CC-NIC: a Cache-Coherent Interface to the NIC

Henry N. Schuh
Google & University of Washington
USA

Arvind Krishnamurthy
Google & University of Washington
USA

David Culler
Google, USA

Henry M. Levy
Google & University of Washington
USA

Luigi Rizzo
Google, Italy

Samira Khan
Google & University of Virginia, USA

Brent E. Stephens
Google & University of Utah, USA

Abstract

Emerging interconnects make peripherals, such as the network interface controller (NIC), accessible through the processor's cache hierarchy, allowing these devices to participate in the CPU cache coherence protocol. This is a fundamental change from the separate I/O data paths and read-write transaction primitives of today's PCIe NICs. Our experiments show that the I/O data path characteristics cause NICs to prioritize CPU efficiency at the expense of inflated latency, an issue that can be mitigated by the emerging low-latency coherent interconnects. But, the coherence abstraction is not suited to current host-NIC access patterns. Applying existing signaling mechanisms and data structure layouts in a cache-coherent setting results in extraneous communication and cache retention, limiting performance. Redesigning the interface is necessary to minimize overheads and benefit from the new interactions coherence enables. This work contributes CC-NIC, a host-NIC interface design for coherent interconnects. We model CC-NIC using Intel's Ice Lake and Sapphire Rapids UPI interconnects, demonstrating the potential of optimizing for coherence. Our results show a maximum packet rate of 1.5Gpps and 980Gbps packet throughput. CC-NIC has 77% lower minimum latency, and 88% lower at 80% load, than today's PCIe NICs. We also demonstrate application-level core savings. Finally, we show that CC-NIC's benefits hold across a range of interconnect performance characteristics.

ACM Reference Format:
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 *28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3617232.3624868

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

ACM ISBN 979-8-4007-0372-0/24/04.
https://doi.org/10.1145/3617232.3624868

1 Introduction

A wide range of new interconnects is emerging for accelerators, disaggregated memory, and multi-GPU systems. PCI Express (PCIe) [37] has long been the standard interconnect between a server and peripheral devices, such as the network interface controller (NIC). While PCIe bandwidth has increased substantially over the seven protocol generations, its interface for host-device communication has remained consistent. Now, new interconnect specifications [2, 3, 35, 36, 49, 50] propose to either replace or build upon the PCIe physical layer, while providing fundamentally different data paths and communication abstractions between the host and the peripheral.

A key attribute of these interconnects is allowing the host and devices to participate in coherence protocols. Hosts can access devices through the processor's highly optimized cache hierarchy, and devices can participate in the CPU's cache coherence protocol while accessing memory. These interconnects enable devices to be integrated into the host processor's coherence domain in different settings. For instance, Compute Express Link (CXL) [3] targets devices housed on expansion cards, Ultra Path Interconnect (UPI) [9, 14] is an inter-socket interconnect that also allows for the integration of hardware devices (e.g., Intel Agilex FPGA [23, 39]), and Cache Coherence Interconnect for Accelerators (CCIX) [2] proposes a coherent interface for chiplet-based systems.

Coherent device access to shared memory is a powerful programming model for data sharing, providing semantics not available with the typical read/write primitives of PCIe transactions. PCIe uses specialized data paths for transfers between the CPU and the device: CPUs access devices using memory-mapped I/O (MMIO) transfers that bypass the cache. Devices access host state using direct memory access (DMA). DMAs traditionally target data in host DRAM and place DRAM on the critical path for device access, although newer platforms add a limited form of cache interactions [15]. In contrast, coherent interconnects integrate with the CPU's existing, highly-optimized memory data path. UPI, CCIX, and

CXL directly interface with the coherence protocol and the L2 cache state, handling cache ownership and data transfers to a peripheral. This not only results in a shorter data path to the CPU core (as the device can target L2 instead of LLC or DRAM) but also enables the device to poll locally on cache-coherent state. Likewise, CPU accesses to the device can utilize the caching memory path instead of MMIO.

Coherent interconnects represent a fundamental change in host-device communication, offering new benefits and posing challenges. This paper aims to understand the value of coherent interconnects in the context of NICs. While many emerging interconnect specifications are in flux, we apply existing UPI hardware as a means to explore cache-coherent host-NIC interface designs and develop principles that could apply across a range of interconnects.

We first study today's PCIe-based NICs, identifying the unique tradeoffs that PCIe imposes on NIC interfaces. PCIe limits the use of shared data structures and imposes CPU overheads for host-initiated interconnect operations. Today's NIC designs, therefore, aim to minimize host PCIe overheads at the expense of transmission latency by introducing additional signaling trips and batching. The impact on packet latency is significant: the host-NIC loopback latency on a Mellanox CX6 NIC is 2.1us at low load and 6.0us at 80% load, almost an order of magnitude higher than switch traversal.

The streamlined datapaths of coherent interconnects can improve latency for existing NIC interface designs. But, we observe performance is highly sensitive to the access pattern on both sides. The producer-consumer patterns typical of existing NIC interfaces incur significant overheads without an optimized combination of access instructions, data ownership, and cache-line layout decisions. Achieving optimal communication requires data structures specifically designed for coherence. Finally, caching must be carefully managed; data and metadata may be retained in remote caches longer than needed, triggering expensive remote communication upon a future local access. Thus, redesigning the host-NIC interface is required to fully take advantage of coherent interconnects, and doing so allows us to benefit from the new signaling and sharing interactions made possible by coherence.

We present CC-NIC, a host-NIC interface optimized for coherent interconnects. We redesign all aspects of the host-NIC interface (namely, data structures, layouts, and signaling) to take advantage of the new data paths and cache interactions supported by coherent interconnects. To design CC-NIC, we consider the space of access type, layout, homing, and prefetching decisions, for each element of the interface. Our redesign not only offers improved latency but also delegates certain buffer management tasks to the NIC, thus reducing host-side costs.

We demonstrate the performance of CC-NIC over UPI on Intel's Ice Lake and Sapphire Rapids server platforms. CC-NIC demonstrates a packet rate of 1.5Gpps and a minimum TX-RX latency of 494ns. Latency under 80% load is 716ns, an even greater reduction relative to PCIe NICs. Compared to an interface matching a current PCIe NIC, on the same UPI link, our proposed design achieves a 3.3× throughput improvement and 52% minimum latency reduction, in addition to decreased latency under load and terabit bandwidth saturation. We evaluate key-value store and RPC workloads; both show that CC-NIC saturates network bandwidth with up to 50% fewer application threads versus PCIe NICs.

We present CC-NIC as a case study of optimizing coherent host-device interactions. The design of CC-NIC can be applied to other coherent interconnects. Our evaluation shows that CC-NIC's design benefits hold, maintaining consistent relative improvement, across varied interconnect performance characteristics.

2 Dissecting the PCIe Host-NIC Interface

We first analyze the host-device interface of today's PCIe NICs. Our goal is to understand how the characteristics of PCIe drive the interface design of existing NICs. In §2.1, we describe the packet queue interface and its data structures in the context of PCIe. Then, in §2.2, we measure the performance of PCIe access mechanisms. We discuss how these performance characteristics lead to tradeoffs in §2.3– the tradeoffs ultimately dictating the design of PCIe NIC metadata structures, data transfer, and buffer management.

2.1 The Host-NIC Interface

In this section, we focus on packet transmit (TX) and receive (RX) queues, the main transfer interface between host and NIC. This interface is consistent across a wide range of NICs and consists of the following components:

Packet buffers store the data payloads transmitted to or received by the NIC. Buffers are pre-allocated memory chunks. A pointer to each buffer is inserted into a *buffer pool* data structure, typically a queue. Allocating and releasing packet buffers involve dequeuing or enqueuing a pointer from the buffer pool. Packet buffers also include application-level metadata, e.g., length of the packet's headers, etc. Although the driver may use this metadata, it is not transferred to the NIC. The address communicated to the device is the start of the packet payload, which is typically cache-aligned and placed after any application metadata.

Descriptors represent the work requests. Each descriptor contains the address of a corresponding packet buffer. Typical descriptors are 16B, including 8B of tightly-packed metadata, such as the data length. They are organized into a ring buffer implemented as a circular array. On the TX path, the driver writes a descriptor for each submitted TX packet. The descriptor is derived from the driver's configuration state and the per-packet metadata in the TX buffer. The address and packet length are critical components of the TX descriptor; the NIC must have these values before

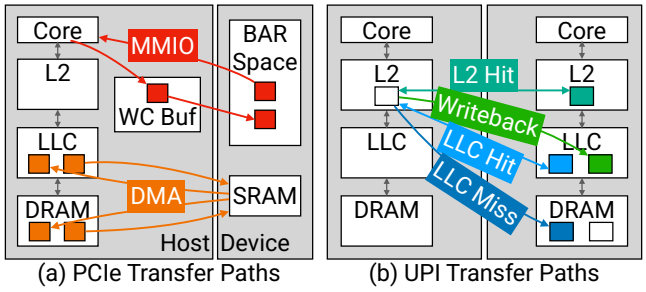

Figure 1. PCIe (a) and UPI (b) transfer paths.

it can access the packet from the host. On the RX path, the host first posts RX descriptors to provide the NIC with blank packet buffers. After the NIC writes a received packet into the RX buffer, it overwrites the descriptor with RX metadata (e.g., completion status).

Head and tail registers serve as signals to coordinate the producer-consumer relationship between the host and NIC. Registers are typically 32 or 64-bit values representing the producer and consumer positions of the ring array. After writing a TX descriptor, the host makes the descriptor available to the NIC by incrementing the TX tail register. When the NIC receives the tail value, it reads and handles packet descriptors up to the new tail index. After the NIC transmits a packet, it signals completion to the host by incrementing the TX head. The host handles transmit completions by returning the freed packets to the buffer pool, reclaiming descriptor ring space.

For the receive path, the host allocates blank RX buffers from the pool, writes their addresses to the RX descriptor ring, and then increments RX head to signal the presence of new RX buffers. When the NIC receives packet data, it uses blank RX buffers at the RX tail index and notifies the host of RX packet availability by incrementing the RX tail. The host handles descriptors up to the RX tail index by returning the RX buffers to the application. In summary, the host writes the TX tail to submit TX packets to the NIC and writes the RX head to submit blank buffers to the NIC. The NIC writes the TX head to indicate transmit completions and writes the RX tail to indicate newly received packets.

