Efficient Remote Memory Ordering for Non-Coherent Interconnects

Wei Siew Liew[*]
u1529306@utah.edu
University of Utah
Salt Lake City, Utah, USA

Md Ashfaqur Rahaman[*]
ashfaq@cs.utah.edu
University of Utah
Salt Lake City, Utah, USA

Adarsh Patil
Adarsh.Patil@arm.com
Arm
Cambridge, UK

Ryan Stutsman
stutsman@cs.utah.edu
University of Utah
Salt Lake City, Utah, USA

Vijay Nagarajan
vijay@cs.utah.edu
University of Utah
Salt Lake City, Utah, USA

Abstract

Software using non-coherent interconnects like PCI Express requires fine-grained memory ordering, but current hardware mandates the use of costly source-side serialization. We show that this architectural mismatch severely limits the performance of two critical applications: (1) the transmission of network packets from a CPU to a NIC (requiring write-to-write ordering) and (2) key-value store lookups by an RDMA-enabled NIC (requiring read-to-read ordering).

We address this by proposing a new destination-based ordering model and the hardware-software co-design comprising PCIe extensions and ISA extensions that allow software to express ordering intent efficiently. Novel microarchitecture at the Root Complex enforces these expressed semantics, eliminating source-side stalls. Our approach significantly improves the throughput of these application kernels and enables new, simpler protocols that outperform the state-of-the-art.

CCS Concepts: • **Hardware → Buses and high-speed links**; *Networking hardware*; • **Computer systems organization** → *Interconnection architectures.*

Keywords: Memory Consistency Models, I/O Interconnects, Non-Coherent Interconnects, RDMA, PCIe

ACM Reference Format:
Wei Siew Liew, Md Ashfaqur Rahaman, Adarsh Patil, Ryan Stutsman, and Vijay Nagarajan. 2026. Efficient Remote Memory Ordering for Non-Coherent Interconnects. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '26), March 22–26, 2026, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3779212.3790156

[*]Both authors contributed equally to this work.

1 Introduction

Modern servers are frequently limited by the performance of the interconnects that link CPUs with devices like network interface cards (NICs) and GPUs. We focus on non-coherent interconnects, like PCI Express (PCIe), which drive today's I/O systems. While coherent I/O interconnects like Compute Express Link (CXL) [8] represent a new direction, thoroughly understanding and optimizing existing systems is a prerequisite to justifying the added complexity and cost of a full-coherence paradigm. This work examines the interaction between a NIC and host CPUs across a PCIe interconnect, exposing a fundamental bottleneck: the high cost of enforcing ordering on remote memory operations.

Remote ordering issues arise when a device (CPU, GPU, NIC, etc.) accesses two or more addresses belonging to another device in a specific sequence, a requirement for many communication patterns. For example, when a host sends packets to a NIC via memory-mapped I/O (MMIO) writes, the packet order must be maintained. Similarly, in an RDMA-based key-value store (KVS), the NIC may first need to acquire a lock and then read the targeted object; violating this ordering constraint would compromise data integrity.

The core problem today is a mismatch between the memory consistency model required by modern I/O software and the capabilities of the PCIe I/O stack. Specifically, the PCIe specification [14] currently lacks the semantics to express and enforce fine-grained memory ordering end-to-end. This forces systems to rely on costly, source-side ordering mechanisms, resulting in serialization and significant performance overheads. For example, in a CPU-NIC transmit path, achieving ordering requires a store fence from the source CPU after every MMIO write, making a direct MMIO-based path completely impractical. This is why modern transmit paths are built on complex workarounds involving MMIO doorbells and DMA reads [19, 20, 30, 31]. Likewise, in a KVS accessed via RDMA, enforcing ordering at the source—either at the server's NIC or at the client—can make ordered remote reads over an order of magnitude slower than their unordered counterparts. These costly mechanisms that serialize at the source to preserve ordering hurt performance significantly.

To overcome this, we propose a new, destination-based ordering paradigm. Our solution co-designs a new PCIe interface and a suite of microarchitectural mechanisms that shift the responsibility for ordering from the initiating source to hardware at or near the remote destination. This allows remote operations to be issued concurrently while ordering is enforced efficiently at the target. Our approach enables a simple and efficient MMIO-based transmit path to achieve line-rate throughput (100 Gb/s on a single core) without store fences and near-zero penalty DMA read ordering.

The foundation of our approach is to make memory ordering an explicit, first-class concern from the instruction set architecture (ISA) down to the interconnect. We introduce acquire/release semantics directly into the PCIe specification, bridging the conceptual gap between how programmers reason about memory consistency and how the interconnect enforces it. We also propose elevating release consistency-style MMIO loads and stores to first-class citizens within the host's ISA, providing a precise, hardware-supported interface for a processor to signal its memory ordering intent. Our key insight is that by making memory ordering explicit, we can enable a more efficient hardware implementation. Specifically, our solution includes a Remote Load-Store Queue (RLSQ) in the PCIe Root Complex capable of enforcing the new acquire/release rules with minimal performance penalties.

In this work [1] we make the following contributions:

- We identify performance pathologies for remote memory ordering with non-coherent interconnects (e.g. PCIe). We analyze the overheads for MMIO and DMA ordering, and we quantify the costs with application kernels.
- We propose a destination-based ordering architecture for non-coherent interconnects. We introduce new acquire/release semantics in the PCIe specification and acquire/release MMIO instructions in the host ISA that explicitly communicate ordering requirements, enabling endpoints to avoid costly serialization at the source.
- We define novel microarchitectural support to efficiently enforce destination-based ordering. Our design includes a Remote Load-Store Queue (RLSQ) in the Root Complex, which leverages the new PCIe semantics to enforce ordering while maximizing parallelism.
- We demonstrate significant performance gains on application kernels. Our results show that our approach enables a simple, fence-free MMIO transmit path that delivers line-rate throughput. An optimized RLSQ improves RDMA-based KVS performance by up to 50.9× for 64 B objects using a single RDMA queue pair in simulation.

[1]https://github.com/icsa-caps/efficient-remote-memory-ordering.git

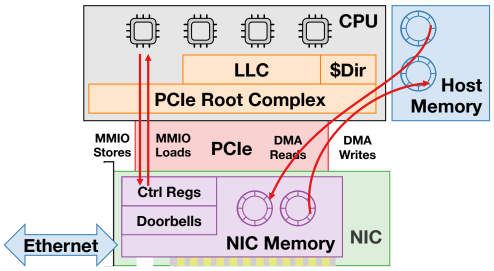


Figure 1. System Memory Interactions. MMIOs from CPU are routed over PCIe via the Root Complex (RC) to the NIC. DMAs from the NIC access host memory via the RC. Our work proposes extending PCIe TLPs for ordering, extending the CPU's ISA to specify MMIO ordering, and a Remote Load-Store Queue at the RC.

Table 1. PCIe Ordering Guarantees

W→W	R→R	R→W	W→R
Yes	No	No	Yes

- We present a design for RDMA-based KVS lookups that exploits our efficient remote memory ordering and is simpler than existing approaches while delivering throughput 1.6× higher than FaRM [11].

2 Remote Memory Ordering Today

Achieving efficient remote memory ordering between a host CPU and a NIC is fundamentally constrained by the asymmetric nature of the underlying PCIe interconnect. Communication in this system model (Figure 1) relies on two primitives: Memory-Mapped I/O (MMIO), where the CPU accesses NIC memory, and Direct Memory Access (DMA), where the NIC accesses host memory.

