Re-architecting End-host Networking with CXL: Coherence, Memory, and Offloading

Houxiang Ji
University of Illinois
Urbana-Champaign
Urbana, IL, USA
hj14@illinois.edu

Yifan Yuan
Meta
Menlo Park, CA, USA
yifanyuan@meta.com

Yang Zhou
University of Illinois
Urbana-Champaign
Urbana, IL, USA
yangz15@illinois.edu

Ipoom Jeong
Yonsei University
Seoul, Republic of Korea
ipoom@yonsei.ac.kr

Ren Wang
Intel Corp.
Portland, OR, USA
ren.wang@intel.com

Saksham Agarwal
University of Illinois
Urbana-Champaign
Urbana, IL, USA
saksham@illinois.edu

Nam Sung Kim
University of Illinois
Urbana-Champaign
Urbana, IL, USA
nskim@illinois.edu

Abstract

The traditional Network Interface Controller (NIC) suffers from the inherent inefficiency of the PCIe interconnect with two key limitations. First, since it allows the NIC to transfer packets to the host CPU memory only through DMA, it incurs high latency, the impact of which becomes more pronounced, especially for small-sized packets. Second, it supports neither shared memory nor full cache coherence between the host CPU and the NIC. Therefore, the host CPU can access NIC memory only through MMIO—which provides higher latency and lower bandwidth than cache-coherent memory access—and software is often responsible for managing consistency between the host CPU and NIC memory. Although built on the PCIe interconnect, Compute Express Link (CXL) efficiently addresses these limitations by providing hardware-managed unified memory and cache coherence between the host CPU and NIC memory. This allows the host CPU and the NIC to access each other's memory using load/store semantics, offering low latency and high bandwidth. In this work, we first present a Type-1 CXL-NIC design that replaces slow legacy PCIe transactions with fast CXL.cache transactions for NIC-to-CPU memory accesses. Second, we extend the Type-1 CXL-NIC to a Type-2 CXL-NIC that introduces cache-coherent NIC memory exposed to the host CPU through CXL.mem, which can buffer packets and descriptors[1]. Lastly, we demonstrate a networking-application co-acceleration by exploiting unique CXL Type-2 device features and near-packet processing on the Type-2 CXL-NIC. Our FPGA-based CXL-NIC prototypes reduce the tail latency of network packet and application request processing by 49% and 39%, respectively, compared to a commodity PCIe-NIC.

[1]Descriptors store information needed for packet processing, including packet sizes and pointers to DMA ring buffers.

CCS Concepts

• **Computer systems organization → Architectures**; • **Hardware → Emerging technologies**.

Keywords

Compute Express Link, Network Interface Controller, Acceleration

ACM Reference Format:
Houxiang Ji, Yifan Yuan, Yang Zhou, Ipoom Jeong, Ren Wang, Saksham Agarwal, and Nam Sung Kim. 2025. Re-architecting End-host Networking with CXL: Coherence, Memory, and Offloading. In *58th IEEE/ACM International Symposium on Microarchitecture (MICRO '25), October 18–22, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3725843.3756102

1 Introduction

The landscape of computer architecture is undergoing a significant transformation, driven by the growing demands of data-intensive applications and the slowing cadence of traditional CPU scaling. This shift necessitates heterogeneous computing paradigms, where the interconnect fabric, which binds host CPUs with I/O devices, accelerators, and memory, emerges as a critical bottleneck. Compute Express Link (CXL) emerges as a pivotal open standard in this context. Strategically built upon the widely used physical and electrical layers of Peripheral Component Interconnect Express (PCIe), it introduces transformative capabilities—most notably unified memory and hardware-managed cache coherence between the host CPU and devices. This innovation promises to fundamentally reshape how the host CPU and devices interact, moving beyond the limitations of existing interfaces.

While CXL offers a versatile suite of protocols (CXL.io, CXL.cache, CXL.mem), much of the initial research and industry momentum has gravitated towards CXL Type-3 devices. These devices primarily utilize the CXL.mem protocol, positioning CXL as a solution for tiered memory architectures and addressing pressing challenges in DRAM capacity and bandwidth scaling by enabling memory expansion modules [31, 35, 53, 54, 63]. However, this focus on memory expansion inadvertently overshadows the potential residing within the *CXL.cache* protocol, a key feature of CXL Type-1 and Type-2 devices. The CXL.cache protocol allows these devices to participate directly in the host CPU's cache coherence domain, enabling fine-grained data sharing and synchronization with unprecedented efficiency that warrants deeper investigation. Therefore, this work shifts the research lens towards CXL Type-1 and Type-2 devices, conducting a focused exploration of the performance advantages unlocked by their inherent cache coherence support.

We contend that Network Interface Controllers (NICs) represent a particularly compelling, indeed a 'killer' application for harnessing the power of CXL's cache coherence. For decades, PCIe has served as the de facto standard for connecting NICs to the host CPU. While PCIe has scaled its bandwidth to accommodate high-performance workloads, its architectural design inherently struggles with the demands of modern network-centric, latency-sensitive applications [7, 26, 29, 58, 61, 62], such as real-time analytics, Large Language Model (LLM) inference, and microsecond-scale distributed systems. These applications are characterized by frequent, fine-grained data interactions where per-operation latency is paramount.

PCIe's shortcomings in these scenarios stem from two core limitations. First, PCIe does not support load/store semantics for device-to-host memory access, instead relying on Direct Memory Access (DMA). While DMA enables bulk data transfers to the host with considerable bandwidth, its setup and completion overhead renders it inefficient for the frequent, small-sized data transfers, *e.g.*, 64B descriptor exchanges between the host CPU and the NIC. Second, PCIe lacks shared memory and cache coherence between the host CPU and devices. In this case, only Memory-Mapped I/O (MMIO) is supported for CPU-to-device communication, suffering from prohibitive latency (>8μs for 512B) and constrained bandwidth (<0.3GB/s). Due to the lack of hardware coherence, PCIe necessitates complex software-managed synchronization schemes (*e.g.*, explicit cache flushing, memory pinning, and memory barriers), which not only complicates programming but also injects substantial latency overhead into communication paths. These PCIe bottlenecks become acutely performance-limiting at high network line rates (100 Gbps and beyond), consuming precious nanoseconds and hindering the overall efficiency of packet processing [29, 37, 49].

To directly address these entrenched PCIe limitations in the networking domain, we design and propose CXL-NIC, a novel NIC architecture meticulously crafted to leverage the CXL.cache protocol for seamless, low-latency, cache-coherent communication between the host CPU and the NIC. Our CXL-NIC design paradigm fundamentally departs from traditional PCIe-based approaches with the following fundamental changes:

Simplifying datapath with coherence. CXL-NIC strategically replaces inefficient PCIe-based synchronization and data movement operations—such as MMIO writes for updating device registers (*e.g.*, queue tail pointers) and DMA reads/writes for fetching/reporting small descriptors—with highly efficient, cache-line-granular operations facilitated by CXL.cache. This allows the host CPU and CXL-NIC to share control structures and packets directly within the coherent memory space. In addition, the unique CXL.cache feature of coherence state control of specific cache lines allows further fine-grained optimizations to reduce back-and-forth coherence traffic. All these designs drastically reduce synchronization overheads and eliminate the need for resource-intensive software interventions like explicit cache flushes.

Coherent NIC memory. Building on CXL Type-1 device's coherence-driven datapath acceleration, CXL-NIC (based on CXL Type-2 device) integrates CXL.mem to enable placing networking data structures directly in NIC memory. The term 'NIC memory' hereafter refers to the coherent DRAM exposed by CXL.mem on the NIC unless otherwise specified. This expands datapath design flexibility, allowing structures like packet buffers or descriptor rings to reside in on-NIC or host memory depending on system requirements. By decoupling data placement from fixed host/device boundaries, CXL-NIC unlocks novel datapath configurations—*e.g.*, hosting descriptors in NIC memory while keeping packet buffer host-resident. However, uninformed data placement in NIC memory risks increased latency of datapaths, as remote NIC memory accesses incur higher overhead than local DRAM. To mitigate this, CXL-NIC co-optimizes data placement and selection of requests underpinning the datapaths to minimize end-host networking latency and reduce host memory pressure. Also, the abundant NIC memory provides the capacity to host scalable buffer pools (*e.g.*, per-queue packet buffers), allowing CXL-NIC to reduce reliance on host memory.

Application-networking co-acceleration. Near-memory processing capability of CXL-NIC extends the acceleration beyond the networking, enabling cross-layer optimization across network and application layers. By co-designing these layers, the system eliminates redundant CPU-NIC data transfers, improves resource efficiency, and reduces end-to-end latency. We take the prevailing Key-Value Store (KVS) as an example application. CXL-NIC hosts the database in NIC-attached memory with device-bias local access, enabling direct request processing without host intervention or coherence checks for local hits. On misses, it leverages CXL-coherent host memory access to serve the request, avoiding the resource-intensive PCIe operations like DMA or RDMA. Meanwhile, CXL.cache coherence guarantees that the host CPU can update the NIC-resident database at any time with simple `ld`/`st` instructions, ensuring consistency across host CPU and NIC.

We implement CXL-NIC on an Intel Agilex-7 FPGA board with CXL 1.1 support. Evaluations based on microbenchmarks demonstrate that CXL-NIC is able to reduce the latency of network packet processing by 49% compared to a commodity PCIe-based NIC, as well as 37% compared to CC-NIC [48], a state-of-the-art coherent NIC design. When evaluated on the KVS application, CXL-NIC achieves a 39% tail latency reduction of KVS requests, compared to a SmartNIC-based design.

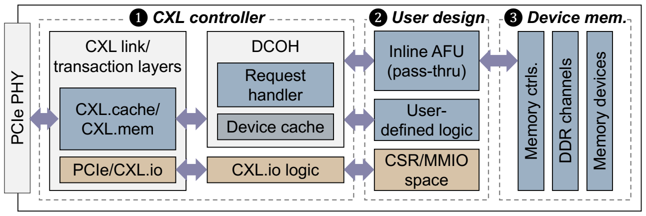


Figure 1: CXL Type 1/2 device architecture. Device memory is only available in CXL Type-2 device.

2 Background

2.1 Compute Express Link (CXL)

CXL is an emerging cache-coherent interconnect standard for processors and peripheral devices, built atop the physical layer of PCIe. CXL devices are classified into three types based on the compositions of the CXL.io, CXL.cache, and CXL.mem protocols.

- **CXL.io**, mandatory for all CXL devices, manages interface initialization and basic I/O operations between the host and devices.
- **CXL.cache**, used for Type-1 and Type-2 devices, enables cache-coherent Device-to-Host (D2H) communication and data movement with hardware-managed coherence.
- **CXL.mem**, used for Type-2 and Type-3 devices, allows Host-to-Device (H2D) memory accesses with regular `ld/st` semantics and Device-to-Device (D2D) memory accesses.

