, resulting in high tail latency [17, 65].

Hardware-offloaded network stacks, represented by Remote Direct Memory Access (RDMA) [26, 39] and TCP Offload Engine (TOE) [59, 80], offer a compelling path to achieve high throughput in the post-Moore's Law era. Unfortunately, fixed-function hardware cannot meet the various requirements of numerous different upper-layer applications [23, 51, 56, 67, 70, 99, 106] and rapid release cycles demanded by cloud environments [52, 56]. For example, RDMA NIC only supports limited transport protocols and is a black box for users, offering only limited "configurable" capabilities that are not truly programmable. Large cloud vendors collaborate with NIC vendors to integrate customized functions into next-gen NICs, which usually take several years. FPGA-based NIC designs can offer a certain programmability using hardware description languages such as Verilog [3, 24, 59, 84, 93, 94], but they still need a much longer development cycle and fail to meet the rapidly evolving demands of the cloud workloads [52, 56]. This leads to an interesting question: **How to build a line-rate, low-overhead network stack while guaranteeing software programmability to modern cloud providers?**

One promising solution is the SmartNIC-centric network stack, which allows offloading host workloads to the specially optimized SmartNIC processing units for better performance/cost efficiency. According to the location of the processing unit, SmartNIC is categorized into two types [50]: on-path and off-path. The former uses the dedicated processing framework and can only be programmed with low-level vendor-specific microcode, posing significant burdens on developers [77, 95]. The latter features an on-NIC Arm SoC [5, 61, 63] and Linux, and developers can seamlessly migrate existing programs. In this paper, we focus on off-path SmartNIC, considering that off-path architecture is the de facto SmartNIC architecture adopted by mainstream cloud vendors (e.g., Alibaba CIPU [56], AWS Nitro [78]). Although the off-path SmartNIC provides the modern cloud required programmability, building a line-rate network stack with off-path SmartNIC is non-trivial due to three identified unique challenges caused by the inherent SmartNIC architecture:

1. High Bandwidth Contention on the Arm-NIC Switch Link. Off-path SmartNIC features a NIC switch to connect the NIC, the Arm, and the host interface. When outgoing traffic and incoming traffic both travel through the Arm, the link bandwidth between the Arm and the NIC switch easily becomes the bottleneck. If each NIC switch endpoint

provides a 400 Gbps duplex bandwidth, a 400 Gbps outgoing traffic from host → Arm → NIC would fully saturate the Arm endpoint bandwidth, and thus leaves no link bandwidth for incoming traffic from NIC to achieve duplex line-rate.

2. Constrained Arm Memory Bandwidth. The network stack requires traffic to be temporarily staged in Arm memory, where both TX and RX paths incur twice the bandwidth occupation. Consequently, consuming a 400 Gbps traffic demands about 1600 Gbps memory bandwidth, far exceeding the bandwidth capability of modern SmartNICs [5, 28, 61, 63], which are commonly equipped with two DDR5 channels and provide about 440 Gbps memory bandwidth. Our empirical evaluation using "Echo Server" shows that this limitation can degrade the attainable throughput by 3.3×.

3. Long latency of Small Packet due to High PCIe Interconnect Latency. To invoke the network stack function, the Work Queue Entry (WQE) and Completion Queue Entry (CQE) are not directly passed to the NIC but detoured to the Arm, which introduces higher latency due to additional PCIe hop and is fatal for latency-sensitive applications. We test on the L2 Reflector application with 64B payload size, the Arm detour incurs an average 10.1μs latency, which is 2.2×/1.4× higher than the RDMA NIC/microkernel-based design.

To this end, we design and implement SmartNS, a SmartNIC-centric network stack with software transport programmability and line-rate packet processing capabilities. To address the above three challenges, SmartNS introduces: 1) a header-only offloading TX path that constructs the custom packet header on Arm and integrates the payload within NIC; 2) an unlimited-working-set in-cache processing RX path that receives packets into LLC, followed by cache self-invalidation after transferring the payload to the host destination; 3) a DMA-only notification pipe that leverages high-performance DMA engines to communicate between host and Arm; and 4) a programmable offloading engine that empowers cloud providers to offload specialized tasks like customize one-sided operations and network functions. In sum, SmartNS achieves line-rate throughput and offers cloud providers the required software transport programmability.

We prototype SmartNS on an off-the-shelf Nvidia Bluefield-3 SmartNIC with 400 GbE [63] connectivity and implemented various different network stacks. Benefitting from our novel design, SmartNS can sustain full-duplex line-rate and comparable single/multiple flow throughput with RDMA NIC. SmartNS achieves 2.2× higher IOPS than the microkernel-based baseline in disaggregated block storage [110] and 1.3× higher throughput than the hardware-offloaded baseline in KVCache transfer [71].

2 Background and Motivation

In modern data center, an ideal network stack should be capable of 1) providing high throughput and low latency; 2) minimizing the host overhead for stack processing (i.e., CPU occupation); 3) avoiding memory bandwidth contention with co-located applications; 4) providing high programmability

Table 1. Comparison of network stack approaches.

Solution	Throughput	Host Overhead	Memory Contention	Programmability
Monolithic kernel-based [90]	low	High	High	Medium
Microkernel-based [52]	Medium	Medium	High	High
Hardware-offloaded [54, 84]	High	Low	None	Low
Naïve SmartNS	Low	Low	None	High
SmartNS	High	Low	None	High

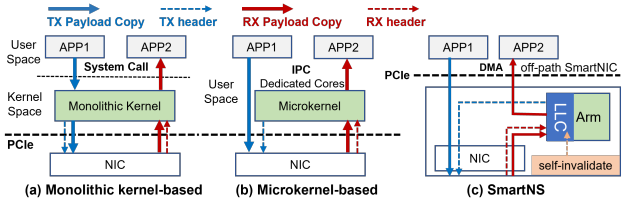


Figure 1. Comparison of different network stack designs.

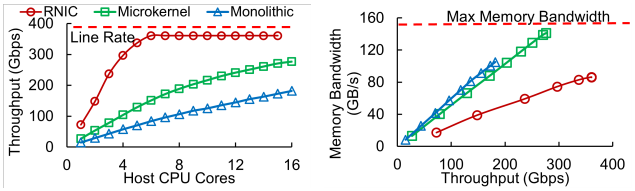
to allow developers to customize data plane policy (transport protocol and multi-path) and control plane policy (QoS, access control and congestion control). However, the existing designs fail to meet all the requirements as shown in Table 1.

2.1 CPU-centric Network Stack

2.1.1 Monolithic Kernel-based Network Stack. Figure 1a illustrates the data flow of a monolithic kernel-based network stack. In the TX path, the user application invokes a system call and switches to the kernel mode, where the network stack processes protocols and constructs packet headers. Then, it copies the user's payload into a pinned DMA-safe memory region and notifies the NIC to transmit. In the RX path, the NIC places received packets in a kernel-space reception queue and triggers an interrupt. After protocol processing, the network stack copies the payload to the user buffer and signals the user application. The system call overhead can incur a slowdown of up to 75%, as reported in prior research [60, 90, 97]. Besides, the high complexity of developing kernel code significantly harms the programmability of monolithic kernel-based network stacks.

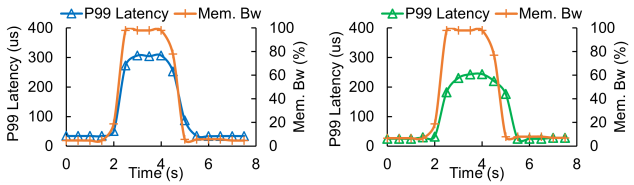
2.1.2 Microkernel-based Network Stack. To mitigate the aforementioned host CPU overhead and programming complexity, many systems adopt the microkernel-based network stack [17, 31, 35, 36, 52, 60, 65, 69] to replace costly system calls with lightweight IPC (Inter-Process Communication), as illustrated in Figure 1b. There are three main benefits. First, compared with system calls, the overhead of IPC in modern CPUs is much lower as it preserves the application cache locality [52]. Second, it allows applications to directly use a pinned DMA-safe memory region, avoiding the memcpy overhead in the TX path. Third, a microkernel-based network stack has much higher programmability, as developing and deploying userspace code is much easier.

2.1.3 Issues of CPU-centric Network Stack. As Moore's law is slowing down, the gap between network and CPU



(a) Network throughput **(b)** Host memory bandwidth

Figure 2. Comparison of different stacks achieved throughput and corresponding host memory bandwidth usage.



(a) Monolithic kernel-based system **(b)** Microkernel-based system

Figure 3. Memory-intensive application causes interference with the network stack.

speeds is rapidly increasing. With 200/400 GbE NICs already deployed in modern datacenters [21, 70, 78] and 800 GbE expected in the near future [55], both CPU-centric designs face two key issues.

I1: High host CPU overhead. The host CPU overhead mainly consists of payload copy, which spends more than 60% of CPU resources [6, 7]. Due to the limited Line Fill Buffer (LFB) per CPU core [91], it can achieve only about 7GB/s memcpy speed per core. Cai et al. [7] propose the decoupling data copy from application threads and using multi-threaded memcpy, but it still needs more than 15 cores to achieve the 400Gbps memcpy without protocol processing.

To illustrate the host CPU overhead, we construct an "Echo Server" using a monolithic kernel-based [97] and a microkernel-based [52] network stack, respectively, and compare with the RDMA NIC [33]. This echo server has 8 DDR5 channels with up to 160GB/s memory bandwidth. Both machines are equipped with a 400 GbE NVIDIA ConnectX-7 NIC [54], and the packet size is 2KB, each CPU core serves 8 connections with 64 TX depth.

Figure 2a illustrates the throughput under different host CPU cores. We observe that due to expensive system calls, the monolithic kernel exhibits approximately 42% lower throughput than the microkernel-based design, which achieves only around 39% per-CPU-core throughput compared with RDMA NIC, mainly due to the additional memcpy in RX path. We predict that this performance disparity would be widened as network speeds approach 800 Gbps in the near future.

I2: High host memory contention. The growth of network speed also places contention on the memory subsystem, which has been extensively explored in prior works [1, 89].

