High-throughput and Flexible Host Networking for Accelerated Computing

Athinagoras Skiadopoulos[*] [1] Zhiqiang Xie[1] Mark Zhao[1] Qizhe Cai[2] Saksham Agarwal[2]
Jacob Adelmann[3] David Ahern[3] Carlo Contavalli[3] Michael Goldflam[3] Vitaly Mayatskikh[3]
Raghu Raja[†] [4] Daniel Walton[3] Rachit Agarwal[2] Shrijeet Mukherjee[3] Christos Kozyrakis[1]

[1]*Stanford University* [2]*Cornell University* [3]*Enfabrica*

Abstract

Modern network hardware is able to meet the stringent bandwidth demands of applications like GPU-accelerated AI. However, existing host network stacks offer a hard tradeoff between performance (in terms of sustained throughput when compared to network hardware capacity) and flexibility (in terms of the ability to select, customize, and extend different network protocols).

This paper explores a clean-slate approach to simultaneously offer high performance and flexibility. We present a co-design of the NIC hardware and the software stack to achieve this. The key idea in our design is the physical separation of the data path (payload transfer between network and application buffers) and the control path (header processing and transport-layer decisions). The NIC enables a high-performance zero-copy data path, independent of the placement of the application (CPU, GPU, FPGA, or other accelerators). The software stack provides a flexible control path by enabling the integration of any network protocol, executing in any environment (in the kernel, in user space, or in an accelerator).

We implement and evaluate *ZeroNIC*, a prototype that combines an FPGA-based NIC with a software stack that integrates the Linux TCP protocol. We demonstrate that *ZeroNIC* achieves RDMA-like throughput while maintaining the benefits of robust protocols like TCP under various network perturbations. For instance, *ZeroNIC* enables a single TCP flow to saturate a $100 Gbps$ link while utilizing only 17% of a single CPU core. *ZeroNIC* improves NCCL and Redis throughput by $2.66\times$ and $3.71\times$, respectively, over Linux TCP on a Mellanox ConnectX-6 NIC, without requiring application modifications.

[*]Work partially done while interning at Enfabrica.
[†]Affiliated with Amazon Web Services, work done while at Enfabrica.

1 Introduction

Modern datacenter applications, such as artificial intelligence (AI), data analytics, and distributed storage, are increasingly reliant on moving massive amounts of data over the network. As a result, datacenter operators are deploying systems capable of hundreds of Gbps of host networking. For instance, the latest NVIDIA DGX-B200 is capable of $3.2 Tbps$ of networking – $400 Gbps$ for each of the 8 GPUs [26]. As compute, memory, and link throughput continue to scale, driven by technologies such as accelerators [5,25,49], the *end-host* network stack is rapidly becoming a dominant bottleneck for these applications [12,13,72,95]. Therefore, the problem of designing host network stacks has come to the forefront.

Existing host network stacks offer a hard tradeoff between performance (in terms of sustained throughput when compared to network hardware capacity) and flexibility (in terms of the ability to select, customize, and extend different network protocols). On the one extreme, RDMA-based host network stacks [8, 34, 38, 54] are able to achieve high performance, but provide minimal to no flexibility. With network protocols baked into the hardware, adapting the protocol to better suit the needs of emerging applications or deployments is either not feasible or requires the time-consuming process of hardware modification. As a result, existing RDMA-based deployments remain fragile due to the possibility of head-of-line blocking, deadlocks, congestion spreading, and/or host congestion [1,2,44,45,63,65,74,96]. On the other extreme, the Linux network stack provides flexibility with a variety of time-tested protocols [4,11,14,39,47,68] and mechanisms that enable the incorporation of new protocols [2,10,13]. Unfortunately, the current Linux stack falls significantly short of exploiting the high-throughput capabilities of modern network hardware [12]. Recent host network stacks [72,89] offer operating points between these two extremes, but suffer from a similar performance-flexibility tradeoff.

We present a clean-slate co-design of the host network hardware and the software stack that simultaneously achieves high performance and flexibility. Our design's key driving idea is the physical separation of the data path (payload transfer between network and application buffers) and the control path (header processing and transport-layer decisions) within the host. Specifically, our NIC hardware enables a *high-performance data path* between the network and the application. The NIC splits the headers from the payload, and directly transfers the payload from/to application buffers, without requiring any intermediate data copy (*zero-copy*). Our software stack enables a *flexible control path*. Users can plug in existing transport stacks, which operate on packets (sans payloads) as before, to make decisions on when to send data (e.g., congestion and flow control) and notify applications upon completion (e.g., acknowledging in-order byte streams). The software control stack orchestrates memory management and signaling between the NIC, the transport stack, and applications. Importantly, our design is independent of the location of application buffers (CPU, GPU, FPGA, or other accelerators) or the transport protocol's execution environment (in the kernel, in user space, or even in an accelerator).

The key challenge in realizing the physical separation of data and control paths is to maintain correct semantics (in-order, exactly-once data delivery) even in presence of network perturbations (data corruption, drops, replication, reordering, etc.). Our hardware implements the bookkeeping needed to correctly transfer incoming data to their designated memory destination even in the presence of network perturbations, while our software stack coordinates across the hardware and the application layer to maintain correct protocol semantics.

We demonstrate the benefits of our approach using an end-to-end prototype, *ZeroNIC*. Our prototype combines an FPGA-based NIC connecting to CPU and GPU memory, with a software stack integrating in-kernel Linux TCP. Our prototype realizes two APIs: the *libibverbs* API [67] used by current RDMA applications and a streaming API for general-purpose socket applications. For both APIs, *ZeroNIC* supports zero-copy data transfers between the NIC and application buffers in CPU or GPU memory. We evaluate *ZeroNIC* across a variety of workloads and network conditions. *ZeroNIC* achieves RDMA-level throughput with low CPU utilization. For instance, we show that *ZeroNIC* allows a single TCP flow to saturate a 100*Gbps* link while utilizing only 17% of a single CPU hyperthread. In comparison, the Linux host network stack on a Mellanox ConnectX-6 NIC achieves at most 50*Gbps* for a single TCP flow at 100% CPU utilization. We also demonstrate that *ZeroNIC* enables a high-performance zero-copy data path between GPU devices, achieving 2.66× higher throughput in NCCL benchmarks [24], NVIDIA's core AI networking library. Finally, we show that *ZeroNIC* benefits from the use of robust network protocols such as the TCP implementation in Linux. *ZeroNIC* maintains its performance under drops and fairness across flows.

To the best of our knowledge, our work is the first to support both send and receive-side zero-copy for reliable protocols like TCP with no constraints (e.g., MTU alignment, API modifications). It supports accelerator devices (e.g., GPUs) and enables protocol termination anywhere (e.g., CPU or control-plane accelerators) without limiting protocol semantics.

2 Motivation and Background

Our goal is to enable high-performance host networking regardless of the data destination (host or accelerator memory) and where/how the control plane is implemented. This flexibility allows the development and tuning of network protocols that improve fabric behavior and overall network efficiency as applications evolve and systems scale.

2.1 RDMA: Performant but Inflexible

Many network-intensive applications, such as AI training using GPUs, frequently use RDMA solutions such as InfiniBand (IB) [6] or RoCE [7]. RDMA solutions bypass the OS network stack and its CPU overheads by terminating the network protocol in specialized hardware and firmware in RDMA NICs (RNICs). RNICs enable high throughput by DMAing network payloads directly from/to application buffers in CPU or GPU memory using information encoded in send/receive requests.

The disadvantage of RDMA solutions is the lack of flexibility. To achieve high throughput, RDMA solutions typically required a lossless fabric such as IB, reliant on certified (short-distance) cabling and specialized switches. Such networks were forced to adopt a restrictive topology, avoiding over-subscription and adding many redundant paths to avoid drops [21,93]. To provide a similar quality-of-service on lossy fabrics, RoCE solutions have increasingly required secondary mechanisms to eliminate drops in the face of congestion, such as priority flow control (PFC) [46] and watchdogs [8, 38]. RoCE solutions still suffer from a host of well-documented challenges, such as end-host congestion [55], major performance degradation under unavoidable network perturbations (e.g., packet drops or reorderings) [44, 103], and excessive buffer requirements [44]. Addressing these challenges is arduous because RoCE's control path is explicitly tied to the implementation of the RNIC, requiring collaboration with and intervention by RNIC vendors. For example, Microsoft required support from its RNIC vendor to address livelocks caused by go-back-0 retransmission [38].

2.2 Kernel Networking: Flexible but Slow

The Linux network stack, built around the TCP/IP protocols, runs on a vast range of commodity hardware, supports diverse topologies, and can adapt to highly-variable network conditions and failures. Its resiliency stems from the fact that developers can optimize network protocol parameters including the congestion scheme and buffer sizes for the needs of emerging applications and deployments.

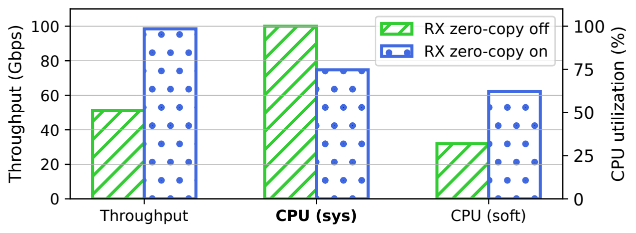


Figure 1: Throughput and receiver CPU utilization with and without receive-side (RX) emulated zero-copy. 100% means that a hyperthread is fully utilized. "CPU sys" refers to the hyperthread running protocol processing. "CPU soft" refers to the hyperthread running the application and the software interrupt handler.

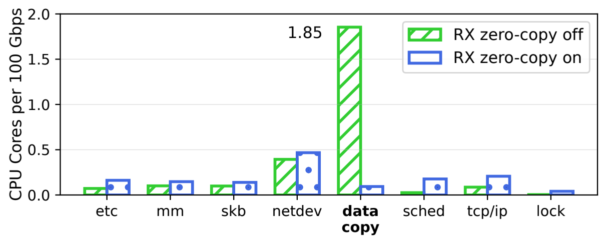


Figure 2: CPU breakdown for the TCP receiver with and without emulated RX zero-copy, normalized to the CPU utilization needed to achieve 100*Gbps* of throughput.