This work focuses on Type-1 and Type-2 devices, as both support CXL.cache. Type-3 devices, in contrast, serve solely as memory expanders and lack CXL.cache support, making them outside the scope of this work.

Device architecture. Figure 1 illustrates a typical CXL device architecture with three major parts: CXL controller, user design, and an optional device memory. In the ❶ CXL controller, the CXL link/transaction layers support functionality for generating CXL transactions and maintain their reliability across the link [8]. Device COHerence engine (DCOH) is the key component that handles data access requests of CXL.cache and CXL.mem protocols in a cache-coherent manner. DCOH's device cache is a small cache region that is in the same coherence domain as the host CPU caches, supporting coherence protocols defined in the CXL specification. The device cache can be divided into Host Memory Cache (HMC) and Device Memory Cache (DMC) to store data from host memory and device memory, respectively [22]. ❷ The programmable user design region consists of inline AFU, user-defined logic, and CSR/MMIO space. Among these, the user-defined logic can interact with the host memory by issuing D2H requests in a CXL Type-1 device, which are routed to and handled by DCOH. In CXL Type-2 device, where ❸ device memory is available, the user-defined logic can also issue Device-to-Device (D2D) requests to access it, while the host CPU can access it through Host-to-Device (H2D) requests via CXL.mem. Together, these mechanisms enable rich interactions between the host CPU and the user-defined logic over both host and device memory.

Control of device cache line state. The CXL.cache protocol uniquely enables explicit control over the state of device cache lines, *a feature absent in other cache-coherent interconnects such as UPI* [18]. Specifically, when user-defined logic issues a data request, it specifies one of four hints for the target cache line: non-cacheable (`NC`), non-cacheable push (`NC-P`), cacheable shared (`CS`) and cacheable owned (`CO`).

- `NC`: For `NC-read`, DCOH reads the corresponding cache line directly from the device cache if present, or fetches it from host or device memory without storing it in device cache. For `NC-write`, DCOH invalidates the cache line in device cache and/or host caches and updates the memory directly.
- (write-only) `NC-P`: Upon receiving `NC-P`, the DCOH updates the cache line in the device cache, pushes the updated cache line to the host CPU's cache hierarchy (typically LLC), and then invalidates the cache line in the device cache.
- (read-only) `CS`: DCOH serves `CS-read` similarly to `NC-read`, but retains the cache line in the device cache in a *Shared* state after fetching it from the host/device side.
- `CO`: The DCOH acquires exclusive ownership of the cache line by invalidating host cache copies and retaining the line in an *Owned* state in the device cache for read or write.

This fine-grained control of cache line state transitions allows software applications and hardware user-defined logic to optimize memory access efficiency by strategically balancing coherence overhead and locality [22]. For example, marking data as *Owned* enables a device to retain exclusive access to frequently used data in its local cache, eliminating the need to repeatedly invalidate host-side copies. This reduces coordination overhead while maintaining fast local access, a level of flexibility that proprietary cache-coherent interconnects cannot provide.

Host/Device-bias mode. CXL.cache protocol supports two modes for D2D access, host- and device-bias modes, to reduce the cache coherence management overhead. In host-bias mode, DCOH ensures coherence by checking the host cache for modified cache lines before accessing device memory. If a modified line exists in the host cache, DCOH updates the cache line in DMC and invalidates the host copy. In device-bias mode, DCOH bypasses host cache checks, enabling lower-latency access to DMC or device memory for faster device-local processing. However, in this mode, the coherence must be managed by software explicitly. The two bias modes can be switched dynamically and separately for different regions in the device memory.

2.2 NIC-to-CPU Datapath

This section describes the Rx and Tx datapaths in modern PCIe-based NICs. Note that these datapaths are applicable to both the kernel-space and user-space networking stacks (*e.g.*, DPDK [16]).

Rx datapath. Figure 2(left) depicts the Rx datapath of network packets. The network stack maintains one or more ring buffers of Rx descriptors (RxDs) to manage incoming network packets. Each RxD in the ring buffer points to a blank Rx packet buffer pre-allocated from a memory pool (commonly referred to as a `mempool`), where incoming packets will be stored. Specifically, ❶ the host CPU assigns blank packet buffers to individual RxDs, preparing them for NIC DMA operations, and ❷ updates the Rx head pointer via MMIO. ❸ The NIC asynchronously DMA-reads the available RxDs from the ring buffer until the Rx head. When a packet arrives, ❹ NIC DMA-writes the received packet to the packet buffer specified in the

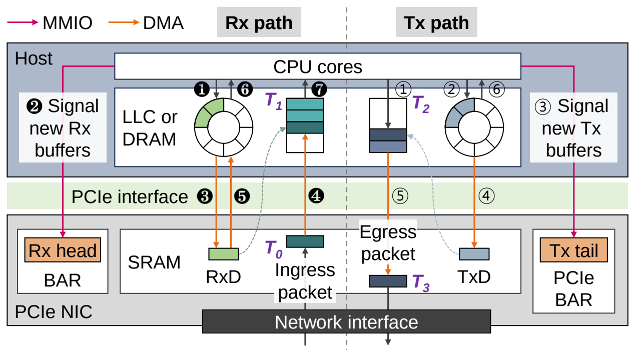


Figure 2: PCIe NIC-to-CPU datapaths: Rx (left) and Tx (right).

fetched RxD. Once the transfer is complete, ❺ the NIC updates the RxD indicating that a new valid packet is available [13]. The NIC then notifies the host CPU of the packet arrival—either via (1) an interrupt or (2) the host CPU polling the RxD in host memory (not shown in the figure). Finally, ❻ the host CPU processes the updated RxDs and ❼ handles the packets stored in the corresponding packet buffer, freeing them for reuse.

Tx datapath. Similar to the Rx datapath, the networking stack manages ring buffers of Tx descriptors (TxDs) for packet transmission. As illustrated in Figure 2(right), the Tx datapath begins when ① the host CPU prepares the packet in a pre-allocated Tx packet buffer and creates a TxD that references the buffer. ② The host CPU writes this TxD into the Tx ring buffer, increments the tail index, and ③ triggers a doorbell notification via MMIO (*i.e.*, by updating the Tx tail pointer) to inform the NIC of the new TxD. Next, ④ the NIC first DMA-reads the TxD from the ring buffer, and then ⑤DMA-reads the packet from the memory location specified in the TxD. After these DMA operations, the NIC performs a DMA write to update the TxD, signaling the completion of the packet transfer to the NIC. It may also generate an interrupt to notify the host CPU to ⑥ free the descriptor entry and associated buffer for future transmissions.

Optimization challenges. To mitigate interrupt overhead, frameworks such as DPDK [16] adopt a user-space polling model in place of traditional interrupt-driven mechanisms. In this approach, the host CPU periodically polls the ring buffer to detect updated descriptors, indicating the arrival of new packets. This design eliminates kernel involvement and context switches, thereby reducing latency. While polling effectively avoids interrupt delays, DPDK still depends on PCIe-based DMA and MMIO operations for updating descriptor rings, triggering doorbells, and transferring packets. However, these operations are inherently limited by PCIe's non-cache-coherent nature, which constrains both latency reduction and throughput gains, limitations we quantify in the following section.

3 PCIe vs CXL in Networking

By dissecting the Rx and Tx datapaths between the host CPU and NIC (§2.2), we identify the non-coherent PCIe operations (*e.g.*, MMIO and DMA) that underpin CPU-NIC communication. In this section, we first quantify their performance impact by measuring

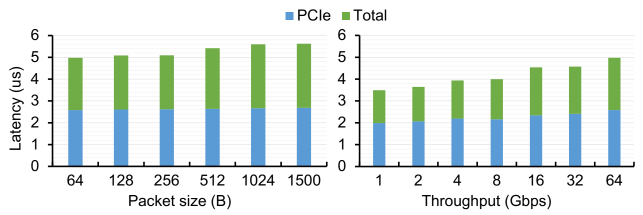


Figure 3: End-to-end packet latency and the portion attributed to PCIe operations with various packet sizes (left) and throughput with a fixed 64B packet size (right).

PCIe's contribution to round-trip latency across packet sizes and rates (§3.1). We then compare these non-coherent PCIe operations with cache-coherent CXL counterparts (*e.g.*, CXL.cache requests) to assess potential performance gains (§3.2). This analysis indicates that cache-coherent interconnects provide a promising option for addressing PCIe's limitations in modern high-speed NICs, motivating our exploration of CXL-based alternatives.

3.1 PCIe Latency Contribution

Because the host CPU and NIC do not share a synchronized clock, we cannot measure the one-way latency directly. We therefore implement a DPDK-based ping–pong microbenchmark. A traffic generator is used to send packets to the host CPU through the NIC, and the host CPU relays packets by copying them into Tx buffers and retransmitting them through the same NIC. We perform timestamping in both the Tx and Rx datapaths to measure PCIe latency, as shown in Figure 2: in the Rx datapath, we timestamp packet arrival at NIC from generator (T_0) and packet availability in host memory (T_1). In the Tx datapath, we record transmission initiation by the host CPU (T_2) and packet arrival at the NIC (T_3). We then calculate the fraction of the end-to-end packet latency attributable to PCIe operations (or PCIe contribution) as $1 - (T_2 - T_1)/(T_3 - T_0)$. The experimental setup is described in Section 6.

Figure 3 shows the end-to-end packet latency and the portion attributable to PCIe operations across varying packet sizes (left) and throughput with a fixed 64B packet size (right). For the smallest 64B packets, PCIe operations account for 52% of total latency, and the fraction decreases to 45% for the largest 1500B (= MTU) packets. The reduction arises because, although PCIe latency grows with packet size, host-side processing latency grows even faster, making PCIe account for a smaller fraction of the total latency for larger packets. When throughput increases from 1 to 64 Gbps, the fraction of latency attributable to PCIe operations decreases from 57% to 52%. This occurs because DPDK's burst processing aggregates packets into batches, thereby reducing the PCIe operation frequency. For example, a single MMIO doorbell can signal multiple descriptors. While the observed trend aligns with prior work [37], our measured PCIe contribution appears lower due to differences in methodology: prior work modified firmware to capture raw hardware-level latencies, whereas our software-level measurements may underestimate it due to optimizations from DPDK. Even though PCIe contributes a smaller *relative* fraction of latency for larger packets or higher throughput, its *absolute* impact remains critical for latency-sensitive workloads. At modern line rates (*e.g.*,

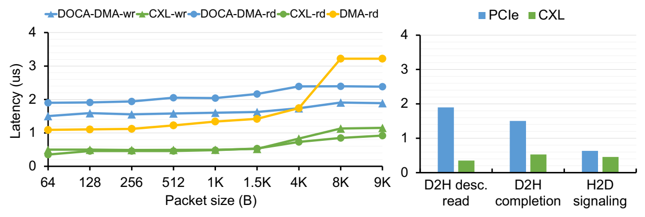


Figure 4: Latency of data movement (left) and synchronization (right) operations by PCIe and CXL.