On the one hand, the extra memcpy adds significant memory pressure to the network stack. For example, only one extra memcpy in the RX path of the 400Gbps network needs at least 200GB/s memory bandwidth to achieve full-duplex line-rate throughput, which requires at least 8 DDR5-4800 memory channels (typical single-socket Intel servers support up to 8 channels [29, 87]). As shown in Figure 2b, both monolithic kernel and microkernel designs consume roughly 1.9× memory bandwidth compared with the RDMA NIC. The monolithic kernel almost fully saturates the available memory bandwidth when its throughput reaches 300Gbps.

On the other hand, CPU-centric network stacks suffer severe interference from co-located memory-intensive applications in multi-tenant environments. As the CPU memory controller is shared among all cores, a core would suffer high memory access latency when other cores issue lots of memory transactions, resulting in high tail latency [17]. To quantify this effect, we use one CPU core as an echo server and other cores running Memory Latency Check (MLC) [27] to simulate a memory-intensive garbage collection application. Figure 3 illustrates the P99 latency of the echo server, MLC starts at 2*s* and ends at 5*s*. We observe that the interference increases P99 latency by 10.5×/9.2× for monolithic kernel and microkernel-based designs, respectively. This indicates that CPU-centric network stacks suffer from high tail latency in the case of co-located memory-intensive applications.

2.2 Hardware-offloaded Network Stack

To address the three aforementioned issues, hardware offloading technologies, e.g., RDMA, are used to offload the entire network stack to the NIC hardware to achieve high throughput, low latency, and low host overhead [54, 75, 78, 84]. However, a hardwired network stack can not meet the various requirements of numerous different upper-layer applications (e.g., GPU communication, disaggregated storage, and high-performance computing). Specifically, their rigid architecture introduces programmability limitations.

I3: Inadequate data/control plane programmability. Commercial NICs only support a limited number of transports. For example, Nvidia ConnectX-7 [54] only supports RoCEv2 [26] and InfiniBand [25]. However, extensive innovations have been proposed to customize the transport layer to address specialized application requirements, like Google Snap [52] 1RMA [84] and Falcon [20], AWS SRD [78], Alibaba Solar [56, 110], Tesla TTPoE [72], and IRN [58]. Likewise, modern data centers impose diverse control plane requirements: GPU communication needs high-precision congestion control [41, 45, 92, 111] and multi-path routing [8, 43, 78]; VPC needs multi-tenancy performance isolation [23, 39, 51, 106] and access control [67]. However, commercial NICs are always treated as black boxes and offer only limited "configurable" capabilities that are not truly programmable. It takes years to collaborate with vendors to integrate custom methods into the next-gen ASIC NIC.

To meet the need for programmability, researchers propose several FPGA-based NIC designs [80, 84, 93, 94], with

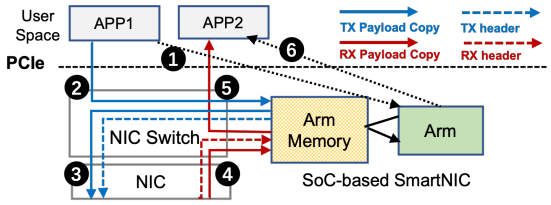


Figure 4. High-level system architecture and TX/RX data flow of a naïve SmartNIC-centric network stack.

which users can update the on-NIC transport module using hardware description languages (HDL) such as Verilog. However, developing using HDL has a much longer development cycle, and fails to meet the rapidly evolving demands of the cloud workloads, which could update weekly or monthly [52, 56].

2.3 Naïve SmartNIC-centric Network Stack

Modern off-path SmartNICs(e.g., NVIDIA BlueField [61, 63], Broadcom Stingray [5], Intel IPU [28]) feature an on-NIC programmable SoC (usually consisting of Arm), and thus provide an opportunity to offload the network stack to the NIC while preserving software programmability.

Figure 4 sketches such a naïve SmartNIC-centric network stack design. In the TX path, the user application notifies Arm to send the packet (❶), then Arm fetches the payload into its memory (❷), prepares the packet header, and encapsulates the header and payload into the corresponding packet and send it out (❸). In the RX path, the incoming packet is first stored in the Arm memory (❹). Then the Arm parses the packet header, processes the protocol, and delivers the payload to the corresponding host buffer (❺), at last the Arm signals the host CPU for notification (❻). This naïve design allows implementing the entire transport layer in the Arm, thus providing software programmability while minimizing host CPU consumption and host memory bandwidth. However, it introduces three severe issues.

C1: High bandwidth contention on the Arm-NIC switch link. An off-path SmartNIC features a NIC switch to connect the NIC, the Arm, and the host interface, as shown in Figure 4. The Arm sits outside the host/NIC path, and that is why it is called off-path SmartNIC. In an off-path architecture, the link bandwidth between the Arm and NIC switch becomes the bottleneck when outgoing traffic and incoming traffic both travel through the Arm. If each NIC switch endpoint provides a 400 Gbps duplex bandwidth, a 400 Gbps outgoing traffic from host → Arm → NIC would fully saturate the link bandwidth of the Arm endpoint, and there would be no link bandwidth for incoming traffic.

C2: Constrained Arm memory bandwidth. Compared to the host CPU, off-path SmartNICs offer comparable general-purpose computing capabilities but feature a significantly weaker memory subsystem [10]. For instance, the BF3 SmartNIC features two 5600MT/s DDR5 channels to deliver the theoretical 716.8Gbps memory bandwidth, and its achievable mixed read-write bandwidth is approximately 440Gbps [10]. However, the network stack requires traffic to be temporarily staged in Arm DRAM, resulting in the payload going through device memory four times, as illustrated in Figure 4. Thus, 400Gbps network throughput requires about 1600Gbps memory bandwidth—a requirement that far exceeds the capability of BF3. BlueField-2, Intel IPU, and Broadcom PS1100R also have similar memory bandwidth shortages.

To illustrate this, we implement a naïve SmartNIC-centric network stack in Figure 4 and conduct the "Echo Server" experiments again (§2.1). We find that it achieves only 120 Gbps throughput (30% of the network link throughput). Through the memory monitor on the Arm, we identify that the primary culprit behind its performance degradation is memory bandwidth exhaustion.

C3: Long latency of small packet due to high PCIe interconnect latency. As shown in Figure 4, to invoke the network stack function, the WQE and CQE are not directly passed to the NIC but detoured to the Arm, which introduces extra PCIe interconnect latency (taking sub-microseconds). Although throughput-sensitive applications are unaffected by the additional latency, it still leads to performance degradation for latency-sensitive applications, particularly when the payload size falls within one MTU. We test on L2 Reflector application and use 64B payload size, the naïve SmartNIC-centric network stack demonstrated an unexpectedly high latency of roughly 10.1μs , which is 2.2× higher than the RDMA NIC and 1.4× higher than the microkernel-based network stack.

3 Design of SmartNS

3.1 Overview

To address the above three challenges, we propose SmartNS, a SmartNIC-centric network stack with software transport programmability and line-rate packet processing capabilities. Figure 5 presents the architecture overview of SmartNS. The main network stack running on the on-NIC Arm cores as a separate user process consists of 1) a header-only offloading TX path (§ 3.2) that constructs custom packet headers on the Arm and directly integrates the host payload within the NIC; 2) a unlimited-working-set in-cache processing RX path (§ 3.3) that receives packets into the LLC, followed by cache self-invalidation after transferring the payload to the host destination; 3) a DMA-only notification pipe (§ 3.4) that solely leverages high performance on-NIC DMA engines to notify the host or Arm core; and 4) a programmable offloading engine (§ 3.5) that allows cloud providers to offload specialized functions like customized one-sided operations and network functions. In contrast to prior works that focus on specific protocols, SmartNS is architected as a general network stack framework to allow for easy implementation of various transport protocols and control plane mechanisms, including QoS policies and congestion control algorithms. In

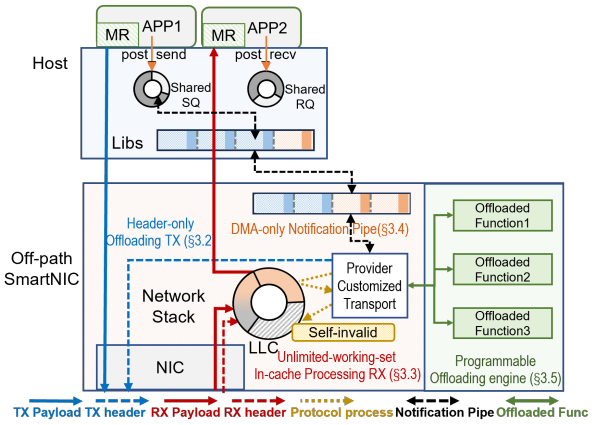


Figure 5. Architecture overview of SmartNS.

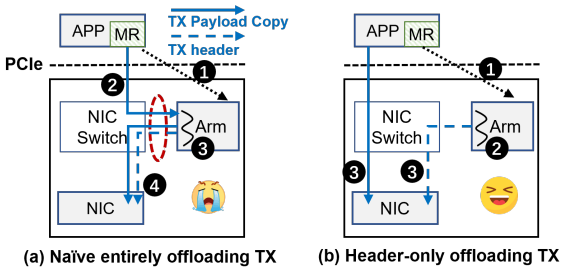


Figure 6. Comparison of TX path strategies.

this paper, we have implemented two demonstrative transport protocols: RoCEv2 [26] and Solar [56], along with the widely adopted DCQCN [111] as the CCA.

On the host side, SmartNS provides end users 1) a user-space runtime library linked to each application; and 2) a kernel module that registers the entire system as a NIC device and communicates with the network stack. This user library forwards control verbs (like `create_qp` and `modify_qp`) to the kernel module and directly passes data verbs (like `post_send` and `post_recv`) to Arm through a DMA-only notification pipe. Since all operations are offloaded to the SmartNIC, this library introduces negligible overhead for the host CPU. By registering the entire system as a NIC device in a kernel module, users can enjoy out-of-the-box IBV verbs compatibility [73], which is important for applying RDMA-aware optimizations [12, 96, 97, 112]. Note that IBV Verbs not only support RDMA RC/UD mode but also support Raw packet mode, which can send/receive raw Ethernet packets. SmartNS relies on Raw packet mode to customize the packet header and support various transport protocols.

3.2 Header-only Offloading TX Path

To address the challenge #1 (**C1**), we propose the Header-only offloading TX path. First, we discuss why the following general and widely used TX path design is inadequate for achieving high throughput.