While PCIe provides strong ordering for *posted* writes, its weak ordering for *non-posted* reads (Table 1) [14] creates significant performance bottlenecks for common Read-to-Read (R→R) patterns. For example, a slow DMA read from main memory can be passed by a faster DMA read that hits in the host cache, violating the fine-grained ordering required by software. For MMIO, however, the bottleneck stems from how the host CPU interacts with the PCIe interface, which causes inefficiencies in both W→W and R→R MMIO ordering. This section analyzes the architectural roots of these overheads for both DMA and MMIO and demonstrates their impact on modern CPU-NIC software stacks.

2.1 DMA Ordering

R→R DMA ordering presents a significant bottleneck. Consider a litmus test where a NIC must read a status flag before its corresponding data; correctness dictates that the NIC not see stale data after seeing an updated flag. Today's only solution is for the NIC to enforce this by serializing the requests—issuing the flag read, waiting for the full PCIe round-trip completion, and only then issuing the data read. This synchronous "stop-and-wait" execution significantly

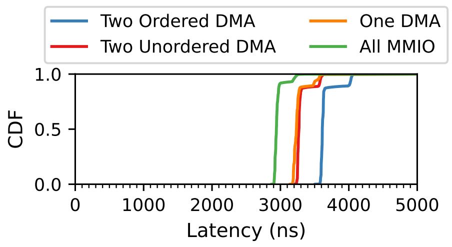


Figure 2. Distribution of RDMA WRITE latency between two hosts using different patterns for operation submission. Including *One DMA* read as part of the submission adds about 300 ns to the delay over *All MMIO* (zero DMAs). Using *Two Ordered DMA* reads adds about another 300 ns, while using *Two Unordered DMA* reads is about the same as *One DMA* since the two DMAs can be overlapped.

reduces read throughput. Simple alternative approaches to efficient ordering are insufficient: pipelining the reads fails because the PCIe fabric can reorder them, and a NIC-side reorder buffer fails because the host-side completions can return out of order (e.g., a cached data value may return before an uncached flag value).

W→W DMA ordering, however, is handled efficiently. Consider a litmus test where a NIC must write data before its corresponding status flag. Because PCIe guarantees that posted writes from the same source are not reordered, the NIC can pipeline data and flag writes, relying on the interconnect to preserve their order with minimal performance impact.

The Cost of DMA Ordering. To evaluate the performance impact of enforcing R→R DMA ordering, we devised an experiment to isolate the latency cost of serializing DMA reads. Our strategy relies on the fact that an RDMA WRITE operation inherently requires the client NIC to read data from host memory. By manipulating how these WRITE operations are submitted—specifically the layout of the Work Queue Element (WQE) and its payload—we can force the NIC into specific DMA read patterns, ranging from fully parallel (unordered) to strictly serialized (ordered).

We used ConnectX-6 Dx 100 Gb/s NICs to implement this (§6.4 provides full details). Each experiment posts one-sided RDMA WRITE operations to a client NIC using different submission techniques (e.g., BlueFlame MMIO vs. standard DMA) to produce the desired DMA read behaviors on the client host. We plot the cumulative distribution function (CDF) of the end-to-end 64 B RDMA WRITE latency (client issue time until completion). All measurements are for a single client thread using one Queue Pair (QP).

Figure 2 shows the results of the experiment. When the RDMA WRITE WQE and the data to be transmitted are provided to the client NIC via MMIO using NVIDIA's BlueFlame optimization (*All MMIO*), each operation takes a median of 2,941 ns. This case issues no DMAs at the client NIC and serves as a baseline that only measures the end-to-end latency of a 64 B RDMA WRITE operation.

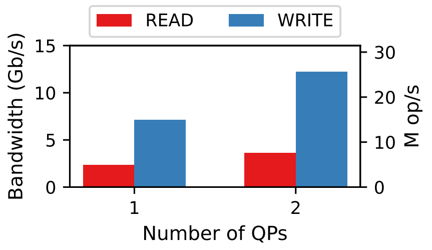


Figure 3. Pipelined RDMA read/write bandwidth for 64 B objects with 1 and 2 QPs. The ordered write bandwidth is significantly higher than the read bandwidth.

In *One DMA*, each WRITE WQE is provided to the client NIC via MMIO, but the 64 B that it must transmit is placed in the client host's memory. Hence, the NIC triggers a single DMA read to fetch the data after receiving the WQE. In this setup, the median operation completes in 3,234 ns, adding 293 ns of delay compared to the case where the NIC does not issue DMA operations; this represents the latency of one 64 B DMA read.

In *Two Unordered DMA*, each WRITE WQE is issued via MMIO, but it specifies (via a scatter-gather list provided as part of the MMIO) two 64 B data buffers for transmission. In this case, the median WRITE completes in 3,271 ns, 330 ns more than when the NIC issues no DMA operations, and just 37 ns slower than when the NIC issues a single DMA operation. This implies that the client NIC overlaps the two DMA reads by relying on the fact that RDMA does not guarantee cache line access order when issuing an RDMA WRITE.

Finally, in *Two Ordered DMA*, WRITEs are not issued via MMIO. Instead, the WQE for the operation is placed in memory and references a 64 B data payload elsewhere in memory. When the client NIC receives the MMIO doorbell write, it must first fetch the WQE for client host memory by issuing a DMA and waiting for its completion to retrieve the address of the payload. Then, it must issue a separate DMA to retrieve the 64 B to transmit. As a result, each operation takes 3,613 ns to complete—672 ns longer than when no DMA is performed and 342 ns longer than *Two Unordered DMA*. This shows that when there is an ordering dependency between the two DMAs, the NIC must issue each operation and wait, resulting in about a 300 ns delay.

What is the impact of this stop-and-wait ordering? Today, without PCIe R→R ordering guarantees, any two RDMA read operations that require ordering must be stalled at the server NIC, which introduces a serialization delay between the reads.

This latency penalty directly constrains the maximum achievable throughput. When we pipeline many 64 B RDMA READ operations over a single QP, we observe that the server NIC performs READs with inter-read latencies similar to the previously measured DMA latency of about 300 ns. Figure 3 shows that with pipelined RDMA READs, throughput reaches approximately 5.0 Mop/s (2.37 Gb/s), implying that the server NIC completes an operation every 200 ns. The RDMA specification does not require the server NIC to enforce RDMA R → R ordering, but these results suggest that if strict ordering were enforced on current hardware, performance could not exceed this limit.

In contrast, RDMA WRITEs provide much higher throughput than RDMA READs (3×). Because of RDMA's strong W→W ordering guarantees, the server NIC can begin processing the next WRITE as soon as the write DMA operations for the previous WRITE are enqueued, allowing the server NIC to issue incoming RDMA WRITEs from the QP efficiently.

Our goal is to make the same high-performance pipelining that is currently possible for writes available for reads.

Impact. The high cost of R→R ordering has significant implications for applications like one-sided key-value stores. A typical get operation requires a "check-before-read" pattern—reading a lock or metadata before the object itself—to ensure correctness against concurrent puts. Due to the lack of R→R ordering guarantees in current PCIe implementations, this ordering must be enforced at the server NIC (assuming a smart NIC capable of lock value checks and subsequent object reads [6, 29]). We observe that enforcing this ordering reduces get throughput by more than an order of magnitude compared to an ideal low-cost ordering primitive. Worse, this ordering is typically enforced in applications by *stalling at the client NIC*—waiting for the round-trip completion of one operation before issuing the next, which results in disastrously low performance.