100+ Gbps), systems handle millions of packets per second, leaving only nanosecond-scale time budgets per packet. In such regimes, even sub-microsecond PCIe delays can lead to notable networking performance degradation [24, 29].

3.2 Microbenchmarking PCIe and CXL

We categorize PCIe operations in the Rx/Tx datapath into data movement and synchronization operations. Data movement operations transfer packet payloads (*e.g.*, D2H DMA-write/read in Rx/Tx), while synchronization operations handle descriptor coordination and host–NIC signaling (*e.g.*, H2D MMIO doorbells in Tx). We measure the latency of these data movement and synchronization operations in the Rx/Tx datapath on the PCIe device and their equivalent CXL requests on the CXL device. Specifically, we use an NVIDIA BlueField-3 SmartNIC (SNIC) [40] as a PCIe device and an Intel Agilex-7 I-Series development kit [17] as a CXL device, which are referred to as BF-3 and Agilex-7 hereafter, respectively. In Agilex-7, a load/store unit is implemented in the user-defined logic (Figure 1–❷) to generate CXL D2H requests to access host memory. Each latency is measured over more than 100 K back-to-back iterations, and median values are presented in Figure 4. Additional setup details are provided in Section 6.

For data movement operations, we compare packet transfer latency in the Rx and Tx paths. We evaluate PCIe DOCA-DMA write/read on BF-3 [39] (DOCA-DMA), CXL.cache D2H `NC-write`/`NC-read` on Agilex-7 (CXL), and DMA D2H read on Agilex-7 (DMA) across various packet sizes, comparing PCIe-based DMA with CXL operations. Accurate DMA latency measurement requires direct control of the NIC's DMA engines and visibility into the DMA request status. Host-initiated DMA transfers require descriptor submission over PCIe, adding extra latency. To avoid this overhead, we use DOCA-DMA on the BF-3's Arm cores to directly control the DMA engines. DOCA-DMA also exposes DMA completion status, enabling precise latency measurement. Measuring DMA operations on Agilex-7 provides a baseline on the same hardware, isolating the effect of CXL. The latency reported in Figure 4 reflects the end-to-end DMA transfer latency with DOCA-DMA, including descriptor preparation and doorbelling, in addition to the actual packet transfer. In contrast, Figure 3 reports packet loopback latency under standard DMA with DPDK, where optimizations such as batching and pipelining amortize descriptor-related overheads across multiple packets, making the observed PCIe latency appear lower than in Figure 4. Using the read/write latency ratio measured with DOCA-DMA, we estimate the standard DMA latency in Figure 3 to be around 1μs, consistent with prior findings [37, 48]. We choose `NC-write`/`NC-read` for CXL D2H operations because they bypass device-side caches, making them comparable to non-coherent DMA. For 64 B packets (the minimum possible packet size for CXL), CXL D2H `NC-write` (Rx) and `NC-read` (Tx) observe 69% and 81% lower latency than their DOCA-DMA counterparts; and CXL D2H `NC-read` achieves 68% lower latency than the DMA D2H read on Agilex-7. We believe this latency improvement arises from two potential factors: (1) CXL.cache avoids DMA setup and completion overheads (*e.g.*, memory pinning before DMA), which render the DMA inefficient for host-NIC communication, especially for small data transfers, and (2) CXL introduces customized link and transaction layer protocols on top of PCIe physical layer to achieve lower latency [11]; we are unable to precisely capture the root cause of the latency benefits, given the limited visibility into the current hardware. As packet sizes increase from 64 B to 9000 B, we observe that CXL maintains similar latency benefits compared to both DOCA-DMA and DMA on Agilex. This is because both CXL and DMA operations observe a similarly modest amount of latency increase with larger packet sizes. CXL D2H `NC-write` and `NC-read` latencies grow only around 1.1× and 1.2×, respectively; DOCA-DMA D2H read and write latencies both grow by ∼1.3×[2]. This is expected for DMA operations because they can keep multiple concurrent PCIe transaction layer packets (TLPs) in-flight from the NIC, delivering a lower effective transfer time per byte [14, 37]. Similarly, CXL.cache also achieves high throughput and low latency for larger packets by allowing multiple cache-line requests in-flight from the CXL device [11, 54].

For synchronization operations in the Rx datapath, we evaluate (1) descriptor fetching with D2H PCIe DOCA-DMA read and CXL `NC-read` and (2) completion signaling with D2H descriptor update by PCIe DOCA-DMA write and CXL `NC-write`, paired with host polling. For synchronization operations in the Tx datapath, we evaluate two H2D signaling methods: (1) PCIe MMIO writes from the host CPU to NIC registers, and (2) CXL D2H `NC-read` requests that poll a shared host memory location updated by the host CPU via `st` [3]. CXL outperforms PCIe-based synchronization methods even when PCIe leverages CPU polling to mitigate interrupt overheads (5–10 μs [30]). Specifically, CXL reduces descriptor read latency by 85% and completion signaling latency by 82% compared to PCIe. For H2D signaling, CXL `NC-read`-based polling cuts latency by 29% over PCIe's MMIO write.

4 Facilitating Communication with Coherence

In this section, we present a CXL Type-1 device-based NIC (Type-1 CXL-NIC) that exploits CXL protocols to overcome PCIe limitations and improve datapath efficiency. We begin with a high-level overview of the CXL-NIC architecture (§4.1), followed by a detailed description of its Rx/Tx datapath (§4.2). We then show how CXL-NIC leverages the unique capabilities of CXL.cache to optimize the datapath and reduce the CPU-NIC communication overhead (§4.3).

[2]DMA on Agilex however increased by 2.9× due to potential inefficiencies in the employed DMA IP

[3]We do not use CXL.io writes to NIC registers for H2D signaling because CXL.io shares the same physical and transaction layers as PCIe MMIO and thus provides comparable performance [22].

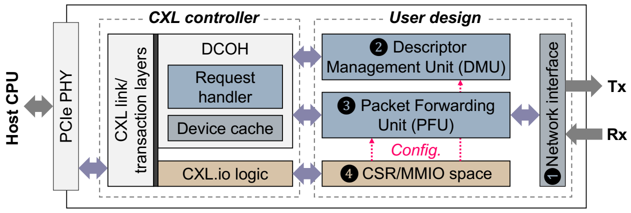


Figure 5: Type-1 CXL-NIC architecture.

4.1 CXL-NIC Architecture

Figure 5 depicts the hardware architecture of Type-1 CXL-NIC without on-device memory. This CXL-NIC consists of four core components: ❶ Network interface facilitates bidirectional packet communication between the NIC and the Ethernet network; ❷ Descriptor Management Unit (DMU) takes care of descriptor-related operations, such as RxD/TxD retrieval from descriptor ring buffers, packet buffer addresses extraction, and descriptor status update to signal packet transfer completion; ❸ Packet Forwarding Unit (PFU) is responsible for packet movement between the NIC and the packet buffers, whose addresses are provided by the DMU. To accommodate bursty traffic, the PFU integrates a dedicated 1 MB SRAM to buffer packets temporarily during NIC-to-CPU transfers. The DMU and PFU connect to the DCOH through separate ports to access host memory via CXL cache requests. This design mitigates contention between descriptor management and data transfer operations. ❹ CSR/MMIO space, mapped to the host memory address space via CXL.io, allows the host CPU to configure CXL-NIC components (*e.g.*, DMU and PFU) using `st` instructions during initialization and runtime. Configuration operations include, for example, provisioning descriptor ring buffer addresses to the DMU and specifying types of CXL.cache requests used for PFU packet transfers. Although backward-compatible with legacy signaling mechanisms (*e.g.*, doorbells), CSR/MMIO is used mainly for infrequent control tasks in CXL-NIC, since CXL.io is non-coherent and has higher latency than CXL.cache (Figure 4).

4.2 Coherence-driven Datapath

CXL-NIC leverages CXL protocols to orchestrate low-latency packet transfers between the host CPU and NIC. Specifically, built on the CXL Type-1 device, CXL-NIC supports both CXL.io and CXL.cache protocols. Due to the absence of coherent on-device memory, the host CPU can only allocate the descriptor ring buffers and packet buffers in host memory, adhering to the same buffer allocation policy as PCIe NICs. We implement a polling-based software framework for the CXL-NIC, CXL-NIC DPDK, which mirrors the structure and functionality of the conventional DPDK [16]. CXL-NIC DPDK manages CXL-NIC configuration as well as networking data structures (*e.g.*, descriptor rings and packet buffers), and executes the operations in Tx/Rx datapaths on the host CPU (*e.g.*, initiating Tx transfers and processing received packets). It also supports zero-copy, as CXL-NIC can directly access user-space buffers allocated in either host or device memory in a cache-coherent manner. We now describe the Rx and Tx datapath spanning the host CPU and CXL-NIC, which replace legacy PCIe operations (*e.g.*, DMA read/write) with CXL.cache requests (*e.g.*, `NC-read/write`,`CO-read`) to optimize the datapath.

Rx datapath. Similar to the Rx datapath in PCIe NIC (§2.2), the host CPU allocates two structures in host memory: (1) Rx packet buffers for storing received packets and (2) a descriptor ring buffer whose entries record the addresses of these buffers. ① During initialization, the host CPU configures the CXL-NIC 's DMU by writing the ring buffer address and length to its MMIO space via CXL.io (Figure 5-❹). ② Before the packet arrives, the DMU issues D2H **`CS-read`** requests to prefetch the available RxDs from the ring buffer address and stores them in the HMC (§2.1). Upon packet arrival, ③ the DMU fetches available RxDs from the HMC by **`CO-read`** and instructs the ④ PFU to issue a D2H **`NC-write`** to transfer the packet to the packet buffer specified in the descriptor. ⑤ After the packet transfer, the DMU issues another D2H **`NC-write`** to update the status field of the corresponding RxD in the ring buffer. The host CPU polls the RxD to detect the completed transfer, then delivers the packet to the application and reclaims the descriptor. Type-1 CXL-NIC can also generate interrupts via CXL.io to notify the host CPU of ready packets; however, the interrupt handling introduces notable latency to the critical path, degrading network performance even with interrupt coalescing optimizations [44].

Tx datapath. The host CPU allocates the Tx descriptor ring buffer and packet buffer in host memory. A key distinction in CXL-NIC is its management of the ring buffer's tail index. In PCIe NICs, the tail index is maintained in a device register, and the host CPU updates it through MMIO writes. In contrast, ① CXL-NIC keeps the tail index directly in host memory, allowing the host CPU to update it via normal store (`st`) instructions. During initialization, the host CPU provides the DMU with the tail index address together with the ring buffer's base address and length. ② For packet transmission, the host CPU allocates the packet buffer and stores the packet into the packet buffer. The host CPU then constructs a TxD that references the buffer, inserts it into the descriptor ring buffer, and updates the tail index via `st`. Batching is supported by deferring tail index updates until multiple descriptors are queued. ③ The DMU polls the tail index by issuing periodic D2H **`CO-read`** requests. ④ When an update is detected, the DMU fetches TxDs up to the tail index using **`NC-read`**, extracts the packet buffer address, and ⑤ instructs the PFU to retrieve the packets using D2H **`NC-read`**. ⑥ Finally, the DMU signals transmission completion via a D2H **`NC-write`** to update the descriptor status, allowing the host to reclaim the packet buffers.