Option: Naïve entirely offloading TX path. Figure 6a depicts the process of naïve entirely offloading TX strategy, which has been widely adopted in prior studies [30, 38, 40]. When the user application initiates packet transmission (❶), the Arm utilizes intra-node DMA/RDMA [10, 95] to read the payload into the Arm memory (❷). The Arm subsequently constructs the corresponding packet header and merges it with the payload into a contiguous buffer (❸). At last, the Arm transmits the assembled packet to the network (❹). This approach benefits from standard IBV verbs, making it straightforward to implement. However, this approach causes significant Arm memory subsystem pressure and saturates the Arm endpoint's full-duplex link bandwidth: although it achieves line rate for TX traffic, there would be no link bandwidth for RX traffic (discussed in §2.3).

Our approach: Header-only offloading TX path. To address the limitations of the above design, we decouple the packet header and payload across from Arm memory and host memory, aiming to reduce the Arm endpoint's link occupation and minimize Arm memory bandwidth pressure. Our key insight is that the network stack typically neither accesses nor modifies the packet payload, thus, we can construct custom header on Arm and allow the payload be fetched directly by the NIC without any additional data movement, while leveraging NIC hardware for residual operations such as CRC computation, IP Checksum, and AES encryptio. Figure 6b illustrates the process. After the Arm prepares the packet header (❷), both the header and the payload are directly retrieved by the NIC (❸). This simple yet effective design offers three key advantages: (1) eliminates the Arm-NIC switch link congestion, (2) reduces bandwidth pressure on the Arm memory subsystem, and (3) decreases the programming complexity of the network stack.

DMA/RDMA operations require the payload buffer to reside in a pre-registered memory region, each memory region associated with a specific context. Since the context includes hardware resources and can't be shared between Arm and host CPU, Arm can't directly use payload information pertaining to the host's memory region to construct WQE. To expose host payload buffer information to the network stack, we introduce the **shadow memory region**. Each time the user application registers a memory region, the kernel module informs the (host VA, Size) to the Arm. The Arm then selects an unused virtual address range (Arm VA), instructs the NIC to establish a hardware mapping between the host VA and the Arm VA, designates the (Arm VA, Size) as the shadow memory region, and finally returns the Arm VA to the user library. Importantly, this Arm VA is not mapped to any physical address and cannot be directly accessed via load/store instructions, thereby introducing no actual memory overhead. As shown in Figure 7a, when the user initiates packet transmission, the library automatically translates the host VA to Arm VA (❶). Subsequently, the Arm utilizes the Arm VA in the shadow memory region to construct the WQE and transmit the packet (❷), thus, NIC hardware resolves the corresponding host VA through that pre-established mapping (❸), fetches the payload directly from host memory,

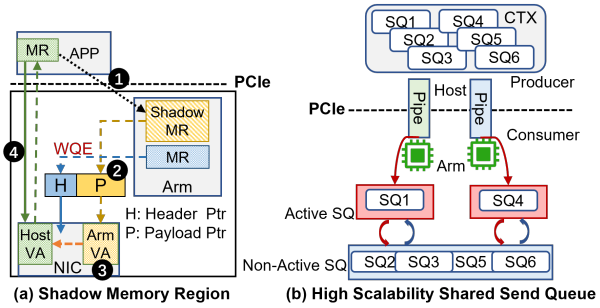


Figure 7. Architecture of Shadow Memory Region and High Scalability Shared Send Queue.

and integrate the custom packet headers generated on Arm (❹). This header-only offloading TX path ensures zero-copy while remaining transparent to the user application.

Another critical concern is the scalability of our software-based TX mechanism. Assuming each SQ is assigned a dedicated notification pipe, the overhead from software looping is negligible when the number of SQs is relatively small ($\leq$ 100). However, as the number of SQs increases, this approach becomes less efficient. The looping time for 2000 SQs can be up to 50 µs [88]. To achieve low overhead and high scalability, we propose the **high-scalability shared send queue**. As shown in Figure 7b, instead of allocating one pipe per SQ, we assign N pipes per context, where N equals the number of Arm cores. Each Arm core is responsible for polling a dedicated pipe within its assigned context. When a new SQ is created in a given context, the driver selects an underloaded Arm core and utilizes its corresponding pipe for SQE transmission. Given that RDMA-based systems typically create a limited number of contexts [96, 101, 112], each Arm core is required to poll only a small number of pipes. This sharing mechanism doesn't break the performance isolation of resources because hardware resources are allocated at the context level, and we only share notification pipes within the same context. Furthermore, to efficiently manage unfinished requests, we maintain an active SQ table that tracks unfinished operations for each Arm core. An SQ is added to the active SQ table when the Arm core receives an SQE related to that SQ, and this SQ will be transitioned to the non-active SQ pool when all the requests are finished.

3.3 Unlimited-working-set In-Cache Processing RX Path

To address challenge #2 (**C2**), we propose unlimited-working-set in-cache processing RX path. In the following, we discuss the challenges, followed by its design.

Challenges. Although TX path optimizations can greatly reduce the precious Arm memory bandwidth incurred by the TX traffic because TX payload does not traverse through the Arm memory, the limited Arm memory bandwidth is still insufficient to support line-rate RX traffic if all RX traffic passes through the Arm memory. Take BF3 as an example, we observe that the Arm memory bandwidth is exhausted when the RX traffic bandwidth is only 60% of the network line rate in the previous "Echo Server" experiment.

Direct Cache Access (DCA) allows PCIe devices such as NICs to write incoming packets to the LLC directly, and thus is widely used in network applications to relieve memory bandwidth problems [2, 13–16, 44, 100]. The Arm on the SmartNIC also supports the DCA mechanism and can use the whole LLC space to serve the DCA mechanism; thus, it can potentially address the above-mentioned memory bandwidth challenge. However, we find that the benefits of DCA are highly constrained: **the working set size must be smaller than the cache size.**

Cache-fitting working set. Figure 8a demonstrates how a network stack receives packets when the working set (i.e., receive buffer) can fit in the cache. We observe that every address in the working set is cached, and an incoming packet can be directly written into the cache without cache eviction. In this case, the DCA mechanism can well address the limited Arm memory bandwidth challenge.

Cache-exceeding working set. However, in a network stack RX path, the working set (i.e., receive buffer) size is usually very large, which far exceeds the cache size. Ideally, the working set size only needs to be larger than the product of the network bandwidth and the average packet processing time. However, packet processing time varies significantly, and the packets may arrive in a burst manner (i.e., a sudden spike in packet arrivals) [1, 16]. To avoid dropping packets, the network stack RX path must ensure that the working set size can absorb the burst. Consequently, multi-hundred-gigabit network stacks typically employ more than 512 descriptors per RX queue [16, 68, 96]. Assuming that there is one RX queue per Arm core and the maximum packet size is 4 KB, the total working set size would be $512*16*4KB = 32MB$ in the 16-core BF3 SmartNIC. Considering that the 16 MB cache of BF3 is also shared with the TX path and other SmartNIC computations, the RX working set size would far exceed the available cache size. Figure 8b demonstrates how a network stack receives packets when the working set exceeds the cache size. We observe that an incoming packet would always incur a cache eviction, thus, the system performance would be bottlenecked by the Arm memory bandwidth.

Unlimited-working-set in-cache processing RX. To this end, we propose unlimited-working-set in-cache processing RX, which avoids cache eviction even if the RX working set size far exceeds the cache size. Our key idea is based on the observation that the packet content is no longer needed by Arm after the network stack RX processes the packet header and forwards the packet to the host. As such, there is no need to actually evict the processed packets from the cache to the Arm memory. Therefore, after the Arm core processes a packet and forwards the packet to the host, the Arm core would explicitly invalidate the cachelines of the packet buffer

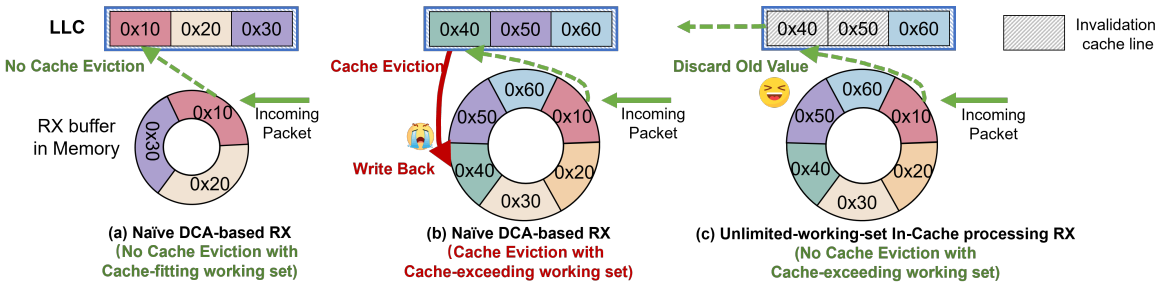


Figure 8. Comparison of RX path strategies that motivate the design of unlimited-working-set in-cache processing RX path.

by invoking the BF3-provided API [64], which would set the related cachelines "invalidate" flag[1].

Generality. SmartNS relies on DCA and self cache-invalidation mechanism to support unlimited-working-set and high-performance RX path. Although our current implementation relies on Nvidia's domain-specific APIs, Intel DDIO (the DCA implementation in x86 processors) has been extensively studied in prior works [1, 14, 16, 44, 100], and some RISC-V processors already support both DCA and self cache-invalidation [2, 10]. We therefore believe our design can be applied to future platforms and NICs.

Figure 8c demonstrates how SmartNS receives packets when the working set exceeds the cache size. As the processed packet is explicitly invalidated, there are always available cachelines for incoming packets, and cacheline stale contents are discarded rather than written back, which allows the incoming packet to overwrite the cacheline directly and avoid unnecessary write back. In the rare case when a packet arrival burst occurs or the processing rate slows down, the incoming packets may find no invalidated cacheline to fill in and incur an eviction. However, the packet would not be dropped, and the eviction would disappear after the burst.

Out-of-order packet processing. Many transport protocols employ Go-Back-N or Selective Repeat (SR) to handle out-of-order packets. For Go-Back-N, SmartNS simply self-invalidates the received packet buffer and sends a NACK signal to the client. For SR, SmartNS maintains a per-connection reorder buffer in Arm memory. When the Arm processor receives an out-of-order packet, it writes back the packet buffer from Arm LLC to Arm memory and sends SACK to the client, followed by a self-invalidate operation. Thus, SmartNS can flexibly support different out-of-order handling policies while remaining compatible with unlimited working-set in-cache processing.