To circumvent this limitation, state-of-the-art key-value stores are forced into complex workarounds such as embedding versioning metadata into every cache line [16]. While functional, these protocols impose their own significant tax: they complicate application development and, as we show later, the overhead of stripping this metadata at the client reduces the performance of otherwise advanced systems like FaRM. This illustrates a clear architectural limitation—software is paying a heavy price in both performance and complexity to compensate for the interconnect's lack of an efficient, hardware-supported ordering mechanism.

2.2 MMIO Ordering

R→R MMIO Ordering is also inefficient due to the weak ordering guarantees of PCIe reads. To ensure ordering between reads, a host CPU must serialize read requests, mirroring the performance bottleneck observed with DMA R→R ordering. This serialization prevents concurrent PCIe read transactions, leading to significant latency and reduced throughput.

W→W MMIO Ordering is also inefficient, but solely due to host CPUs' microarchitecture. The bottleneck is the store fence instruction needed to enforce order between writes

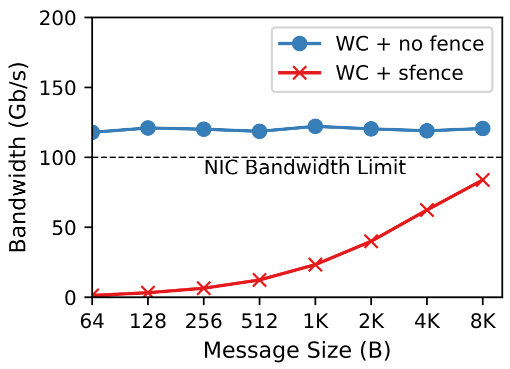


Figure 4. MMIO Write Bandwidth for Combined Stores to a ConnectX-6 Dx. sfences thwart achieving line rate transmission.

to a write-combining memory region. Write-combining efficiently batches MMIO transfers [21, 31], but the CPU does not guarantee these buffered writes reach the Root Complex in program order. If two cache line-sized writes must be written in order from the host to the NIC's memory, software must insert a store fence between the writes. This fence forces a hard serialization point, stalling the processor until the first write is flushed to the Root Complex, negating the pipelining benefits of PCIe's otherwise efficient posted writes.

The Cost of W→W MMIO Ordering. Given the expected similarity in cost between R→R MMIO and DMA ordering, we focused our experimental evaluation on W→W MMIO ordering. We replicate an experiment from prior works [22, 31]. The experiment measures the throughput of write-combined stores to NIC memory, enforcing order by inserting a store fence after every B bytes (emulating a packet of size B), and it compares this to a baseline without fences.

Figure 4 corroborates recent results [22, 31]. Without ordering, we achieved a throughput of 122 Gb/s. Enforcing ordering, even with packet sizes as large as 512 bytes, reduced throughput by 89.5%. We also confirmed that using strictly non-cacheable stores—which also enforce order—yields even worse performance. These results provide quantitative evidence that the sfence is the primary architectural bottleneck, validating our analysis of W→W MMIO ordering.

Impact. The above benchmark models a CPU-to-NIC transmit path, where maintaining packet order is crucial. The results reveal a common misconception: the bottleneck is not that CPUs cannot saturate the PCIe bus, but rather the architectural cost of ordering. While unordered MMIO writes can exceed 100 Gb/s, the sfence required for ordering slashes throughput by an order of magnitude for small packets.

This severe performance penalty explains why modern systems abandon the simple, direct MMIO transmit path. Instead, they rely on a costly workaround: the CPU writes packet data to host memory and then writes to an MMIO "doorbell" register. This doorbell triggers the NIC to initiate a separate DMA operation to fetch the data. This complex, indirect path adds significant round-trip latency and still struggles to achieve line rate without specialized hardware [30]. These workarounds are a direct consequence of a missing architectural primitive for software to efficiently express its fine-grained ordering requirements to the hardware.

3 Fast Remote Memory Ordering Overview

Our approach is a hardware-software co-design with new architectural and microarchitectural support to eliminate remote ordering bottlenecks. We propose PCIe and host ISA extensions to express ordering, as well as changes at the Root Complex (RC) to enforce these ordering semantics efficiently.

Efficient R→R DMA Ordering. Our solution to the R→R DMA bottleneck is a two-part co-design. First, we extend the PCIe specification to allow a NIC to pipeline ordered reads (e.g., a lock check and a subsequent data read) by explicitly annotating their required order. Second, we enhance the Root Complex (RC) to enforce this ordering against the host's coherent memory. This second step is crucial: merely preserving order across the interconnect is insufficient, as parallel requests to the host memory system can complete out of order (e.g., a data read that hits in the cache can pass the lock read that misses). Even a simple, sequential enforcement at the RC could provide a benefit.

This gain stems from shifting the serialization point from the source (the NIC) to the destination (the RC). In the baseline, NIC-side serialization incurs the full round-trip latency of the interconnect and the host memory access, a stall of ≈500 ns. This limits throughput to roughly 2 million ordered reads per second (Mops/s). By moving enforcement to the RC, our approach allows the NIC to pipeline read requests, amortizing the long interconnect latency. The throughput bottleneck then becomes the RC's sequential access to host memory via the RC's Remote Load-Store Queue (RLSQ), which is ≈100 ns per read. This improves throughput by 5× to 10 Mops/s.

To eliminate this remaining serialization and achieve near-ideal performance, our advanced design draws inspiration from speculative memory ordering techniques employed in out-of-order processors [7, 12]. It allows the RLSQ to execute reads speculatively and in parallel, buffering the results and delivering the data to the waiting PCIe requests while honoring the required ordering. This "out-of-order execute, in-order commit" model allows the latency of multiple memory accesses to be overlapped, significantly boosting throughput.

Correctness is maintained by cleanly integrating the RLSQ with the host's existing cache coherence protocol. The queue tracks in-flight speculative reads and snoops the coherence fabric, much like a CPU cache. An intervening write from a host core to a speculatively read address triggers an invalidation message to the queue. This invalidation squashes the speculation and retries the read to fetch the up-to-date value. This mechanism ensures that in the common case without such conflicts, ordered reads perform at nearly the same speed as fully unordered reads, effectively solving the bottleneck.

Efficient W→W MMIO Ordering. The CPU-NIC transmit path is severely throttled by the architectural cost of enforcing W→W MMIO ordering. To ensure writes arrive at the Root Complex in order, current systems must execute a store fence after each packet. This serialization adds ≈100 ns of latency per packet, capping the achievable throughput for 64-byte packets at ≈5 Gb/s—a fraction of a modern 100 Gb/s link. To eliminate this bottleneck, we propose labeling transactions with sequence numbers and using a reorder buffer at the destination, taking inspiration from our previous work [22]. To realize this idea, we propose enhancements to the host ISA to explicitly identify remote MMIO/PCIe write operations and their ordering requirements. This allows the CPU to issue a stream of packets without stalling on a fence. A reorder buffer (ROB) at the Root Complex then uses these sequence numbers to reconstruct the correct program order before forwarding the writes to the NIC in an ordered manner.