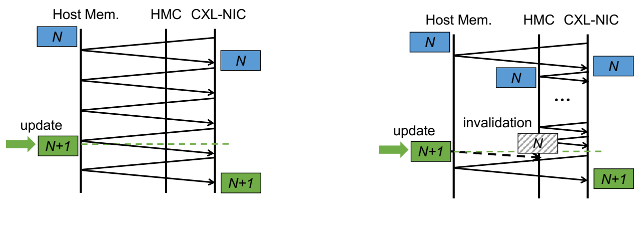


Figure 6: Polling Tx tail by CXL-NIC using `NC-read` (left) and `CO-read` (right).

4.3 Optimizations with Cache Line State Control

CXL.cache protocol provides four types of requests, each specifying the desired cache line state in the device cache (§2.1). In CXL-NIC, we leverage these request types deliberately at each step of the Rx/Tx flow described in Section 4.2. By tailoring request selection to datapath requirements, such as reducing device cache pollution, we demonstrate how CXL.cache enables optimizations that PCIe cannot provide.

RxD prefetch with `CS-read`. Before packets arrive at the network interface, the DMU prefetches available descriptors into the HMC of CXL-NIC via `CS-read` commands, storing them in the *Shared* state. Unlike PCIe NICs, which require explicit software coordination to prefetch descriptors into non-coherent storage (*e.g.*, on-NIC SRAM) [13, 24, 41], CXL's hardware-managed coherence automatically invalidates cached RxDs if host updates them after prefetching, guaranteeing freshness without software intervention. `NC-read` can also fetch RxDs from host memory, but it bypasses the CXL device cache without storing RxDs in HMC. `CO-read` can cache descriptors in the *Owned* state, but it incurs unnecessary coherence probes for ownership acquisition, which is not required since RxDs are read-only before packet arrival. In contrast, `CS-read` caches RxDs in the *Shared* state, enabling caching without ownership overhead and aligning with the RxD access pattern. When packets arrive, `CO-read` request to read RxD can hit the HMC directly, reducing read latency by 3.3× compared to accessing host memory.

Tx tail index polling with `CO-read`. In PCIe NICs, the host CPU updates the tail register in the device by MMIO to signal the availability of new TxDs. CXL-NIC eliminates this MMIO operation by storing the Tx tail index in host memory and polling it with device-initiated `CO-read` requests. Unlike `NC-read`, which generates traffic on the CXL interconnect on every poll, `CO-read` caches the tail index in HMC in the *Owned* state, enabling local polling without CXL traffic until the host CPU updates the tail index. When the host CPU updates the index, CXL hardware-managed coherence automatically invalidates the cached copy, and the next poll fetches the updated value from host memory as shown in Figure 6, creating an **event-driven Tx datapath** with minimal CXL bandwidth consumption. While `CS-read` could cache the index in the *Shared* state, its lack of exclusive ownership risks race conditions between host CPU updates and NIC polling. Building on this event-driven Tx datapath, CXL-NIC adopts the inline signaling technique from CC-NIC [48] seamlessly, merging the Tx tail index poll and TxD read into a single CXL request. Each TxD includes a host-managed status flag that signals descriptor readiness. By polling TxDs directly with `CO-read` and checking this flag inline, CXL-NIC avoids a dedicated `NC-read` to fetch TxD, reducing latency. The *Owned* state provided by `CO-read` enforces *mutual exclusion*: the host CPU holds ownership while preparing descriptors, and CXL-NIC acquires it to update status after transmission, ensuring single-writer semantics without additional software-managed synchronization.

Packet transfer with `NC-read`/`NC-write`. CXL-NIC adopts `NC-write` and `NC-read` for packet transfer in Rx and Tx datapaths, respectively. In the Rx datapath, packets are written directly to host memory via `NC-write`, bypassing the NIC cache. This ensures no copies of the packet persist in device cache, preserving the device cache to store metadata like RxDs. Meanwhile, `NC-read` is demonstrated to achieve the fastest packet transfer in evaluation (§7). In the Tx path, packet buffers are read from host memory using `NC-read`, also bypassing the NIC cache. This avoids coherence invalidation traffic when the host CPU reuses Tx packet buffers, unlike `CS-read` or `CO-read`, which cache the buffers in the device and trigger invalidations on reuse. CC-NIC [48], which is built on another cache-coherent interconnect UPI [18], has to rely on software cache line flushes (`CLFLUSH`) to invalidate Tx packet buffer copies in device cache before the host CPU can safely reuse packet buffers. CC-NIC proposes to mitigate the cache line flush overheads with buffer recycling allocators, but the design forces Rx and Tx datapaths to share a single global buffer pool, complicating software and limiting scalability for multi-pool use cases (*e.g.*, isolated per-tenant queues or QoS partitions). In contrast, with no cached copies after transmission, buffer reuse in CXL-NIC incurs neither flushes nor invalidation traffic, allowing the host CPU to recycle buffers freely across workloads without allocators or cross-path coordination.

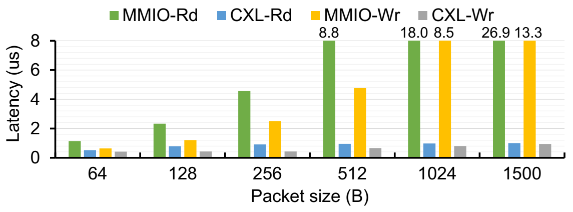


Figure 7: Device memory access by MMIO and CXL.mem.

RxD/TxD completion update with `NC-write`. After a packet is successfully transferred to the host memory (Rx datapath) or transmitted by the NIC (Tx datapath), the DMU issues D2H `NC-write` requests to update the status of descriptors in the ring buffer. This update is safe because CXL-NIC already holds exclusive ownership of the RxDs and TxDs via `CO-read`, preemptively resolving any race conditions between accesses from the host CPU and CXL-NIC. The `NC-write` also automatically invalidates cached copies of prefetched RxDs and used TxDs in the device cache, eliminating the need for software cache flushes and preventing coherence traffic when descriptors are reused.

5 Acceleration with Coherent NIC Memory

Section 4 has demonstrated how Type-1 CXL-NIC alleviates PCIe bottlenecks using CXL.cache requests. Building on this foundation, we turn to CXL Type-2 device, which extends CXL Type-1 device with coherent device memory (via CXL.mem) to further accelerate end-host networking. We first characterize the coherent device memory access latency and compare it with that of non-coherent on-device memory in PCIe NICs. Next, we redesign the Rx/Tx datapaths in CXL Type-2 device-based CXL-NIC (Type-2 CXL-NIC) to exploit coherent memory. This involves optimizing the networking data memory layouts (§5.2) and introducing a CXL-specific optimization with `NC-P` (§5.3). Finally, we present a case study of networking-application co-acceleration on the Type-2 CXL-NIC (§5.4).

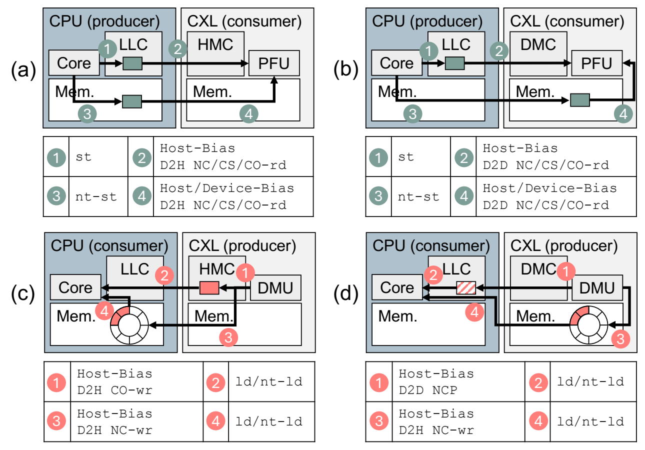
5.1 Access to Device Memory

To quantify the benefits of the coherent CXL device memory, we leverage Agilex-7 as a PCIe device to measure H2D access via prefetchable MMIO and as a CXL Type-2 device to measure H2D access over CXL.mem. As shown in Figure 7, `ld` and `st` over CXL.mem achieve 5.6× and 4.5× lower latency on average, respectively, compared to MMIO. The latency reduction stems from CXL.mem's native support for CPU `ld`/`st` semantics, eliminating strict ordering requirements on PCIe MMIO transactions. This reduction in H2D access latency positions CXL device memory as a seamless extension of the host memory hierarchy and makes it feasible to place latency-sensitive networking data structures directly in the device memory.

5.2 Networking Data Memory Layout

CXL Type-2 device provides hardware-managed coherent memory via CXL.mem, enabling host access to device DRAM at cache line granularity. Building on this capability, we build up Type-2 CXL-NIC to exploit the coherent CXL device memory (referred to as NIC memory) to further enhance networking efficiency. The coherent NIC memory enables flexible placement of Tx/Rx data structures, such as packet buffers and descriptor ring buffers, across host memory and NIC memory using generic NUMA-aware memory allocation mechanisms. The host CPU can dynamically designate these structures to local (host) or remote (NIC) NUMA nodes, eliminating device-specific low-level software modifications required in prior work [41]. The Rx/Tx datapaths are adapted to these flexible memory layouts with careful selection of CXL.cache and CXL.mem requests.

To simplify analysis, we model synchronization and data movement between the host CPU and the NIC as a producer-consumer model. For instance, in the Tx datapath, the host CPU acts as the producer by generating packets, and the NIC acts as the consumer by reading them for transmission. Figure 8(a) and (b) show cases where the host CPU is the producer and CXL-NIC is the consumer, while Figure 8(c) and (d) invert these roles. As an example, in Tx packet transfer, Figure 8(a) places the packet buffer in host memory, while Figure 8(b) places it in CXL-NIC memory. Both buffer allocations support two possible data paths: cache-based (① + ②) and memory-based (③ + ④), with distinct types of requests in the affiliated tables. In the cache-based data path, the host CPU ① writes the packet to host LLC via `st` and PFU has to issue ② host-bias D2H/D2D CXL.cache read requests to fetch the packet from the host LLC. The selection of D2H or D2D requests depends on where the packet buffer is allocated in host or CXL-NIC memory. In the memory-based data path, the host CPU employs ③ `nt-st` to write the packet to host/CXL-NIC memory directly, invalidating any cached copy in the CPU cache hierarchy. Since `nt-st` invalidates copies in host cache, the PFU can ④ read the packet in either host- or device-bias mode. Figure 8(c) and (d) illustrate the opposite case, where CXL-NIC is the producer and the host CPU is the consumer, using Rx descriptor updates as an example. In this example, Figure 8(c) places the descriptor buffer in host memory, whereas Figure 8(d) places it in CXL-NIC memory. Both placements again support two possible datapaths, with their request types detailed in the table. For brevity, we omit detailed discussion of these variants. A unique case arises in Figure 8(d), where data produced in CXL-NIC memory and consumed by the host CPU can be transferred using the `NC-P` operation. We provide further analysis of `NC-P` and its usage in Section 5.3.