Towards supporting 800 Gbps networks. With unlimited-working-set in-cache processing RX, the required cache size only needs to be larger than the product of the network bandwidth and the average packet processing time. For a 400 Gbps network and a 10μs average network stack processing time, the required cache size is only 500 KB, which is far smaller than the BF3 Arm LLC size (32× smaller). Even when the network bandwidth scales to 800 Gbps and the processing latency is doubled, our approach only needs $800Gbps \times 20us = 2MB$ LLC. Therefore, it only requires upgrading Arm LLC bandwidth from the current 1200 Gbps to 1600 Gbps, which is relatively easy for the next generation Arm to accomplish. As such, we believe **SmartNS RX path can also fit the next-generation high-bandwidth network**.

3.4 DMA-only Notification Pipe

Common RNICs utilize *WQE-by-MMIO* and *Doorbell* mechanisms to update SQ/RQ/CQ pointers with the RNIC [32, 74]. In the first case, the WQEs are transferred via 64B write-combined MMIOs; in the second case, the host updates the NIC Doorbell, followed by the NIC fetching WQEs using one or more DMAs. Although these mechanisms perform well for common RNIC, off-path SmartNICs adopt an emulated MMIO interface between the Arm and the host CPU [61, 63] and suffer from low MMIO throughput. Our observations reveal that BF3 achieves fewer than 1K/s MMIO write rate, which is markedly insufficient for the demands of 400Gbps networks. Moreover, exclusively using the doorbell mechanism incurs latency due to an extra PCIe round-trip.

Therefore, we propose a DMA-only notification pipe that relies solely on a high-performance DMA engine for communication and coordination. To maximize efficiency, the design enforces a single producer and a single consumer, ensuring lockless access, and each element is aligned to the cache line size. Each element contains a dedicated 1-bit flag signal to indicate its validity. The producer sets this flag in the producer buffer and initiates the DMA engine, while the consumer continuously polls the next element in the consumer buffer until its flag becomes valid. Upon reaching the buffer's end, the flag toggles to indicate wrap-around. Since the producer typically batches multiple elements per DMA transfer, cache contention is avoided [76]. For SQ/RQ, the user application is the producer and the ARM is the consumer. To further realize memory alignment and keep cache locality, SQ elements (64B) are transferred immediately, whereas RQ entries are grouped in batches of four (4 × 16B). For the CQ element (64B), roles are reversed: the Arm produces entries, and the user application consumes them. But it involves a question about how the ARM tracks the user application's progress in handling CQE. To address this, we implement a consumer

[1]We align each packet buffer to the cacheline granularity (64B) to ensure that the invalidation would not affect other packets.

counter located in the user library: each time the consumer processes an element, it automatically increases the counter, and the producer periodically reads it via one DMA read after every n elements (customized by developers).

Latency-sensitive flow optimization. We observe that latency-sensitive applications typically have payload sizes limited to a single MTU and employ two-sided operations. To address challenge #3 (**C3**), we introduce a low-latency QP optimized for MTU-sized, two-sided operations, comprising two complementary mechanisms: In the TX path, we extend the SQE to carry inline payloads, thereby eliminating one PCIe round-trip. Payloads are delivered to the Arm via our high-performance notification pipe followed by inline send [32]. In the RX path, we allow the NIC to directly place the payload section in the application buffer while routing the header section to the Arm. The application polls the first byte of the receive buffer to detect incoming packets. In this case, the transport protocol is still processed on the SmartNIC, while the application should detect and handle out-of-order and duplicate packets at the user level.

3.5 Programmable Offloading Engine

In addition to the network stack, exploiting the spare SmartNIC computing resources to mitigate the host CPU occupation is a trendy way to lower the data center tax [37, 49, 50]. However, prior works [10, 34, 83, 102, 103] require ad-hoc design and deployment tailored to specific scenarios and applications. SmartNS proposes a programmable offloading engine, allowing easy offloading of various applications. The engine runs on top of the network stack transport and provides a rich set of programming APIs to enable cloud providers to offload the application logic easily. In short, a programmable offloading engine allows cloud providers to register an unused transport opcode and hook up with a customized function that runs on the spare SmartNIC Arm cores. An incoming packet with a specific opcode would trigger the execution of the hooked-up function after the packet has been processed by the transport. This offloading engine executes application-layer functions on dedicated Arm cores isolated from the network stack and can invoke lookaside accelerations on the SmartNIC to provide computing intensity. Cloud providers configure the number of Arm cores allocated to this engine at SmartNS startup.

This offloading engine provides a rich set of programming API (Table 2) to help cloud providers deploy their offloading tasks. As such, cloud providers can focus on task logic instead of dealing with low-level network stack details. We take a simple batched RDMA READ as an example and show how to use these APIs to implement it; the example code is provided in supplementary material. The programmer first chooses an user-defined opcode and uses `register_opcode` to register a batched RDMA READ handler function. Before establishing the QP connection, the programmer invokes `register_dma_region` to register a host memory region for subsequent DMA accesses. Upon receiving a corresponding request with the batched RDMA READ opcode, that registered handler function is invoked and executed as a user-space coroutine. In this coroutine, `alloc_resp` is called to allocate a pinned Arm memory for the response packet. The programmer then invokes `submit_dma` to execute the DMA operation, and all tasks are enqueued into a task pool and executed asynchronously via coroutines; once the DMA transfer completes, `wait_dma_finish` returns control and execution resumes. Finally, the programmer uses `submit_resp` to deliver the response packet back to the client.

Table 2. Programmable offloading engines APIs.

register_opcode(opcode, qp, func)
Registering the target opcode to the network stack, this function will be invoked when a target packet is received.
register_dma_region(host_addr, size)
Registering the host memory on the Arm for following DMA operations.
alloc_resp(context, size)
Allocating *size* bytes on pinned Arm memory for response packet, and return the address point.
submit_dma(context, op, host_addr, arm_addr, size)
Submit a DMA operation to the associated address, op can be READ or WRITE, return an ID for trace.
wait_dma_finish(context, dma_id)
Wait the dma_id corresponding dma operation to finish.
submit_resp(context, addr, size)
Submit the response packet with the specific address.

4 Implementation

We build a fully functional prototype of SmartNS using Nvidia BlueField-3 SmartNIC [63] with a PCIe5.0×16 interface and 2×200 Gbps Ethernet ports. SmartNS's implementation consists of the core network stack running as a separate user process on the Arm processor, the kernel modules running in the host kernel, and the user-space libraries linking with user applications. The core network stack is implemented in 6,890 lines of C++20 code, the kernel module in 1,350 lines of C98 code, and the user-space libraries in 3,012 lines of C++20 code. Moreover, we introduced several modifications (500 lines of code) to the mlx5 driver on the Arm to enable more efficient DMA and cache operation interfaces, while **keeping the host mlx5 driver unchanged**. Our entire system runs in an unmodified Linux environment. We left more implementation details in supplementary material.

5 Evalution

Our evaluations aim to answer the following questions:

- How does the performance of SmartNS compare to other network stacks (§5.2)?
- How effective is the header-only offloading TX path (§5.3)?

- How effective is the unlimited-working-set in-cache processing RX path (§5.4)?
- How effective is the DMA-only notification pipe (§5.5)?
- How effective is programmable offloading engine (§5.6)?
- How much performance acceleration can SmartNS achieve for block storage and KVCache transfer workloads (§5.7)?

5.1 Experimental Setup

Hardware Testbed. Our hardware testbed consists of two servers, each having two 16-core Intel Xeon Gold 6426Y running at 2.5GHz, 512 GiB (16x32 GiB) 4800 MHz DDR5 memory, and a 37.5 MiB LLC. Each server is equipped with an Nvidia BlueField-3 B3220 400GbE NIC and connected back-to-back using two 200GbE QSFP56 cables.

Snap Baseline. We implement the baseline "Snap" that runs the network stack as a separate user program and dedicated CPU cores. As a representative of microkernel-based network stacks, Snap leverages DPDK [11] to achieve high throughput and adopts a simple go-back-N mechanism for packet loss recovery. Since Snap [52] is not open-sourced, we construct our network stack based on the descriptions provided in the paper while minimizing any unnecessary CPU overhead.

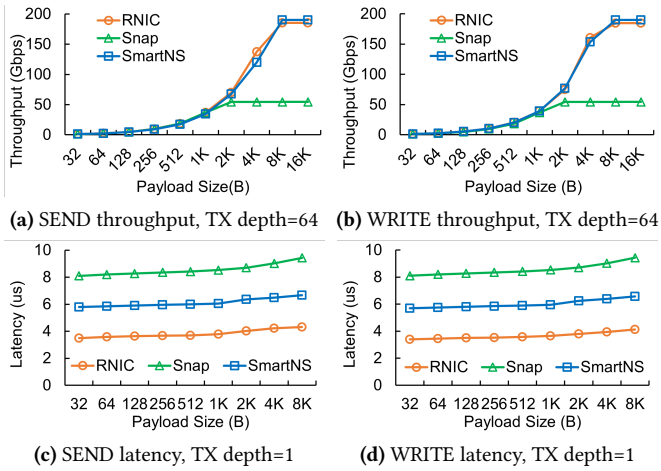
RDMA NIC Baseline. We implement the baseline "RNIC" that directly uses the hardware-offloaded RDMA stack provided by the state-of-the-art RNIC Nvidia ConnectX-7 [54], leveraging RC mode and RoCEv2 protocol to achieve the highest performance. We use dedicated busy-looping CPU cores to execute IBV verbs like `post_send` and `poll_cq`.

Solar-CPU Baseline. We implement the Solar [56] transport protocol on dedicated host CPU cores, strictly following the specifications outlined in the paper. Solar is the storage network stack for Alibaba Cloud's EBS service and has been deployed on a large scale. Due to the current lack of commodity RNIC support for the Solar protocol, we deploy it on the host CPU and leverage CRC hardware offload along with DSA engines to achieve optimal performance.

5.2 Comparison with other Network Stacks

In this section, we focus on two basic performance metrics: throughput and latency. We use the LibR benchmark tool [9], which is similar to perftest [19] but has finer-grained control. These tools can run on SmartNS without any modification, and the dedicated CPU cores assigned to the network stack remain in a spin state to maximize performance.