4 Architectural Support

This section details the architectural support underpinning our approach to fast remote memory ordering. This support encompasses two key components: extensions to the PCIe specification to express the ordering requirements of PCIe reads (enabling R→R ordering), and extensions to the host ISA allowing it to express MMIO loads, stores, and their ordering.

4.1 PCIe Extensions for Remote Memory Ordering

To address the inefficiencies in R→R ordering, we propose extending the PCIe specification to enable devices to express ordering requirements for their read requests. Similar to the existing distinction between ordered and unordered writes in PCIe, our extension introduces the ability to differentiate between ordered and unordered reads. However, prevalent producer-consumer communication patterns in host-device interactions necessitate a more nuanced approach than simply adopting strong and relaxed ordering semantics.

Consider a common scenario where the host writes a series of data items to memory and subsequently sets a flag to signal their availability. A device then polls the flag, and upon observing it set, proceeds to read the previously written data. Examining the device's actions (a flag DMA read followed by data DMA reads), we find that the data reads must occur after the flag read, but the data reads themselves do not have an inherent order relative to each other. This pattern cannot be accurately and efficiently expressed using only strong or relaxed ordering. Marking all reads as strong would be correct but overly conservative, hindering potential performance optimizations. Conversely, marking only the flag read as strong would fail to enforce the necessary ordering of data reads after the flag.

The ideal interface for expressing such producer-consumer relationships is through the use of *acquire* and *release* semantics, a model effectively employed by several ISA memory consistency models like ARM and RISC-V. Therefore, we advocate for the adoption of acquire and release semantics within the PCIe specification. Revisiting our example, the DMA read for the flag would be marked as an acquire operation, ensuring that all subsequent memory accesses by the device (the data reads) observe the state of memory at or after the flag was read. The subsequent DMA reads for the data items could then be marked as relaxed, allowing them to be reordered with respect to each other while still being correctly ordered after the flag read—precisely the desired behavior for maximizing performance in this common pattern. Conversely, we also advocate for the use of PCIe release writes and unordered writes instead of simply strongly ordered writes and weakly ordered writes.

Encoding acquire and release ordering in PCIe is straightforward. For writes, we can re-purpose an existing *relaxed ordering* bit. When this bit is set for a write, it can be interpreted by the Root Complex and PCIe devices as a *release* operation, signaling that prior actions should become visible to other agents. For reads, we can add a new, analogous *acquire* bit in the TLP header. Setting this bit for a read would indicate to the Root Complex and the requesting device that subsequent actions should see the results of this read.

4.2 Host ISA extensions for MMIO

To efficiently enforce remote ordering, a host CPU ISA must be rich enough to express ordering constraints for MMIO loads and stores. Current host ISAs typically lack explicit mechanisms to differentiate between local memory accesses and MMIO operations targeting remote devices. Instead, today's systems rely on the host processor's support for mapping addresses as non-cacheable or write-combining [1], augmented with host memory ordering instructions (fences) for enforcing ordering between the processor and the Root Complex. Hence, many ISAs have complex and non-intuitive interactions between their memory ordering guarantees and PCIe ordering rules. For example, x86 processors strictly serialize reads to uncached MMIO regions, stalling execution to preserve order at the source. This performance penalty is effectively wasted, however, as the PCIe fabric is permitted to reorder these requests in flight!

The RISC-V ISA, with its flexible fence that expresses both MMIO ordering and host memory ordering (`fence iorw,iorw`) offers a clearer path forward. Its fences are already expressive enough to describe the necessary software constraints, from simple MMIO Write-after-Write (`fence o,o`) to complex memory-to-I/O patterns (`fence w,o`). The key limitation is the implementation that has to be conservative, owing to a mismatch between the CPU's behavior and the interconnect's guarantees. Today, a compliant CPU must still stall at the fence until prior operations drain. Our model reinterprets this. Instead of a stall, the fence acts as a directive for the microarchitecture to inject ordering metadata (e.g., sequence numbers) into the MMIO stream. This provides the same ordering guarantee to downstream hardware without processor stalls, transforming the fence from a costly serialization point into a lightweight ordering directive.

While reinterpreting fences is a pragmatic step, the most principled solution is to elevate ordered remote operations to first-class citizens in the ISA. This aligns the host ISA with the acquire/release semantics we proposed for PCIe, ensuring a unified ordering model from the CPU to the device. We therefore propose four new instruction variants: MMIO-Store, MMIO-Release, MMIO-Load, and MMIO-Acquire. To be correct and intuitive, their semantics must integrate with the host's memory model: an MMIO-Release must ensure all prior host memory operations are visible before the MMIO write is observed, and an MMIO-Acquire must ensure all subsequent host memory operations happen only after the MMIO read completes. This provides a clean programming model for managing ordering across the complex host-device boundary.

5 Microarchitectural Support

This section details the two key microarchitectural components of our design: an enhanced Remote Load-Store Queue (RLSQ) in the Root Complex to efficiently order DMA requests, and the host CPU support required to implement our new MMIO instructions.

5.1 Remote Load-Store-Queue

The RLSQ in the PCIe Root Complex is the microarchitectural bridge that enforces the PCIe interconnect's ordering rules on the host's coherent memory system.

Baseline. In the baseline design [10, 32], the RLSQ's behavior is a reflection of PCIe's guarantees. Because PCIe reads are weakly-ordered, the RLSQ dispatches incoming DMA read requests to the coherence directory in parallel.

Conversely, because PCIe writes are strongly-ordered, the RLSQ processes DMA write requests serially to ensure they are applied to memory in the correct order. However, the RLSQ can optimize by issuing the coherence requests (e.g., invalidations or ownership requests) for multiple pending writes in parallel. While these coherence actions are in flight, the actual data writes to the cache or memory controller are strictly serialized, committing only from the head of the RLSQ's FIFO queue after all preceding operations have completed. This allows high latency coherence messages to be overlapped, improving throughput while ensuring that writes become visible to the host in the correct PCIe order.

Proposed design: Release-Acquire RLSQ. Our proposed design faithfully implements the ordering guarantees of the new *acquire* and *release* operations. Relaxed PCIe reads and writes are issued concurrently from the RLSQ. However, a PCIe acquire blocks the issue of all subsequent requests until its own coherent request completes. Conversely, a PCIe release stalls until all prior requests have been completed before its own coherent request is issued.

These semantics are key to enabling high-performance, optimistic NICs without sacrificing correctness. Consider a NIC that, to hide latency, speculatively pipelines an acquire read of a key-value store lock followed by a data read. In a baseline Root Complex, these parallel requests can race within the host memory system, potentially allowing the data read to return a stale value even after the acquire read completes. This violates the acquire semantic and breaks correctness. Our RLSQ prevents this race by stalling the speculative data read until the acquire is fully resolved, guaranteeing a consistent memory view and making aggressive, high-performance NIC designs safe.

Optimization: Thread-specific Ordering. The simple Release-Acquire RLSQ is overly conservative, creating false dependencies by enforcing order globally across all NIC traffic. An acquire from one thread context (e.g., a Queue Pair) will needlessly stall an independent request from a different thread. To solve this, we propose extending the PCIe TLP to carry a thread ID. The RLSQ then uses this ID to enforce acquire/release semantics on a per-thread basis, allowing requests from different threads to proceed in parallel. This is a logical extension of the ID-based Ordering (IDO) principle that the PCIe specification already provides for writes [14], applying a similar thread-aware model to our new domain of ordered reads.