(a)

①	st	②	Host-Bias D2H NC/CS/CO-rd
③	nt-st	④	Host/Device-Bias D2H NC/CS/CO-rd

(b)

①	st	②	Host-Bias D2D NC/CS/CO-rd
③	nt-st	④	Host/Device-Bias D2D NC/CS/CO-rd

(c)

①	Host-Bias D2H CO-wr	②	ld/nt-ld
③	Host-Bias D2H NC-wr	④	ld/nt-ld

(d)

①	Host-Bias D2D NCP	②	ld/nt-ld
③	Host-Bias D2H NC-wr	④	ld/nt-ld

Figure 8: Datapaths and their associated requests with different memory layouts in CXL-NIC.

With CXL.cache and CXL.mem, Type-2 CXL-NIC offers diverse datapaths. However, these datapaths are shaped by architectural asymmetries from CXL and performance asymmetries from hardware heterogeneity, necessitating careful selection of datapath and requests to meet application demands. Architecturally, CXL devices support device-bias mode, which bypasses coherence enforcement for local device-memory accesses. Performance asymmetry stems from hardware heterogeneity: for instance, our Type-2 CXL-NIC prototype employs Agilex-7 operating at 400 MHz, much lower than CPUs, reflecting typical resource disparities in heterogeneous systems. Together, these asymmetries make H2D and D2H transfers inherently unequal in latency and bandwidth. In contrast, CC-NIC adopts design choices predicated on UPI's symmetric NUMA coherence and overlooks practical hardware asymmetries. We evaluate these choices in our prototype and find they yield suboptimal performance relative to our CXL-aware design, which explicitly accounts for device-bias mode and hardware heterogeneity (§7.2).

5.3 Intelligent Usage of `NC-P`

As shown in Figure 8(d), when CXL-NIC is the producer, it can either proactively push the data into the host LLC using `NC-P` or update device memory by `NC-write`. When the host CPU subsequently issues a `ld`, `NC-P` reduces latency by serving it from the LLC, avoiding a remote fetch from NIC memory. DDIO can also inject I/O data into the host LLC; however, because it relies on non-coherent DMA, software must manage coherence explicitly, and unprocessed data may be prematurely evicted to DRAM (the "leaky DMA" problem) [60]. Under high throughput, unrestricted `NC-P` poses a similar risk: frequent cache injections may overwhelm the LLC and displace unprocessed data. We mitigate this with two mechanisms. **(1) Adaptive push-write gating** employs a runtime-configurable flag register on CXL-NIC, set by the host CPU via

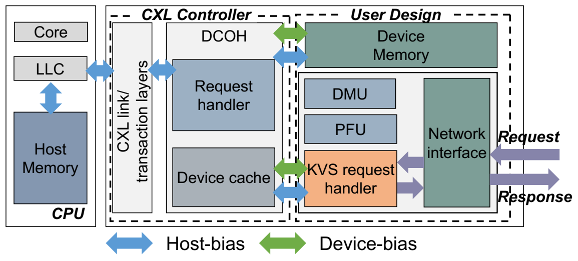


Figure 9: Key-value store application running on CXL-NIC.

CXL.io, to dynamically enable or disable NC-P. When LLC utilization approaches a threshold, the host CPU disables NC-P and reverts to NC-write for device memory updates. Unlike DDIO's static LLC redirection, this provides fine-grained control over cache pressure. **(2) Post-push write-back** issues a delayed NC-write to the same address *N* cycles after NC-P, invalidating the cache line in host LLC and writing it back to the device memory. Evicting the line after *N* cycles prevents LLC saturation but introduces a timing trade-off: if the host CPU does not access the data within that window, the delayed NC-write evicts it to device memory and the latency benefit of NC-P is lost. Considering this, CXL-NIC allows the host CPU to tune the value of *N* in a device register to balance cache utilization and access latency based on workload characteristics.

5.4 Networking-Application Co-acceleration

In this section, we extend CXL-NIC beyond the networking stack to provide holistic acceleration across network and application domains. By co-designing these layers, CXL-NIC eliminates redundant data movements, optimizes resource utilization, and achieves end-to-end application performance improvements.

Key-value stores (KVS) are foundational to cloud and data center systems [5, 29], serving workloads with stringent latency requirements. KVS implementations typically use efficient data structures like hash tables to map keys to values, prioritizing low-latency lookups and updates. In hash table-based KVS, keys are hashed to indices that point to buckets or entries storing corresponding values to read/update/insert. Chaining and cuckoo hashing are two common ways used to alleviate the hash collisions but they introduce extra memory accesses. For example, cuckoo hashing employs multiple hash functions and probes multiple buckets per lookup concurrently to locate the matching entry. While Type-1 CXL-NIC expedites KVS by accelerating the request routing to the host, Type-2 CXL-NIC leverages coherent, gigabyte-scale memory and implements an on-NIC KVS request handler to process KVS requests locally (Figure 9). The KVS request handler performs hash computation, bucket lookup, and insertion. The NIC memory hosts a subset of KVS database preloaded by the host CPU with frequently accessed ('hot') key-value pairs, with host memory serving as a scalable backend for cold data. Type-2 CXL-NIC employs *dynamic bias mode switching* to minimize coherence overhead when serving the KVS requests.

Upon receiving a KVS request, the KVS handler extracts the key from the received packet, computes its hash, and probes the hash table residing in CXL-NIC memory with D2D CS-read. If the key-value pair is found in NIC memory (a hit), the handler reads the value by D2D NC-read to serve the request locally or updates the entry via D2D NC-write without generating traffic to host memory. All these D2D requests are in **device-bias mode**, bypassing coherence checks with the host CPU and thus serving requests in a faster path. If the key-value pair is absent in NIC memory (a miss), the KVS request handler switches to **host-bias mode**, issuing a D2H CO-read to fetch the key-value pair from the database residing in host memory. It immediately answers the request with the retrieved data to meet latency requirements, and asynchronously stores the pair into NIC memory via a D2D NC-write. Concurrently, the least recently used (LRU) key-value entry in the NIC memory is evicted to host memory through a background swap, transparent to applications. This tiered database placement design, unified through CXL's hardware-coherent memory semantics, ensures low-latency access to 'hot' data in NIC memory and efficient capacity expansion for 'cold' data in host memory, achieving a balance between performance and resource utilization across the hierarchy.

Table 1: Host machine and devices.

Name	Description
CPU & Memory	Intel Xeon 6538Y+ CPU @2.2 GHz, 32 cores (SMT disabled) 60 MB LLC, 8× DDR5-4800 channels
CXL Device	Intel Agilex®-7 I-Series development kit@400MHz [17] CXL 1.1 over PCIe 5.0, 2× DDR4-2400
SNIC	NVIDIA BF-3 [40], 8 ARM A72 cores@2.5GHz 16GB on-board DDR4-1600 DRAM

6 Implementation

CXL-NIC is prototyped on the system with 5th-generation Intel Xeon CPUs. We use the Agilex-7 FPGA board as the CXL device and BF-3 as the PCIe NIC device, both of which are connected to the host CPU through PCIe-5.0 ×16 lanes. Their specifications are summarized in the Table 1. Agilex-7 consists of a primary FPGA chiplet integrated with two DDR4-2400 DRAM controllers for device memory and up to three 'R-Tile' chiplets, each with one instance of the ASIC-based hardened CXL ×16 endpoint IP and four instances of the hardened PCIe IP. It also includes a 'Soft R-Tile Wrapper' and 'Soft Support Logic,' which interface between the FPGA and the hardened CXL IP. Agilex-7 can serve as either a CXL Type-1 device or CXL Type-2 device depending on user configurations. BF-3 incorporates a 400 Gbps RDMA NIC with Arm CPU cores alongside specialized subsystems, including acceleration engines for various network functions and a DDR5-5200 DRAM controller connected to 32GB DRAM. To ensure measurement stability, we disable hyper-threading and fix the host CPU frequency at 2.2 GHz [1, 19].

We implement the CXL-NIC components shown in Figure 5 on the FPGA chiplet and integrate the KVS functions into the on-NIC KVS request handler. To isolate the benefits of CXL, the ideal baseline would implement the same components on the FPGA chiplet while replacing CXL requests with PCIe-based DMA or MMIO. However, the current Agilex-7 platform lacks mature hardware and software support to enable this setup. Additionally, as discussed in Section 5.1, H2D access to memory on a conventional PCIe NIC requires intrusive firmware or OS modifications and typically exposes only a limited amount of SRAM to the host. This constraint hinders a comprehensive evaluation of different networking data memory layouts (§5.2). Similarly, due to hardware constraints that prevent Agilex-7 from directly interfacing with other off-the-shelf NICs, we develop a custom packet generator on the FPGA chiplet to simulate inbound network traffic. This generator supports runtime configuration of packet rates and sizes. Current synthesis limits and CXL IP constraints restrict the FPGA clock frequency to 400 MHz, which bounds the maximum achievable packet rate from the generator. We expect these limitations to be mitigated by more mature CXL IP designs and future ASIC-based CXL devices, which can achieve higher performance with dedicated hardware at the cost of programmability.

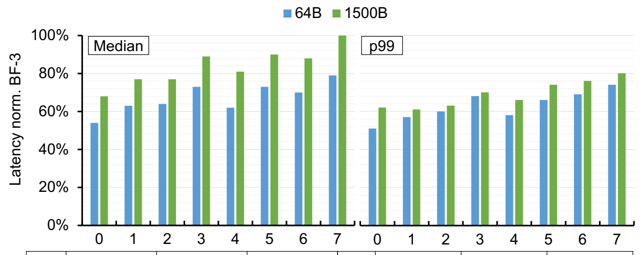
’

7 Evaluation

We evaluate CXL-NIC with loopback tests and application benchmarks. The loopback tests follow the same methodology as PCIe profiling in Section 3.1 to compare CXL-NIC against BF-3. More specifically, a hardware packet generator emulates inbound traffic and embeds timestamps in payloads. CXL-NIC receives the packets from the generator, transfers them to the Rx packet buffer, and notifies the host via RxD updates. Dedicated CPU cores poll on the Rx ring buffer, copy the packets to Tx packet buffer, and put TxDs to the Tx ring buffer. CXL-NIC polls on the ring buffer to detect new TxD, fetches the packet, and calculates the loopback latency as the interval from the packet generation to its return to CXL-NIC after traversing the Rx and Tx datapaths. For Type-1 CXL-NIC, we evaluate performance across CXL.cache request types following the datapaths described in Section 4.2. By testing each request type with host-memory-resident data structures, we identify the optimal configuration for latency and bandwidth contention. For Type-2 CXL-NIC, we extend the analysis to evaluate the benefits of NIC memory by testing networking data allocations across host and NIC memory alongside various types of CXL.cache/CXL.mem requests. We also compare against the layouts suggested by CC-NIC [48]. For application evaluation, we follow the RAMBDA [61] methodology as the baseline, where BF-3 processes requests on ARM cores and retrieves missing key-value entries from the host via RDMA. To evaluate CXL-NIC, we adapt a packet generator to issue KVS requests, implementing the workflow in Section 5.4, where the KVS request handler processes the requests with local NIC memory and issues CXL.cache requests to host memory on misses.