Unfortunately, the Linux stack cannot achieve high throughput ($\geq$ 100*Gbps*) as single-thread CPU performance is a key bottleneck [12]. Specifically, receive-side data copies from kernel to application buffers dominate end-to-end performance, limiting a single TCP flow even after major optimizations (e.g., TSO/GRO, jumbo frames, and packet steering).

We built a proof-of-concept experiment to showcase the single-flow performance potential of removing the data copy. We modified the iperf benchmark [28] to support send-side zero-copy via the Linux sender ZC API [53]. We emulated receive-side zero-copy by truncating payloads in the kernel, avoiding the additional copy to user space [1]. We enabled TSO, GRO, and jumbo frames. We also pinned the iperf process and steered the receiver flow so that the interrupt handler (`soft`) and TCP processing (`sys`) are located in the two hyperthreads of the same physical core (sharing the L1 cache).

Figure 1 shows the sustained throughput and CPU utilization. Even with send-side zero-copy on, regular kernel networking can only achieve 50*Gbps* for a single flow. Similar to [12], we observe that CPU utilization is the bottleneck – specifically the TCP protocol processing receiver core (`sys`). Throughput cannot scale and the interrupt handling thread (`soft`) is underutilized. Figure 2 shows that the majority of CPU cycles for TCP processing are spent on data copies. Enabling receive zero-copy eliminates data copy overheads and drastically improves throughput, saturating the 100*Gbps* link. This experiment suggests that a flexible receiver zero-copy mechanism that copies data to application buffers in CPU, GPU, or storage devices can enable a wide range of protocols/stacks, including Linux TCP and other user space or hardware protocols/stacks [32, 48, 51, 72, 76, 77, 90], to meet the throughput requirements of network-intensive applications.

[1]Code available at https://github.com/enfabrica/iperf

2.3 Towards Control & Data Path Separation

The core challenge with existing network solutions is the tight integration of the control and data paths, leading users to either integrate the data path into the kernel, sacrificing performance, or embed the control path in hardware, sacrificing flexibility. We propose the physical separation of these two paths. The data path provides robust support for zero-copy from NICs to application buffers on devices like CPUs and GPUs. The control path supports various transport protocols executing in software or hardware. This separation allows the control path to be optimized without overhauling the efficient data path.

There are many implementations of send-side (TX) zero-copy such as those in RDMA NICs, the MSG_ZEROCOPY flag in the Linux send system call [27], and the io_uring API for asynchronous I/O [18]. In contrast, existing *receive-side (RX) zero-copy* approaches are severely limited.

The challenge of page alignment. Linux includes a page-remapping mechanism for RX zero-copy in socket APIs [17, 59]. It allows the NIC to DMA the entire payload to a memory location and then remap the payload's physical address to the application buffer's virtual address at page granularity. This approach requires page-aligned payloads, making it difficult for applications to transmit arbitrary data lengths, as they can do with the socket or verbs APIs. The page-alignment requirement may also be incompatible with GPUs or flash devices [3], limiting the applicability of this approach. Moreover, page-remapping incurs high CPU overheads due to the need for TLB flushing after altering page table entries [59, 94].

The challenge of API compatibility. Several proposals facilitate RX zero-copy by altering application interfaces [9, 50, 79, 101, 102]. They require extensive changes to applications using common APIs like sockets or IB verbs, which typically rely on read/write operations from a contiguous buffer. These proposals asynchronously transfer packets from the NIC to application buffers, either as a linked list of scattered payloads or with headers and data interleaved in a buffer, which are released after being processed by the application. Hence, applications must adapt to handling non-contiguous data addresses during read operations.

The challenge of packet perturbations. Packet reordering, drops, and retransmissions disrupt the expected order of packet arrivals and complicate the correct copying of payloads into application buffers. A simple solution, employed by many

RNICs, is to discard out-of-order packets and default to a go-back-N retransmission strategy, at the expense of throughput (§2.1). An alternative is to temporarily buffer out-of-order packets in the NIC until the missing packets arrive, potentially through a selective retransmission mechanism. This approach can quickly exhaust the SRAM capacity of state-of-the-art NICs [74, 96], especially in large bandwidth-delay-product (BDP) environments such as hosts with 400*Gbps*+ networking per GPU, and limits the effective rate at which data is transferred to the application.

The challenge of reliable protocols. Some systems sacrifice reliable transport semantics, directly copying incoming payloads to the next-available application buffer. This limits RX zero-copy support to unreliable protocols like UDP [15, 57]. Recent attempts to support reliable connections (RC) have constrained applicability. SRNIC [96] handles sequential and out-of-order packets via separate fast and slow data paths. IRN [74] requires the sender to explicitly define a receiver buffer identifier in the header. 1RMA [91] requires application involvement for managing ordering and handling failure recovery. Flor [62] separates the control and data paths for RDMA transports to reconcile the control path differences across different RNIC generations. However, Flor primarily supports unreliable connections (UC). To extend to reliable semantics, Flor uses an additional reliability sequence number in the RDMA work request and requires the sender and the receiver to establish a common chunk size for data transfers. Flor must dynamically tune the chunk size to tradeoff between high throughput (larger chunks) and managing congestion, drops, and retransmissions (smaller chunks).

Other related work. Nicmem [80], PayloadPark [36], and Ribosome [87] have recently explored separating the control and data paths in distinct contexts from our goals. They focus primarily on NFV (Network Function Virtualization) workloads that do not process payloads, but rather operate only on metadata to deliver packets to their next destination. To optimize resource usage such as PCIe traffic, they split packet headers and payloads and send only the header to the host. SplitRPC [56] uses a control and data path separation, but it is limited to unreliable protocols like UDP and use-cases like end-user requests/responses for AI inference. Our work tackles a broader range of applications that continuously process payloads and benefit from transport layer functionalities.

3 Performant and Flexible Host Networking

We co-design the hardware and software to *physically separate the data and control paths* in host networking, but *logically couple them after separation*. The physical separation enables a high-throughput, zero-copy data path to application buffers for payloads, and an independent control path for header processing. Figure 3 provides a high-level view of our approach. The data path connects to *any endpoint* (e.g., accelerators, storage, host memory, etc.), and the control path

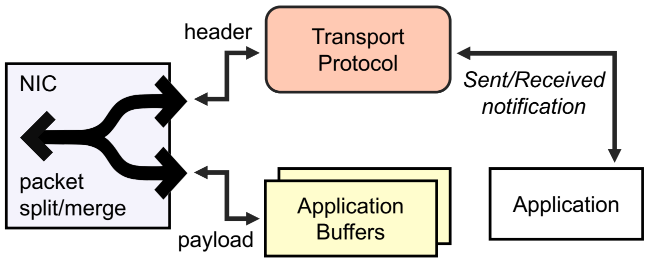
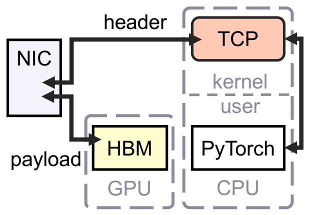
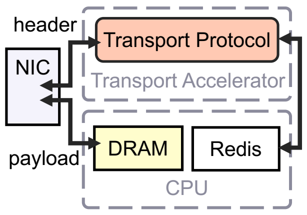


Figure 3: Host networking with physically separated control and data paths.



(a) Application buffers in GPU memory (HBM), protocol (in this case, TCP) in kernel space.



(b) Application buffers in CPU memory (host DRAM), protocol in a transport accelerator.

Figure 4: Examples of control and data path separation.

executes *arbitrary transport protocols* in *any execution environment* (in user or kernel space software on a CPU, in SmartNIC software or hardware, or even in a protocol accelerator), as illustrated by the two examples in Figure 4. The logical coupling allows the control path to have full control of protocol semantics, i.e., when data is correctly received or sent, how to handle events like reorderings and retransmissions, and when to notify the application – even if the transport protocol and data live in completely different devices.

The key challenge in providing a zero-copy data path managed by transport protocols external to the NIC is that the NIC must decide *if, when,* and *where* to copy incoming payloads, *prior* to the transport protocol addressing out-of-order deliveries and retransmissions. Additionally, regardless of when data is copied, it should only be exposed to the application when protocol semantics allow (e.g., in-order delivery).

We begin by reviewing how packets travel throughout our network stack (§3.1). The NIC (§3.2) splits and merges headers and data to enable zero-copy data transfers directly to arbitrary devices (e.g. GPUs), even under reorderings, retransmissions, and drops. Our software stack is composed of the control stack and the provider library. The control stack (§3.3), which can execute in an arbitrary execution environment (e.g., in the kernel as a driver) is the coordinator between the NIC, an arbitrary transport protocol, and the application. The transport protocol acts only on packet headers, while the control stack proxies its actions to data in remote memory (i.e., NIC or application buffers). Our provider library implements both

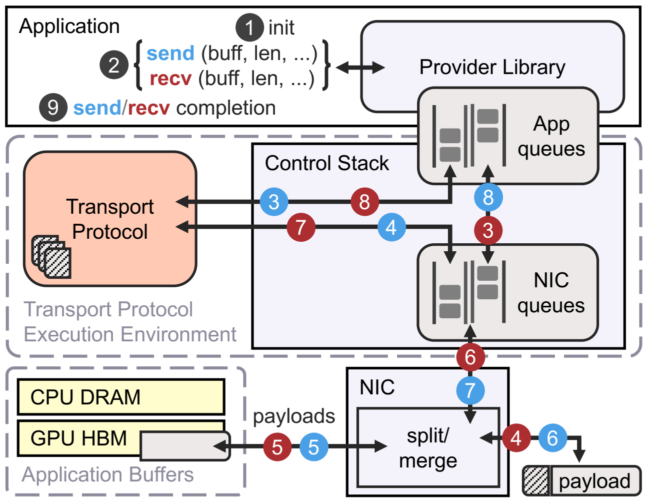


Figure 5: End-to-end send (blue) and receive (red) paths.

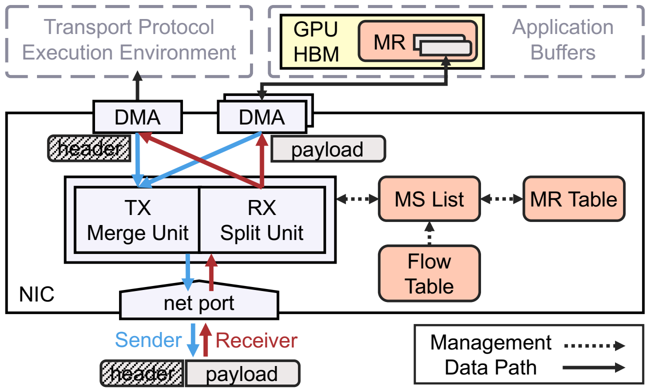


Figure 6: NIC hardware block diagram.

message and streaming APIs (§3.4) to allow an easy mapping of popular networking libraries onto our design. Finally, we discuss how we address various challenges in the host network stack such as retransmissions and compatibility with current optimizations (§3.5).