2.2 PCIe Microbenchmarks

We now perform a measurement characterization of host-device accesses to understand how PCIe performance dictates host-NIC interface designs. While existing work has identified PCIe limitations [7, 32, 42, 43, 54], we aim to understand the extent to which performance limitations hold on current server platforms and NIC interfaces.

PCIe interconnect latency. PCIe presents an asymmetric interface to the device and the host. Figure 1a shows the mechanisms for transfers initiated by the host and by the device: *MMIO* and *DMA*, respectively.

Host-to-NIC reads and writes are performed via memory-mapped IO (MMIO). The device exposes a memory area mapped into the host address space as an uncacheable (UC) or write-combining (WC) memory type. This allows the host to issue loads and stores to the device, which are executed as PCIe read and write transactions. The UC and WC memory types do not provide cache coherence or operate within the cache hierarchy. Instead, CPU loads always require a PCIe roundtrip, resulting in expensive accesses. On the ICX CPU platform, targeting an Intel E810 NIC (testbed described in §5.1), we measure a median MMIO read latency of 982ns (8B) and 1026ns (64B AVX512).

PCIe devices read and write host memory using Direct Memory Access (DMA). DMAs may be significantly larger (e.g., 4KB) than the 64B MMIO write-combining buffer size and typically access standard writeback host memory. While conventional PCIe NICs do not expose DMA latency statistics, we expect DMA roundtrip latency to be comparable to that of MMIO. SmartNICs that provide a low-level DMA controller interface, such as Marvell's LiquidIO [27], show a minimum DMA read latency of at least 1μs [25, 44].

Implication: The high latency of MMIO and DMA accesses suggests that each PCIe roundtrip contributes significantly to overall packet latency. Host-NIC data structures should also be designed to minimize high-cost CPU operations such as polling or reading across the PCIe bus (MMIO loads).

MMIO write throughput. Unlike loads, MMIO stores are posted, so a store does not incur a PCIe roundtrip delay from the host's perspective. However, MMIO writes are still expensive in the context of both UC and WC memory types. The UC memory type bypasses host caches altogether, so each MMIO access results in a PCIe transfer. To preserve ordering, only one MMIO access may be in flight between the CPU core and PCIe root, thus limiting throughput. The WC memory type offers more flexibility via write-combining store buffers, which can merge contiguous stores within a 64B-aligned region into a single PCIe transfer. This can reduce PCIe protocol overhead but complicates write ordering since writes may be buffered for an arbitrary time before being flushed. To ensure writes are flushed, it is necessary to issue a fence instruction, e.g., `sfence`, or ensure that each 64B buffer is completely filled in sequential order. These flush conditions make it difficult to achieve fine-grained control over PCIe write ordering.

Figure 2 compares the write throughput of WC MMIO accesses to a device, WC-mapped local DRAM, and write-back DRAM. We run this experiment on the ICX platform, targeting the E810 NIC for MMIO accesses using a single thread. We repeat the experiment with write sizes between 64B and 8KB, issuing an `sfence` barrier after each write.

With the WC data path, writing to both PCIe MMIO and DRAM, we find that the barriers, which may be needed to ensure ordering, impact throughput. This is not the case with

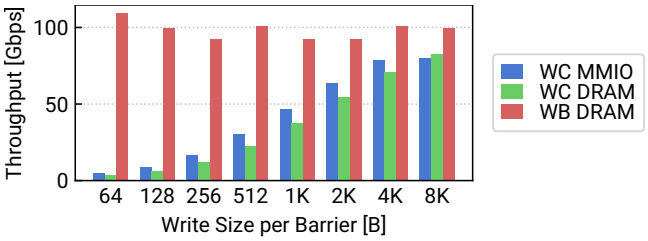

Figure 2. Single-threaded write throughput for WC MMIO (to E810 NIC), WC-mapped DRAM, and regular WB DRAM.

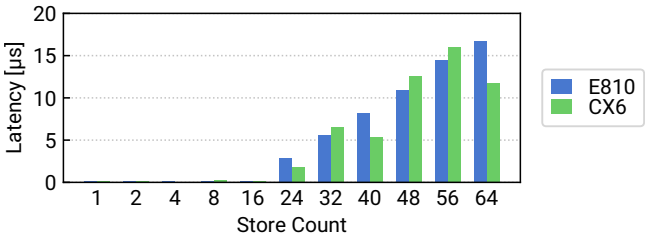

Figure 3. MMIO store latency versus iteration count for PCIe 4.0 ×16, from Ice Lake host to CX6 and E810 NICs.

WB DRAM, where throughput is consistent regardless of barrier frequency. Our results suggest that the MMIO WC data path cannot achieve high throughput without extensive batching. Near-maximum single-threaded throughput requires writing at least 4KB per barrier; with 64B packets, this means a batching factor of 64. This batched throughput is still only 76% of singleton 64B WB performance.

Implication: The WC and UC data paths are throughput-limited relative to standard write-back memory. Our measurements represent a NIC design that uses MMIO for bulk data/metadata transfer, unlike the signaling-only MMIO operations of current NICs. For streaming writes using the WC datapath, the barriers required to ensure ordering and flush to the device limit throughput.

MMIO write-combining latency. WC memory also introduces the limitation of a fixed number of store buffers. When all WC buffers are occupied, issuing a store within a 64B region not already buffered results in stalling until a buffer is flushed. Figure 3 shows the cumulative latency of N 32-bit MMIO stores to the E810 and CX6 NICs, up to $N = 64$. Latency remains uniform and low (< 20ns) until $N = 24$, where all WC buffers are utilized for the N stores. Beyond that, store latency is at least 15× greater and increases with N, as store buffers are flushed on the critical path.

Implication: When the MMIO data path is used for bursts of small stores, limited store buffer availability leads to expensive, high-latency accesses. This represents a NIC interface design that applies MMIO stores for metadata transfer, e.g., submitting descriptors to the device.

2.3 PCIe NIC Interface Design

The nature of PCIe and its performance characteristics impose constraints on the host-NIC interface. We identify three issues:

1. Since PCIe is not a coherent interconnect, local data structure updates must be communicated or signaled with explicit PCIe transactions.
2. PCIe operations incur high latency, so reducing the number of interconnect traversals is critical to achieving low-latency packet transmissions (§2.2).
3. Data and metadata writes over PCIe are expensive for the CPU in terms of both throughput and high-latency stalls (§2.2, §2.2).

The above issues define the performance tradeoffs imposed by the PCIe interface. An ideal design would achieve high packet throughput, low latency, and high CPU efficiency, but the PCIe prevents us from achieving all three goals simultaneously. Today's PCIe NICs prioritize CPU efficiency and throughput at the expense of latency by making the following design decisions:

Data structures are host-local, and updates are explicitly signaled. The host maintains packet buffers and descriptor rings in its local memory (as opposed to device MMIO) to reduce the CPU overheads for data structure access and updates. Requests to transmit a packet and newly received packets are signaled explicitly to the NIC. Other arrangements (e.g., host or NIC polling across the PCIe) waste PCIe bandwidth for each polling access and consequently are not used. For instance, in the transmit path, the host writes TX packets and TX descriptors into host memory and writes only a TX signal to the queue tail register maintained on the device side via MMIO. This results in a tradeoff: minimal data transfer over MMIO at the cost of extra interconnect roundtrips to read descriptors and packets from host memory.[1]

Descriptor transfer is batched. Given the host-side CPU stalls for MMIO writes to uncacheable NIC-side registers, the host may enqueue a large group of descriptors per MMIO register signal. This batching optimization again trades off latency for CPU efficiency.

The host handles all buffer management. The PCIe read-write interface, without cache coherence, limits the sharing of data structures between host and NIC. The synchronization mechanisms that support multi-core pool accesses are incompatible with PCIe reads and writes. Thus, the host performs all buffer management. This includes preallocating RX buffers to be used by the NIC and freeing completed TX buffers after NIC transmission. This results in additional bookkeeping communication over PCIe and

[1]Some NICs, e.g., the CX6, implement an alternative data path that writes descriptors over MMIO; however, this is typically enabled only for low-throughput, latency-critical workloads.

also limits the NIC from performing memory optimizations based on the properties (such as size) of received packets.

In the Evaluation, §5.3, we provide end-to-end measurements of PCIe NIC latency, throughput, and core utilization.

3 System Design for Coherent Interconnects

In this section, we describe the design of CC-NIC, a host-NIC interface optimized for cache-coherent interconnects such as UPI. First, in §3.1, we contrast a cache-coherent interconnect with PCIe, given our analysis of today's NIC interface designs. Then, we discuss the CC-NIC design in terms of metadata structures (§3.2), data accesses (§3.3), and packet buffer management (§3.4).

We present the design of CC-NIC as a series of empirically-backed design decisions, with the eventual design having the following desirable properties: (1) low-latency packet transmissions through the use of cache-to-cache transfers and hardware-supported signaling, (2) high throughput for data and descriptor communications using the efficient write-back datapath, and (3) reduced CPU management overheads realized by sharing buffer management and optimizing buffer placements for both TX and RX paths. Figure 4 compares the TX path for the CC-NIC interface with that of a PCIe NIC.

CC-NIC provides a data plane interface analogous to DPDK `mempool` and `ethdev` APIs, with burst semantics to enable batched TX/RX and buffer management operations. Figure 5 shows the core software interface.

3.1 Contrasting Coherent Interconnects and PCIe

Coherent interconnects, such as UPI and CXL, are tightly integrated with the CPU's memory data paths. Cross interconnect accesses may target DRAM and caches (see Figure 1b). The coherence protocol manages shared cache state, transferring lines into local caches when memory is accessed. The protocol ensures a writer gains exclusive control of a cache line before writing, invalidating any copies in remote caches. The protocol allows multiple caches to share reader access to a line, and lines may be forwarded between caches.

Overall, coherent interconnects provide a fundamentally different interface from PCIe. The coherence abstraction enables new forms of signaling and data structure sharing without the constraints of the PCIe read and write interfaces. Coherent interconnects integrate with the memory datapath and cache hierarchy, unlike PCIe MMIO, and also provide a symmetric interface, avoiding the tradeoffs between MMIO and DMA operations. However, cross-interconnect transfers depend on cache presence and coherence states, in terms of latency, memory controller requests, protocol metadata overhead, and roundtrips. There are limited means of manipulating these cache line states and caching behavior in general. As a result, implementing NIC data structures in the context of a coherent interconnect leads to both opportunities and challenges. We identify three factors that call for a different design:

1. **Coherence enables interface signaling and shared data structures.** PCIe NICs typically implement TX signaling with a separate mechanism, MMIO, from data and metadata DMA transfers. This results in an extra interconnect roundtrip to retrieve TX metadata via DMA after receiving the signal. Cache coherence performs signaling in hardware: when the remote side performs a write, the coherence protocol will invalidate any locally cached copy and fetch the new value upon subsequent access. Further, a coherent interconnect also enables the use of shared data structures between host and NIC, thus allowing for shared management of the buffer pool.