7.1 Benefits of Coherence

In this section, we evaluate the performance of Type-1 CXL-NIC, which keeps networking data and metadata structures (*e.g.*, packet buffers) in host memory to isolate the benefits attributed to the CXL.cache protocol. We use loopback tests to measure latency and throughput across different CXL.cache request combinations. Since each CXL.cache operation supports multiple request types, we have a big request type combination space to explore. To narrow it, we focus on three operations: Rx/Tx packet transfer for data movement and TxD polling for synchronization, as loopback latency tests are most sensitive to these. Other operations are either off the critical path, such as RxD prefetch and TxD completion update, or insensitive to the request types, such as RxD completion, where CPU polling obscures latency differences across request types.



ID	Rx	TxD	Tx	ID	Rx	TxD	Tx
0	NC-write	NC-read	NC-read	4	CO-write	NC-read	NC-read
1			CS-read	5			CS-read
2		CO-read	NC-read	6		CO-read	NC-read
3			CS-read	7			CS-read

Figure 10: Unloaded loopback median (left) and p99 (right) latency in CXL-NIC with various request type combinations (*Comb.* 0-7 on X-axis) normalized to BF-3.

Latency. Figure 10 shows the unloaded loopback latency between CPU and CXL-NIC, normalized to BF-3. Each bar corresponds to a unique request combination (*Comb.*) with IDs 0 through 7. The table below the plot details these combinations, where **Rx** represents Rx packet transfer, **TxD** for TxD polling, and **Tx** for Tx packet transfer. We provide two request types per operation due to space limits. *Comb.* 0 achieves the lowest median and 99th percentile (p99) tail latency among the configurations by using only non-coherent (`NC`) requests for all operations, reducing median latency by 46% (32%) and p99 latency by 49% (38%) 64B (1500B) packet relative to the BF-3 baseline. Switching from `NC-read` to `CS-read` for Tx packet transfers (*Comb.* 1) introduces 17% (64B) and 13% (1500B) median latency increase as `CS-read` forces the cache line into a *Shared* state in NIC cache, incurring extra coherence traffic. `CO-write` for Rx packet transfers (*Comb.* 4) raises median latency by 15% (64B) and 19% (1500B) compared to `NC-write` (*Comb.* 0), due to two factors: (1) `CO-write` requires exclusive cache line ownership, incurring extra coherence traffic and (2) CPU reads packet from HMC across the CXL when `CO-write` stores the packet in HMC in *Owned* state, whereas `NC-write` ’s direct, uncached writes to host memory allow the CPU to read from local DRAM. While `CO-read`-based TxD polling (*Comb.* 2) reduces bandwidth consumed by polling traffic, as shown in Figure 6, it suffers from a 19% (64B) and 13% (1500B) latency penalty compared to *Comb.* 1 due to exclusiveness acquiring traffic. In this sense, CXL-NIC favors *decoupled coherence strategies*: data movement operations employ `NC` requests to avoid cache-to-cache transfers and reduce coherence traffic caused by cache line state control, while synchronization operations requires cacheable operations (*e.g.*, `CO-read`) to amortize bandwidth costs and prevent control contention between the host CPU and NIC, despite moderate latency penalties.

Throughput. We measure CXL-NIC’s maximum achievable throughput under continuous 1500B packet transfers to minimize synchronization overhead, with sustained descriptor availability and no host CPU bottlenecks. RxDs are prefetched to the NIC cache via `CS-read`, while `CO-read`-based TXD polling is used to reduce CXL bandwidth consumption. Under the FPGA synthesis limitations and CXL IP constraints, the FPGA frequency is up to 400 MHz issuing single 64B CXL.cache request per cycle, resulting in a 204 Gbps theoretical throughput. Figure 11 shows the Rx and Tx throughput with various CXL.cache requests for data movement, normalized to this limit. For the Rx datapath, `NC-write` and `NC-P` reach 90% of the throughput limit, while `CO-write` sustains only 73% due to coherence overhead. For the Tx datapath, `NC-read` reaches 62% of the throughput limit, whereas `CO-read` and `CS-read` achieve only 48%. This disparity arises from (1) in the Tx datapath, CXL D2H reads traverse the interconnect twice to fetch data, whereas in the Rx datapath, D2H writes are single-trip; and (2) additional bandwidth consumed by TxD fetching. While evaluated with a single descriptor ring, CXL-NIC natively supports multiple ring buffers by replicating DMU-PFU pairs with unique descriptor/packet buffer addresses per instance, enabled by software-level isolation and non-interfering coherence semantics. Our dual-ring implementation confirms this functional viability while we do not see notable throughput improvement due to FPGA frequency constraints and CXL IP bandwidth limitations. Higher throughput is expected with more mature hardware (*e.g.*, ASICs) and CXL IP designs.

Batching improves throughput by amortizing synchronization overhead across packets. CXL-NIC supports batching by setting batch size to DMU (Figure 5-❸). For a batch size *N*, in Rx datapath, DMU passes *N* packet buffer addresses to PFU, which then issues CXL.cache requests back-to-back to write *N* packets while only updates the final RxD to signal completion. In Tx datapath, when TxD update is detected, DMU reads *N* TxDs concurrently and forwards *N* packet buffer addresses to PFU for parallel packet reads. Figure 12 shows the Rx and Tx throughput of 1500B packet with various batch sizes using `NC` requests, normalized to the corresponding maximum achievable throughput. At the batch size of 8, Rx datapath achieves 95% of maximum throughput, while Tx requires a batch size of 32 to achieve 88%, still marking a significant improvement over the 31% without batching.

7.2 Benefits of NIC Memory

In this section, we evaluate the Type-2 CXL-NIC, which incorporates coherent NIC memory exposed to the CPU as remote NUMA nodes. This broadens the design space by enabling flexible memory allocation of networking data structures alongside request type selections (Figure 8). While we conducted extensive experiments for tens of design choices, we focus here on two key findings: (1) the impact of packet buffer placement, and (2) a latency-optimal design derived from co-optimizing data structure placement and request types.

Packet buffer placement. To isolate the impact of packet buffer placement, we allocate descriptor rings in host memory using `NC-write` for RxD completions and `CO-read` for TxD polling, while restricting data movement to use `NC-write`/`NC-read`. When packet buffers are allocated in NIC memory, we use D2D `NC-write`/`NC-read` in host-bias mode. We evaluate four **L**ayouts: **L1**: both Rx and Tx packet buffer in host memory; **L2**: Rx in NIC memory, Tx in host memory; **L3**: Rx in host memory, Tx in NIC memory; **L4**: both Rx and Tx in NIC memory. Figure 13 shows the median and p99 latency of 64B and 1500B packet, normalized to latency in L1. Layouts with buffers in device memory (L2–L4) consistently underperform L1. Specifically, L2, L3 and L4 exhibit 23%(38%), 54%(35%) and 71%(74%) higher median latency for 64B (1500B) packet than L1. This degradation arises from two sources: (1) the CPU accesses to NIC memory over the CXL interconnect, and (2) coherence protocol overhead in host-bias mode, which requires coherence checking with CPU, which nullifies the spatial-locality benefit of D2D access to NIC memory. The degradation implies that an uninformed networking data placement in CXL-NIC memory hurts the end-host networking performance, necessitating a elaborated cooperation with request selection. Although naïve NIC-resident buffers degrade latency, CXL.mem enables scalable NIC memory expansion to store networking data structures (*e.g.*, hundreds of descriptor rings to support concurrent connections).

Latency-optimal configuration. Our comprehensive evaluation identifies the latency-optimal CXL-NIC configuration through co-optimizing memory placement and coherence request selections. The design allocates the Rx descriptor ring, Rx packet buffer, and Tx packet buffer in NIC memory, while retaining the Tx descriptor ring in host memory. For the Rx datapath, CXL-NIC employs `CS-read` to cache RxD in NIC cache and `NC-P` for Rx packet transfers and completion. For the Tx datapath, the host CPU `nt-st` the packet to Tx packet buffers and CXL-NIC employs `NC-read` to poll TxD and device-bias mode `NC-read` for packet transfer. Figure 14

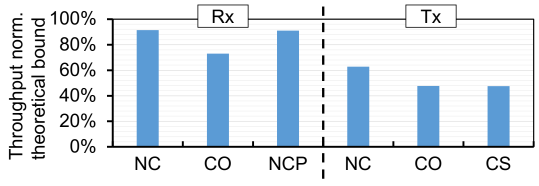


Figure 11: Maximum achievable Rx and Tx throughput with different types of CXL requests.

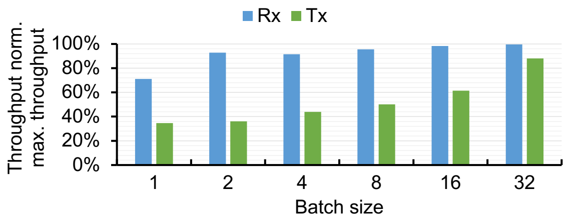


Figure 12: Throughput with various batch sizes for 1500B packet normalized to maximum achievable throughput.

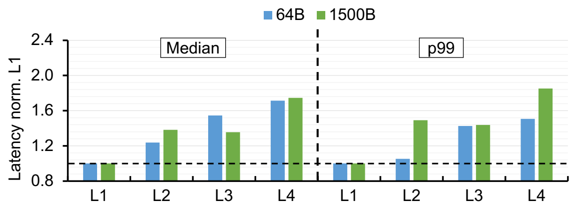


Figure 13: Latency with different packet buffer allocations in CXL-NIC.

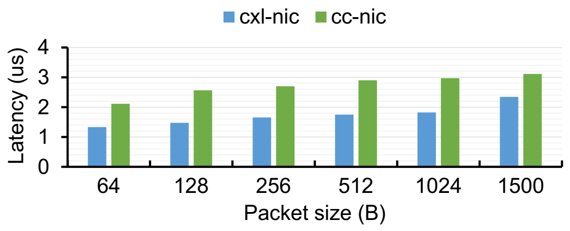


Figure 14: Latency of CXL-NIC and CC-NIC.