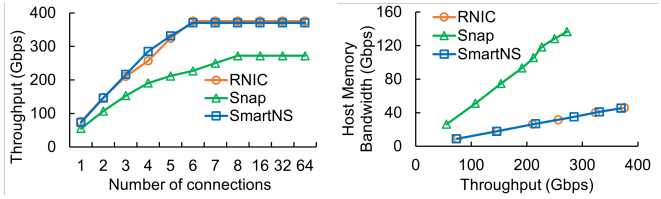
Figure 9 illustrates the single connection throughput and latency of the SEND/WRITE operations across various network stacks. We set TX depth to 64 for the throughput test and 1 for the latency test. We have two observations. First, SmartNS achieves comparable throughput to RNIC and up to 3.5× higher throughput than Snap in both SEND and WRITE tests. This is because SmartNS offloads massive stack workloads to NIC hardware similar to RNIC, but Snap relies on the host CPU to execute, constraining single-connection throughput. Second, SmartNS exhibits 1.5× higher latency



(a) SEND throughput, TX depth=64 **(b)** WRITE throughput, TX depth=64

(c) SEND latency, TX depth=1 **(d)** WRITE latency, TX depth=1

Figure 9. RDMA SEND/WRITE throughput and latency between a pair of connections on different hosts.



(a) Aggreagate WRITE throughput **(b)** Host memory bandwidth occupation

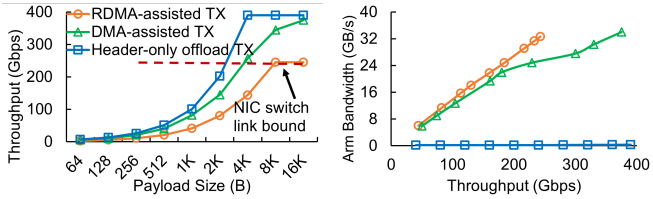
Figure 10. Aggreagate RDMA WRITE throughput of multiple connections and corresponding host memory bandwidth.

than RNIC but still 1.4× lower than Snap, because WQE and CQE in SmartNS must cross the Arm-NIC switch link and suffer PCIe interconnect latency relative to RNIC. However, the proximity of the Arm to the NIC enables SmartNS to reduce network latency compared to Snap.

Figure 10 shows the throughput of WRITE operations with multiple connections, and the corresponding host memory bandwidth across various network stacks, where the TX depth of each connection is set to 64 and payload size is set to 2KB. We observe that SmartNS demonstrates nearly linear scaling with the increase in connections, reaching line rate with more than 5 connections, comparable to RNIC and 1.4× higher than Snap. SmartNS also keeps minimal host memory bandwidth occupation and reduces the 2.7× bandwidth compared with Snap. This efficiency is attributed to its SmartNIC-based packet handling and zero-overhead on the host CPU.

5.3 Header-only Offloading TX Path

In this section, we examine the performance of our header-only offloading TX path. As the baseline, we implement the naïve RDMA-assisted entirely offloading TX and DMA-assisted entirely offloading TX approaches discussed in (§3.2). Moreover, since BF3 has two ports and each connection only uses one port, we use two connections and set TX depth to 64, finally we collect and count the aggregated bandwidth.



(a) WRITE throughput, TX depth=64 **(b)** Arm memory bandwidth occupation

Figure 11. Comparison of RDMA WRITE throughput under 2 connections across three TX Path choices and corresponding Arm memory bandwidth.

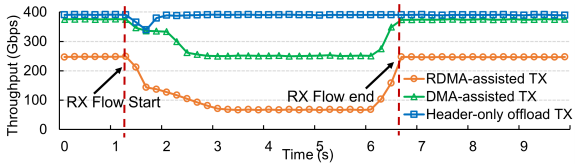


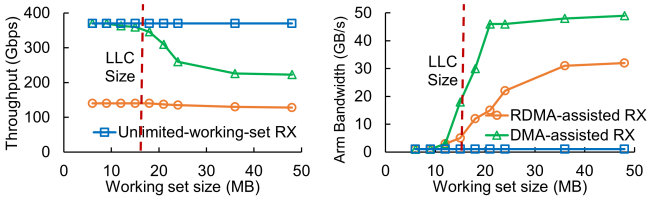
Figure 12. Using 8 connections and 2KB RDMA WRITE payload to achieve each TX design upper limit, and insert a 400Gbps RX flow at the 1 second that lasts 5 seconds.

Figure 11a shows the different TX approaches achieved throughput, and Figure 11b illustrates the corresponding Arm memory bandwidth occupation. We have three observations. First, when the payload size is below 8KB, header-only offloading TX achieves 2.7× and 1.5× higher throughput, because this mechanism only needs to construct the packet header and achieve zero copy, and thus significantly reduces the Arm core and memory subsystem pressure. Second, header-only offloading TX and DMA-assisted entirely offloading TX reach line rate, whereas RDMA-assisted entirely offloading TX is constrained by Arm-NIC switch link utilization, as discussed in Section (§3.2). Third, header-only offloading TX maintains Arm memory usage below 0.5GB/s, independent of network throughput and 70× lower than DMA-assisted TX. This efficiency is achieved by the header-only offloading TX bypassing the packet payload storage.

To further illustrate the benefits, we use 8 connections and 2KB payload to repeat the above experiment. Once throughput stabilized at its maximum, we insert a 400 Gbps RX flow on different cores from the server Arm core to the client Arm core, running for 5 seconds. Figure 12 shows the aggregated TX throughput achieved by the various approaches. Notably, our approach maintains line rate despite the RX flow, whereas the other two experience 72% and 35% throughput reduction. This is because our mechanism avoids contention on the Arm-NIC switch link with the RX flow.

5.4 Unlimited-working-set In-Cache Processing RX Path

In this section, we evaluate the effectiveness of the unlimited-working-set in-cache processing RX path. We use 12 Arm cores on both client and server, each core handles a single



(a) WRITE throughput, 8KB payload **(b)** Arm memory bandwidth occupation

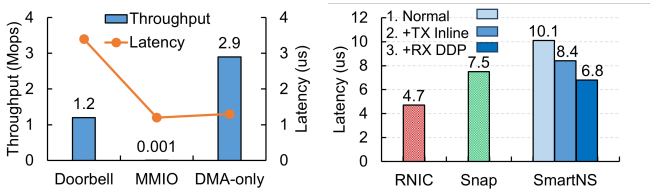
Figure 13. Comparison of 8KB RDMA WRITE throughput under different working set sizes across three RX path choices and corresponding Arm memory bandwidth.

connection. We equip each RX queue with 8KB queue element to receive jumbo packets. On the server, we vary the number of RX queue elements to emulate different working set sizes. The client sends 8KB RDMA WRITE requests, which are processed by the server's Arm cores, and then the payload is transferred to the host buffer by inter-node RDMA/DMA. Similar to the TX path, we compare three strategies: (1) using naïve intra-node RDMA for transferring payload from Arm to Host, (2) using naïve DMA for transferring, and (3) using unlimited-working-set in-cache processing RX.

Figure 13a shows the different RX working set sizes achieved throughput, while Figure 13b illustrates the corresponding Arm memory bandwidth. We have two observations. First, the unlimited-working-set mechanism maintains line-rate throughput even as the working set far exceeds LLC size; specifically, SmartNS achieves 1.6×/2.9× higher throughput than naïve DMA/RDMA transfer when the working set size is above 48MB, primarily due to a significant reduction in pressure on the Arm memory subsystem. Notably, even under 48MB working set size, each RX queue only contains fewer than 512 elements, which is relatively small. Second, the unlimited-working-set mechanism in-cache processing RX introduces less than 0.8GB/s of memory bandwidth usage even when throughput reaches line rate, demonstrating that nearly all operations (receive, process, and transfer) are processed in cache, without incurring additional memory bandwidth overhead. Third, the naïve RDMA/DMA exhibits substantial Arm bandwidth utilization once the working set size exceeds the LLC, indicating a pronounced leaky DMA problem effect that limits achievable throughput.

5.5 DMA-only Notification Pipe

In this section, we compare *Doorbell*, *WQE-by-MMIO*, and DMA-only notification mechanisms for WQE submission latency and achieved throughput. Here, we define latency as the elapsed time between host submission of a 64-byte WQE and its receipt by the Arm; both tests employ one host CPU and one Arm core. Figure 14a shows the throughput and latency of different notify mechanisms. We have two observations. First, DMA-only notification pipe achieves comparable latency with MMIO, and 2.6× lower than Doorbell, this is



(a) Throughput and latency of different notify mechanisms

(b) L2 reflector latency for different network stack

Figure 14. Performance of DMA-only notification pipe and low-latency QP.

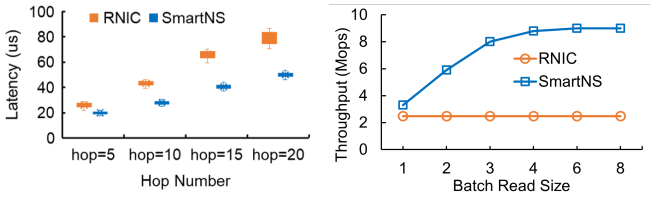
mainly because it omits the additional PCIe round-trip. Second, it delivers the highest throughput, which is 2.4× higher than the Doorbell mechanism; this is also caused by the extra PCIe round trip. Surprisingly, the MMIO mechanism exhibits such low throughput, less than 1K/s. This is because most off-path SmartNICs suffer limited capabilities emulated MMIO interface, the Arm must communicate with NIC firmware to get the content of incoming requests, which is very costly.

We also evaluate the latency of different network stacks via an L2-reflector application. The client sends a 64-byte packet to the server, and the server swaps the source and destination MAC addresses of each packet and returns it to the client. As shown in Figure 14b, unoptimized SmartNS suffers 2.2×/1.4× higher latency than RNIC and Snap, respectively. However, with our TX inline and RX direct data placement optimization, SmartNS can achieve 1.11× lower latency than Snap. Although it is still approximately 2μs slower than RNIC, this margin is modest. Consequently, developers can freely choose SmartNS for its programmability or RNIC when the absolute lowest latency is required.