3.1 Receive and Send Path Overview

Figure 5 illustrates the receive (RX) and send (TX) paths through our stack.

Receive path. ① A receiving application begins by performing an initialization step using the provider library. As usual, this step establishes a connection and binds to a network interface. It also allocates application and NIC queues (§3.3) to coordinate between the control stack, the provider library, and the NIC hardware. The application also registers shared memory with the NIC for zero-copy transfers. These memory buffers can be anywhere in the system (e.g., GPU memory). Applications can periodically register (and deregister) shared memory space as needed. ② After initialization, the application invokes receive calls and the provider starts polling for completions. For every receive call, the provider enqueues an RX request entry into the application queue. RX request entries contain the receive call's buffer location and length. ③ The control stack steers the entry to the appropriate NIC queue. The NIC parses RX request entries to store application buffer information into dedicated hardware structures.

④ As packets arrive in the NIC from the sender, the NIC parses their headers and decides on dropping, buffering, or accepting each packet (§3.2). When a packet is accepted, the NIC splits it into the header and the payload. ⑤ The NIC identifies the payload's correct memory location in the designated device buffer and DMAs it accordingly. ⑥ The NIC creates and forwards RX header entries, composed of headers and metadata, to NIC queues leading to the control stack. ⑦ When the data DMA completes, the control stack forwards the headers from each RX header entry to the transport protocol. The protocol processes the header, reasoning about data acknowledgment (e.g., ACKing only in-order data). ⑧ When the protocol allows, the control stack posts a completion entry into the application queue.

Send path. The send path is also designed so that the transport protocol maintains control over data transmission. ① As with the receive path, the application begins with initialization steps that establish a connection, bind with a network device, and allocate and bind with the application and NIC queues needed for coordination. ② Upon a (non-blocking) send, the provider library enqueues a TX request entry to the application queue. The TX request entry contains the application buffer's location and length. ③ The control stack then forwards the entry to the transport protocol. ④ The transport protocol creates packet headers and allows progress according to its flow and congestion control mechanisms. When transmission is allowed, the control stack forwards the constructed header alongside the TX request entry to the NIC queue.

⑤ The NIC parses the TX request entries in-order and DMAs data from the application buffers directly into NIC memory. ⑥ The NIC then merges data with headers to form packets, optionally applying optimizations such as TSO, and transmits packets over the network. ⑦ Upon transmission, the NIC enqueues completion entries back to the NIC queue. ⑧ The control stack polls for completions and forwards them to the application queue. ⑨ Finally, the provider library polls for entries and notifies the application upon completion.

3.2 NIC Hardware Design

Figure 6 presents the NIC hardware design that implements key data structures to split (merge) packets, transfer headers and payloads to (from) the control stack and application, and track payload placement on a per-flow basis so that data can be zero-copied to their correct application buffer.

Memory management hardware data structures. The NIC

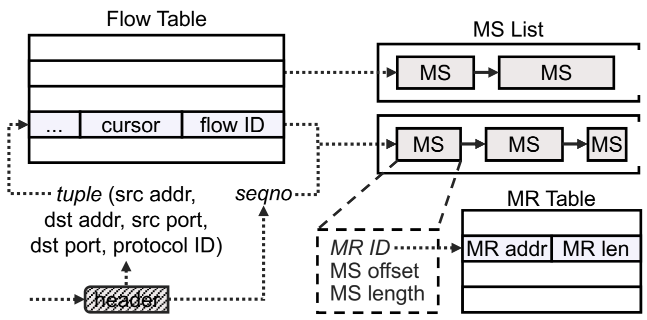


Figure 7: NIC hardware data structures.

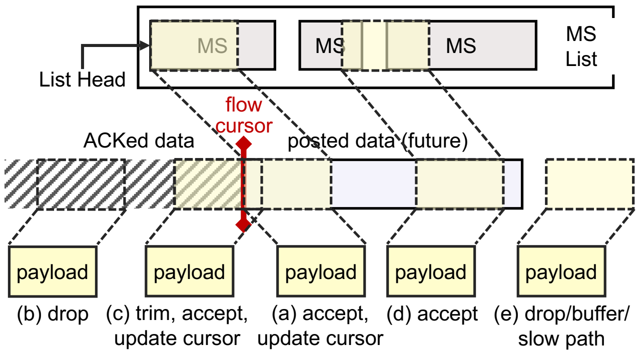


Figure 8: Flow cursor logic given packet arrival conditions.

implements the data structures shown in Figure 7 to track application buffers. At initialization, the application registers a set of *Memory Regions (MR)* using the provider library. An MR is a contiguous part of the application's virtual address space. The NIC maintains an MR Table entry for each registered MR. Virtual MR addresses are translated via an IOMMU in the NIC, which caches translations for efficiency. MRs can be anywhere in the system reachable by the IOMMU. For example, in the context of GPUs, CUDA allocates GPU memory, associates it to the PCIe address space (PCIe BAR), and maps it to host application buffers. In this case, our NIC IOMMU stores CPU to PCIe address space translations. Meanwhile, the NVIDIA driver translates between PCIe and GPU memory, as in GPUDirect.

The control stack creates a *Memory Segment (MS)* for each send and receive operation. An MS corresponds to a contiguous user buffer and is defined by its MR ID, its offset within the MR, and its length[2]. As the application makes asynchronous send and receive calls, the control stack enqueues the RX and TX MSs to the NIC into a per-flow *MS List*. The MS List is essentially a linked-list containing the application buffers involved with pending requests. Note that each flow (and thus MS List) maps to a distinct NIC queue in the control stack.

[2]For simplicity of presentation, we assume contiguous Memory Segments, although MSs may map to more complex data structures.

The *Flow Table* tracks flow metadata used for incoming packets. The most important fields in each *Flow Table* entry are the flow ID used to index to the flow's corresponding MS List, and the *flow cursor*. The flow cursor is the sequence number corresponding to the last in-order consumed packet in the flow. The MS List and the flow cursor are combined to make decisions over the packet's payload, as explained below.

Receiving a packet. The hardware structures described above allow the NIC to map incoming packets to application buffers for zero-copy DMA, as shown in Figure 6. We focus on the process of handling reliable protocols. Unreliable protocols simply land data in the next available MS.

The NIC parses the header of an incoming packet to construct a tuple that indexes the Flow Table (Figure 7), and obtain the corresponding MS List and flow cursor. The MS List and flow cursor are combined with the header's sequence number to derive the packet's position in the flow. The NIC then decides one of four actions: *accept*, *drop*, *buffer*, or *defer*.

We begin with Case (a) in Figure 8. In the absence of network perturbations, the arriving packet contains the next in-order unconsumed payload that the flow is expecting, i.e., the data immediately following the flow cursor. The packet is accepted: the hardware examines the top MS in the flow's MS List and uses the MS and MR information to derive the application address to DMA (zero-copy) the payload. The cursor is updated to reflect the next unconsumed position. MS boundaries and the size of the payload do not have to align. The packet's payload may consume a fraction of the MS or may need to span into the next MS in the list. Fully consumed MSs are retired when the flow cursor passes them.

Case (b) receives a packet with a sequence number that suggests all the bytes in the payload have been previously received and ACKed according to the flow cursor. This may be due to a re-transmission when an ACK is lost or delayed. This packet is dropped and no further action is taken. Case (c) receives a packet that includes some bytes that are previously ACKed and some new bytes. The hardware drops the repeated part and accepts the rest of the packet as in Case (a).

Case (d) receives a packet beyond the flow cursor (i.e., with a hole). This may be the result of packet reordering or a drop of an earlier packet. The hardware will walk the MS List, and by accumulating MS lengths, it will identify the right segment for the data. The data will be accepted and DMAed to the proper application buffer address. Even if future MSs are fully filled, they will not be retired until the cursor passes them. The control stack periodically sends the latest acknowledged byte to the NIC to update the cursor.

Finally, under rare conditions, a packet may not match any MS (Case (e)). This may be the result of excessive drops or the receiver posting buffers at a slow pace. We can drop the packet, buffer it in NIC memory and retry later, or defer the packet to a non-zero-copy path. We implemented the last option (defer) in our prototype system (§4). We also use this approach if a packet arrives for a flow that has no Flow Table entry (i.e., the table has reached its capacity limits).

When a packet is accepted, it is passed to the Split/Merge unit shown in Figure 6, which splits the packet into header and payload. A DMA engine copies payload data directly to application memory (e.g., in CPU user space or GPU device memory). Another DMA engine then forwards headers to the control path address space (e.g., in CPU kernel memory).

Note that the NIC lands data in application memory before any transport processing happens. It also allows the overwrite of future data if a packet is transmitted multiple times since their MSs are not retired. However, correctness is maintained because the application is notified when it is safe to use its buffer by the control path, after protocol processing is done.

Sending a packet. The send hardware path is simple, as packets are sent in the order of requests. Upon receiving a header from the control stack, MSs enter the MS List and are handled by hardware in FIFO order. A DMA engine copies the payload directly from the corresponding application buffer. The Split/Merge unit merges the header and payload to form a packet, which is transmitted over the network.

Hardware requirements for scalability. Maintaining per-flow state raises scalability concerns. Our design supports 10K high-performance flows with ~10MB of NIC memory.

Most proposed structures have a low memory footprint. To support 10K flows, the Flow Table and MR Table require ~700KB and ~100KB respectively, to store all necessary metadata. MS Lists are the most resource-intensive structures. To support long, potentially out-of-order, packet runs with low memory footprint, our NIC does not buffer payloads. Instead, the NIC DMAs future payloads to their correct memory destination by finding the correct MS. For maximum efficiency, we allocate a minimum number of MS List entries per flow to keep the flow pipeline humming, and pull additional MS List entries as needed (a CIR/PIR – committed/peak information rate system) [43].