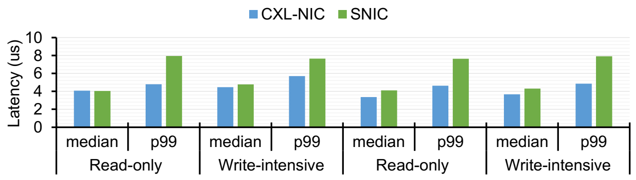


Figure 15: Latency performance of different KVS designs on SNIC and CXL-NIC on two workloads.

compares our latency-optimal design against CC-NIC [48] design, which proposes writer-homed descriptor ring buffers and places all packet buffers in host memory. We implement CC-NIC using `CS-read` and `CO-write` for data movement and synchronization to align with UPI `ld/st`. CXL-NIC achieves 37% lower median latency than CC-NIC on average across the packet size from 64B to 1500B. The reduction can mainly be attributed to (1) the intelligent usage of `NC-P` (§5.3), which pushes the RxD and Rx packet into host LLC for faster host CPU access and (2) the `NC-read` for data movement in Tx, as observed in Section 7.1.

KVS performance. Figure 15 compares the median and p99 tail latencies of KVS requests across two workloads: read-only (100% GET operations) and write-intensive (50% GET and 50% PUT operations), under Zipfian and uniform access distributions. Under a Zipfian distribution with read-only workload, CXL-NIC achieves 18% median latency reduction and 39% p99 latency reduction compared to SNIC. This improvement stems from the fast NIC memory accesses under device-bias mode, which aligns with the locality inherent in Zipfian distributions. When switching from Zipfian to uniform distribution, CXL-NIC exhibits an average 20% median latency increase across both workloads, with no comparable change in p99 latency. This occurs because uniform distribution spreads requests more evenly, forcing more frequent host memory accesses in host-bias mode, which incurs higher latency.

8 Discussion

Multiple-producer multiple-consumer support. Current CXL 1.1-based devices lack support for atomic operations such as Compare-and-Swap (CAS), which are essential for implementing lock-free Multiple-Producer-Multiple-Consumer (MPMC) queues. In the TX datapath, where the CXL-NIC fetches descriptors from the Tx descriptor ring buffer, multiple CXL-NIC engines may concurrently update the shared head index. Without CAS, this coordination is unsafe, leading to potential descriptor duplication or loss. In the RX datapath, where the CXL-NIC writes completions, multiple engines must safely update the shared tail index of the Rx descriptor ring buffer to signal newly received packets without write conflicts. These synchronization requirements make CAS or similar atomic operation support critical for correct MPMC operation on the CXL-NIC side. As CXL 1.1 lacks CAS support, such MPMC coordination on the NIC side is deferred to future work, pending CAS availability in CXL 2.0 and beyond [3].

Security implications & Mitigation. CXL-NIC allows the host to access larger device memory regions than PCIe-NIC, which only exposes limited memory through PCIe BARs. However, the host can protect these regions through its MMU as it does for its remote NUMA node. Meanwhile, CXL.cache, used by CXL-NIC, enables devices to access host memory regions in a cache-coherent manner. Since both PCIe and CXL devices (without the IOMMU and an equivalent mechanism) can directly access them with Host Physical Addresses (HPAs)—through DMA for the former and D2H accesses for the latter—they can cause out-of-bound memory accesses. To prevent them, CXL 2.0 introduces a protection mechanism that restricts memory access of each CXL device to pre-assigned HPA ranges, allowing only valid requests to be translated and served [52]. However, this model supports only a limited number of HPA regions per device, making it coarse-grained and inflexible. MMU-like protections (*e.g.*, using PCIe Address Translation Services) may address the limitation but they are required to be implemented by device manufacturers. Side-channel attacks on the shared caches between two processes have been explored, such as Flush+Reload [59] and Prime+Probe [34]. As CXL devices can access the host LLC, they naturally become strong candidates for executing such attacks. We expect that existing solutions to defend against similar attacks [56, 64] can be adapted for CXL devices.

Virtualization. Virtualizing CXL-NIC allows multiple virtual machines (VMs) to share a single device, improving hardware utilization and enabling scalable deployment in multi-tenant environments. Since CXL is built on top of PCIe, CXL devices can seamlessly support Single Root I/O Virtualization (SR-IOV) as one of PCIe's extended capabilities [8]. SR-IOV enables a single physical device (*i.e.*, Physical Function (PF)) to expose multiple Virtual Functions (VFs) to VMs. However, CXL devices should implement MMU-like mechanisms within the device for secure sharing, as they are not protected or enforced by the IOMMU like PCIe devices. Otherwise, the hypervisor must emulate MMU-like mechanisms by managing the HPA range filters to confine each VF to its assigned memory regions in host memory and CXL memory. For CXL-NIC specifically, the hypervisor may isolate networking data structures (*e.g.*, queues and doorbell registers) across VFs and mitigate coherence-related side channels introduced by shared LLC access via CXL.cache as discussed before. These remain open challenges in current CXL implementations.

9 Related Work

CPU-NIC interaction. The interaction between host and NICs is increasingly constrained by PCIe limitations, particularly as network speeds exceed 100 Gbps. Early work demonstrated that PCIe's non-coherent memory access and protocol overheads lead to underutilized bandwidth and microsecond-level delays, even with advanced features like Intel DDIO [37]. Over time, there have been efforts on moving the NIC closer or more integrated to the CPU [2, 9, 15, 33, 38]. However, these proposals always lack practicality, as they are always intrusive and not modular, leading to high design cost and total cost of ownership (TCO). Recent efforts like CC-NIC [48], Dagger [26], and Enzian [7] address these challenges by leveraging proprietary coherence protocols like Intel UPI to integrate NICs into the CPU coherence domain, achieving sub-50 ns latency and hardware-managed cache synchronization. However, such proprietary protocols and implementations restrict generalizable insights, as their design choice relies on the case-by-case measurement and lacks exploration of low-level fine-grained capabilities like cache-line state manipulation. Such limitations render their inefficiency with real CXL devices as we demonstrated in the Section 7. Meanwhile, CXL's open standard and protocol flexibility enable scalable, low-latency host-NIC interaction, positioning it as a sustainable alternative for next-gen infrastructure. With CXL-NIC framework, more analysis and exploration can also be conducted for any domain-specific use cases.

From the software stack perspective, there have also been attempts to facilitate CPU-NIC communication, such as user-space networking stack like DPDK [16] and mTCP [20]. Research like Enso [43] proposes communication models different than the current packetized one, motivating a streaming interface to better support NIC interaction. CXL-NIC is complementary to these innovations. For example, CXL-NIC can also take Enso's approach to pre-allocate a large buffer and write a batch of data before syncing with the memory system.

NIC offload. A growing body of research targets NICs as platforms for offloading computationally intensive OS functions and stacks, along with applications, to reduce host CPU overhead and improve scalability. Networking functions and protocol stacks are a natural focus for NIC offloading, with SNICs accelerating operations such as TCP/IP processing, virtualization-layer packet handling, and traffic management [4, 12, 27, 32, 36, 45, 47, 51]. Beyond networking, OS-layer abstractions like memory management [21, 28, 57] and RPC frameworks [10, 26, 42, 50, 55] have been offloaded to NICs to eliminate host intervention. At the application level, NIC offloading enables high performance for common datacenter workloads [5, 23, 29, 46]. Workloads such as distributed storage systems [25] and interactive services [6] can execute directly on NICs, bypassing host CPUs entirely. Our CXL-enabled NIC design complements these existing approaches while introducing new optimization opportunities. By leveraging cache coherence and low-latency communication offered by CXL, the architecture can enhance offloading efficiency by replacing the non-coherent, inefficient PCIe operations while potentially simplifying offloading mechanisms through native hardware-managed coherence.

10 Conclusion

We present CXL-NIC, a novel hardware-software co-design for high-performance NIC, leveraging the great potential of the emerging CXL protocols. Leveraging cache coherence, memory capacity, and offloading capability, CXL-NIC greatly facilitates CPU-NIC synchronization, data movement, and function/application acceleration. Our FPGA-based prototype demonstrates that, compared to a commodity PCIe-based NIC, CXL-NIC is able to achieve 49% and 39% lower latency for packet processing and KVS request processing, respectively.

Acknowledgments

This work was supported in part by grants from PRISM, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA and the IBM-Illinois Discovery Accelerator Institute (IIDAI); by the MSIT, Korea, under the Global Scholars Invitation Program (RS-2024-00456287) supervised by the IITP; by the Technology Innovation Program (No. RS-2024-00420541, 2410000802) funded by the Ministry of Trade, Industry and Energy of Korea; and by a generous gift from AMD-Xilinx HACC. Nam Sung Kim is the corresponding author.

References

[1] Bilge Acun, Phil Miller, and Laxmikant V Kale. 2016. Variation Among Processors Under Turbo Boost in HPC Systems. In *Proceedings of the 2016 International Conference on Supercomputing (ICS'16)*.

[2] Mohammad Alian and Nam Sung Kim. 2019. Netdimm: Low-Latency Near-Memory Network Interface Architecture. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'19)*.

[3] Altera. accessed in 2025. Agilex 7 R-Tile Compute Express Link 1.1/2.0 FPGA IP User Guide.

[4] Emmanuel Amaro, Zhihong Luo, Amy Ousterhout, Arvind Krishnamurthy, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. 2020. Remote Memory Calls. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks (HotNets'20)*.

[5] Michaela Blott, Kimon Karras, Ling Liu, Kees Vissers, Jeremia Bär, and Zsolt István. 2013. Achieving 10gbps Line-rate Key-Value Stores with FPGAs. In *Proceedings of the 5th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'13)*.

[6] Sean Choi, Muhammad Shahbaz, Balaji Prabhakar, and Mendel Rosenblum. 2020. λ-NIC: Interactive Serverless Compute on Programmable SmartNICs. In *Proceedings of the 2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS'20)*.

[7] David Cock, Abishek Ramdas, Daniel Schwyn, Michael Giardino, Adam Turowski, Zhenhao He, Nora Hossle, Dario Korolija, Melissa Licciardello, Kristina Martsenko, Reto Achermann, Gustavo Alonso, and Timothy Roscoe. 2022. Enzian: An Open, General, CPU/FPGA Platform for Systems Software Research. In *Proceedings of the 27th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'22)*.

[8] Compute Express Link Consortium. accessed in 2025. Compute Express Link Specification Revision 3.1. https://computeexpresslink.org/wp-content/uploads/2024/02/CXL-3.1-Specification.pdf.

[9] Alexandros Daglis, Stanko Novaković, Edouard Bugnion, Babak Falsafi, and Boris Grot. 2015. Manycore Network Interfaces for In-Memory Rack-Scale Computing. In *Proceedings of the 2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA'15)*.

[10] Alexandros Daglis, Mark Sutherland, and Babak Falsafi. 2019. RPCValet: NI-Driven Tail-Aware Balancing of μs-Scale RPCs. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*.

[11] Debendra Das Sharma, Robert Blankenship, and Daniel Berger. 2024. An Introduction to the Compute Express Link (CXL) Interconnect. *ACM Comput. Surv.* (2024).