2. **CC-NIC has to choose between different data transfer mechanisms and homing options.** Coherent interconnects provide a diverse set of transfer mechanisms. For instance, CC-NIC can target write-back memory in addition to the cache-bypass data path and home data structures on either the host or the NIC, thus providing it with multiple transfer options to choose from. Cross-interconnect data transfers and cache state transitions also depend on the current cache residency of an object, possible prefetching, and the cache states caused by previous accesses. As a result, small objects, e.g., signals and descriptor metadata, are highly sensitive to layout.

3. **CC-NIC has to carefully manage caching.** The coherence protocol exchanges cache line ownership across the interconnect. Therefore, a remote access may result in additional communication when a later local access is performed. This is specifically problematic for producer-consumer workloads. For instance, the typical TX path transfers metadata and data from the host to the NIC, then the NIC performs the data transmission, and the data is not needed again by the NIC. In the coherent interconnect context, retaining the packet buffer or descriptor in the NIC-side cache is unhelpful; it adds overhead to subsequent host accesses that would have to perform remote cache invalidations. This suggests that minimizing overhead requires selectively invalidating cached data, which is unsupported on typical x86 platforms, or re-designing the data structures to avoid this access pattern.

3.2 Metadata Structures

In this section, we discuss the key questions that inform CC-NIC's handling of metadata, such as TX/RX descriptors.

How can we take advantage of cache coherence to reduce software overhead? Cache-coherent interconnects provide an underlying hardware mechanism to transfer and signal the availability of new data, via cache state transitions. This avoids the need for software-based signaling via head and tail index registers. To this end, CC-NIC applies

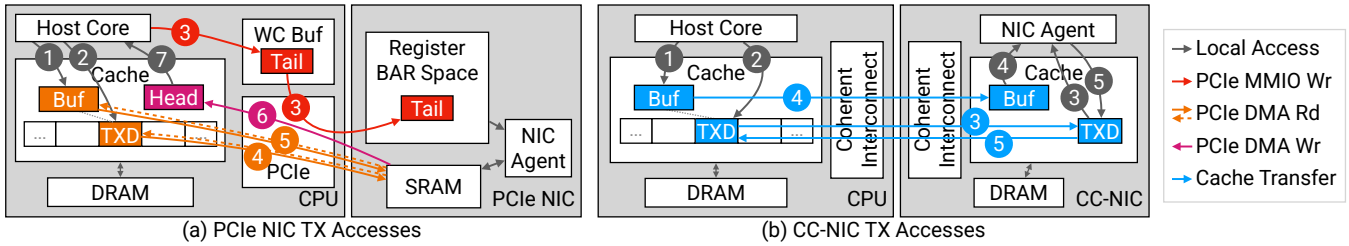

Figure 4. Comparison of TX path accesses for (a) the Intel E810, and (b) the CC-NIC interface. Numbers denote access order.

```
int ccnic_buf_alloc(struct ccnic_pool *pool,
    struct ccnic_buf **bufs, unsigned count);
void ccnic_buf_free(struct ccnic_pool *pool,
    struct ccnic_buf **bufs, unsigned count);

int ccnic_tx_burst(int txq_index,
    struct ccnic_buf **bufs, unsigned count);
int ccnic_rx_burst(int rxq_index,
    struct ccnic_buf **bufs, unsigned count);
```

Figure 5. The core CC-NIC data plane interface, maintaining the semantics of DPDK `mempool` and `ethdev` APIs.

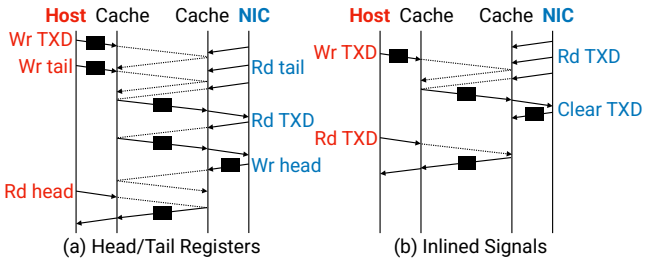

Figure 6. Signaling communication with (a) registers, and (b) signals inlined in the descriptor. Dotted lines represent coherence metadata.

an *inlined* signal in the descriptor, implemented as a flag indicating whether the descriptor is ready for consumption or free. Integrating the signal and descriptor eliminates a cache line transfer per signal and saves a cross-socket cache line access delay, as shown in Figure 6. For transmission, instead of polling a register containing the queue tail index, the NIC polls the next descriptor in the ring. The descriptor metadata includes a ready flag, which the host sets after other descriptor fields are written. Once the flag is set, the NIC receives the signal and the descriptor content in one access.

Event-driven implementation. A coherent NIC ASIC could further optimize signaling communication by directly handling coherence protocol messages. Instead of accessing descriptors through the cache polling abstraction, the device would directly take action in response to snoop messages received over the interconnect. Handling the coherence messages as signals avoids the scalability limitations of software-based polling in the presence of large queue counts.

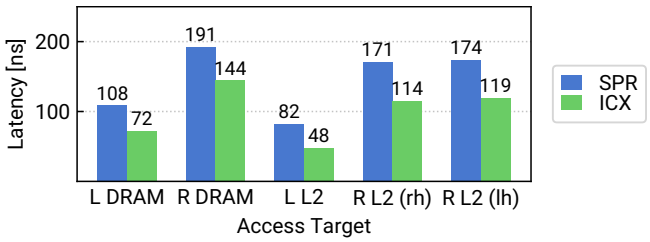

Figure 7. Local and cross-UPI access latency for Sapphire Rapids and Ice Lake hosts with various cache states.

What is the ideal data path for metadata transfers? Since a coherent interconnect may retrieve data from DRAM or multiple levels of the cache hierarchy, we measure the performance of each transfer case to understand performance implications for signaling communication. Figure 7 shows the median access latency of a 64B-aligned object in various cache states for both local and remote (cross-UPI) memory on Ice Lake (ICX) and Sapphire Rapids (SPR) server platforms. We find that accessing remote uncached DRAM incurs approximately twice the latency of local DRAM access. Accessing data cached in remote L2 is faster: 171ns on SPR and 114ns on ICX for memory homed on the remote socket (*rh* case), and slightly higher for memory homed on the local socket (*lh*). In these cases, the remote CPU has written to and retained a line in its L2 cache in the `M` (modified) state, and then the local reader accesses the address. When an `M`-state object exists in a remote L2 cache, it cannot exist in any other L2, so there is always a local L2 miss. With reader-homed memory, the reader's L2 miss causes a speculative memory read in addition to the remote request to the writer's cache. This speculative read is unneeded and causes lower performance when an object is reader-homed, increasing bus utilization with spurious traffic. Regardless of homing, remote L2 accesses are faster than a remote DRAM access, suggesting that cache-to-cache transfers achieve the best-case latency.

CC-NIC applies these observations in its design. CC-NIC places metadata structures in writer-homed memory: the TX

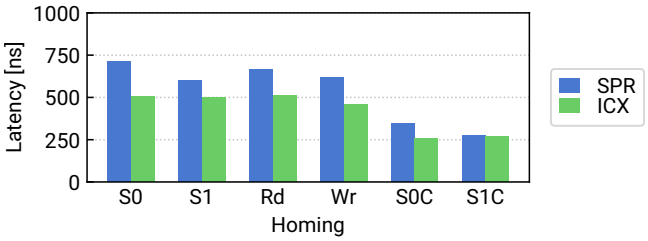

Figure 8. UPI pingpong experiment showing median latency with different memory layout choices.

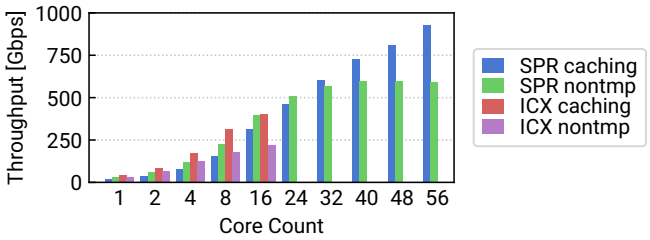

Figure 9. Stream transfer experiment comparing UPI throughput with caching and nontemporal accesses.

descriptor ring is host-homed, and the RX ring is NIC-homed. It enhances the possibility of cache-to-cache transfers by utilizing write-back memory with regular caching accesses instead of nontemporal stores that target memory. However, the working set size of the NIC interface affects performance, as do prefetch accesses (see §3.3).

How does memory layout affect metadata? NIC metadata, such as descriptors and signals, exhibit a producer-consumer access pattern in which each descriptor is written by one side and read by the other. For instance, TXDs are written by the host and RXDs by the NIC. Performance depends on the access pattern of a cache line, as this determines the protocol communication necessary to ensure coherence.

We use a pingpong microbenchmark to analyze producer-consumer accesses. We run a single thread on both sockets, accessing two shared 64-bit registers. The first thread increments the first value, while the second thread polls. The polling thread increments the second register after reading the updated value. We report roundtrip time, from writing the first register to reading the same value in the second register. Figure 8 shows median latency with both registers allocated in separate cache lines; both homed on the same socket (*S0/S1* cases); both homed on the respective reader/writer sockets (*Rd/Wr*); and both co-located on one cache line (*S0C/S1C*). In the *S0/S1* and *Rd/Wr* cases, each register exists on a separate cache line written by one CPU and read by the other. These correspond to PCIe NIC signaling, where host-to-NIC registers exist in the MMIO address space and NIC-to-host registers in write-back memory.

With separate cache lines, we find that writer-homed memory yields the lowest latency, consistent with Figure 7. These scenarios all show $1.7-2.4\times$ higher latency than when the values are on one cache line, homed on either socket. Co-locating producer and consumer structures on a single cache line achieves the best overall latency. With separate cache lines, each read transfers a cache line over the interconnect, and each write incurs another roundtrip to invalidate the reader cache. The co-located case instead uses one cache line for two-way communication. The $1.7-2.4\times$ latency difference shows the benefit of applying memory layouts that enable this two-way communication. Measuring *offcore response* perf counters shows a reduction of remote-socket requests from 4 to 2 per pingpong. This indicates reduced interconnect utilization in addition to improved latency.