For example, a large bandwidth-delay-product (BDP) of $100Gbps \cdot 0.2ms = 2.5MB$ would require $\frac{2.5MB}{4KB} = 625$ MSs. Instead of allocating 625 entries for all 10K MS Lists ($10K \cdot 625 \cdot 8B = 50MB$), we allocate a minimum of 128 committed entries to each MS List, while supporting thousands of peak entries (e.g. 8K) that are allocated to flows on demand from a large entry backing store (e.g. 1M entries). For 10K high-performance flows, the total buffer requirement is $\max(1M \cdot 8B, 10K \cdot 128 \cdot 8B) = 9.77MB$. Downsizing MS Lists adds the additional requirement to buffer RX NIC requests during the lifetime of their respective MSs. Besides supporting thousands of zero-copy flows, our design additionally supports non-zero-copy flows that do not occupy the newly proposed data structures.

The required hardware resources for our NIC are significantly lower than those of most RNICs [96]. Modern SmartNICs also require several processor cores, tens of MBs of processor caches, and external memory like DDR, LPDDR, or HBM that can handle payload buffering for high BDPs.

3.3 Control Stack Design

The goal of the control stack is to enable an arbitrary transport protocol with our zero-copy data path, while maintaining efficiency and correctness. The control stack does so by separating and defining a clean interface between three components: *a) the application, b) the transport protocol,* and *c) the NIC*. The control stack maintains connections between each application and the NIC using two sets of queues.

Application queues. The control stack is co-located with the transport protocol [3], *application queues* are allocated in shared memory between the control stack and the application, and connect the provider library with the control stack. Each set of application queues contains a send queue, a receive queue, and their respective completion queues. TX and RX requests are enqueued by the provider library into the send and receive queues, respectively, while the control stack notifies applications upon completions via the completion queue.

NIC queues. The control stack also establishes a set of *NIC queues* for TX and RX requests, incoming RX header entries, and completions. NIC queues connect the control stack to the NIC. They are implemented in the control stack and are accessed by the NIC via DMA. In the send direction, the control stack enqueues MSs and headers to the NIC, constructed from TX requests enabled by the transport protocol. In the receive direction, the control stack sends MSs from RX requests to the NIC. As they are consumed by incoming data, the corresponding headers are split from incoming packets to form RX header entries directed to the control stack.

Supporting arbitrary transport protocols. Current solutions that leverage a single queue pair to provide zero-copy functionality struggle to support protocols not executing in either end of the queue (in the NIC or in user space). In contrast, our control stack uses two separate sets of queues to interpose the transport protocol between the application and NIC. The control stack polls the application send queue for requests and the NIC queue for receive-path headers and invokes the transport protocol to generate send-path headers and acknowledgments, respectively. The control stack can support arbitrary transport protocols by translating application requests to the respective transport API (e.g., TCP sockets).

Enhancing efficiency. In addition to eliminating data copies, the control stack benefits from reduced system call and interrupt overheads when submitting work and receiving completions. Specifically, polling on application queues avoids system calls, resulting in performance benefits similar to mechanisms such as `io_uring` [18]. Since the control stack is co-located with the transport protocol, it directly invokes it without system calls. Similarly, the control stack polls the NIC for completions and headers, avoiding software interrupts. To address applications with sparse communication, mechanisms such as combining polling and doorbells can also be applied.

[3] We assume the control stack executes as a kernel module; §6 discusses supporting transport protocols external to the kernel (e.g., in user space).

Maintaining correctness. Finally, the control stack maintains correctness by logically coupling the physically separated control and data paths. On the sender side, the control stack invokes the protocol's flow and congestion control to enqueue control entries in the NIC and to trigger data DMAs. This is equivalent to the congestion control algorithm acting on headers physically accompanied by their data. On the receiver side, payloads are separated from their headers in the NIC and DMAed directly to application buffers. RX header entries sent to the control stack incorporate information (e.g. sequence number) to bind to their corresponding data. Data becomes visible to the user upon consulting the protocol's acknowledgment policy (e.g., in-order delivery). Thus, the transport protocol maintains ownership of the data without ever touching the data itself, allowing us to reap all the benefits of current transport protocols (robustness, fairness, etc.).

3.4 API Design

The primary goal of our API is to allow current applications (more precisely, current networking libraries) to use our network stack with minimal effort. Current applications use either *message-based* or *streaming* semantics. Message semantics (e.g., RDMA verbs) require the network stack to deliver messages corresponding to contiguous memory buffers. A message size is well-defined by the side initializing communication (one- or two- sided). Streaming interfaces (e.g., sockets) allow senders to continuously transmit byte streams of arbitrary length. The receiver can keep invoking receive calls to consume data in the stream as the network stack progressively signals reception on a byte-stream basis. The stream memory layout can be irregular (non-contiguous) and different on the sender and receiver sides. Our design explicitly supports *both* message-based and streaming semantics. We implement the *libibverbs* API [67] and a socket-like interface.

Supporting message interfaces. Most high-performance applications rely on message semantics [24, 33, 37, 71, 73]. We support their transparent interoperability by implementing the libibverbs API. We dynamically link the libibverbs `verbs_context_ops` to our provider library. The provider in turn connects to our control stack and exposes our application queues to the user as `struct ibv_qp`.

Supporting streaming interfaces. Our design also supports streaming applications by exposing a socket-like API, with slight modifications to support our software stack. The application performs initialization similar to *libibverbs* (find a device, allocate a protection domain for memory regions, and initialize queues). Connection is established via the ordinary socket API (not requiring our fast data path). The above structures are wrapped in a `struct comm_ctx`. `send` and `recv` calls are asynchronous and extended with an argument containing the `comm_ctx`. To relieve the responsibility of registering and de-registering memory regions from the application, our `send` and `recv` calls post their buffer argument as an MR on their invocation. MRs can reside within any endpoint.

3.5 Addressing Challenges

Retransmissions. Section 3.2 explains how the NIC chooses the correct MSs, including when packets are retransmitted. However, in the presence of potential retransmissions, an already consumed MS may need to be reused multiple times. Both TX and RX sides post buffers that ultimately create MSs which must therefore be carefully retired or replenished.

On the sender side, the control stack clones the socket buffer (containing only metadata), before sending it to the transport stack and keeps it alive until the protocol receives an acknowledgment. If the transport decides on retransmitting the packet, the socket buffer is cloned again. The hardware will create the same MS and the retransmission will be accommodated. On the receiver side, the NIC only retires MSs directly following the flow cursor (§3.2), allowing overwrites of future retransmitted data. Permitting overwrites simplifies our retransmission handling logic, especially when arriving packets contain both new and previously delivered data.

Multiple flows. Multiplexing flows in the same NIC queues or MS Lists creates significant complexity in tracking which flow is served on each access. We bypass this issue by assigning NIC queues on a per-flow basis. Before binding with a NIC queue, the control stack creates a flow entry rule in the NIC Flow Table. The unique *flow ID* is used to index NIC queues and MS Lists, ensuring exclusivity. Hence, zero-copy flows are limited to the number of queues supported by the NIC. Despite this issue, we support large enough flow counts with moderate resource requirements (§3.2).

Associating messages with application buffers. Our design supports message semantics with a streaming protocol underneath. In contrast to streams, messages do not have to fully consume a user buffer before using the next one. Thus, there is no clear signal to determine if an incoming packet is the continuation of the currently served message (and MS) or refers to the next message (and MS). We address this issue by adding a message sequence number within the packet transport header (e.g. in the “options” field for TCP). Combined with the stream sequence number, we can point to the correct MS and retire previous MSs that can be consumed even if they were not fully filled.

Compatibility with offload mechanisms. Popular offload optimizations such as GRO/LRO (receiver) and TSO (sender) are compatible with our design. The control stack can transparently support software offload mechanisms like GRO; consecutive headers will be merged into a single socket buffer while their payloads have already been DMAed to consecutive Memory Segments. The user is notified about the latest in-order data, as usual. Similarly for LRO, headers are combined in the NIC after they are split from their payloads. For TSO, to support headers corresponding to more than an MSS (maximum segment size), the NIC segments them into smaller headers. The MS in the send request will now serve for multiple DMAs, one for each segmented header.

4 Implementation

We implemented *ZeroNIC*, an end-to-end prototype of our proposed design. *ZeroNIC* consists of an FPGA-based NIC that implements the key hardware functionality (§3.2), a software control stack that uses TCP as the transport protocol (§3.3), and a provider library exposing the API (§3.4).

NIC. We built a 100*Gbps* Ethernet NIC prototype using a commodity Xilinx Virtex UltraScale+ FPGA [97]. The NIC has three x16 PCIe 3.0 links that connect to an x86 CPU socket and two NVIDIA GPUs (Quadro RTX 4000). In essence, our NIC also acts as a switch between the CPU and the GPUs.

The NIC-to-CPU link is controlled by the QDMA IP [99] from Xilinx that presents the NIC as an endpoint device to the CPU. The CPU is the PCIe root port device. The QDMA block allows for DMA transfers in both directions at full PCIe bandwidth (100*Gbps*). Each NIC-to-GPU link is controlled by the Xilinx XDMA IP [98] that presents the NIC as the root port device to the GPU. The GPU is a PCIe endpoint device. Unfortunately, under this configuration the XDMA block supports a limited number of outstanding PCIe transactions. This imposes a hardware limit on the sustained PCIe bandwidth for DMA transfers between the NIC and the GPU. When moving data from the GPU to the FPGA (GPU is the sender), the maximum PCIe bandwidth is 85.0*Gbps*. When moving data from the FPGA to the NIC (GPU is the receiver), the maximum PCIe bandwidth is 38.6*Gbps*. This limitation of our FPGA system and IP blocks is not fundamental to our design. An ASIC implementation of our NIC would saturate available bandwidth for transfers to GPUs.

Our NIC implements a split/merge unit for the 100*Gbps* Ethernet port. We use context-addressable memories to implement the MR Table and the Flow Table that is addressed by the 5-tuple from the TCP/IP header (source and destination addresses and ports, and protocol ID). The split-merge unit connects to the PCIe ports. The *ZeroNIC* design is modular and can be extended to support multiple 100*Gbps* Ethernet ports using replicated split/merge units. It can also support more root-ports in order to connect more than two GPUs.