5.6 Programmable Offloading Engine

In this section, we evaluate the performance advantage of two programmable offloading functions. First is *linked list traversal*; we consider a short linked list where each element contains an 8-bytes unique key and 64-bytes value. To locate a target key, the list is traversed starting from the head until a match is found, then returns the value pointed by the value pointer to the client. While RNIC performs the traversal on the client side, SmartNS executes the traversal within the programmable offloading engine using lightweight intra-node DMA operations instead of expensive inter-node RDMA operations. Second is *batched RDMA READ*. As the baseline, RNIC sends a series of 64-bytes RDMA READ packets. In our design, the client-side network stack aggregates multiple target addresses into one consolidated request, after which the server-side offloading engine issues concurrent DMA transfers to fetch all requested values in parallel and returns the aggregated data in a single response.

Figure 15a shows the traversal latency for varying hop counts, SmartNS achieves 1.7× reduction in latency compared to RNIC, owing to the lightweight intra-node DMA operations, which incur lower latency than inter-node RDMA



(a) Linked list traversal

(b) Batched RDMA READ

Figure 15. Programmable offloading functions performance.

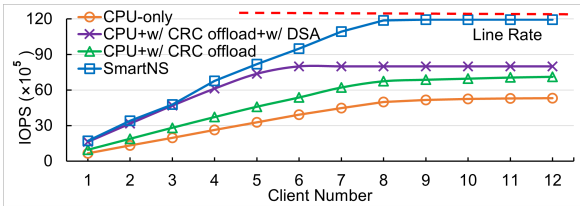


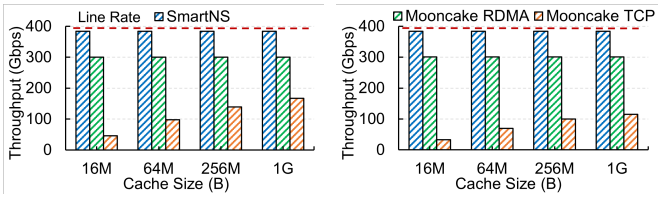
Figure 16. Performance comparison when running the block storage application with Solar protocol.

operations. SmartNS also exhibits more stable latency performance, primarily because intra-node DMA operations are less susceptible to network fluctuations. Figure 15b shows the throughput achieved by Batched RDMA READ on a single connection, SmartNS achieved 3.5× higher throughput than RNIC, this is mainly because RDMA READ has limited outstanding number and simply increasing READ operations number doesn't increase throughput, but SmartNS can perform READ concurrently with the powerful DMA engine, thereby enabling significantly enhanced throughput.

5.7 End-to-end Application Performance

Disaggregate block storage. The compute-storage separation architecture has become the de facto paradigm in modern data centers [57, 81, 82, 105]. In this design, compute servers and storage servers are organized into separate clusters, allowing each to be independently designed and optimized for specific workloads. The storage agent (SA), embedded within compute servers, converts storage operations into network transactions. Alibaba Solar [56] is the transport protocol used in SA and has been widely deployed. Leveraging the high flexibility offered by SmartNS, we implemented the Solar protocol in fewer than 2,000 lines of C++ code. As the baseline, we executed the protocol on dedicated CPU cores, similar to Snap. Additionally, we employ CRC NIC offload and Intel Data Streaming Accelerator (DSA) [42] engine to further improve CPU performance. Similar to Solar [56, 110], we select widely used 4KB READ request IOPS as metrics and limit the Solar-CPU network stack to 8 dedicated CPU cores. Each client is assigned to a dedicated core and set TX depth to 32.

Figure 16 illustrates the IOPS achieved by various approaches. We have two observations. First, SmartNS reaches line rate with more than 8 clients, delivering 2.5× (2.2×) higher IOPS compared to the "CPU-only" baseline for 1 client (12 clients). This is because SmartNS offloads intensive CRC



(a) CPU mem to CPU mem **(b)** CPU mem to GPU mem

Figure 17. Throughput comparison of KVCache transfer.

calculation to NIC hardware and massive memcpy to DMA engine, leaving only lightweight in-cache protocol processing for the Arm core. Second, even utilizing CRC offload and DSA engine, SmartNS still achieves 1.5× higher IOPS than "Solar-CPU". Benefit from header-only offloading TX and unlimited-working-set in-cache processing RX, SmartNS can easily operate at line rate, while "Solar-CPU" is constrained by the limitations of the CPU and DSA engine.

KVCache Transfer. Modern large language models (LLMs) are based on the Transformer architecture, and each inference request is logically divided into two stages: the prefill (P) stage and the decoding (D) stage. To meet stringent SLO requirements, the P/D disaggregation architecture is widely discussed and adopted [47, 66, 71, 109]. In this architecture, prefill nodes generate the KVCache and transfer it to the decoding nodes to continue decoding. Thus, efficient KVCache transfer from the prefill node pool to the decoding node pool is crucial. The Mooncake transfer engine [71] provides a high-performance data transfer framework supporting RDMA/TCP protocols and Nvidia GPUDirect. However, Mooncake only chooses limited QP connections for each transfer task, which induces QP hash collision and one of the physical ports is underutilized (similar to ECMP hash collision [70, 98]). Therefore, we integrated the packet spraying mechanism [8, 78] in SmartNS that dynamically varies the source UDP port on each packet, substantially mitigating hash collisions and fully utilizing the bandwidth of both physical ports. We replace Mooncake RDMA with SmartNS and compare it with the original Mooncake performance. Notably, SmartNS also supports Nvidia GPUDirect, enabling the network stack to transfer payloads directly to/from pinned GPU memory. For our evaluation, we utilize 8 CPU cores to submit transfer tasks, bond two ports to one `mlx5_bond_0` port [62], and set TX Depth to 1.

Figure 17a shows the transfer latency for various KVCache sizes between CPU memories, while Figure 17b shows the latency for transfers from CPU memory to pinned GPU memory. Leveraging header-only offloading TX and unlimited-working-set in-cache processing RX, SmartNS achieves line rate performance for KVCache sizes exceeding 16MB, exhibits 3.2× lower transfer latency compared to Mooncake TCP. Remarkably, SmartNS also outperforms Mooncake RDMA by 1.3×, attributed to the flexibility of SmartNS can easily adopt the packet spraying mechanism to fully utilize the bandwidth of both ports.

6 Related Work

Network Stack as a Service. IsoStack [79], ZygOS [69], TAS [35], NetKernel [60], FreeFlow [36], and RoUD [22] use dedicated CPU cores to construct an efficient and low-latency network stack. Snap [52] is an industry framework proposed by Google and provided in a general-purpose, multi-tenant cloud environment. Shenango [65] designs a busy looping IOKernel and uses an entire core to thread scheduling and packet steering functions. SRM [88] and KRCORE [97] share QP connections in kernel mode and user mode, respectively, and perform as a network stack. However, they all introduce the extra memcpy and cause host overhead, memory bandwidth pressure, and application interference. In contrast, SmartNS offloads the entire network stack to off-path SmartNIC, minimizing the host overhead and interference.

Hardware-offloaded Network Stack. Extensive studies have offloaded the entire network stack to hardware and customized RDMA transport. From the industry, Google 1RMA [84] and Falcon [20], AWS SRD [78], Meta [18] and Microsoft Hyperscale [4, 21], Alibaba Solar [56] and HPN [70], are driving RDMA transport customizations at scale. This is supplemented by academic contributions like SRNIC [94] and StaR [93] focus on scalability, SmartDS [103] and RPC-NIC [102] focus on message split. ZeroNIC [85] splits the packet header to host kernel stack and passes the payload directly to the destination. All of them can achieve high throughput but offer low programmability and low flexibility. Instead, SmartNS maintains line-rate and high flexibility through the novel designs.

Offloading to SmartNIC. Offloading host workloads to SmartNICs has recently attracted significant attention in both academia and industry. Many prior works [30, 38, 48, 86, 95, 104] offload tasks to off-path SmartNICs. IO-TCP [38] offloads disk I/O and TCP packet transfer to SmartNIC and reduces the burden on the CPU for online content delivery. Xenic [77] offloads distributed transactions to SmartNIC. SCR [107] uses datapath accelerator (DPA) to fine-grain perform the congestion control algorithm. These works leverage SmartNIC to alleviate host CPU pressure but do not provide a comprehensive study on the offload network stack.

7 Conclusioin

This paper presents SmartNS, a SmartNIC-centric network stack with software transport programmability and line-rate packet processing capabilities. To tackle the limitations of SmartNIC-induced challenges, SmartNS introduces a header-only offloading TX path, an unlimited-working-set in-cache processing RX path, a DMA-only notification pipe, and a programmable offloading engine. SmartNS is immediately deployable, which maintains compatibility with IBV verbs and leverages off-the-shelf SmartNICs. The experimental results show that SmartNS achieves 2.2× higher IOPS in block storage and 1.3 × higher throughput in KVCache transfer. We will make SmartNS open-source to benefit our community.

References

[1] Mohammad Alian, Siddharth Agarwal, Jongmin Shin, Neel Patel, Yifan Yuan, Daehoon Kim, Ren Wang, and Nam Sung Kim. Idio: Network-driven, inbound network data orchestration on server processors. In *MICRO*. IEEE, 2022.

[2] Shashank Anand, Michal Friedman, Michael Giardino, and Gustavo Alonso. Skip it: Take control of your cache! In *ASPLOS*, 2024.

[3] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling programmable transport protocols in {High-Speed}{NICs}. In *NSDI*, 2020.

[4] Wei Bai, Shanim Sainul Abdeen, Ankit Agrawal, Krishan Kumar Attre, Paramvir Bahl, Ameya Bhagat, Gowri Bhaskara, Tanya Brokhman, Lei Cao, Ahmad Cheema, et al. Empowering azure storage with {RDMA}. In *NSDI*, 2023.

[5] Broadcom. BCM958804-PS1100R. https://gtmteknoloji.com/wp-content/uploads/2020/08/PS1100R-PB100.pdf, 2020.

[6] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. Understanding host network stack overheads. In *SIGCOMM*, 2021.

[7] Qizhe Cai, Midhul Vuppalapati, Jaehyun Hwang, Christos Kozyrakis, and Rachit Agarwal. Towards μ s tail latency and terabit ethernet: disaggregating the host network stack. In *SIGCOMM*, 2022.

[8] Guo Chen, Yuanwei Lu, Bojie Li, Kun Tan, Yongqiang Xiong, Peng Cheng, Jiansong Zhang, and Thomas Moscibroda. Mp-rdma: enabling rdma with multi-path transport in datacenters. *TON*, 2019.

[9] Xuzheng Chen and Jie Zhang. Libr: Yet another rdma perftest. https://github.com/carlzhang4/libr, 2025.