Microarchitecturally, this per-thread ordering can be implemented efficiently. While physically partitioning the RLSQ into separate queues is possible, a more resource-efficient design uses a single, logically partitioned queue [24]. This approach maintains a small amount of per-thread state (e.g., an "awaiting acquire" flag). An incoming request is stalled only if its thread ID matches a thread that is currently in this waiting state. This design eliminates false dependencies and maximizes parallelism between independent contexts.

Optimization: Speculative DMA Ordering. To eliminate stalls within a single thread, our most advanced design employs speculation, similar to modern processors [12]. For an Acquire→Read sequence (e.g., an acquire on X, then a read on Y), the RLSQ issues both requests to the host memory system speculatively and in parallel. To preserve correctness, it buffers the result of the speculative read (Y) and responds to the NIC only after the acquire (X) has completed. This maintains the illusion of serial execution while overlapping the memory latencies.

Correctness against concurrent host writes is ensured by integrating the RLSQ with the host's coherence protocol. This requires no changes to the protocol itself—which is notoriously hard to design and verify—but simply treats the RLSQ as a new coherent agent, akin to adding another cache. The RLSQ is tracked as a temporary sharer for in-flight speculative reads, allowing it to snoop coherence traffic. An intervening host write to a speculative address triggers a standard directory invalidation, which squashes the buffered result and forces a retry of that single read to fetch the up-to-date value. Crucially, unlike a CPU's Load-Store Queue, only the conflicting read is squashed, not all subsequent speculative operations. Because the RLSQ speculates on known addresses, the penalty for mis-speculation is low, making the approach highly efficient.

This speculative principle also applies to Write→Release ordering. The RLSQ can speculatively issue the coherence actions (e.g., invalidations) for a release concurrently with the preceding data writes. Once the data writes are confirmed complete by the memory system, the release can also complete, having already finished its high-latency coherence work in parallel.

5.2 Host Support for MMIO Operations

Next, we turn our attention to the microarchitectural support required for the new MMIO load and store instructions introduced in §4.2; specifically we describe efficient implementations for the MMIO-Load, MMIO-Store, MMIO-Acquire, and MMIO-Release instructions, ensuring that their ordering semantics are enforced when interacting with remote devices over the PCIe interconnect.

Elevating MMIO operations to first-class citizens in the ISA lets the microarchitecture manage their memory ordering more effectively. The key microarchitectural support involves associating a sequence number with each MMIO operation. For instance, an MMIO-Store to address X followed by an MMIO-Release to address Y would be assigned strictly increasing sequence numbers, explicitly denoting their order. We then maintain a reorder buffer (ROB) at the Root Complex to reconstruct this order. If the MMIO-Release (with a higher sequence number) arrives at the root complex before the MMIO-Store (with a lower sequence number), the ROB recognizes that a later operation has been received out of order. The Root Complex delays issuing the PCIe write corresponding to the MMIO-Release to the device until the PCIe write for the earlier MMIO-Store (with the lower sequence number) has also arrived and been issued.

As with DMA operations, we incorporate the hardware thread ID as part of the sequence number. This allows the ROB to distinguish and independently manage the ordering of MMIO operations originating from different hardware threads within the CPU.

The microarchitectural implementation of the ROB is straightforward. We can maintain a simple state machine that tracks the highest sequence number for which all preceding sequence numbers have also been received. Once such a contiguous sequence is identified, all the corresponding MMIO operations within that sequence can be dispatched as ordered PCIe writes towards the target device.

This sequence number-based approach is flexible in its placement of the ROB. The Root Complex is an obvious choice, but this mechanism would also support ROBs at device endpoints. Placing the ROB at the endpoint opens up the possibility of using unordered PCIe reads and writes throughout the interconnect for MMIO reads and writes since end-to-end ordering can be guaranteed solely by the sequence numbers and the ROB at the final destination (the device).

By embedding ordering information within the transactions themselves, intermediate links—including the PCIe fabric and the Root Complex—no longer need to enforce strict ordering. This allows the Root Complex to aggressively forward PCIe reads and writes without serialization, significantly increasing interconnect utilization and performance.

6 Evaluation

In this section, we assess the performance benefits of our proposed architectural and microarchitectural enhancements for fast remote memory ordering using a two-pronged approach. The first relies on simulation, and the second emulates remote ordering performance on existing NVIDIA ConnectX NICs to understand what the gains might be in practice.

We first describe the simulation infrastructure and benchmarks used for evaluation. Next, we present the overall performance improvement achieved by our complete approach, incorporating PCIe extensions, ISA modifications, and the optimized RLSQ design compared to today's baseline techniques that rely on store fences for W→W MMIO ordering and implicit serialization for R→R DMA. Finally, we provide an estimate of the area and static power overheads of the RLSQ and ROB.

6.1 Simulation Infrastructure

Our proposed designs are simulated on gem5 [5] using the classic cache model. For the DMA experiments, we use a SimpleTimingCPU model, as these operations are device-initiated; this focuses the evaluation on the I/O path performance rather than the detailed microarchitecture of the host core.

Table 2 lists the simulation configuration for DMA read experiments. We modeled a baseline Root Complex described in prior art [10, 32]. In particular, tracker entries are used to track requests that access the same cache line. Our RLSQ model is integrated into this Root Complex. In gem5, DMA requests are split into 64 B packets on the I/O bus. We use a one-way I/O bus latency of 200 ns, estimated from the 600 ns round-trip latency for DMA reads reported in prior work [27]

Table 2. Simulation Configurations for DMA Experiments

Processor	
Core	1 SimpleTimingCPU, 3 GHz
Cache Hierarchy	
L1 Instruction	16 KiB, 2-way associative, 2 cycle latency
L1 Data	64 KiB, 2-way associative, 2 cycle latency
L1 to L2 bus	256-bit wide, 1 cycle latency
L2	256 KiB, 8-way associative, 20 cycle latency
Memory	
Memory bus	128-bit wide, 7 cycle latency
Memory Interface	DDR3-1600 in 8x8 configuration
Bandwidth	8 channels, 12.8 GB/s per channel
I/O System	
I/O bus	128-bit wide, 200 ns latency
Root Complex	17 ns latency, 256 tracker entries
RLSQ	256 entries
NIC	3 ns DMA request issue latency

Table 3. Simulation Configurations for MMIO Experiments

Processor	
Core	1 O3CPU, 3 GHz
Cache Hierarchy	Same as configuration as Table 2
Memory	Same as configuration as Table 2
I/O System	
I/O bus	128-bit wide, 200 ns latency
Root Complex	60 ns latency, 16 entry buffer
NIC	10 ns MMIO processing latency

Table 3 lists the simulation configuration for MMIO write experiments, which use the O3CPU model to accurately capture the performance of core-initiated operations. Posted PCIe writes are modeled with zero latency response packets on the non-coherent crossbar that models the PCIe interconnect. The Root Complex schedules a response packet to the CPU cache controller without delay. Writes without source ordering are modeled by the cache controller acknowledging uncacheable MMIO writes without delay. Fence instructions stall until a response from the root complex is received.

6.2 Benchmarks

We use three main benchmark kernels to demonstrate the benefits of remote ordering.

Ordered DMA Reads: The first is a microbenchmark that simulates clients issuing DMA reads of various sizes (similar to a NIC handling a workload of RDMA READs). We vary the approach the NIC uses to order PCIe reads within each DMA read; this benchmark shows the effectiveness of remote ordering and speculative remote ordering. This microbenchmark is simulated using a NIC that issues DMA read requests from a trace of increasing addresses.