CC-NIC applies two-way communication for signaling and descriptor transfers. Unlike the PCIe head and tail register layout, the host and NIC communicate by writing and clearing each descriptor and its inlined signal. This results in an access pattern matching the minimum pingpong latency.

How do we optimize for both latency-sensitive and high-bandwidth regimes? With inlined signaling, the host and NIC directly poll descriptor ring memory rather than separate registers. This results in the cache line thrashing between sockets when writing and polling a series of descriptors smaller than the 64B cache line. This thrashing increases latency compared to a cache-aligned case where descriptors are padded to 64B. Cache-aligned descriptors result in significant wasted space (e.g., 48 out of 64B), impacting the maximum packet rate. Furthermore, both scenarios prevent batching multiple descriptors per signal, a technique existing NICs typically rely on to maximize packet rate. To address these tradeoffs, CC-NIC implements a balanced solution: it packs bursts of up to $4\times$ 16B descriptors into a cache line, with unused entries zeroed out, and uses one signal per cache line. If the consumer reaches a blank descriptor in the middle of a group, it skips to the next cache line to poll for the subsequent descriptor group. This eliminates wasted space in the high-throughput case while avoiding thrashing in the low-throughput, un-batched case. With one signal per descriptor group, CC-NIC applies batching to utilize each descriptor cache line fully in high-throughput scenarios.

3.3 Data Accesses

Next, we discuss the questions guiding the design of packet data transfer in CC-NIC.

How should we write packet data? We run a streaming write microbenchmark to compare caching and nontemporal store throughput. For each case, we run a writer thread on the local CPU and a reader thread on the remote CPU. The writer thread sequentially writes data to a shared memory region, signaling the reader via a register for each 1MB written. The reader accesses 1MB per signal, copying into a thread-local buffer. We run a copy of the workload on each pair of threads,

up to all 16 ICX and 56 SPR cores. Figure 9 shows the results using two access types. In the *caching* case, the writer applies cacheable stores, which follow the typical memory datapath and enter the cache. This results in cross-interconnect cache transfers from the writer to the reader. In the *nontemporal* case, the writer uses cache-bypassing nontemporal stores to target reader-socket DRAM. This case aligns with the PCIe MMIO datapath, where stores are submitted over the interconnect directly without entering the cache. Our results show that the cache-to-cache transfer path enables higher throughput on both platforms: 1.8× (ICX) and 1.6× (SPR).

Additionally, we measure the maximum achievable interconnect throughput on our platforms using the Intel `mlc` benchmark utility [17]. This measurement uses a read-only remote access workload, which shows higher throughput than other patterns. On SPR and ICX, we find a maximum data throughput of 1020Gbps and 443Gbps, respectively. This suggests that the reader-writer streaming workload reaches 91% of best-case read-only throughput by using cache-to-cache transfers.

Based on this result, we apply caching stores to write packet data, both at the host application (while generating TX packets) and at the CC-NIC (while writing inbound RX packets).

How can we minimize coherence protocol overhead for data transfers? Packet buffers also demonstrate the producer-consumer access pattern described in §3.2; NIC's TX buffer accesses are read-only, and RX buffer accesses are write-only. Like descriptors, performance depends on underlying coherence state transitions. After the NIC completes a packet transmission, the buffer is likely to remain in the NIC's local cache. Although the buffer contents are no longer needed by the NIC, when the buffer is next allocated on the host side, writing to the buffer memory requires cross-socket access to invalidate this unnecessarily cached data.

There is no ideal instruction available to purge the consumed buffer memory out of the consumer-side cache. While `CLFLUSHOPT` does trigger cache invalidation, it is an expensive instruction that must be called on a per-cache-line basis and may incur memory accesses after the invalidation. Other cache control instructions, such as `CLWB`, do not help, as they do not result in cache invalidation.

Instead, CC-NIC implements a buffer recycling allocator to reuse the most recently freed TX buffers as RX buffers and vice versa. In both cases, the goal is to allocate buffer memory still present in the writer's cache. CC-NIC's buffer recycling provides similar application-level semantics to the TX-RX buffer reuse implemented by some PCIe NIC drivers (such as the i40e kernel driver [48]). However, these existing mechanisms are software-only driver optimizations and thus do not affect interconnect communication. CC-NIC's buffer recycling takes place at both the NIC and the host, and addresses the unique producer-consumer overheads imposed by cache coherence. We implement buffer recycling using host- and NIC-local stacks, which cache free buffer addresses from the pool of packet buffer memory. This technique is suitable for applications with a single buffer pool for TX and RX traffic, the typical design pattern among DPDK NIC drivers. But, in cases where there are multiple references to the TX buffer payload, and the host retains the TX buffer after transmission (e.g., for potential retransmission), this optimization falls back to standard buffer allocate/release behavior. The CC-NIC buffer allocator is described fully in §3.4.

Where should data be homed? Our measurements of remote-socket accesses, in §3.2, demonstrate a latency benefit to homing memory on the writer socket. As a result, CC-NIC places the TX descriptor ring on the host socket and the RX ring in NIC memory. However, we allocate packet buffer memory entirely homed on the host. Since applications may arbitrarily access packet buffer data, placing it in remote memory could have unexpected application-level consequences. Applications may, for instance, submit RX buffers to a TX queue, so writer-homing does not universally apply to packet buffer memory. Instead, CC-NIC's recycling buffer allocation policy and locality-oriented optimizations, described next, aim to minimize the cost of host and NIC buffer accesses.

How can we maximize cache locality for packets? As measured in §3.2, maximum remote access performance is achieved when an object is present in remote cache. As such, it is important to maximize the caching of shared data on both the host and NIC sides. Using small packet buffer sizes for small packets reduces the overall memory footprint of the NIC interface. When supported by the application, an MTU-sized buffer, for instance, 4KB, is subdivided into 32× 128B packet buffers. When the host allocates a buffer to write a TX packet, it selects either a large or small buffer based on the packet data size, if known in advance. The RX side follows the same logic (see §3.4). This increases the cache efficiency of small packet transfers. Unlike PCIe NIC drivers, which may inline packet data into the descriptor ring, this approach does not require copying the packet data payload into another location.

Relative to PCIe, cache coherence brings the additional challenge of potential remote prefetching. When buffers are allocated sequentially from a contiguous region of memory, the sequential access pattern on the consumer side may result in hardware prefetching of the buffer memory just beyond the current packet buffer. These remote prefetches contend with local writes of that same packet buffer before it is submitted to the descriptor ring. This behavior occurs when the NIC handles one posted TX buffer and prefetches subsequent buffer memory while the host is writing to the next buffer allocated sequentially in memory. We avoid this contention by filling the memory pool with buffers such that

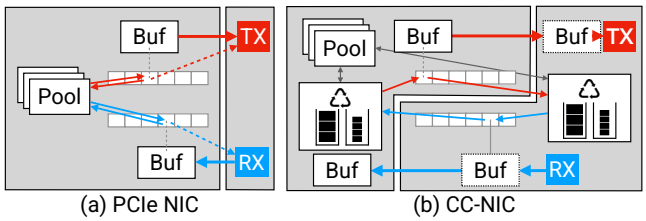

Figure 10. Buffer management approaches. Thin lines denote buffer allocation/release; thick lines denote buffer data transfer.

repeated buffer allocations do not yield sequential memory addresses. This policy avoids unwanted cache state transitions, increasing the efficiency of producer-side packet buffer writes.

3.4 Buffer Management

Several of the above design features, including the recycling buffer allocator (§3.3), subdividing buffers for small packets (§3.3), and cache-aligned descriptor groups (§3.2), are critical to CC-NIC's efficient utilization of the interconnect. Each of these optimizations requires allocating buffers and writing RX descriptors based on the properties of the workload. In a typical PCIe NIC interface, RX buffers are allocated and posted to RX descriptors by the host prior to the actual reception of packets by the NIC. This makes it impossible to apply knowledge of the RX packet burst at the time of assigning buffers to RX descriptors. CC-NIC overcomes this by taking advantage of cache coherence to share the responsibility of buffer management with the host. Cache coherence allows the host and NIC to access the buffer pool data structure concurrently without the restrictions associated with simultaneous PCIe DMA and CPU accesses (e.g., lack of atomic operations). A shared buffer pool structure allows the NIC to release TX buffers to the pool after transmission. Likewise, the NIC can allocate RX buffers on demand and write their addresses into the RX descriptor ring. This results in a symmetric design that avoids extra bookkeeping passes over the queues to free completed TX packet buffers and post blank RX packet buffers. Finally, shared buffer management enables CC-NIC's buffer allocation and descriptor layout optimizations. Figure 10 compares CC-NIC's buffer management design to that of PCIe NICs.

4 CC-NIC Implementation

To demonstrate the benefits of the CC-NIC design, we implemented CC-NIC on a dual-socket server where one socket acts as a software NIC. In this implementation, all host-NIC communication occurs over the UPI interface. In addition to being a coherent interconnect that we can experiment with now, UPI also provides bandwidth higher than contemporary PCIe generations (see Table 1). Sapphire Rapids CPUs provide a terabit-throughput UPI interface, allowing us to model terabit NIC communication.

Protocol	GT/s	1 Link GB/s	Max Total GB/s
PCIe 4.0	16	2.0	31.5 (×16)
PCIe 5.0, CXL 1.0-2.0	32	3.9	63.0 (×16)
PCIe 6.0, CXL 3.0	64	7.6	121 (×16)
Ice Lake UPI	11.2	22.4	67.2 (×3)
Sapphire Rapids UPI	16	48	192 (×4)

Table 1. Comparison of PCIe, CXL, and UPI bandwidth.

We designate one CPU and its local-socket memory as the host and the second socket as the NIC. NIC-socket memory represents coherent device memory, and the NIC cores represent the processing units of the NIC. The software flexibility enables experimenting with data structure designs and communication patterns since we are not restricted by the hardware interfaces of existing NICs. We believe the software-initiated nature of NIC accesses does not change the host-NIC interactions required to transfer packets; hardware-initiated transfers would map to equivalent coherence protocol operations and show comparable interconnect performance.