[10] Xuzheng Chen, Jie Zhang, Ting Fu, Yifan Shen, Shu Ma, Kun Qian, Lingjun Zhu, Chao Shi, Yin Zhang, Ming Liu, et al. Demystifying datapath accelerator enhanced off-path smartnic. *arXiv preprint arXiv:2402.03041*, 2024.

[11] DPDK. Data Plane Development Kit. https://www.dpdk.org/, 2025.

[12] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: distributed transactions with consistency, availability, and performance. In *SOSP*, 2015.

[13] Alireza Farshin, Tom Barbette, Amir Roozbeh, Gerald Q Maguire Jr, and Dejan Kostić. Packetmill: toward per-core 100-gbps networking. In *ASPLOS*, 2021.

[14] Alireza Farshin, Amir Roozbeh, Gerald Q Maguire Jr, and Dejan Kostić. Make the most out of last level cache in intel processors. In *EuroSys*, 2019.

[15] Alireza Farshin, Amir Roozbeh, Gerald Q Maguire Jr, and Dejan Kostić. Make the most out of last level cache in intel processors. In *EuroSys*, 2019.

[16] Alireza Farshin, Amir Roozbeh, Gerald Q Maguire Jr, and Dejan Kostić. Reexamining direct cache access to optimize {I/O} intensive applications for multi-hundred-gigabit networks. In *ATC*, 2020.

[17] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *OSDI*, 2020.

[18] Adithya Gangidi, Rui Miao, Shengbao Zheng, Sai Jayesh Bondu, Guilherme Goes, Hany Morsy, Rohit Puri, Mohammad Riftadi, Ashmitha Jeevaraj Shetty, Jingyi Yang, et al. Rdma over ethernet for distributed training at meta scale. In *SIGCOMM*, 2024.

[19] sshaulnv gilr8, HassanKhadour. perftest: Infiniband verbs performance tests. https://github.com/linux-rdma/perftest, 2025.

[20] Google Falcon. Google Falcon. https://github.com/opencomputeproject/OCP-NET-Falcon, 2024.

[21] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. Rdma over commodity ethernet at scale. In *SIGCOMM*, 2016.

[22] Zhiqiang He, Yuxin Chen, and Bei Hua. Roud: Scalable rdma over ud in lossy data center networks. In *CCGrid*, 2023.

[23] Zhiqiang He, Dongyang Wang, Binzhang Fu, Kun Tan, Bei Hua, Zhi-Li Zhang, and Kai Zheng. Masq: Rdma for virtual private cloud. In *SIGCOMM*, 2020.

[24] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Changhoon Kim, and Nick McKeown. The nanopu: A nanosecond network stack for datacenters. In *OSDI*, 2021.

[25] Infiniband. Infiniband architecture. https://www.infinibandta.org/, 2008.

[26] Infiniband. RoCEv2 Update from the IBTA. https://www.openfabrics.org/images/eventpresos/workshops2014/DevWorkshop/presos/Wednesday/pdf/02_RoCEv2forOFA.pdf, 2014.

[27] Intel. Intel Memory Latency Checker. https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html, 2024.

[28] Intel. Intel® Infrastructure Processing Unit. https://www.intel.com/content/www/us/en/products/details/network-io/ipu.html, 2024.

[29] JEDEC. JEDEC DDR5 compare. https://www.jedec.org/category/technology-focus-area/main-memory-ddr3-ddr4-sdram, 2024.

[30] Zhen Jin, Yiquan Chen, Mingxu Liang, Yijing Wang, Guoju Fang, Ao Zhou, Keyao Zhang, Jiexiong Xu, Wenhai Lin, Yiquan Lin, et al. Os2g: A high-performance dpu offloading architecture for gpu-based deep learning with object storage. In *ASPLOS*, 2025.

[31] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for {μsecond-scale} tail latency. In *NSDI*, 2019.

[32] Anuj Kalia, Michael Kaminsky, and David G Andersen. Design guidelines for high performance {RDMA} systems. In *ATC*, 2016.

[33] Anuj Kalia, Michael Kaminsky, and David G Andersen. {FaSST}: Fast, scalable and simple distributed transactions with two-sided rdma datagram rpcs. In *OSDI*, 2016.

[34] Sagar Karandikar, Aniruddha N Udipi, Junsun Choi, Joonho Whangbo, Jerry Zhao, Svilen Kanev, Edwin Lim, Jyrki Alakuijala, Vrishab Madduri, Yakun Sophia Shao, et al. Cdpu: Co-designing compression and decompression processing units for hyperscale systems. In *ISCA*, 2023.

[35] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr Sharma, Arvind Krishnamurthy, and Thomas Anderson. Tas: Tcp acceleration as an os service. In *EuroSys*, 2019.

[36] Daehyeok Kim, Tianlong Yu, Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Shachar Raindel, Chuanxiong Guo, Vyas Sekar, and Srinivasan Seshan. {FreeFlow}: Software-based virtual {RDMA} networking for containerized clouds. In *NSDI*, 2019.

[37] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostić, Youngjin Kwon, Simon Peter, and Emmett Witchel. Linefs: Efficient smartnic offload of a distributed file system with pipeline parallelism. In *SOSP*, 2021.

[38] Taehyun Kim, Deondre Martin Ng, Junzhi Gong, Youngjin Kwon, Minlan Yu, and KyoungSoo Park. Rearchitecting the tcp stack for i/o-offloaded content delivery. In *NSDI*, 2023.

[39] Xinhao Kong, Jingrong Chen, Wei Bai, Yechen Xu, Mahmoud Elhaddad, Shachar Raindel, Jitendra Padhye, Alvin R Lebeck, and Danyang Zhuo. Understanding {RDMA} microarchitecture resources for performance isolation. In *NSDI*, 2023.

[40] Adithya Kumar, Anand Sivasubramaniam, and Timothy Zhu. Splitrpc: A {Control+ Data} path splitting rpc stack for ml inference serving. *POMACS*, 2023.

[41] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan MG Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, et al. Swift: Delay is simple and effective for congestion control in the datacenter. In *SIGCOMM*, 2020.

[42] Reese Kuper, Ipoom Jeong, Yifan Yuan, Ren Wang, Narayan Ranganathan, Nikhil Rao, Jiayu Hu, Sanjay Kumar, Philip Lantz, and Nam Sung Kim. A quantitative analysis and guidelines of data streaming accelerator in modern intel xeon scalable processors. In *ASPLOS*,

2024.

[43] Jialong Li, Haotian Gong, Federico De Marchi, Aoyu Gong, Yiming Lei, Wei Bai, and Yiting Xia. Uniform-cost multi-path routing for reconfigurable data center networks. In *SIGCOMM*, 2024.

[44] Qiang Li, Qiao Xiang, Derui Liu, Yuxin Wang, Haonan Qiu, Xiaoliang Wang, Jie Zhang, Ridi Wen, Haohao Song, Gexiao Tian, et al. From rdma to rdca: Toward high-speed last mile of data center networks using remote direct cache access. *arXiv preprint arXiv:2211.05975*, 2022.

[45] Qijing Li, Xinyang Huang, Bowen Liu, Pengbo Li, Junxue Zhang, and Kai Chen. Cache-aware i/o rate control for rdma. In *APNet*, 2025.

[46] Wenxue Li, Xiangzhou Liu, Yunxuan Zhang, Zihao Wang, Wei Gu, Tao Qian, Gaoxiong Zeng, Shoushou Ren, Xinyang Huang, Zhenghang Ren, et al. Revisiting rdma reliability for lossy fabrics. In *SIGCOMM*, 2025.

[47] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

[48] Bowen Liu, Xinyang Huang, Qijing Li, Zhuobin Huang, Yijun Sun, Wenxue Li, Junxue Zhang, Ping Yin, and Kai Chen. Ceio: A cache-efficient network i/o architecture for nic-cpu data paths. In *SIGCOMM*, 2025.

[49] Guowei Liu, Laiping Zhao, Yiming Li, Zhaolin Duan, Sheng Chen, Yitao Hu, Zhiyuan Su, and Wenyu Qu. Fuyao: Dpu-enabled direct data transfer for serverless computing. In *ASPLOS*, 2024.

[50] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto smartnics using ipipe. In *SIGCOMM*. 2019.

[51] Jiaqi Lou, Xinhao Kong, Jinghan Huang, Wei Bai, Nam Sung Kim, and Danyang Zhuo. Harmonic: Hardware-assisted {RDMA} performance isolation for public clouds. In *NSDI*, 2024.

[52] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C Evans, Steve Gribble, et al. Snap: A microkernel approach to host networking. In *SOSP*, 2019.

[53] Mellanox. ConnectX®-6 DX Card. https://www.nvidia.com/content/dam/en-zz/Solutions/networking/ethernet-adapters/connectX-6-dx-datasheet.pdf, 2022.

[54] Mellanox. ConnectX®-7 EN Card. https://www.nvidia.com/content/dam/en-zz/Solutions/networking/ethernet-adapters/connectx-7-datasheet-Final.pdf, 2023.

[55] Mellanox. ConnectX®-8 EN Card. https://resources.nvidia.com/en-us-accelerated-networking-resource-library/connectx-datasheet-c, 2024.

[56] Rui Miao, Lingjun Zhu, Shu Ma, Kun Qian, Shujun Zhuang, Bo Li, Shuguang Cheng, Jiaqi Gao, Yan Zhuang, Pengcheng Zhang, et al. From luna to solar: the evolutions of the compute-to-storage networks in alibaba cloud. In *SIGCOMM*, 2022.

[57] Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. Gimbal: enabling multi-tenant storage disaggregation on smartnic jbofs. In *SIGCOMM*, 2021.

[58] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. Revisiting network support for rdma. In *SIGCOMM*, 2018.

[59] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. AccelTCP: Accelerating network applications with stateful TCP offloading. In *NSDI*, 2020.

[60] Zhixiong Niu, Qiang Su, Peng Cheng, Yongqiang Xiong, Dongsu Han, Keith Winstein, Chun Jason Xue, and Hong Xu. Netkernel: Making network stack part of the virtualized infrastructure. *TON*, 2021.

[61] Nvidia. NVIDIA BLUEFIELD-2 DPU. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-2-dpu.pdf, 2022.

[62] Nvidia. Link Aggregation. https://docs.nvidia.com/networking/display/bluefielddpubspv422/link+aggregation, 2024.