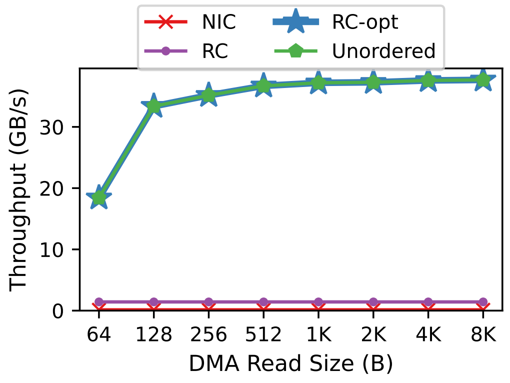


Figure 5. Throughput of DMA reads in simulation using one QP. Speculative ordering achieves ordering at no cost.

Key-value Store (KVS) Gets: RDMA-based KVSs that hold small items in memory accessed by a user-supplied key have become important in recent years [9, 11, 17, 25, 26, 33]. Often the algorithms that get items from a KVS have subtle ordering constraints that result in complexity, stalls, and extra round trips on today's unordered interconnects. We benchmark several KVS *get* implementations showing the benefits of remote ordering for throughput and reducing complexity. We validate these results by comparing our simulation results to implementations of these algorithms on existing NICs. In order to better represent real applications that batch get requests [28], our simulation includes batch size and issue interval parameters. We use batch sizes of 100 and 500 requests with an inter-batch issue interval of 1 µs based on the `halo3d` and `sweep3d` communication patterns [15].

NIC Packet Transmission: Ethernet link speeds of 100 Gb/s and higher have made it crucial for software to be able to coordinate the transfer of packet data to NICs efficiently [30]. We show that our proposed Release ordering for MMIO makes MMIO efficient enough that software may be able to directly transfer packet data while preserving correct ordering without costly `sfence` instructions that prevent this today. In simulation, cache line sized MMIO writes are modeled by writes to addresses that are one cache line apart. MMIO writes are issued to increasing addresses to simulate writes with increasing sequence numbers. The simulated NIC checks if the write packets arrive in the correct order.

6.3 Simulation Results

Ordered Reads. We simulate a single thread of execution on a NIC performing reads of varying length sequential regions. Figure 5 shows the results. Today, when a NIC issues a DMA read, each cache line in the DMA is read in an arbitrary order, making it efficient enough to saturate a 100 Gb/s network link even with 64 B granularity accesses (*Unordered*). If an application requires cache lines to be read in a specific order (e.g., lowest-to-highest address), then a NIC would need to execute each cache line access synchronously (*NIC*), destroying throughput both for small and large accesses since the number of stalls is proportional to the number of cache lines read. Delegating this responsibility to the Root Complex (*RC*) reduces the length of these stalls, but it still prevents the read bandwidth from scaling. However, speculative ordering (*RC-opt*) ensures that ordered reads can be pipelined to the host coherence subsystem without stalling, allowing ordered read performance to match that of unordered accesses.

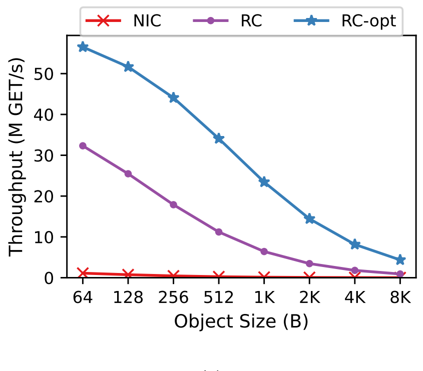
Key-Value Stores. Next, we benchmark RDMA-based key-value store *get* operations. In our benchmarks, we vary the approach we use to order the PCIe reads triggered by the RDMA READs as part of these get operations, comparing the performance of reads that are ordered by the NIC, remotely by the Root Complex (*RC*), and with speculative ordering at the Root Complex (*RC-opt*).

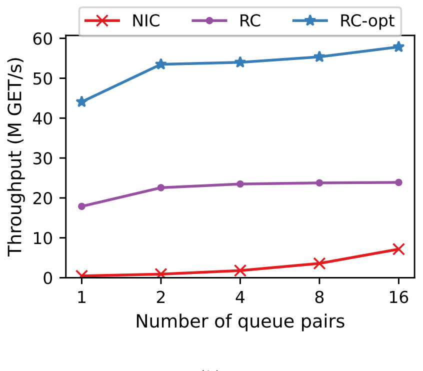
In our benchmark, we use the optimistic, validation-based get algorithm described by Jasny et. al. [16]. In this approach, an item get operation is performed at a server NIC using two RDMA READs. In the first RDMA READ, a client reads an item header version number and the item. After receiving the results from the first RDMA READ the client issues a second RDMA READ that fetches a second copy of the item header version number. If the version numbers match, the read item is returned to the caller. If the version numbers do not match, the protocol repeats.

This protocol is unsafe today because PCIe reads are unordered within an RDMA read; in the first RDMA READ it is possible that the cache line containing the header version number is read after the cache lines containing the data item. Even if an ordered writer correctly updated the data item between its updates to the header, the reader could see stale or torn reads. With our proposed read ordering, the unmodified protocol works correctly.

Figure 6a shows the results when a single client (one QP) issues batches of 100 gets. All operations within a batch are executed according to their order in the batch. As with our ordered DMA reads benchmark, using the NIC to order reads results in a 440 ns synchronous stall between the NIC and the CPU coherence subsystem to fetch each cache line separately; this results in poor performance, and it cannot be amortized across larger items. Ordering at the RC reduces the stall between fetching cache lines substantially improving performance by 29.1× over NIC ordered reads. RC-Opt's speculative ordering allows all cache line reads within each request and all requests themselves to be pipelined to the memory system, incurring stalls only between whole batches of requests making it 50.9× faster than NIC ordering.

Figure 6b shows that even as more clients/QPs offer requests to the server, these gains hold. Increasing client count helps NIC-based ordering the most since it can overlap the PCIe reads of up to 16 get operations at a time. However, the





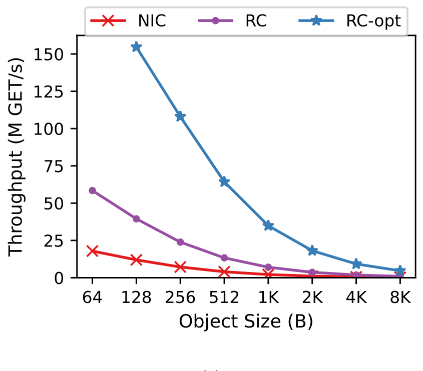


Figure 6. Key-Value Get Throughput. (a) uses a single client QP submitting batches of 100 get requests; (b) scales the workload across multiple QPs/clients; (c) uses 16 QPs/clients each submitting batches of 500 get requests. Across all these configurations RC-opt shows robust performance gains.

CloudLab sm110p (one as client, one as server)	
CPU	1× Intel Xeon Silver 4314 16-cores @ 2.40 GHz (Ice Lake)
RAM	128 GB ECC DDR4-3200
NIC	NVIDIA ConnectX-6 Dx EN 100 Gb/s PCIe 4.0 ×16

Table 4. Hardware Setup for Emulation Experiments (Figure 7).

increased parallelism is not enough for NIC ordering to make up for its long stalls. In practice, the NVIDIA ConnectX-6 Dx NICs that we have benchmarked do not scale performance substantially beyond 16 QPs for deeply pipelined RDMA READs, suggesting that NIC-based ordering is unlikely to ever converge to RC's performance even with an unbounded number of clients.