To evaluate the PCIe and CC-NIC interfaces in isolation, we focus on loopback performance. Prior work finds that PCIe can contribute the majority of network TX/RX latency observed by the end-host [32]. While these experiments exclude Ethernet transmission, they demonstrate the most significant component of overall latency.

To understand end-to-end throughput and core utilization, we implement a CC-NIC Overlay interface atop a PCIe NIC. With a PCIe NIC installed on the second socket, we utilize *overlay* threads on the NIC socket to bridge between the CC-NIC UPI interface and a PCIe NIC. These threads poll both UPI TX and PCIe RX queues, copying packet data and writing descriptors between each respective pair of queues. This allows applications running on the first socket to perform network TX/RX via CC-NIC. While overlay packet forwarding adds latency and burns cores on the second CPU, it allows us to measure application throughput and core utilization.

5 Evaluation

Our evaluation is guided by the following questions:

1. How does CC-NIC perform relative to PCIe NICs? §5.2
2. What performance does CC-NIC achieve on a terabit UPI interconnect? §5.3
3. What are the gains from optimizing metadata structures, data accesses, and buffer management? §5.4
4. How does batching affect performance? §5.5
5. Does CC-NIC's design increase coherence communication efficiency? §5.6
6. Can CC-NIC save CPU cores with a key-value store application and TCP RPC stack? §5.7
7. How sensitive is the design to hardware prefetching, interconnect bandwidth, and latency? §5.8, §5.9

5.1 Evaluation Setup

We use two server platforms with the following specifications. The ICX server is a dual-CPU Intel Ice Lake Xeon Gold 6346, running at 3.1GHz, with PCIe 4.0 support and 3×11.2GT/s UPI links. Each ICX CPU has 16 cores (32 hyperthreads), 1.25MB per-core L2, 36MB LLC, and 12×16GB DDR4 at 3200MHz. This server contains two PCIe NICs, an Intel E810-2CQDA2 (E810) and Nvidia ConnectX-6 Dx MT42822 (CX6), both 2×100GbE devices. Our SPR server contains dual Intel Sapphire Rapids CPUs, running at 2.0GHz, with PCIe 5.0 support and 16GT/s UPI. Each SPR CPU has 56 cores (112 hyperthreads), a 2MB per-core L2 cache, 105MB LLC, and 8 × 64GB DDR5 at 4800MHz.

We apply these two server platforms to evaluate the following comparison points:

- **CC-NIC on ICX (UPI).** We deploy CC-NIC on the ICX server to compare UPI- and PCIe-based NIC communication on the same CPU platform.
- **CX6, E810 on ICX (PCIe).** For our PCIe NIC measurements on the ICX server, we follow the vendor-published system and driver configuration steps [18, 34] and verify that packet-forwarding performance matches these official DPDK performance reports. We enable standard platform-level optimizations such as DDIO [15].
- **CC-NIC on SPR (UPI).** To measure CC-NIC's performance across a terabit coherent interconnect, we also deploy CC-NIC on the SPR platform with the above specifications.
- **Unoptimized UPI on SPR, ICX.** To demonstrate coherent NIC performance without CC-NIC's design features, we implement the Intel E810 NIC interface over the UPI interconnect. We use writeback memory and caching accesses but maintain the E810 data structure layout and register-based signaling. This baseline scenario represents a case where future coherent NICs apply the same software interface as today's PCIe NICs.

Loopback setup. We implement a traffic generator using DPDK [4] to evaluate both CC-NIC and PCIe NICs. Each NIC serves as a loopback between pairs of TX and RX queues. Each application thread configures private queues, allocates TX buffers, and writes full, timestamped payloads for each TX packet burst; it polls RX queues and accesses each RX payload before freeing the buffer. This is more work per packet than minimal RX-TX forwarding due to payload accesses and separate TX and RX flows. We vary TX rates from one inflight packet to the maximum sustainable rate and measure median roundtrip latency and RX data throughput.

Overlay setup. We use the CC-NIC Overlay (§4) to evaluate per-thread application throughput with the CC-NIC interface. An application uses the CC-NIC UPI interface for TX/RX; remote-socket overlay threads transfer packets between corresponding PCIe NIC queues. As a baseline, the application interfaces directly with the same socket-local PCIe NIC. While the PCIe NIC limits throughput in both cases, this setup allows us to compare CPU utilization.

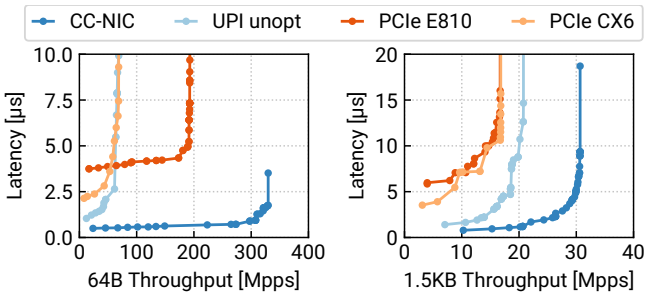

Figure 11. Throughput-latency curves, comparing CC-NIC, unoptimized UPI, and PCIe loopback performance on the ICX server, for 64B and 1.5KB packet sizes

5.2 Performance Comparison Overview

Figure 11 shows a comparison of four host-NIC interfaces on the ICX server: the CX6 and E810 PCIe NICs, a naive implementation of the E810 interface over UPI, and CC-NIC. These results show that CC-NIC provides a significant opportunity for latency improvement and higher throughput over PCIe. CC-NIC's minimum latency is 77% and 86% lower than that of the CX6 and E810. As detailed in §5.3, CC-NIC's latency reduction over the CX6 is more significant when considering latency under load for both large and small packets. CC-NIC also achieves a 1.7× and 4.3× higher peak packet rate than the E810 and CX6. With 1.5KB packets, we observe 1.8× higher data throughput over both PCIe NICs.

The unoptimized UPI (*unopt*) scenario shows that a coherence-optimized design is critical. This case applies the E810 interface across UPI and achieves lower 64B packet rates than the native PCIe E810 despite a higher-bandwidth interconnect. Relative to CC-NIC, this version shows 79% lower throughput and 2.1× higher minimum latency.

5.3 Detailed Performance Results

This section compares CC-NIC, CX6, and E810 results on the ICX platform. Figure 12 shows throughput-latency profiles for CC-NIC and CX6, with 64B and 1.5KB packet sizes. We also measure CC-NIC on the SPR platform's terabit UPI interconnect, shown in Figure 13.

Latency. CC-NIC demonstrates low minimum latency and loaded latency relative to both PCIe NICs. We measure a minimum loopback latency of 490ns (ICX) and 650ns (SPR) versus a best-case PCIe latency of 2116ns (CX6) and 3809ns (E810). The latency difference between CC-NIC and the CX6 is more significant when the load is increased. At 80% load, CC-NIC's 64B latency is 88% lower than the CX6 (85% lower

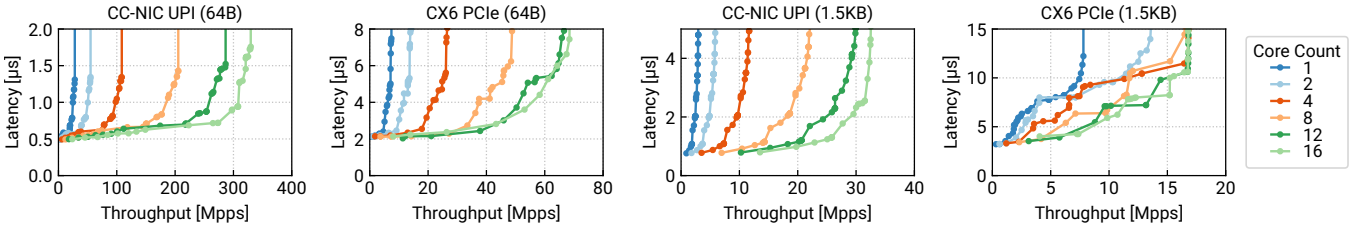

Figure 12. Loopback throughput-latency curves for CC-NIC and CX6 on the ICX server, with 64B and 1.5KB packet sizes.

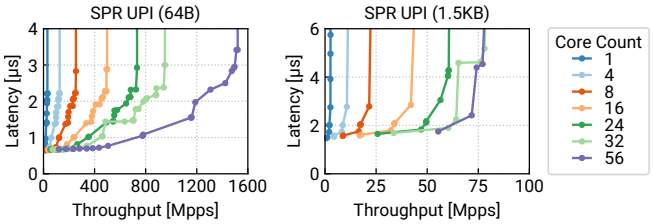

Figure 13. Loopback throughput-latency curves for CC-NIC on Sapphire Rapids UPI, with 64B and 1.5KB packet sizes.

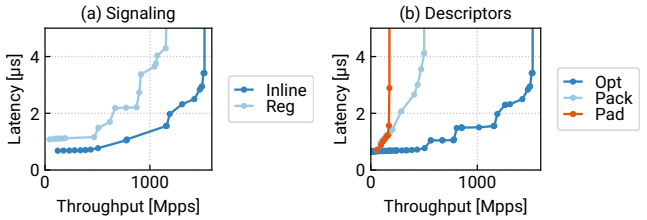

Figure 14. Throughput and latency varying (a) register and inline signaling options, and (b) descriptor layouts.

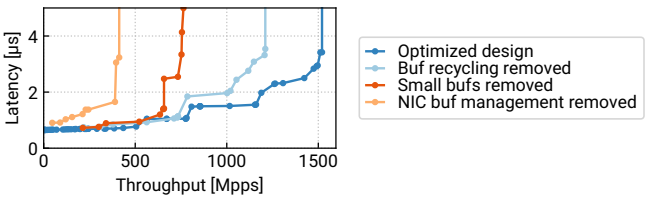

Figure 15. Performance impact of CC-NIC buffer management features.

than the E810). With large 1.5KB packets, minimum latency is 76% lower than the CX6 (87% lower than the E810). As with small packets, at 80% load, CC-NIC achieves a greater improvement over PCIe: 88% for both CX6 and E810.