[63] Nvidia. NVIDIA BLUEFIELD-3 DPU. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-3-dpu.pdf, 2024.

[64] NVIDIA. Cache Invalidate Operation. https://docs.nvidia.com/doca/sdk/mmap+advise/index.html#src-3543226633_id-.MmapAdvisev2.10.0-CacheInvalidateOperation, 2025.

[65] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high {CPU} efficiency for latency-sensitive datacenter workloads. In *NSDI*, 2019.

[66] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting. In *ISCA*. IEEE, 2024.

[67] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. The design and implementation of open {vSwitch}. In *NSDI*, 2015.

[68] Boris Pismenny, Adam Morrison, and Dan Tsafrir. {ShRing}: Networking with shared receive rings. In *OSDI*, 2023.

[69] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *SOSP*, 2017.

[70] Kun Qian, Yongqing Xi, Jiamin Cao, Jiaqi Gao, Yichi Xu, Yu Guan, Binzhang Fu, Xuemei Shi, Fangbo Zhu, Rui Miao, et al. Alibaba hpn: A data center network for large language model training. In *SIGCOMM*, 2024.

[71] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: Trading more storage for less computation—a {KVCache-centric} architecture for serving {LLM} chatbot. In *FAST*, 2025.

[72] Eric Quinnell. Tesla transport protocol over ethernet (ttpoe): A new lossy, exa-scale fabric for the dojo ai supercomputer. In *HCS*, 2024.

[73] RDMA Core. RDMA core userspace libraries and daemons. https://github.com/linux-rdma/rdma-core, 2024.

[74] Feng Ren, Mingxing Zhang, Kang Chen, Huaxia Xia, Zuoning Chen, and Yongwei Wu. Scaling up memory disaggregated applications with smart. In *ASPLOS*, 2024.

[75] Zhenghang Ren, Yuxuan Li, Zilong Wang, Xinyang Huang, Wenxue Li, Kaiqiang Xu, Xudong Liao, Yijun Sun, Bowen Liu, Han Tian, et al. Enabling efficient {GPU} communication over multiple {NICs} with {FuseLink}. In *OSDI*, 2025.

[76] Hugo Sadok, Nirav Atre, Zhipeng Zhao, Daniel S Berger, James C Hoe, Aurojit Panda, Justine Sherry, and Ren Wang. {Ensō}: A streaming interface for {NIC-Application} communication. In *OSDI*, 2023.

[77] Henry N Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. Xenic: Smartnic-accelerated distributed transactions. In *ASPLOS*, 2021.

[78] Leah Shalev, Hani Ayoub, Nafea Bshara, and Erez Sabbag. A cloud-optimized transport protocol for elastic and scalable hpc. *IEEE micro*, 2020.

[79] Leah Shalev, Julian Satran, Eran Borovik, and Muli Ben-Yehuda. IsoStack—Highly efficient network processing on dedicated cores. In *ATC*, 2010.

[80] Rajath Shashidhara, Tim Stamler, Antoine Kaufmann, and Simon Peter. {FlexTOE}: Flexible {TCP} offload with {Fine-Grained} parallelism. In *NSDI*, 2022.

[81] Junyi Shu, Kun Qian, Ennan Zhai, Xuanzhe Liu, and Xin Jin. Burstable cloud block storage with data processing units. In *OSDI*, 2024.

[82] Junyi Shu, Ruidong Zhu, Yun Ma, Gang Huang, Hong Mei, Xuanzhe Liu, and Xin Jin. Disaggregated raid storage in modern datacenters. In *ASPLOS*, 2023.

[83] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. Strom: smart remote memory. In *EuroSys*, 2020.

[84] Arjun Singhvi, Aditya Akella, Dan Gibson, Thomas F Wenisch, Monica Wong-Chan, Sean Clark, Milo MK Martin, Moray McLaren, Prashant Chandra, Rob Cauble, et al. 1rma: Re-envisioning remote memory access for multi-tenant datacenters. In *SIGCOMM*, 2020.

[85] Athinagoras Skiadopoulos, Zhiqiang Xie, Mark Zhao, Qizhe Cai, Saksham Agarwal, Jacob Adelmann, David Ahern, Carlo Contavalli, Michael Goldflam, Vitaly Mayatskikh, et al. High-throughput and flexible host networking for accelerated computing. In *OSDI*, 2024.

[86] Qiang Su, Shaofeng Wu, Zhixiong Niu, Ran Shu, Peng Cheng, Yongqiang Xiong, Zaoxing Liu, and Hong Xu. Meili: Enabling smartnic as a service in the cloud. *arXiv preprint arXiv:2312.11871*, 2023.

[87] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, et al. Demystifying cxl memory with genuine cxl-ready systems and devices. In *MICRO*, 2023.

[88] Jian Tang, Xiaoliang Wang, and Huichen Dai. Scalable rdma transport with efficient connection sharing. In *INFOCOM*. IEEE, 2023.

[89] Amin Tootoonchian, Aurojit Panda, Chang Lan, Melvin Walls, Katerina Argyraki, Sylvia Ratnasamy, and Scott Shenker. {ResQ}: Enabling {SLOs} in network function virtualization. In *NSDI*, 2018.

[90] Shin-Yeh Tsai and Yiying Zhang. Lite kernel rdma support for datacenter applications. In *SOSP*, 2017.

[91] Midhul Vuppalapati, Saksham Agarwal, Henry Schuh, Baris Kasikci, Arvind Krishnamurthy, and Rachit Agarwal. Understanding the host network. In *SIGCOMM*, 2024.

[92] Hao Wang, Han Tian, Jingrong Chen, Xinchen Wan, Jiacheng Xia, Gaoxiong Zeng, Wei Bai, Junchen Jiang, Yong Wang, and Kai Chen. Towards {Domain-Specific} network transport for distributed {DNN} training. In *NSDI*, 2024.

[93] Xizheng Wang, Guo Chen, Xijin Yin, Huichen Dai, Bojie Li, Binzhang Fu, and Kun Tan. Star: Breaking the scalability limit for rdma. In *ICNP*, 2021.

[94] Zilong Wang, Layong Luo, Qingsong Ning, Chaoliang Zeng, Wenxue Li, Xinchen Wan, Peng Xie, Tao Feng, Ke Cheng, Xiongfei Geng, et al. {SRNIC}: A scalable architecture for {RDMA}{NICs}. In *NSDI*, 2023.

[95] Xingda Wei, Rongxin Cheng, Yuhan Yang, Rong Chen, and Haibo Chen. Characterizing off-path smartnic for accelerating distributed systems. In *OSDI*, 2023.

[96] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing {RDMA-enabled} distributed transactions: Hybrid is better! In *OSDI*, 2018.

[97] Xingda Wei, Fangming Lu, Rong Chen, and Haibo Chen. {KRCORE}: A microsecond-scale {RDMA} control plane for elastic computing. In *ATC*, 2022.

[98] Yunhong Xu, Keqiang He, Rui Wang, Minlan Yu, Nick Duffield, Hassan Wassel, Shidong Zhang, Leon Poutievski, Junlan Zhou, and Amin Vahdat. Hashing design in modern networks: Challenges and mitigation techniques. In *ATC*, 2022.

[99] Zhuolong Yu, Bowen Su, Wei Bai, Shachar Raindel, Vladimir Braverman, and Xin Jin. Understanding the micro-behaviors of hardware offloaded network stacks with lumina. In *SIGCOMM*, 2023.

[100] Yifan Yuan, Mohammad Alian, Yipeng Wang, Ren Wang, Ilia Kurakin, Charlie Tai, and Nam Sung Kim. Don't forget the i/o when allocating your llc. In *ISCA*. IEEE, 2021.

[101] Rohit Zambre, Aparna Chandramowlishwaran, and Pavan Balaji. Scalable communication endpoints for mpi+ threads applications. In *ICPADS*. IEEE, 2018.

[102] Jie Zhang, Hongjing Huang, Xuzheng Chen, Xiang Li, Ming Liu, and Zeke Wang. Rpcacc: A high-performance and reconfigurable pcie-attached rpc accelerator. *arXiv preprint arXiv:2411.07632*, 2024.

[103] Jie Zhang, Hongjing Huang, Lingjun Zhu, Shu Ma, Dazhong Rong, Yijun Hou, Mo Sun, Chaojie Gu, Peng Cheng, Chao Shi, et al. Smartds: Middle-tier-centric smartnic enabling application-aware message split for disaggregated block storage. In *ISCA*, 2023.

[104] Qizhen Zhang, Philip Bernstein, Badrish Chandramouli, Jiasheng Hu, and Yiming Zheng. Dds: Dpu-optimized disaggregated storage. *arXiv preprint arXiv:2407.13618*, 2024.

[105] Qizhen Zhang, Xinyi Chen, Sidharth Sankhe, Zhilei Zheng, Ke Zhong, Sebastian Angel, Ang Chen, Vincent Liu, and Boon Thau Loo. Optimizing data-intensive systems in disaggregated data centers with teleport. In *SIGMOD*, 2022.

[106] Yiwen Zhang, Yue Tan, Brent Stephens, and Mosharaf Chowdhury. Justitia: Software {Multi-Tenancy} in hardware {Kernel-Bypass} networks. In *NSDI*, 2022.

[107] Chenxingyu Zhao, Jaehong Min, Ming Liu, and Arvind Krishnamurthy. {White-Boxing}{RDMA} with {Packet-Granular} software control. In *NSDI*, 2025.

[108] Yimeng Zhao, Ahmed Saeed, Ellen Zegura, and Mostafa Ammar. Zd: a scalable zero-drop network stack at end hosts. In *CoNEXT*, 2019.

[109] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. {DistServe}: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *OSDI*, 2024.

[110] Lingjun Zhu, Yifan Shen, Erci Xu, Bo Shi, Ting Fu, Shu Ma, Shuguang Chen, Zhongyu Wang, Haonan Wu, Xingyu Liao, et al. Deploying user-space {TCP} at cloud scale with {LUNA}. In *ATC*, 2023.

[111] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale rdma deployments. *SIGCOMM*, 2015.

[112] Pengfei Zuo, Jiazhao Sun, Liu Yang, Shuangwu Zhang, and Yu Hua. One-sided {RDMA-Conscious} extendible hashing for disaggregated memory. In *ATC*, 2021.