Control Stack. We implemented the control stack as a Linux kernel driver, which binds the provider library with the NIC. The driver directly invokes the *unmodified Linux kernel TCP stack* for protocol processing, translating application requests and NIC queue entries into Linux TCP socket calls. While our design supports arbitrary protocols and execution locations, we select the kernel TCP protocol for the first prototype as it is robust, but challenging to make performant (see §2.2). Application queues live in shared memory between the kernel and provider library. NIC queues live in the kernel's virtual network device. Both are implemented as ring buffers of user-configurable sized entries.

Table 1: Evaluation system setup.

System	TCP / RoCE Baselines	*ZeroNIC*
NIC	Mellanox ConnectX-6	Prototype built on Xilinx Virtex UltraScale+
Topology	2-node direct-conn 100G eth	2-node direct-conn 100G eth (38.6G max for GPU)
Protocol	TCP bbr / RoCEv2 RC	TCP bbr
Setup	TCP: TSO, LRO, 9K MTU RoCE: 4K MTU	TSO, GRO, 9K MTU (always TCP)
CPU	32 core AMD EPYC 7502 L1,L2,L3: 2MB,16MB,128MB	32 core AMD EPYC 7502 L1,L2,L3: 2MB,16MB,128MB

5 Evaluation

We evaluate the efficiency of our host networking approach using the *ZeroNIC* prototype, with the Linux kernel's TCP transport in our control stack. We compare the performance of *ZeroNIC* against two popular baselines: a TCP baseline that uses the Linux network stack without our high-performance data path, and a RoCE baseline that terminates the transport protocol in the NIC. Both baselines use a Mellanox ConnectX-6 NIC. We summarize the specific configurations of these systems in Table 1. All *ZeroNIC* measurements utilize large-segment offloading (TSO and GRO) and jumbo frames, unless otherwise specified. We do not require MTUs to be page-aligned.

5.1 *ZeroNIC* Throughput Evaluation

***ZeroNIC* provides RDMA-level throughput to application buffers in CPU memory at low CPU utilization.** Table 2 shows the throughput achieved by *ZeroNIC* for a single flow between a sender and a receiver application using CPU memory. We compare against the Mellanox RoCE baseline (MLX RoCE), as well as against kernel TCP using the Mellanox NIC (MLX TCP) with and without send-side zero-copy (TX ZC on/off). We enabled send-side zero-copy for Mellanox TCP as discussed in §2.2. For *ZeroNIC* and Mellanox TCP, we pin the protocol processing thread, and either the queue polling (for *ZeroNIC*) or the software interrupt handling (for Mellanox TCP) thread to hyperthreads in the same physical core to maximize cache locality.

Table 2 breaks down the receiver-side CPU utilization between the kernel (`sys`) and other CPU cycles (`usr/soft`). `sys` includes protocol processing and the *ZeroNIC* driver, while `usr/soft` includes the *ZeroNIC* provider library, interrupts, and the application itself. Note that 100% CPU utilization means that *a single CPU hyperthread* (2 per core) is fully utilized. RoCE offloads protocol processing to the RNIC and

Table 2: Throughput and receiver-side CPU utilization for CPU-to-CPU transfers. "CPU sys" refers to the hyperthread running protocol processing and *ZeroNIC*'s driver. "CPU usr/soft" refers to the hyperthread running the application, software interrupt handler, and *ZeroNIC*'s provider library.

System	Throughput (Gbps)	CPU sys (%)	CPU usr/soft (%)	Estimated max Tput
MLX TCP TX ZC off	43.89 ± 1.35	94.15 ± 3.45	29.55 ± 2.62	46.61
MLX TCP TX ZC on	50.63 ± 0.55	100.0 ± 0.00	32.36 ± 0.80	50.63
MLX RoCE	98.03 ± 0.00	N/A	9.58 ± 0.81	N/A
ZeroNIC	96.37 ± 0.60	17.20 ± 1.96	33.50 ± 1.11	560.29

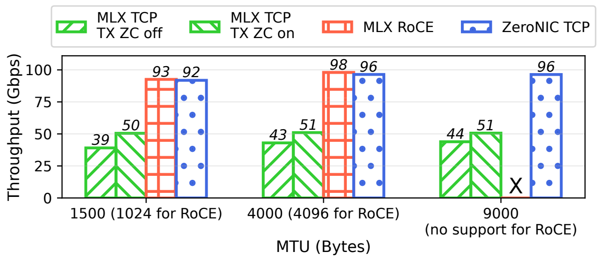


Figure 9: Achieved throughput across multiple MTU sizes. RoCE only supports power-of-two MTU sizes up to 4096B.

lands data directly into application buffers. Thus, we do not observe `sys` CPU utilization. For *ZeroNIC* and RoCE, the throughput benchmark polls for completions. We do not include cycles spent on the polling loop for *ZeroNIC* and RoCE, as cycles spent polling do not limit throughput (e.g. the RoCE application hyperthread shows as 100% utilized).

We observe that Mellanox TCP is constrained by CPU utilization, despite the optimizations used (TSO, LRO, 9K MTU). The Linux TCP stack uses a single thread for protocol processing for each flow. With send-side zero-copy disabled, the sender-side protocol processing thread saturates (not shown), while the receiver thread almost reaches full utilization at 43*Gbps*. Enabling sender zero-copy exposes the receive-side bottleneck as the receiver thread saturates at 50*Gbps*.

ZeroNIC copies RX data directly to user space application buffers, eliminating the CPU cycles spent on data copy as shown in Figure 2. This reduces the protocol processing thread's CPU utilization from 100% at 50.63*Gbps* to 17.20% at 96.37*Gbps*. *ZeroNIC* also eliminates the majority of system calls via the control stack's polling architecture (§3.3), achieving an even lower `usr/soft` utilization than what baseline TCP is projected to need at 100*Gbps* (§3.3, Figure 1). This allows *ZeroNIC* to reach a throughput comparable to RoCE. However, *ZeroNIC* maintains the flexibility of the Linux stack,

Table 3: Achieved throughput and receiver-side CPU utilization for communication across different CPU/GPU endpoints.

System	Throughput (Gbps)	CPU sys (%)	CPU usr/soft (%)	Estimated max Tput
ZeroNIC CPU-CPU	96.37 ± 0.60	17.20 ± 1.96	33.50 ± 1.11	560.29
ZeroNIC CPU-GPU	84.78 ± 0.41 [4]	16.31 ± 0.54	36.33 ± 2.21	519.80
ZeroNIC GPU-GPU	38.59 ± 0.07 [4]	9.12 ± 0.21	32.50 ± 2.07	423.14

while RoCE implements its entire control path in the RNIC.

ZeroNIC is now bound by the link capacity. Given additional or faster links, *ZeroNIC* can scale beyond 100*Gbps*. The last column in Table 2 estimates the maximum throughput that *ZeroNIC* can achieve with the kernel TCP stack, by scaling the protocol processing thread (`sys`) to saturate CPU utilization (indeed, as we will see in Table 3, the `usr/soft` thread has minor variations for different peak bandwidth settings). *ZeroNIC* is projected to scale to > 500*Gbps* for a single flow of the kernel TCP stack. This is a $11\times$ higher throughput than the current TCP network stack achieves for a single flow using the Mellanox NIC.

Finally, Figure 9 demonstrates that *ZeroNIC*'s benefits hold across various MTU sizes. For smaller MTUs (1500 or 1024 bytes), throughput on both *ZeroNIC* and RoCE slightly reduces due to higher packets-per-second DMA overheads.

***ZeroNIC* enables high-throughput data transfers directly to device (GPU) memory.** *ZeroNIC* is able to extend zero-copy benefits to arbitrary endpoints, including GPUs. Hence, *ZeroNIC* can directly transfer data from and to GPU HBM, bypassing the host CPU memory, similar to GPUDirect [22]. Table 3 presents *ZeroNIC*'s single-flow throughput for CPU-to-CPU, CPU-to-GPU, and GPU-to-GPU communication. In all cases, the control path uses the Linux TCP stack. *ZeroNIC* is able to saturate the bandwidth supported by the hardware on all paths, given the prototype IP limitations discussed in Section 4: ~100*Gbps* for CPU-to-CPU, 85*Gbps* for CPU-to-GPU, and 38.6*Gbps* for GPU-to-GPU transfers.

To validate that the *ZeroNIC* design scales to higher throughput in the absence of prototype limitations, Table 3 also reports CPU utilization. As in Table 2, we split CPU utilization between protocol processing and driver (`sys`) and other cycles (`usr/soft`). As we can see by comparing the CPU-to-CPU and GPU-to-GPU results, the `usr/soft` CPU cycles do not strongly scale with maximum throughput. The limiting factor for higher throughput for a single flow would be the protocol processing overheads of the kernel's TCP

[4]This is the maximum throughput supported by our hardware prototype due to FPGA IP limitations (Section 4).

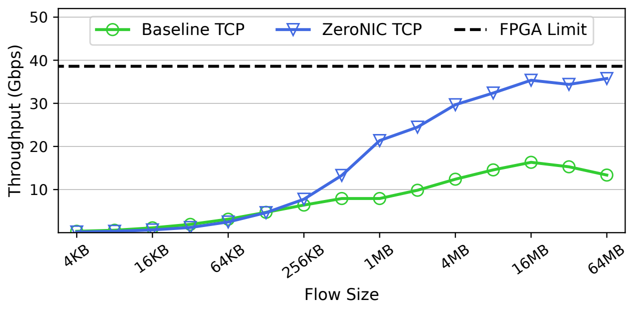


Figure 10: NCCL throughput using *ZeroNIC* across different flow sizes.

stack. Hence, we calculate the maximum throughput *ZeroNIC* can reach when the kernel thread saturates (100% utilization) to be above 400*Gbps* for CPU-to-CPU, CPU-to-GPU, and GPU-to-GPU flows, This means that *ZeroNIC* can replace the RoCE back-end network in GPU-based AI clusters that is used to communicate activations and gradients during AI training [75]. A *ZeroNIC*-based system with 8 GPUs would require 8 CPU hyperthreads to support a total of 3.2*Tbps* of GPU-to-GPU networking, while gaining the benefits of using any network protocol, such as the robustness of TCP.

5.2 End-to-End Workloads

ZeroNIC supports popular APIs (§ 3.4) that enable application integration *without modifications*, simply by linking against our provider library. We demonstrate this using three important workloads; NCCL benchmarks, PyTorch, and Redis.