Throughput. On the ICX server, CC-NIC demonstrates a maximum 64B packet rate of 330Mpps (169Gbps). This is a substantially higher packet rate than PCIe NICs on the same platform: 192Mpps (E810) and 76Mpps (CX6). For 1.5KB packets, CC-NIC reaches 403Gbps out of a maximum 443Gbps measured data throughput on the interconnect; both PCIe NICs reach their rated 200Gbps line rate on the 252Gbps PCIe link. While the ICX server core count limits CC-NIC's 64B packet rate, the SPR results demonstrate full interconnect utilization. The SPR CC-NIC loopback reaches a maximum packet rate of 1520Mpps (778Gbps) with 64B packets. Including the descriptor metadata transferred with each packet, this corresponds to 96% of the measured maximum UPI data throughput. For 1.5KB packets, we measure 986Gbps data throughput or 97% of UPI throughput.

Core count. With 64B packets, 48 of 56 (SPR) and 14 of 16 (ICX) host cores are required to reach 90% of the maximum rate. Large 4KB packets decrease the core counts to 18 (SPR) and 8 (ICX). Each core accesses full TX/RX payloads, placing a higher burden on the host cores than workloads such as forwarding with header-only accesses. We measure core utilization with application workloads in §5.7.

5.4 Design Feature Analysis

Next, we evaluate the impact of CC-NIC's design features. Recent work analyzing data center workloads in the context of transport protocols [31] and data stores [46] emphasize the prevalence of small packets. Thus, we examine small packet workloads and evaluate packet-handling efficiency.

Signal Inlining. Figure 14a shows the impact of inlining signals into the descriptor ring versus maintaining external queue tail doorbell registers. For 64B packets, inlined signals reduce minimum latency by 37% and increase maximum packet rate by 1.3×.

Descriptor Layout. Using the same workload, we evaluate different descriptor layout choices: the optimized layout (§3.2), 16B descriptors equivalent to the E810 NIC (*pack* case), and the same format with each descriptor padded to a cache line (*pad*). Figure 14b shows the results. Due to the 64B granularity of UPI cache transfers and the direct descriptor polling required for inlined signals, memory layout substantially affects performance. Cache-aligning (padding) each descriptor achieves low latency by avoiding thrashing. Packing singleton 16B descriptors into a cache line improves throughput by 2.9× but causes thrashing as the host and NIC each access multiple signals and metadata fields per line. Finally, the optimized descriptor layout incorporates a single signal and a group of descriptors per cache line. This layout achieves a 3.0× throughput improvement while matching the best-case minimum latency of the padded case.

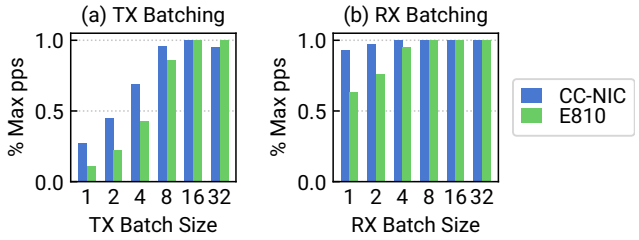

Figure 16. 64B packet rate relative to maximum, varying TX (a) and RX (b) batch sizes, for CC-NIC and E810 (PCIe).

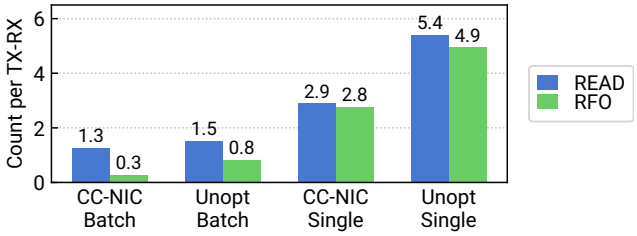

Figure 17. NIC remote accesses per TX-RX loopback, batched and singular descriptor cases.

Buffer Management Optimizations. In Figure 15, we evaluate the performance impact of buffer management optimizations. We begin with the optimized design, removing features sequentially. All measurements use 56 SPR cores and 64B packets. First, we disable same-socket buffer reuse and nonsequential allocation (§3.3), so all allocations and frees access the buffer pool, not the buffer reuse cache for each core. This is representative of a workload where TX buffers are retained by the application after transmission, not returned to the buffer pool. We observe a 20% throughput reduction in this case. Second, we disable the small buffer optimization (§3.3), so each 64B packet is written into a separate 4KB buffer. This results in a larger shared memory footprint and a further 37% throughput decrease. Finally, we disable shared access to the buffer pool (§3.4), instead posting and freeing buffers exclusively on the host side. This change prevents the NIC from adaptively filling RX descriptors based on the available burst count and increases host bookkeeping, decreases maximum throughput by 46%, and increases latency by 1.3×. This final case is comparable to PCIe NIC buffer management.

5.5 Batching Effects

Batching is critical to achieving high NIC packet rates. For PCIe NICs, TX batching enables submitting multiple packets with one MMIO doorbell; larger batch sizes reduce the rate of MMIO operations. For CC-NIC, TX batching allows multiple descriptors to be transferred within a single cache line. Host-side RX batching primarily affects access patterns on the descriptor ring and buffer pool, determining whether buffers are handled individually or in bulk. Figure 16 shows 64B packet rate at a given host TX/RX batch size, relative to the highest achievable packet rate. We define batch size as the maximum number of buffers transmitted per polling loop iteration, i.e., the TX/RX *burst* count used in DPDK APIs. We repeat the experiment with CC-NIC and the E810 PCIe NIC, both on the ICX server. We vary the TX batch size while using a fixed RX batch size of 32, and vice versa. For TX, CC-NIC achieves higher packet rates with lower batching factors; the unbatched case shows 27% of peak throughput for CC-NIC versus 12% for the E810. Because CC-NIC uses lightweight per-cache-line signals instead of MMIO doorbells, the need for large batching factors is reduced; CC-NIC achieves peak packet rates when descriptor cache lines are filled. For poll-mode RX, host-side batching is less crucial to performance since the host does not perform PCIe MMIO accesses and releases RX descriptors lazily. Both NICs show significantly less sensitivity to the RX batch factor: CC-NIC maintains at least 93% of peak throughput across batch sizes, and the E810 achieves at least 63%.

5.6 Interconnect Communication

To measure the coherence communication required to facilitate host-NIC interactions, we measure *offcore response* PMU counters of the NIC CPU. As an additional comparison point, we deploy NIC and host threads on a single CPU. This setting eliminates UPI communication altogether, revealing the interconnect contribution to latency and bandwidth overheads.

Remote Access Counters. We measure *offcore response* PMU counters of the NIC and host CPU to quantify interconnect communication. Figure 17 compares the remote accesses performed by CC-NIC and the unoptimized UPI baseline per 64B TX-RX loopback operation. The figure shows NIC CPU accesses; due to the symmetric TX-RX design of CC-NIC, we observe symmetric host-side access counts. Each remote access consists of a *read* or *read for ownership* (RFO) interconnect operation. We evaluate singleton and fully-batched (4 descriptors per cache line) cases. The batched case shows a throughput-oriented workload, processing descriptors in bursts of 8 and filling ring cache lines without wasted space. The singleton case represents a low-throughput, low-latency workload, where the host transmits one packet at a time and immediately polls for completion status. This case maximizes contention on shared cache lines, with the host and NIC accessing descriptors and signals simultaneously.

With the batched workload, CC-NIC performs one read access per packet, plus one read and one RFO per descriptor group (0.25 per packet). This suggests that CC-NIC effectively amortizes metadata cache transfers. The unoptimized UPI baseline, which uses register-based signaling, incurs one additional read and two additional RFO accesses per descriptor group. For the singular scenario, each individual packet requires full cache-line transfers for the descriptor, packet

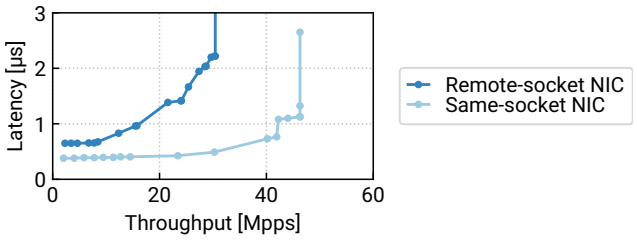

Figure 18. Single-thread 64B loopback performance, comparing CC-NIC thread running on local CPU versus cross-UPI remote CPU.

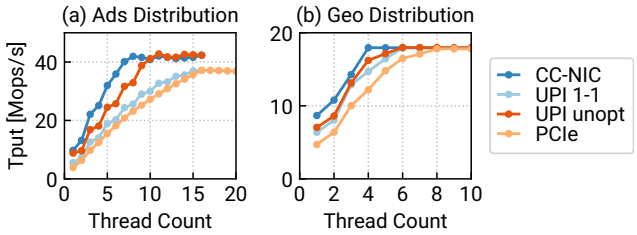

Figure 19. Throughput versus thread count for key-value store workloads, comparing CC-NIC Overlay and PCIe NIC interfaces.

	PCIe Mops/s	CC-NIC Mops/s	Thread Count
KV store (ads)	37.0	42.3	16 → 8
KV store (geo)	17.8	17.9	8 → 4
TCP echo RPC	58.3	64.6	5 → 3

Table 2. Peak throughput and core count for KV Store and TCP Echo RPC applications, comparing CX6 and CC-NIC Overlay interfaces.

memory, and, in the unoptimized version, registers. When we compare batched and singular cases, our results show the importance of efficient descriptor cache-line layouts. Packing multiple descriptors into one cache line (§3.2) significantly reduces coherence communication, for both CC-NIC and the unoptimized case. Comparing optimized and unoptimized interface designs with the singleton workload, CC-NIC is able to recycle locally-cached buffer memory (§3.3) and avoid separate cache transfers for register signaling (§3.2). This reduces interconnect communication, even in the presence of contented host-NIC accesses.

Same-Socket Comparison. We deploy CC-NIC and host threads on a single NUMA node to understand the contribution of the UPI interconnect on loopback latency and per-thread throughput. This setting exhibits host-NIC interactions between local CPU cores, eliminating transfers across the UPI physical link. Figure 18 shows one-thread 64B loopback performance between host and CC-NIC threads on the same SPR CPU, compared to the cross-UPI deployment used for all other results. Comparing both minimum and loaded latencies shows that the interconnect accounts for approximately 40-50% of TX-RX loopback latency. The increased latency of cross-UPI accesses increases stalling for the host application, which impacts maximum per-thread throughput; the same-socket experiment shows 1.5× greater per-thread throughput.

5.7 Application-Level Performance

Table 2 summarizes the thread count reduction enabled by CC-NIC for the key-value store echo RPC applications discussed below. We compare the CC-NIC Overlay interface (forwarding to the CX6 PCIe NIC) to the direct interface with the CX6, both on the ICX platform.