Finally, since RC-Opt can fully pipeline memory operations for requests within a batch, Figure 6c shows that if clients use larger batch sizes and offer more concurrency, speculative remote ordering becomes crucial for scaling throughput. For small object sizes—the most challenging case for interconnect overhead—RC-opt is the only approach that maintains correctness of the protocol while approaching the 100 Gb/s link that modern NICs support.

6.4 Emulation on Existing NICs

To validate our simulation and gain further confidence in our results, we emulate our architecture's expected best-case performance using real hardware. Our simulation results (§6.3) show that in read-only workloads where there are no conflicts, the performance of our speculative ordering mechanism is identical to that of today's fully unordered hardware. This key finding allows us to use existing NVIDIA ConnectX-6 Dx NICs as a realistic performance proxy to emulate our proposed architecture. Even though these NICs do not enforce order, their unordered throughput represents the performance our ordered speculative design can achieve. We validate this assumption in Section 6.5, where we show

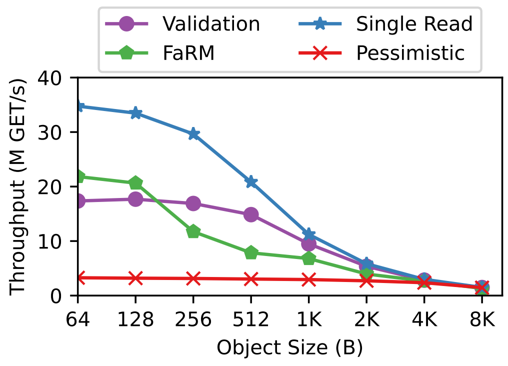


Figure 7. Throughput of gets on a key-value store implemented via RDMA using various algorithms on a 100 Gb/s NVIDIA ConnectX-6 Dx. Validation and Single Read require R→R ordering for correctness. Our proposed Single Read algorithm, enabled by hardware ordering, is simpler and outperforms all baselines, including a 1.6× gain over FaRM for 64 B objects.

that our simulation results closely match the performance measured on the real hardware (Table 4).

A recent paper reimplemented several approaches to fetching and synchronizing access to remote records in a KVS [16]. Their work compares numerous state-of-the-art approaches, groups common approaches (e.g., optimistic protocols versus pessimistic, locking-based approaches), and ensures that the uniform platform and benchmark harness illustrate differences in the algorithms rather than the implementations. Building on their harness, we repeat their experiments here on our hardware, and we extend the results to include an algorithm that is only safe with a NIC and an interconnect that supports ordered reads.

In the implementation of all of these algorithms we batch 32 concurrent get operations before polling for operation completion at the client to ensure each of the 16 client threads offers substantial concurrency to the server NIC and to amortize polling costs at the client.

Pessimistic [16]: Pessimistic protocols are prevalent in several state-of-the-art RDMA-enabled KVSs [16, 23, 37]. In this protocol, a client pipelines an RDMA fetch-and-add to increment the reader count for a key-value item and an RDMA READ for the item to the server's NIC, which also includes a lock bit for the writer. If the lock bit is set by the writer, then the operation is restarted. Otherwise, the client asynchronously issues another RDMA fetch-and-add to decrement the reader count, and it makes the read value available to the caller of the get operation.
Optimistic with Validation [26]: We described this algorithm in §6.3. It uses two RDMA READs per get request; it requires R⟶R order for correctness. Though our system does not actually order PCIe reads, based on our simulation results, in cases like this where there are no writes the results should be representative.
FaRM [11]: FaRM and XStore [35] get operations fetch key-value items using a single RDMA READ. In this approach, each item contains a header with a version number, and each cache line that is part of the item also includes (part of) the item version number. Writers first update the header version number and then update each cache line with the new data and the new version number embedded in it. Clients issue a single RDMA READ for a whole item. If the version number in the header matches the version number in every cache line, then the data in the cache lines is safe to use, and it is returned to the caller. If there is a mismatch, the RDMA READ is repeated. The version numbers in cache lines ensure the protocol is correct even if PCIe reads within the RDMA READ are reordered.
Single Read: Single Read is a simpler protocol that previously was not possible due to PCIe read reordering. In it, each item includes a header version and a footer version. Clients issue a single RDMA READ for a whole item including its version numbers. If the two version numbers match, the item is returned to the caller; if the version numbers do not match, the RDMA READ is repeated. Unlike FaRM, this protocol does not require any version information embedded in the individual cache lines of each of the items. For correctness, writers must work from back to front, first updating the footer version number, then the item data, then the header version number. This avoids a race where a reader and writer could interleave in an order where the reader sees the updated header, stale data, and an updated footer. Some past systems have used a similar approach, though they were incorrect since they implicitly relied on DMA read ordering within each RDMA READ [16, 26, 34, 38].

Figure 7 shows the get throughput of each of these algorithms for a range of item sizes. At sizes less than 4 KB, Pessimistic suffers from the high overhead of the RDMA

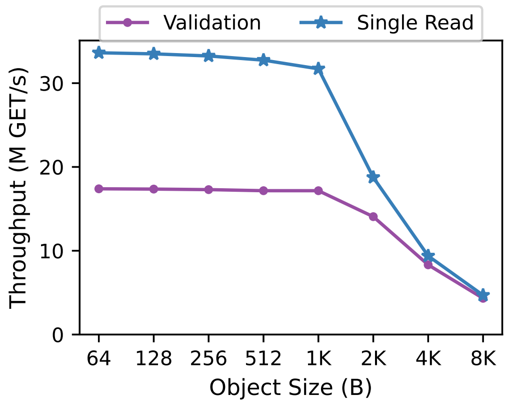


Figure 8. Throughput of get operations in simulation using both Single Read and Validation with 16 queue pairs and batch size 32.

fetch-and-add needed to lock access to each item. Validation's protocol only relies on RDMA READs, and it performs much better. For example, with 512 B items it is able to transfer more than 60 Gb/s, letting applications use most of the 100 Gb/s link between the client and server. However, a correct implementation of Validation would only be able to perform this well using the remote ordering that we describe in this paper. Single Read substantially improves on Validation at every item size with about double the throughput with small items, and it uses less network bandwidth.

Like with Single Read, FaRM get operations are processed with a single RDMA READ. However, since FaRM embeds ordering metadata within items, clients must strip metadata out of items before returning items to most applications. Effectively, FaRM requires an extra deserialization step at clients that requires copying item data into a contiguous buffer. At the high > 10 GB/s data rates of modern NICs this becomes a substantial overhead. In our setup this additional copying limits FaRM get throughput to less than what Validation can achieve for all but the smallest item sizes. Relying on remote ordering allows Single Read to avoid the need for this extra metadata and copying without hurting efficiency.

Overall, these results show that remote ordering may enable simpler algorithms that beat state-of-the-art performance for important applications. We simplified the descriptions of each algorithm to ignore how they safely coordinate concurrent writes to items. Each individual paper provides full details; however, this can generally be done in a straightforward manner for each of the algorithms, e.g., by having writers perform a compare-and-swap on the version number.