NCCL. NCCL [24] is the dominant communication library for distributed AI using GPUs. It implements and optimizes collective communication primitives that are commonly used in AI training and multi-GPU inference. Different phases of inter-node collective communication (all-reduce in data parallelism, all-to-all in expert parallelism, point-to-point in pipeline parallelism, etc.) use tens of megabytes as their collective bucket size [58, 64, 85]. The number of flows scales with the number of nodes (N). For example, NCCL's tree algorithm, the predominant inter-node collective implementation algorithm, creates $2\log N$ flows per node [19]. Exposed communication increases with system size [88], making network performance critical, especially as cluster sizes increase beyond 10,000 GPUs and 1,000 nodes [20]. Improving NCCL performance directly reduces exposed communication, leading to faster AI training and inference [41, 70, 82, 100].

We ran the broadcast NCCL benchmark [23]. For 2 nodes, broadcast sends the full collective size unidirectionally between two *ZeroNIC* GPU servers. Since broadcast is the core primitive used to build other collectives, improved broadcast throughput directly translates to higher collective throughput in general. We compare *ZeroNIC* to a baseline TCP implementation which uses the *ZeroNIC* NIC hardware, but always forwards the entire packet directly to the unmodified Linux network stack (no zero-copy).

Table 4: Average training epoch latency (in seconds) for different PyTorch distributed data parallel models using RoCE GPUDirect and *ZeroNIC*.

System	ResNet50	ResNet101	ResNet152
MLX RoCE	3.52 ± 0.04	6.12 ± 0.07	8.80 ± 0.04
ZeroNIC	3.57 ± 0.02	6.22 ± 0.08	8.83 ± 0.08

Figure 10 shows the throughput achieved by NCCL as we vary the collective size. For small sizes, *ZeroNIC* and the baseline deliver the same throughput. The throughput bottleneck for small collectives is actually NCCL itself. It implements a higher-level protocol with significant processing overheads that cannot saturate the link with a single flow for small collectives, regardless of whether RDMA or TCP is used. As the collective size increases, the bottleneck becomes packet processing in the TCP stack. For the baseline TCP (no zero-copy), NCCL saturates at ~16*Gbps* for flows beyond 16*MB*. For large collective sizes, *ZeroNIC* manages to hit the maximum throughput allowed by our FPGA prototype, 2.66× higher than the baseline. If the FPGA limitation is removed, *ZeroNIC* will saturate the Ethernet link. These results show that the *ZeroNIC* data path is especially powerful for devices such as GPUs. It eliminates two data copies: a copy from the kernel buffer to the application buffer and a copy from the CPU-based application buffer to a GPU buffer.

PyTorch. PyTorch [78] is the most popular AI framework. For distributed training, PyTorch implements various parallelization strategies, leveraging communication backends such as NCCL. For example, in data parallelism, training data is partitioned while each node holds a full copy of the model. During each iteration's backward pass, all model gradients are averaged across all ranks using the all-reduce collective.

We trained different sizes of ResNet [42] using PyTorch's distributed data parallelism [81] with NCCL. We compared the average training epoch latency on two *ZeroNIC* nodes using TCP, against two Mellanox nodes using RoCE with GPUDirect [22]. Our baselines, ResNet50, ResNet101, and ResNet152 are composed of 25.6, 45.5, and 60.2 million parameters, and require synchronizing 51.2, 91.0, and 120.4 MBs worth of gradients in every iteration, respectively. Each epoch is composed of 100 iterations. Table 4 shows that *ZeroNIC* achieves GPUDirect-level performance, within 2% of RoCE's latency.

As NCCL supports the IB verbs API, we ran both the PyTorch and NCCL experiments on *ZeroNIC* without any application/library modifications. These results demonstrate that our design can be effortlessly used in AI clusters that rely on high performance, while maintaining the flexibility and robustness of the Linux network stack.

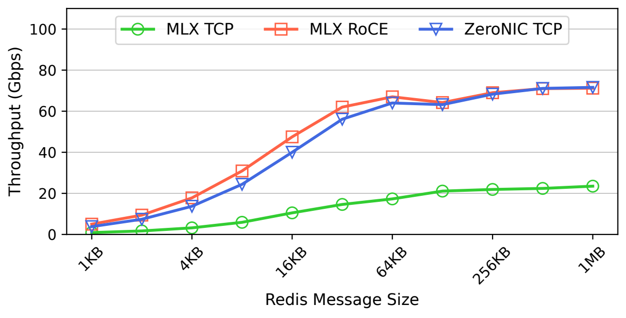


Figure 11: Redis throughput across different payload sizes.

Redis. Redis [83] is an in-memory key-value database widely used for in-memory caching. Redis' performance is crucial for a wide range of data-intensive web-scale applications. We used Redis with the *libibverbs* API, and we evaluated its throughput on *ZeroNIC* against Mellanox RoCE (MLX RoCE) and kernel TCP using the Mellanox NIC (MLX TCP). We ran the *redis-benchmark* [84] using the `SET` operation and varied the payload size from $1KB$ to $1MB$. We used a total of 4 clients with 10 outstanding requests per client in order to saturate the Redis server thread.

Figure 11 illustrates that *ZeroNIC* achieves end-to-end performance on par with RoCE, averaging 89% of RoCE's throughput across all the payload sizes. Compared to Mellanox TCP, *ZeroNIC* achieved a $3.71\times$ higher throughput on average, benefiting from the lower CPU overheads. For instance, for the $16KB$ message size, 71% of CPU cycles are consumed by networking stack overhead for Mellanox TCP. In contrast, both *ZeroNIC* and Mellanox with RoCE allow for nearly 99% of the cycles to be dedicated to application-level processing.

5.3 *ZeroNIC* Robustness Evaluation

While *ZeroNIC* achieves high throughput, it *also* gains the robustness offered by transport protocols such as TCP. To demonstrate the benefits of a flexible control path, we evaluate *ZeroNIC* under various network perturbations and conditions.

***ZeroNIC* supports interleaved packets across multiple zero-copy and non-zero-copy flows.** To evaluate *ZeroNIC*'s ability to handle and scale to multiple flows, we performed an incast experiment combining 2MB zero-copy flows, and 64KB bidirectional short flows that used the unmodified non-zero-copy socket API. The receiver *ZeroNIC* server used a single core (two hyperthreads) to perform application and network processing for all zero-copy flows.

Figure 12 shows (a) the throughput and CPU utilization for the long flows and (b) the p50 latency for the short flows. *ZeroNIC* is able to steer interleaved incoming packets to their correct NIC queues, avoiding flow collision. It maintains fairness across all flows, evenly distributing bandwidth of up to 8 zero-copy flows, the FPGA's hardware limit. Meanwhile, total CPU utilization (protocol processing, driver, provider, and application) remains approximately constant, and the aggregate throughput across flows saturates the available bandwidth. Overall, *ZeroNIC* achieves high throughput with roughly constant CPU resource demands as the flow count scales. This result, together with our moderate hardware memory requirements to support thousands of high-performance flows (§3.2), validates our design's scalability. Additionally, regardless of the number of high-performance flows, *ZeroNIC* can still concurrently support non-zero-copy flows. Figure 12b shows that their latency is not affected by the number of long flows.

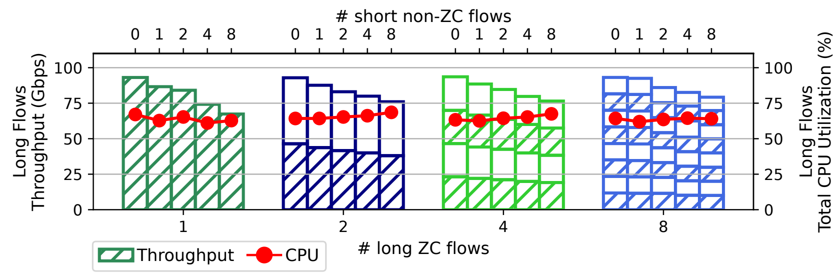
***ZeroNIC* extends the drop-resistance of TCP to GPU-direct data paths.** A primary motivation for physically separating the control and data paths is to combine zero-copy performance with a mature network transport. This experiment examines packet losses, which is a significant problem for RDMA solutions, like RoCE, that were designed assuming a lossless fabric (§2.1). We injected packet drops by adding probabilistic filtering rules at the RX side for both *ZeroNIC* and the Mellanox TCP baseline with TX zero-copy. We could not replicate this experiment for RoCE because none of our available drop rules [35, 66, 92] could intercept RDMA traffic.

Figure 13 shows the throughput of a single GPU-to-GPU flow as we raised the probabilistic drop rate from 0.1% to 10% for the two systems. As expected, both TCP-based systems perform well at low drop rates. *ZeroNIC* maintains near-full throughput even at 1% drops, taking advantage of TCP's mechanisms for drop resistance (retransmitting minimal data). RoCE is known to collapse to near-zero throughput at 1% drops due to the use of go-back-N for retransmissions [69, 103].

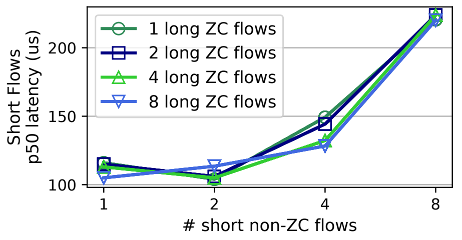
Adding drop resistance to GPU-to-GPU traffic is particularly important for AI clusters. It removes the pressure to design a perfect congestion control mechanism and to oversize switch buffers. It also allows switch chips to be configured to use cut-though switching instead of store-and-forward switching. The latter is forced by the need for forward-error-correction (FEC) in order to reduce noise-induced packet losses to zero.

6 Discussion

Zero-copy is critical but is not a panacea. We demonstrated that a data path with both receive and send-side zero-copy improves host networking even with mature network protocols. However, as network links scale to $800Gbps$ and beyond for workloads like artificial intelligence, the control path will become the next bottleneck. Recent proposals to reduce packet processing overheads include hardware offload [40,60], system call mitigation [18, 31], extending segmentation offload [16, 29, 52], and cache-aligned reorganization [61]. To improve metadata I/O between the NIC and software, systems like Enso [86] and PacketMill [30] introduce optimizations



(a) Long flow (zero-copy) throughput and CPU utilization under collision with short flows.



(b) Short flow (non-zero-copy) p50 latency.

Figure 12: Robustness experiment with long (zero-copy) and short (non-zero-copy) CPU-to-CPU flows colliding in *ZeroNIC*.