Key-Value Store Throughput. We implement a key-value store following the design of CliqueMap [46], with DPDK's `rte_hash` table as the index. Server threads poll NIC RX queues to handle get and set RPCs. Gets are zero-copy, applying multi-segment TX (DPDK `extbuf`) to submit the header and object payload to the NIC. This requires two buffer addresses per TX descriptor, increasing host-NIC metadata but avoiding object `memcpy`. We deploy the key-value store on one ICX server with a CX6 NIC, plus two remote clients, enough to saturate the server. We evaluate two production object distributions from Google, *Ads* and *Geo* [46], limiting sizes to a 9600B MTU (truncating the largest 0.01% of Ads). Ads consists of smaller objects; 61% are less than 100B, compared to 13% in Geo. For both, we evaluate 95% gets, 5% sets on 1M objects, following a Zipf access pattern with a coefficient of 0.75.

Figure 19 shows key-value request throughput with CC-NIC and CX6 interfaces across the range of application thread counts (hyperthreading enabled). Since all scenarios perform TX/RX via the CX6 NIC, peak throughput is determined by its packet rate. However, the CC-NIC Overlay interface achieves peak throughput with fewer application threads. For Ads, 8 threads saturate throughput with the CC-NIC Overlay interface, compared to 16 with the CX6. The high rate of small objects stresses the host-NIC interface, especially with multi-segment TX. The Geo workload demonstrates a reduction of 8 to 4 threads, showing core savings with a distribution skewed towards larger objects. The *UPI 1-1* series uses one overlay thread per application thread. Relative to the direct CX6 interface, the same number of threads access PCIe NIC queues, but this work is offloaded from application threads. This increases per-thread throughput up to 31%, but the overlay thread count limits performance. Comparing CC-NIC to the unoptimized UPI (*unopt*) baseline shows the benefits of coherence-optimized buffer management: CC-NIC shows a savings of 3 (Ads) and 2 (Geo) threads at peak throughput.

TCP RPC Throughput. We evaluate a TCP RPC server built using TAS [22], a high-performance userspace TCP service. We run the RPC server implemented by the TAS authors, a basic TCP application dynamically linked to TAS, overriding the kernel sockets interface. A set of userspace

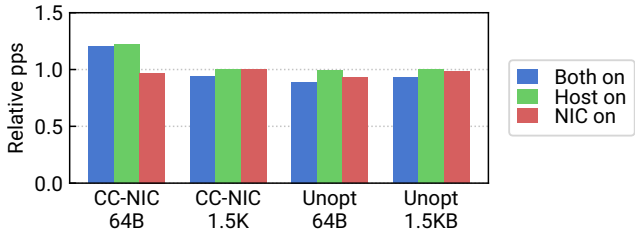

Figure 20. Impact of SPR hardware prefetching on 64B packet throughput, relative to the prefetching-disabled case.

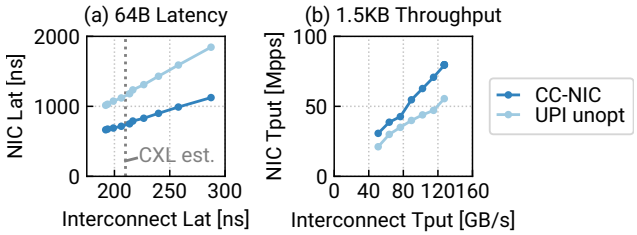

Figure 21. Performance with reduced UPI throughput and latency.

TAS *fast-path* threads to handle the TCP data plane via DPDK, achieving state-of-the-art TCP performance. We replace TAS's fast-path PCIe TX/RX with the CC-NIC Overlay to evaluate the benefits of a coherent NIC interface. We do not offload any aspects of TAS but instead deploy a drop-in replacement NIC interface.

We evaluate a workload of 64B echo RPCs, deploying one application thread and measuring the number of TAS fast-path threads required to achieve 95% peak throughput. In the PCIe baseline case, we run all application and TAS threads on the CX6's local socket CPU. For the CC-NIC Overlay case, we deploy overlay threads on the CX6 socket and all TAS and application threads on the remote CPU. On a second machine, we run the client application with all threads and a total of 96 flows, enough to saturate the server. Table 2 compares RPC throughput with the CC-NIC Overlay and direct CX6 interfaces. Applying the CC-NIC Overlay results in NIC saturation with 3 TAS threads versus 5 with the PCIe interface. The CX6 case shows slightly lower peak throughput due to internal TAS overheads, which increase with the fast-path thread count. Both scenarios are limited to the CX6 NIC packet rate.

5.8 Sensitivity to Hardware Prefetching

In Figure 20, we compare the impact of hardware prefetching on packet rates for CC-NIC and the unoptimized UPI baseline, on the SPR platform. We enable prefetching on the host, NIC, and both CPUs, measuring packet rate relative to the case of prefetching disabled. We find that the optimized CC-NIC interface is able to benefit from host-side prefetching for small-packet workloads: prefetching increases packet rate 1.2× for 64B packets. This gain comes from the CPU's *DCU IP Prefetcher* and affects packet buffer accesses in particular. Both designs achieve maximum throughput with prefetching enabled on the host CPU only (we use this setting for all other experiments). For the unoptimized interface design, without CC-NIC's locality-oriented buffer pool optimizations (§3.3), prefetching strictly decreases performance by up to 7%. These differences suggest that the NIC interface design dictates whether prefetching improves performance or increases interconnect overheads.

5.9 Sensitivity to Interconnect Performance

We analyze CC-NIC's sensitivity to interconnect bandwidth and latency by varying the NIC socket uncore frequency. This allows us to study CC-NIC under reduced interconnect performance. However, this approach results in pessimistic measurements, as downclocking the uncore impacts purely local access performance in addition to remote UPI accesses. Across the range of supported uncore frequencies (maximum is the default), we measure host-to-NIC-socket DRAM access latency and read throughput and loopback performance with CC-NIC and the unoptimized UPI interface. Figure 21 shows 64B packet loopback latency relative to access latency, and 1.5KB packet throughput relative to interconnect data throughput, measured on the SPR server.

According to the CXL Consortium, the expected access latency for CXL-attached DRAM is 170-250ns [40]. This range is corroborated by research on CXL.mem prototypes, which finds that CXL.mem load latency is approximately 1.5× higher than cross-UPI remote DRAM [47]. In Figure 21a, we observe that CC-NIC's latency increase closely tracks the increase in host-to-NIC interconnect access latency. With a 1.11× increase in interconnect latency to 211ns (middle of the CXL range), CC-NIC loopback latency increases by 1.13×. CC-NIC maintains its relative improvement over the unoptimized UPI interface, which incurs a 1.16× latency increase. Figure 21b shows that performance is also stable over a range of interconnect throughputs. 1.5KB loopback throughput scales well and maintains a consistent improvement over the unoptimized case. When interconnect throughput is set to a minimum of 40%, CC-NIC throughput is 39%.

6 Discussion

Hardware DMA. Hardware bulk transfers, on both host and NIC sides, can potentially increase efficiency over CPU accesses. While our application-level results (§5.7) show that CC-NIC can reduce core utilization without DMA, efficient hardware transfers could benefit large-packet workloads. On-chip DMA engines, such as Intel's Data Streaming Accelerator [16], are one possible mechanism for CPU-initiated bulk transfers. For device-initiated DMA, a CXL-attached NIC could leverage both CXL.cache for metadata and small

packet transfers, plus CXL.io DMA for bulk packet operations.

Security and Isolation. We expect coherent host-device interconnect standards to provide mechanisms for protecting and isolating host resources. We expect that current techniques to control PCIe DMA access to the host address space, e.g., IOMMU translation, apply to coherent device accesses. Likewise, the BAR space abstraction of current PCIe devices offers a means of isolation by limiting host-device coherence to a portion of the address space.

Network Function Workloads. While we studied workloads involving full packet access, cache-coherent NICs could bring additional benefits for middlebox workloads like packet switching. Packet-switching through a PCIe NIC incurs unneeded interconnect and memory bandwidth utilization. Even if the application only operates on packet headers, the full packet payload is still transferred to and from host memory. In the case of DDIO, this may result in cache pollution. Instead, a coherent NIC may retain payloads in the NIC cache while the host operates on the header, avoiding interconnect transfers for packet data the host does not access.

7 Related Work

TinyNF [38], NIQ [7], PacketMill [6], and others [4, 8, 11, 19, 20, 22, 26, 42, 45] propose optimizations to the host software interface of PCIe NICs through the elimination of driver and stack overheads. Since our work maintains the packet queue model, these optimizations carry to the software stack running atop a coherent NIC interface.

NanoPU [13], Direct Cache Access [12], and Semi-Coherent DMA [28] propose new CPU-NIC data paths. Like our work, these systems demonstrate that tighter integration between the NIC and CPU caches enables higher performance. Rather than propose new data paths, our work leverages the faster paths of an existing cache-coherent interconnect.

Scale-out NUMA [33], and Dagger [23] apply cache coherence in conjunction with new communication models beyond NIC packet RX/TX. Scale-out NUMA enables remote coherent access to host memory by integrating an RDMA-like interface with the cache hierarchy. Dagger applies a UPI-attached FPGA as a target for offloaded RPCs with coherent host access. Our work focuses specifically on optimizing the producer-consumer data transfers associated with packet RX/TX. However, these systems and others [5, 21, 29, 30, 53] apply similar producer-consumer interactions, e.g., RDMA work queues. In the context of a coherent interconnect, the design we propose applies to these data structures.

Prior work on microkernel and shared-memory message passing [1, 41], as well as IO virtualization [51, 52], describes optimizations for producer-consumer accesses in the shared memory setting. With coherent host-device interconnects, these considerations (e.g., optimizing for cache-to-cache transfers) become newly important to host-device interactions. The specific context of NIC interactions presents new opportunities for optimization and requires a specialized design. For instance, while existing message queue systems optimize for cache alignment, applying it to NIC RX queues requires broader changes to the buffer management system. CC-NIC applies a new combination of design decisions to optimize for the unique properties of both host-device coherence and NIC TX/RX descriptor communication.

Pond [24] and DirectCXL [10] explore CXL as a means of providing disaggregated memory resources. Their analysis of CXL datapath performance pertains to CXL-attached NIC interactions.