[12] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Lavier Jack, Lam Norman, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Madhan Silva, Ganriel nd Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, Dvid A. Maltz, and Albert Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*.

[13] Mario Flajslik and Mendel Rosenblum. 2013. Network Interface Design for Low Latency Request-Response Protocols. In *Proceedings of the 2013 USENIX Annual Technical Conference (ATC'13)*.

[14] Wentao Hou, Jie Zhang, Zeke Wang, and Ming Liu. 2024. Understanding Routable {PCIe} Performance for Composable Infrastructures. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI'24)*.

[15] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Changhoon Kim, and Nick McKeown. 2021. The nanoPU: A Aanosecond Network Stack for Datacenters. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI'21)*.

[16] Intel. accessed in 2025. Data Plane Development Kit (DPDK). https://www.dpdk.org.

[17] Intel Corporation. accessed in 2025. Agilex™ 7 FPGA I-Series Development Kit. https://www.intel.com/content/www/us/en/products/details/fpga/development-kits/agilex/agi027.html.

[18] Intel Corporation. accessed in 2025. Intel Xeon Processor Scalable Family Technical Overview. https://www.intel.com/content/www/us/en/developer/articles/technical/xeon-processor-scalable-family-technical-overview.html.

[19] Intel Corporation. accessed in 2025. Optimizing Computer Applications for Latency: Part 1: Configuring the Hardware. https://www.intel.com/content/www/us/en/developer/articles/technical/optimizing-computer-applications-for-latency-part-1-configuring-the-hardware.html.

[20] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*.

[21] Houxiang Ji, Mark Mansi, Yan Sun, Yifan Yuan, Jinghan Huang, Reese Kuper, Michael M. Swift, and Nam Sung Kim. 2023. STYX: Exploiting SmartNIC Capability to Reduce Datacenter Memory Tax. In *Proceedings of the 2023 USENIX Annual Technical Conference (ATC'23)*.

[22] Houxiang Ji, Srikar Vanavasam, Yang Zhou, Qirong Xia, Jinghan Huang, Yifan Yuan, Ren Wang, Pekon Gupta, Bhushan Chitlur, Ipoom Jeong, et al. 2024. Demystifying a CXL Type-2 Device: A Heterogeneous Cooperative Computing Perspective. In *Proceedings of the 57th IEEE/ACM International Symposium on Microarchitecture (MICRO'24)*.

[23] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. Netcache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th symposium on operating systems principles (SOSP'17)*.

[24] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2016. Design Guidelines for High Performance RDMA Systems. In *Proceedings of the 2016 USENIX annual technical conference (ATC'16)*.

[25] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostić, Youngjin Kwon, Simon Peter, and Emmett Witchel. 2021. LineFS: Efficient SmartNIC Offload of a Distributed File System with Pipeline Parallelism. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP'21)*.

[26] Nikita Lazarev, Shaojie Xiang, Neil Adit, Zhiru Zhang, and Christina Delimitrou. 2021. Dagger: Efficient and Fast RPCs in Cloud Microservices with near-Memory Reconfigurable NICs. In *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*.

[27] Yanfang Le, Hyunseok Chang, Sarit Mukherjee, Limin Wang, Aditya Akella, Michael M Swift, and TV Lakshman. 2017. UNO:Unifying Host and Smart NIC Offload for Flexible Packet Processing. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC'17)*.

[28] Seung-seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. 2021. Mind: In-Network Memory Management for Disaggregated Data Centers. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP'21)*.

[29] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. 2017. KV-Direct: High-performance In-memory Key-Value Store with Programmable NIC. In *Proceedings of the ACM SIGOPS 26th Symposium on Operating Systems Principles (SOSP'17)*.

[30] Bojie Li, Zihao Xiang, Xiaoliang Wang, Han Ruan, Jingbin Zhou, and Kun Tan. 2023. FastWake: Revisiting Host Network Stack for Interrupt-mode RDMA. In *Proceedings of the 7th Asia-Pacific Workshop on Networking (APNet'23)*.

[31] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. 2023. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23)*.

[32] Jialin Li, Ellis Michael, and Dan RK Ports. 2017. Eris: Coordination-Free Consistent Transactions Using In-Network Concurrency Control. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*.

[33] Kevin Lim, David Meisner, Ali G. Saidi, Parthasarathy Ranganathan, and Thomas F. Wenisch. 2013. Thin Servers with Smart Pipes: Designing SoC Accelerators for Memcached. In *Proceedings of the 2013 ACM/IEEE 40th Annual International Symposium on Computer Architecture (ISCA'13)*.

[34] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. 2015. Last-level cache side-channel attacks are practical. In *Proceedings of the 2015 IEEE symposium on security and privacy (S&P'15)*.

[35] Jinshu Liu, Hamid Hadian, Yuyue Wang, Daniel S. Berger, Marie Nguyen, Xun Jian, Sam H. Noh, and Huaicheng Li. 2025. Systematic CXL Memory Characterization and Performance Analysis at Scale. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'25)*.

[36] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. 2020. AccelTCP: Accelerating Network Applications with Stateful TCP Offloading. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*.

[37] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W Moore. 2018. Understanding PCIe Performance for End Host Networking. In *Proceedings of the 2018 ACM SIGCOMM Conference (SIGCOMM'18)*.

[38] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. 2014. Scale-out NUMA. In *Proceedings of the 19th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*.

[39] NVIDIA Corporation. accessed in 2025. DOCA Documentation. https://docs.nvidia.com/doca/sdk/doca+dma/index.html.

[40] NVIDIA Corporation. accessed in 2025. NVIDIA BlueField-3 DPU. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-3-dpu.pdf.

[41] Boris Pismenny, Liran Liss, Adam Morrison, and Dan Tsafrir. 2022. The Benefits of General-Purpose On-NIC Memory. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'22)*.

[42] Arash Pourhabibi, Mark Sutherland, Alexandros Daglis, and Babak Falsafi. 2021. Cerebros: Evading the RPC Tax in Datacenters. In *Proceedings of the 54th IEEE/ACM International Symposium on Microarchitecture (MICRO'21)*.

[43] Hugo Sadok, Nirav Atre, Zhipeng Zhao, Daniel S. Berger, James C. Hoe, Aurojit Panda, Justine Sherry, and Ren Wang. 2023. Enso: A Streaming Interface for NIC-Application Communication. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI'23)*.

[44] Khaled Salah. 2007. To Coalesce or Not To Coalesce. *AEU-International Journal of Electronics and Communications* 61, 4 (2007), 215–225.

[45] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. 2017. In-Network Computation is a Dumb Idea Whose Time Has Come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks (HotNets'17)*.

[46] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtárik. 2021. Scaling Distributed Machine Learning with In-Network Aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI'21)*.

[47] Anirudh Sarma, Hamed Seyedroudbari, Harshit Gupta, Umakishore Ramachandran, and Alexandros Daglis. 2022. Nfslicer: Data Movement Optimization for Shallow Network Functions. *arXiv preprint arXiv:2203.02585* (2022).

[48] Henry N Schuh, Arvind Krishnamurthy, David Culler, Henry M Levy, Luigi Rizzo, Samira Khan, and Brent E Stephens. 2024. CC-NIC: A Cache-Coherent Interface to the NIC. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'24)*.

[49] Henry N Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. 2021. Xenic: SmartNIC-Accelerated Distributed Transactions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP'21)*.

[50] Hamed Seyedroudbari, Srikar Vanavasam, and Alexandros Daglis. 2023. Turbo: SmartNIC-enabled Dynamic Load Balancing of μs-scale RPCs. In *Proceedings of the 2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA'23)*.

[51] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. 2020. StRoM: Smart Remote Memory. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys'20)*.

[52] Samuel W Stark, A Theodore Markettos, and Simon W Moore. 2023. How Flexible is CXL's Memory Protection? Replacing a sledgehammer with a scalpel. *Queue* (2023).

[53] Yan Sun, Jongyul Kim, Zeduo Yu, Jiyuan Zhang, Siyuan Chai, Michael Jaemin Kim, Hwayong Nam, Jaehyun Park, Eojin Na, Yifan Yuan, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. 2025. M5: Mastering Page Migration and Memory Management for CXL-based Tiered Memory Systems. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'25)*.

[54] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, et al. 2023. Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices. In *Proceedings of the 56th IEEE/ACM International Symposium on Microarchitecture (MICRO'23)*.

[55] Mark Sutherland, Siddharth Gupta, Babak Falsafi, Virendra Marathe, Dionisios Pnevmatikatos, and Alexandros Daglis. 2020. The NeBuLa RPC-Optimized Architecture. In *Proceedings of the 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA'20)*.

[56] Venkatanathan Varadarajan, Thomas Ristenpart, and Michael Swift. 2014. Scheduler-based defenses against {Cross-VM} side-channels. In *Proceedings of the 23rd USENIX security symposium (USENIX security 14)*.

[57] Qing Wang, Youyou Lu, Erci Xu, Junru Li, Youmin Chen, and Jiwu Shu. 2021. Concordia: Distributed Shared Memory with In-Network Cache Coherence. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST'21)*.

[58] Daliang Xu, Hao Zhang, Liming Yang, Ruiqi Liu, Gang Huang, Mengwei Xu, and Xuanzhe Liu. 2025. Fast On-device LLM Inference with NPUs. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'25)*.

[59] Yuval Yarom and Katrina Falkner. 2014. {FLUSH+ RELOAD}: A high resolution, low noise, l3 cache {Side-Channel} attack. In *Proceedings of the 23rd USENIX security symposium (USENIX security'14)*.

[60] Yifan Yuan, Mohammad Alian, Yipeng Wang, Ren Wang, Ilia Kurakin, Charlie Tai, and Nam Sung Kim. 2021. Don't Forget the I/O When Allocating Your LLC. In *Proceedings of the 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA'21)*.

[61] Yifan Yuan, Jinghan Huang, Yan Sun, Tianchen Wang, Jacob Nelson, Dan RK Ports, Yipeng Wang, Ren Wang, Charlie Tai, and Nam Sung Kim. 2023. Rambda: RDMA-driven Acceleration Framework for Memory-intensive μs-scale Datacenter Applications. In *Proceedings of the 2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA'23)*.

[62] Shaoxun Zeng, Minhui Xie, Shiwei Gao, Youmin Chen, and Youyou Lu. 2025. Medusa: Accelerating Serverless LLM Inference with Materialization. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'25)*.

[63] Yuhong Zhong, Daniel S Berger, Carl Waldspurger, Ryan Wee, Ishwar Agarwal, Rajat Agarwal, Frank Hady, Karthik Kumar, Mark D Hill, Mosharaf Chowdhury, et al. 2024. Managing Memory Tiers with CXL in Virtualized Environments. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI'24)*.

[64] Ziqiao Zhou, Michael K Reiter, and Yinqian Zhang. 2016. A software approach to defeating side channels in last-level caches. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS'16)*.