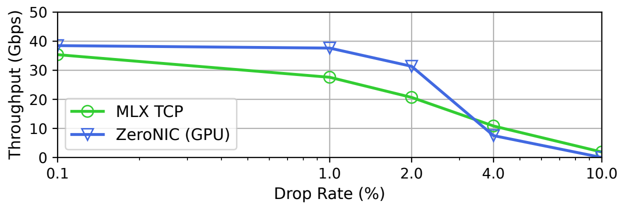


Figure 13: Throughput under instrumented drops in *ZeroNIC* (GPU-to-GPU) and Mellanox TCP (CPU-to-CPU).

for the efficient use of PCIe bandwidth. Our design is well-positioned to help with and benefit from this evolution.

Our design enables an agile evolution of the data and control path. Our design creates a triangle between the NIC, data path devices, and control path devices with well-defined interfaces and responsibilities. This allows the fast and largely independent evolution and optimization of control and data path devices. For example, a host architect can quickly swap GPUs for other AI accelerators without re-implementing or re-optimizing the network data path or the control path.

Control path flexibility is equally important. Achieving the right balance of features and resources integrated in the NIC is difficult. Unlike RNICs that jointly implement the control and data paths in inflexible hardware or opaque firmware, our NIC design implements only a necessary subset of features to support remotely executed control paths. Via our design, a system architect may use a different network protocol in order to improve fabric performance (high utilization, reduced drops, reduced hot spots, etc.) or use a specialized hardware component for faster header processing (CPU with specialized cores, FPGAs, or specialized accelerator). Our design facilitates changes in the control path of the triangle without necessitating changes in the performant data path or the application layer.

The bounds of the maximum bandwidth and minimum latency of communication between the elements of the triangle are set by the link specifications that connect them. Our prototype uses PCIe links and inherits PCIe's bandwidth and latency profiles. As higher throughput and/or lower latency links, such as CXL and NVLink, gain acceptance in industry, our design will benefit from their characteristics.

7 Conclusion

Current end-host network stacks inherently couple the control and data path, resulting in implicit trade-offs between the flexibility and performance of network solutions. In this paper, we showed that a physical separation of the data and control paths allows host network stacks to achieve *both* flexibility and performance. To this end, we presented a co-designed hardware and software stack, which enables a zero-copy receive and send data path between the NIC and any device memory, controlled by any arbitrary transport protocol. We showcased *ZeroNIC*, a prototype that combines an FPGA-based NIC and the TCP stack in the Linux kernel as the transport protocol. Our prototype saturates available network bandwidth on CPU and GPU benchmarks. It improves TCP-based NCCL and Redis throughput by $2.66\times$ and $3.71\times$, respectively, over Linux TCP on a Mellanox ConnectX-6, all while maintaining the robustness of the TCP transport.

Acknowledgments

We are grateful to the anonymous reviewers and to our shepherd, Aurojit Panda, whose comments have greatly helped improve this paper. We would also like to acknowledge the contributions of Antonis Michaloliakos, Grigori Inozemtsev, George Prekas, and many others at Enfabrica who have played a vital role in this endeavor. This research was partly supported by the Stanford Platform Lab and its affiliates, and by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. This research was also partly supported by NSF CNS-2047283. Athinagoras Skiadopoulos was supported by a Stanford Graduate Fellowship. Mark Zhao was supported by a Stanford Graduate Fellowship and a Meta Ph.D. Fellowship. Qizhe Cai was supported by a Meta Ph.D. Fellowship.

References

[1] Saksham Agarwal, Rachit Agarwal, Behnam Montazeri, Masoud Moshref, Khaled Elmeleegy, Luigi Rizzo, Marc Asher de Kruijf, Gautam Kumar, Sylvia Ratnasamy, David Culler, and Amin Vahdat. Understanding host interconnect congestion. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*, pages 198–204, 2022.

[2] Saksham Agarwal, Arvind Krishnamurthy, and Rachit Agarwal. Host congestion control. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 275–287, 2023.

[3] David Ahern and Shrijeet Mukherjee. Merging the networking worlds. In *The Technical Conference on Linux Networking (Netdev 0x16)*, 2023.

[4] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, page 63–74, New York, NY, USA, 2010. Association for Computing Machinery.

[5] AMD. Amd hbm, 2023. https://www.amd.com/en/technologies/hbm.

[6] InfiniBand Trade Association. Infiniband architecture specification, 2023. https://www.infinibandta.org/ibta-specification/.

[7] InfiniBand Trade Association. Infiniband architecture specification: Rocev2, 2023. https://www.infinibandta.org/ibta-specification/.

[8] 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 *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 49–67, 2023.

[9] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. {IX}: a protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, 2014.

[10] Lawrence Brakmo. Tcp-bpf: Programmatically tuning tcp behavior through bpf. In *The Technical Conference on Linux Networking (Netdev 2.2)*, 2017.

[11] L.S. Brakmo and L.L. Peterson. Tcp vegas: end to end congestion avoidance on a global internet. *IEEE Journal on Selected Areas in Communications*, 13(8):1465–1480, 1995.

[12] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. Understanding host network stack overheads. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 65–77, New York, NY, USA, 2021. Association for Computing Machinery.

[13] Qizhe Cai, Midhul Vuppalapati, Jaehyun Hwang, Christos Kozyrakis, and Rachit Agarwal. Towards μs tail latency and terabit ethernet: disaggregating the host network stack. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 767–779, 2022.

[14] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. Bbr: Congestion-based congestion control. *ACM Queue*, 14, September-October:20 – 53, 2016.

[15] Solarflare Communications. Onload user guide, issue 20, 2015. Section D.6 at https://www.smallake.kr/wp-content/uploads/2015/12/SF-104474-CD-20_Onload_User_Guide.pdf.

[16] Jonathan Corbet. Jls2009: Generic receive offload, 2009. https://lwn.net/Articles/358910/.

[17] Jonathan Corbet. A reworked tcp zero-copy receive api, 2018. https://lwn.net/Articles/754681/.

[18] Jonathan Corbet. The rapid growth of io_uring, 2020. https://lwn.net/Articles/810414/.

[19] NVIDIA Corporation. Massively scale your deep learning training with nccl 2.4, 2019. https://developer.nvidia.com/blog/massively-scale-deep-learning-training-nccl-2-4/.

[20] NVIDIA Corporation. Massively scale your deep learning training with nccl 2.4, 2019. https://developer.nvidia.com/blog/massively-scale-deep-learning-training-nccl-2-4/.

[21] NVIDIA Corporation. Nvidia dgx superpod: Scalable infrastructure for ai leadership, 2021. https://images.nvidia.com/aem-dam/Solutions/Data-Center/gated-resources/nvidia-dgx-superpod-a100.pdf.

[22] NVIDIA Corporation. Gpudirect rdma, 2023. https://docs.nvidia.com/cuda/gpudirect-rdma/.

[23] NVIDIA Corporation. NCCL Tests, 2023. https://github.com/NVIDIA/nccl-tests.

[24] NVIDIA Corporation. Nvidia nccl, 2023. https://developer.nvidia.com/nccl.

[25] NVIDIA Corporation. Nvidia blackwell architecture, 2024. https://www.nvidia.com/en-us/data-center/technologies/blackwell-architecture/.

[26] NVIDIA Corporation. Nvidia dgx b200, 2024. https://www.nvidia.com/en-us/data-center/dgx-b200/.

[27] Linux Networking Documentation. Msg_zerocopy, 2023. https://www.kernel.org/doc/html/v4.15/networking/msg_zerocopy.html.

[28] Jon Dugan, Seth Elliott, Bruce A Mah, Jeff Poskanzer, and Kaustubh Prabhu. iperf – the ultimate speed test tool for tcp, udp and sctp. 2021.

[29] Eric Dumazet and Coco Li. Big tcp. In *The Technical Conference on Linux Networking (Netdev 0x15)*, 2021.

[30] Alireza Farshin, Tom Barbette, Amir Roozbeh, Gerald Q Maguire Jr, and Dejan Kostić. Packetmill: toward per-core 100-gbps networking. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1–17, 2021.

[31] The Linux Foundation. Linux foundation docuwiki: napi, 2016. https://wiki.linuxfoundation.org/networking/napi.

[32] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020.

[33] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al. Open mpi: Goals, concept, and design of a next generation mpi implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 11th European PVM/MPI Users' Group Meeting Budapest, Hungary, September 19-22, 2004. Proceedings 11*, pages 97–104. Springer, 2004.

[34] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, et al. When cloud storage meets {RDMA}. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 519–533, 2021.

[35] Steven Gordon. baidu-research/tensorflow-allreduce, 2013. https://github.com/baidu-research/tensorflow-allreduce.

[36] Swati Goswami, Nodir Kodirov, Craig Mustard, Ivan Beschastnikh, and Margo Seltzer. Parking packet payload with p4. In *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies*, pages 274–281, 2020.

[37] OpenFabrics Interfaces Working Group. Libfabric openfabrics, 2023. https://ofiwg.github.io/libfabric/.

[38] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. Rdma over commodity ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 202–215, New York, NY, USA, 2016. Association for Computing Machinery.

[39] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: A new tcp-friendly high-speed tcp variant. *SIGOPS Oper. Syst. Rev.*, 42(5):64–74, jul 2008.

[40] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. Packetshader: a gpu-accelerated software router. *ACM SIGCOMM Computer Communication Review*, 40(4):195–206, 2010.

[41] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, and Phil Gibbons. Pipedream: Fast and efficient pipeline parallel dnn training. *arXiv preprint arXiv:1806.03377*, 2018.

[42] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[43] Juha Heinanen and Roch Guerin. Rfc2698: A two rate three color marker, 1999.

[44] Torsten Hoefler, Duncan Roweth, Keith Underwood, Bob Alverson, Mark Griswold, Vahid Tabatabaee, Mohan Kalkunte, Surendra Anubolu, Siyan Shen, Abdul Kabbani, et al. Datacenter ethernet and rdma: Issues at hyperscale. *arXiv preprint arXiv:2302.03337*, 2023.

[45] Shuihai Hu, Yibo Zhu, Peng Cheng, Chuanxiong Guo, Kun Tan, Jitendra Padhye, and Kai Chen. Deadlocks in datacenter networks: Why do they form, and how to avoid them. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, pages 92–98, 2016.

[46] IEEE802. 802.1qbb - priority-based flow control, 2011. https://1.ieee802.org/dcb/802-1qbb/.