8 Conclusion

This paper makes a case for redesigning the host-NIC software interface in the context of emerging cache-coherent interconnects. These interconnects are capable of high performance, but the interface design of current PCIe NICs performs poorly in the coherent setting. We present CC-NIC, a NIC interface designed to benefit from cache coherence. Our results, modeling CC-NIC over the coherent UPI interconnect, demonstrate high throughput, low latency, and CPU-efficient host-NIC communication.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, Michio Honda. This work was supported in part by NSF grant CNS-2006349 and ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

References

[1] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. User-Level Interprocess Communication for Shared Memory Multiprocessors. *ACM Trans. Comput. Syst.*, 9(2):175–198, may 1991.

[2] CCIX Consortium Inc. CCIX Base Specification 1.0. https://www.ccixconsortium.com/library/specification/.

[3] Compute Express Link Consortium Inc. CXL 3.0 Specification. https://www.computeexpresslink.org/download-the-specification.

[4] DPDK Project. Data Plane Development Kit. https://www.dpdk.org/.

[5] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, Seattle, WA, Apr. 2014. USENIX Association.

[6] A. Farshin, T. Barbette, A. Roozbeh, G. Q. Maguire Jr., and D. Kostić. PacketMill: Toward per-Core 100-Gbps Networking. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 1–17, New York, NY, USA, 2021. Association for Computing Machinery.

[7] M. Flajslik and M. Rosenblum. Network Interface Design for Low Latency Request-Response Protocols. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 333–346, San Jose, CA, June 2013. USENIX Association.

[8] S. Gallenmüller, P. Emmerich, F. Wohlfart, D. Raumer, and G. Carle. Comparison of Frameworks for High-Performance Packet IO. In *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for*

Networking and Communications Systems, ANCS '15, page 29–38, USA, 2015. IEEE Computer Society.

[9] Gen-Z Consortium. Gen-Z Specifications. https://genzconsortium.org/specifications/.

[10] D. Gouk, S. Lee, M. Kwon, and M. Jung. Direct Access, High-Performance Memory Disaggregation with DirectCXL. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 287–294, Carlsbad, CA, July 2022. USENIX Association.

[11] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy. SoftNIC: A Software NIC to Augment Hardware. Technical Report UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015.

[12] R. Huggahalli, R. Iyer, and S. Tetrick. Direct cache access for high bandwidth network i/o. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 50–59, 2005.

[13] S. Ibanez, A. Mallery, S. Arslan, T. Jepsen, M. Shahbaz, C. Kim, and N. McKeown. The nanoPU: A Nanosecond Network Stack for Datacenters. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 239–256. USENIX Association, July 2021.

[14] Intel Corporation. An Introduction to the Intel QuickPath Interconnect. https://www.intel.ca/content/dam/doc/white-paper/quick-path-interconnect-introduction-paper.pdf.

[15] Intel Corporation. Intel Data Direct I/O Technology. https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html.

[16] Intel Corporation. Intel Data Streaming Accelerator Architecture Specification. https://cdrdv2-public.intel.com/671116/341204-intel-data-streaming-accelerator-spec.pdf.

[17] Intel Corporation. Intel Memory Latency Checker v3.9a. https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html.

[18] Intel DPDK Validation Team. Intel Ethernet Performance Report with DPDK 21.11. http://fast.dpdk.org/doc/perf/DPDK_21_11_Intel_NIC_performance_report.pdf.

[19] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 489–502, Seattle, WA, Apr. 2014. USENIX Association.

[20] A. Kalia, M. Kaminsky, and D. Andersen. Datacenter RPCs can be General and Fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 1–16, Boston, MA, Feb. 2019. USENIX Association.

[21] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA Efficiently for Key-Value Services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 295–306, New York, NY, USA, 2014. Association for Computing Machinery.

[22] A. Kaufmann, T. Stamler, S. Peter, N. K. Sharma, A. Krishnamurthy, and T. Anderson. TAS: TCP Acceleration as an OS Service. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[23] N. Lazarev, S. Xiang, N. Adit, Z. Zhang, and C. Delimitrou. Dagger: Efficient and fast rpcs in cloud microservices with near-memory reconfigurable nics. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 36–51, New York, NY, USA, 2021. Association for Computing Machinery.

[24] H. Li, D. S. Berger, L. Hsu, D. Ernst, P. Zardoshti, S. Novakovic, M. Shah, S. Rajadnya, S. Lee, I. Agarwal, M. D. Hill, M. Fontoura, and R. Bianchini. 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, Volume 2*, ASPLOS 2023, page 574–587, New York, NY, USA, 2023. Association for Computing Machinery.

[25] M. Liu, T. Cui, H. Schuh, A. Krishnamurthy, S. Peter, and K. Gupta. Offloading Distributed Applications onto SmartNICs Using IPipe. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 318–333, New York, NY, USA, 2019. Association for Computing Machinery.

[26] M. Marty, M. de Kruijf, J. Adriaens, C. Alfeld, S. Bauer, C. Contavalli, M. Dalton, N. Dukkipati, W. C. Evans, S. Gribble, N. Kidd, R. Kononov, G. Kumar, C. Mauer, E. Musick, L. Olson, E. Rubow, M. Ryan, K. Springborn, P. Turner, V. Valancius, X. Wang, and A. Vahdat. Snap: A Microkernel Approach to Host Networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 399–413, New York, NY, USA, 2019. Association for Computing Machinery.

[27] Marvell. Marvell LiquidIO III. https://www.marvell.com/content/dam/marvell/en/public-collateral/embedded-processors/marvell-liquidio-III-solutions-brief.pdf.

[28] S. Min, M. Alian, W.-M. Hwu, and N. S. Kim. Semi-coherent dma: An alternative i/o coherency management for embedded systems. *IEEE Computer Architecture Letters*, 17(2):221–224, 2018.

[29] C. Mitchell, Y. Geng, and J. Li. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 103–114, San Jose, CA, June 2013. USENIX Association.

[30] C. Mitchell, K. Montgomery, L. Nelson, S. Sen, and J. Li. Balancing CPU and Network in the Cell Distributed B-Tree Store. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 451–464, Denver, CO, June 2016. USENIX Association.

[31] B. Montazeri, Y. Li, M. Alizadeh, and J. Ousterhout. Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 221–235, New York, NY, USA, 2018. Association for Computing Machinery.

[32] R. Neugebauer, G. Antichi, J. F. Zazo, Y. Audzevich, S. López-Buedo, and A. W. Moore. Understanding PCIe Performance for End Host Networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 327–341, New York, NY, USA, 2018. Association for Computing Machinery.

[33] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot. Scale-out NUMA. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, page 3–18, New York, NY, USA, 2014. Association for Computing Machinery.

[34] NVIDIA Corporation. NVIDIA Mellanox NICs Performance Report with DPDK 21.11. http://fast.dpdk.org/doc/perf/DPDK_21_11_Mellanox_NIC_performance_report.pdf.

[35] NVIDIA Corporation. NVIDIA NVSwitch Technical Overview. https://images.nvidia.com/content/pdf/nvswitch-technical-overview.pdf.

[36] OpenCAPI Consortium. OpenCAPI Specifications. https://opencapi.org/technical/specifications/.

[37] PCI-SIG. PCI Express Specifications. https://pcisig.com/specifications/.

[38] S. Pirelli and G. Candea. A Simpler and Faster NIC Driver Model for Network Functions. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 225–241. USENIX Association, Nov. 2020.

[39] PK Gupta. Intel Xeon+FPGA Platform for the Data Center. https://reconfigurablecomputing4themasses.net/files/2.2%20PK.pdf.

[40] Prakash Chauhan and Mahesh Wagh. CXL Memory Challenges. https://hc34.hotchips.org/assets/program/tutorials/CXL/Hot%20Chips%202022%20CXL%20MemoryChallenges.pdf.

[41] Y. Ren, G. Liu, V. Nitu, W. Shao, R. Kennedy, G. Parmer, T. Wood, and A. Tchana. Fine-Grained Isolation for Scalable, Dynamic, Multi-tenant Edge Clouds. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 927–942. USENIX Association, July 2020.

[42] L. Rizzo. netmap: A Novel Framework for Fast Packet I/O. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 101–112,

Boston, MA, June 2012. USENIX Association.

[43] L. Rizzo, P. Valente, G. Lettieri, and V. Maffione. PSPAT: Software packet scheduling at hardware speed. *Computer Communications*, 120, 02 2018.

[44] H. N. Schuh, W. Liang, M. Liu, J. Nelson, and A. Krishnamurthy. Xenic: SmartNIC-Accelerated Distributed Transactions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 740–755, New York, NY, USA, 2021. Association for Computing Machinery.

[45] L. Shalev, J. Satran, E. Borovik, and M. Ben-Yehuda. IsoStack—Highly Efficient Network Processing on Dedicated Cores. In *2010 USENIX Annual Technical Conference (USENIX ATC 10)*. USENIX Association, June 2010.

[46] A. Singhvi, A. Akella, M. Anderson, R. Cauble, H. Deshmukh, D. Gibson, M. M. K. Martin, A. Strominger, T. F. Wenisch, and A. Vahdat. CliqueMap: Productionizing an RMA-Based Distributed Caching System. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 93–105, New York, NY, USA, 2021. Association for Computing Machinery.

[47] Y. Sun, Y. Yuan, Z. Yu, R. Kuper, I. Jeong, R. Wang, and N. S. Kim. Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices, 2023.

[48] The Linux Kernel Archives. Linux Base Driver for the Intel Ethernet Controller 700 Series. https://www.kernel.org/doc/html/latest/networking/device_drivers/ethernet/intel/i40e.html.

[49] Universal Chiplet Interconnect Express. UCIe 1.0 Specification. https://www.uciexpress.org/specification.

[50] Universal Chiplet Interconnect Express. UCIe 1.0 Specification. https://www.uciexpress.org/specification.

[51] Virtio. Libvirt Virtualization API. https://wiki.libvirt.org/Virtio.html.

[52] VMWare Incorporated. Performance Evaluation of VMXNET3 Virtual Network Device. https://www.vmware.com/pdf/vsp_4_vmxnet3_perf.pdf.

[53] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast In-Memory Transaction Processing Using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 87–104, New York, NY, USA, 2015. Association for Computing Machinery.

[54] Y. Yuan, M. Alian, Y. Wang, R. Wang, I. Kurakin, C. Tai, and N. S. Kim. Don't forget the i/o when allocating your llc. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 112–125, 2021.