[47] V. Jacobson. Congestion avoidance and control. *SIGCOMM Comput. Commun. Rev.*, 18(4):314–329, aug 1988.

[48] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a highly scalable user-level TCP stack for multicore systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 489–502, Seattle, WA, April 2014. USENIX Association.

[49] Norman P Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, et al. Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings. *arXiv preprint arXiv:2304.01433*, 2023.

[50] Magnus Karlsson and Björn Töpel. The path to dpdk speeds for af xdp. In *Linux Plumbers Conference*, 2018.

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

[52] The kernel development community. Segmentation offloads, 2016. https://www.kernel.org/doc/Documentation/networking/segmentation-offloads.txt.

[53] The kernel development community. Msg_zerocopy, 2017. https://www.kernel.org/doc/html/v4.15/networking/msg_zerocopy.html.

[54] 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 *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 113–126, 2019.

[55] Xinhao Kong, Yibo Zhu, Huaping Zhou, Zhuo Jiang, Jianxi Ye, Chuanxiong Guo, and Danyang Zhuo. Collie: Finding performance anomalies in {RDMA} subsystems. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 287–305, 2022.

[56] Adithya Kumar, Anand Sivasubramaniam, and Timothy Zhu. Splitrpc: A {Control+ Data} path splitting rpc stack for ml inference serving. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 7(2):1–26, 2023.

[57] Yanfang Le, Brent Stephens, Arjun Singhvi, Aditya Akella, and Michael M Swift. Rogue: Rdma over generic unconverged ethernet. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 225–236, 2018.

[58] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668*, 2020.

[59] Bojie Li, Tianyi Cui, Zibo Wang, Wei Bai, and Lintao Zhang. Socksdirect: Datacenter sockets can be fast and compatible. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 90–103. 2019.

[60] Bojie Li, Kun Tan, Layong Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 1–14, 2016.

[61] Coco Li. Analyze and reorganize core networking structs to optimize cacheline consumption, 2023. https://lore.kernel.org/netdev/20231129072756.3684495-1-lixiaoyan@google.com/.

[62] Qiang Li, Yixiao Gao, Xiaoliang Wang, Haonan Qiu, Yanfang Le, Derui Liu, Qiao Xiang, Fei Feng, Peng Zhang, Bo Li, et al. Flor: An open high performance {RDMA} framework over heterogeneous {RNICs}. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 931–948, 2023.

[63] Qiang Li, Qiao Xiang, Yuxin Wang, Haohao Song, Ridi Wen, Wenhui Yao, Yuanyuan Dong, Shuqi Zhao, Shuo Huang, Zhaosheng Zhu, et al. More than capacity: Performance-oriented evolution of pangu in alibaba. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 331–346, 2023.

[64] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704*, 2020.

[65] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, et al. Hpcc: High precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 44–58. 2019.

[66] NVIDIA Cumulus Linux. Buffer and queue management, 2023. https://docs.nvidia.com/networking-ethernet-software/cumulus-linux-43/Layer-1-and-Switch-Ports/Buffer-and-Queue-Management/.

[67] linux rdma. Rdma-core libibverbs, 2021. https://github.com/linux-rdma/rdma-core.

[68] Shao Liu, Tamer Başar, and Ravi Srikant. Tcp-illinois: A loss and delay-based congestion control algorithm for high-speed networks. In *Proceedings of the 1st international conference on Performance evaluation methodolgies and tools*, pages 55–es, 2006.

[69] Yuanwei Lu, Guo Chen, Zhenyuan Ruan, Wencong Xiao, Bojie Li, Jiansong Zhang, Yongqiang Xiong, Peng Cheng, and Enhong Chen. Memory efficient loss recovery for hardware-based transport in datacenter. In *Proceedings of the First Asia-Pacific Workshop on Networking*, APNet'17, page 22–28, New York, NY, USA, 2017. Association for Computing Machinery.

[70] Liang Luo, Jacob Nelson, Luis Ceze, Amar Phanishayee, and Arvind Krishnamurthy. Parameter hub: a rack-scale parameter server for distributed deep neural network training. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 41–54, 2018.

[71] Piotr Luszczek, Jack J Dongarra, David Koester, Rolf Rabenseifner, Bob Lucas, Jeremy Kepner, John McCalpin, David Bailey, and Daisuke Takahashi. Introduction to the hpc challenge benchmark suite. Technical report, Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), 2005.

[72] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: A microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 399–413, New York, NY, USA, 2019. Association for Computing Machinery.

[73] Dave Minturn. Nvm express over fabrics. In *11th Annual OpenFabrics International OFS Developers' Workshop*, 2015.

[74] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. Revisiting network support for rdma. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 313–326, 2018.

[75] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Zhihao Jia, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, Liang Luo, Jie (Amy) Yang, Leon Gao, Dmytro Ivchenko, Aarti Basant, Yuxi Hu, Jiyan Yang, Ehsan K. Ardestani, Xiaodong Wang, Rakesh Komuravelli, Ching-Hsiang Chu, Serhat Yilmaz, Huayu Li, Jiyuan Qian, Zhuobo Feng, Yinbin Ma, Junjie Yang, Ellie Wen, Hong Li, Lin Yang, Chonglin Sun, Whitney Zhao, Dimitry Melts, Krishna Dhulipala, KR Kishore, Tyler Graf, Assaf Eisenman, Kiran Kumar Matam, Adi Gangidi, Guoqiang Jerry Chen, Manoj Krishnan, Avinash Nayak, Krishnakumar Nair, Bharath Muthiah, Mahmoud khorashadi, Pallab Bhattacharya, Petr Lapukhov, Maxim Naumov, Ajit Mathews, Lin Qiao, Mikhail Smelyanskiy, Bill Jia, and Vijay Rao. Software-hardware co-design for fast and scalable training of deep learning recommendation models. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 993–1011, New York, NY, USA, 2022. Association for Computing Machinery.

[76] 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. *IEEE/ACM Transactions on Networking*, 30(3):999–1013, 2021.

[77] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019.

[78] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[79] Simon Peter, Jialin Li, Irene Zhang, Dan RK Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. *ACM Transactions on Computer Systems (TOCS)*, 33(4):1–30, 2015.

[80] Boris Pismenny, Liran Liss, Adam Morrison, and Dan Tsafrir. The benefits of general-purpose on-nic memory. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1130–1147, 2022.

[81] PyTorch. Pytorch distributed data parallel, 2023. https://pytorch.org/docs/stable/generated/torch.nn.parallel.DistributedDataParallel.html.

[82] Saeed Rashidi, Matthew Denton, Srinivas Sridharan, Sudarshan Srinivasan, Amoghavarsha Suresh, Jade Nie, and Tushar Krishna. Enabling compute-communication overlap in distributed deep learning training platforms. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 540–553. IEEE, 2021.

[83] Redis. Redis, 2023. https://redis.io.

[84] Redis. Redis benchmark, 2023. https://redis.io/docs/management/optimization/benchmarks/.

[85] Joshua Romero, Junqi Yin, Nouamane Laanait, Bing Xie, M Todd Young, Sean Treichler, Vitalii Starchenko, Albina Borisevich, Alex Sergeev, and Michael Matheson. Accelerating collective communication in data parallel training across deep learning frameworks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1027–1040, 2022.

[86] 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 *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 1005–1025, 2023.

[87] Mariano Scazzariello, Tommaso Caiazzi, Hamid Ghasemirahni, Tom Barbette, Dejan Kostić, and Marco Chiesa. A {High-Speed} stateful packet processing approach for tbps programmable switches. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1237–1255, 2023.

[88] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

[89] Leah Shalev, Hani Ayoub, Nafea Bshara, and Erez Sabbag. A cloud-optimized transport protocol for elastic and scalable hpc. *IEEE Micro*, 40(6):67–73, 2020.

[90] Rajath Shashidhara, Tim Stamler, Antoine Kaufmann, and Simon Peter. {FlexTOE}: Flexible {TCP} offload with {Fine-Grained} parallelism. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 87–102, 2022.

[91] Arjun Singhvi, Aditya Akella, Dan Gibson, Thomas F. Wenisch, Monica Wong-Chan, Sean Clark, Milo M. K. Martin, Moray McLaren, Prashant Chandra, Rob Cauble, Hassan M. G. Wassel, Behnam Montazeri, Simon L. Sabato, Joel Scherpelz, and Amin Vahdat. 1rma: Re-envisioning remote memory access for multi-tenant datacenters. In *ACM SIGCOMM*, 2020.

[92] Enterprise SONiC. Acl (access control list), 2023. https://support.edge-core.com/hc/en-us/articles/900000214926--Enterprise-SONiC-ACL-Access-Control-List-#h_01FWN7B0ED07H7A9CWD8243KZW.

[93] Mellanox Technologies. Introduction to infiniband. Technical report, 2003. https://network.nvidia.com/pdf/whitepapers/IB_Intro_WP_190.pdf.

[94] PATH to TCP 4K MTU and RX zerocopy. Eric dumazet, 2020. https://www.infinibandta.org/ibta-specification/.

[95] Weiyang Wang, Moein Khazraee, Zhizhen Zhong, Manya Ghobadi, Zhihao Jia, Dheevatsa Mudigere, Ying Zhang, and Anthony Kewitsch. TopoOpt: Co-optimizing network topology and parallelization strategy for distributed training jobs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 739–767, Boston, MA, April 2023. USENIX Association.

[96] 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 *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1–14, 2023.

[97] Inc. Xilinx. Virtex ultrascale+ fpga product brief, 2021. https://www.xilinx.com/content/dam/xilinx/support/documents/product-briefs/virtex-ultrascale-product-brief.pdf.

[98] Inc. Xilinx. Dma/bridge subsystem for pci express v4.1, 2022. https://www.xilinx.com/support/documents/ip_documentation/xdma/v4_1/pg195-pcie-dma.pdf.

[99] Inc. Xilinx. Qdma subsystem for pci express v4.0, 2022. https://www.xilinx.com/support/documents/ip_documentation/qdma/v4_0/pg302-qdma.pdf.

[100] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating*

Systems Design and Implementation (OSDI 22), pages 521–538, 2022.

[101] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, et al. The demikernel datapath os architecture for microsecond-scale datacenter systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 195–211, 2021.

[102] 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 *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 673–687, 2023.

[103] 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. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 523–536, New York, NY, USA, 2015. Association for Computing Machinery.