6.5 Cross-Validating Simulation and Emulation

This final step performs a crucial cross-validation. First, it validates that our simulation infrastructure is well-calibrated by showing it can accurately model the performance of real hardware. Second, it validates that our hardware emulation is a meaningful proxy for our proposed architecture's potential.

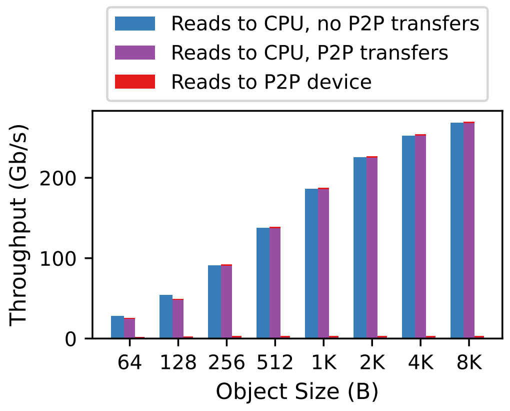


Figure 9. For each request size, the left bar represents the baseline with no P2P transfers, the middle bar represents *P2P-VOQ*, the right bar represents *P2P-noVOQ*.

To do this, we repeated the Validation and Single Read benchmarks in our simulation, configuring them to match the real NIC's behavior of serially issuing RDMA READs from each QP. The results show a strong correlation: the simulated throughput in Figure 8 closely tracks the real hardware performance from Figure 7. The curves diverge only when limited by different bottlenecks—the wider PCIe bus in simulation versus the narrower Ethernet link on the real hardware. This close match gives us high confidence in both our simulation's predictions and our emulation's findings, suggesting the benefits of our architecture will hold on real systems.

6.6 Peer-to-Peer Experiments

The previous sections studied ordered read requests issued by a single NIC to a single CPU root complex; however, systems can have topologies involving multiple destination devices. One example is peer-to-peer (P2P) transfers in PCIe [14]. This section discusses how read requests originating from the same source device that target different destination devices can be handled. There are two cases to consider:

Case 1: Requests originate from the same process and R→R order is required. An example of such a scenario would be a client application running at a device (e.g. a NIC) that needs to read data from GPU memory after reading some synchronization variable from CPU memory. Even if read requests are ordered by the interconnect, we must guarantee that a read request R_2 at device D_2 will be handled only after a read request R_1 at device D_1 has been handled. In this case, we revert to ordering at the source NIC by issuing R_2 to D_2 only after receiving the completion for R_1 issued to D_1.

Case 2: Requests originate from different processes and R→R order is not required. An example of this case would be different clients on a single machine reading from two different devices. Since these read requests originate from distinct clients, ordering is not required.

If strict ordering is enforced by the interconnect, or if network resources are naively shared, a slow device can penalize a fast destination. Specifically, suppose application A_1 sends read request R_1 to device D_1 and application A_2 sends read request R_2 to device D_2. If D_1 is congested, requests targeting it may fill the interconnect buffers. If the switch uses a single shared queue, request R_2 (targeting the uncongested D_2) gets stuck behind R_1. This is Head-of-Line (HOL) blocking. We propose using Virtual Output Queues (VOQs) [24] to isolate these flows and prevent such degradation.

To quantify the impact of HOL blocking and the efficacy of VOQs, we simulated three system configurations:

- RC-opt (Baseline): A standard system where the source device, a NIC, issues ordered reads to the CPU.
- P2P-VOQ: Adds a congested P2P device to the baseline. The NIC connects to the CPU and the congested device via a crossbar switch equipped with VOQs (separate queues for each destination).
- P2P-noVOQ: Adds the same congested P2P device, but the crossbar switch uses a single shared 32-entry queue for all incoming requests, regardless of destination.

We used the Single Read protocol (§6.4) with two distinct threads running on the NIC:

- Thread A (CPU Flow): Issues batches of 100 requests to the CPU with a 1 µs inter-batch interval.
- Thread B (P2P Flow): Issues requests to the P2P device. To simulate congestion, the P2P device is modeled with a service time of 100 ns per request and an input limit of one request at a time. Thread B attempts to issue requests at the same rate as Thread A but without the inter-batch delay, ensuring the P2P device is constantly saturated.

In the P2P-noVOQ configuration, the slow consumption rate of the P2P device causes the shared 32-entry switch queue to fill rapidly. Once full, the switch rejects new requests from both threads. The NIC handles this backpressure using a round-robin scheduler to retry failed requests. Hence, the high-speed CPU flow is throttled to match the drain rate of the slow P2P flow, as it must wait for free buffer space.

Figure 9 illustrates the results. In the P2P-noVOQ scenario, the shared queue causes significant performance degradation: read throughput to the CPU drops by up to 167× for 8192 B objects compared to the baseline. In contrast, the P2P-VOQ setup successfully isolates the flows. Throughput is restored to near-baseline levels across all object sizes.

6.7 Ordered MMIO Writes

Section 2.2 showed that enforcing W→W MMIO ordering in CPU-NIC transmit paths via `sfence` significantly hurts throughput compared to unordered MMIO stores. Figure 4 illustrates this performance degradation observed on an NVIDIA ConnectX-6 Dx NIC. To validate this observation, we replicated the experiment in our simulator. The simulator

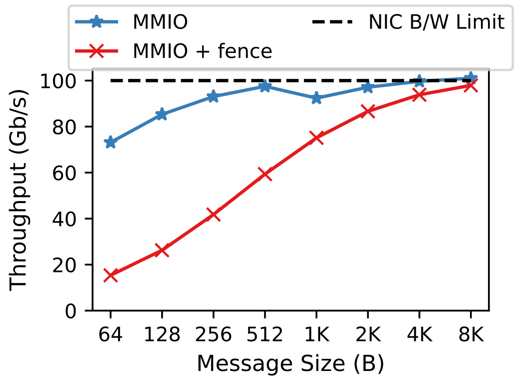


Figure 10. MMIO write throughput in simulation. Inserting a fence after every message enforces message order at the source.

Table 5. Estimate of Hardware Area

	Area in mm^2	% of I/O Hub
RLSQ	0.9693	0.6853
ROB	0.2330	0.1647
I/O Hub [10]	141.44	100

Table 6. Estimate of Static Power

	Static Power in mW	% of I/O Hub
RLSQ	49.2018	0.4920
ROB	4.8092	0.0481
I/O Hub [10]	10000	100

configuration is summarized in Table 3. Figure 10 shows similar trends to the results on the NIC. MMIO write throughput drops significantly when fences are used to ensure ordering.

6.8 Hardware Area and Static Power Overhead

We estimate the area required and the static power consumption for the RLSQ and ROB using CACTI [4]. The RLSQ and ROB are modeled as caches with 64 B blocks.

The RLSQ is modeled as a 256 block, fully-associative cache with 1 read port, 1 write port and 1 search port. The RLSQ is fully associative so that speculative loads can be invalidated when invalidation messages are received. The ROB is modeled as a 32 block, direct-mapped cache (indexed by the sequence number) with 1 read port and 1 write port. The ROB uses 32 blocks to implement separate virtual networks for relaxed stores and release stores, each with 16 entries.

For comparison against the reported die area and idle power consumption of the Intel I/O Hub design [10], the CACTI models for the RLSQ and the ROB both use the same 65 nm process technology.

The hardware area estimate is presented in Table 5 and static power estimate in Table 6. Overall, adding the RLSQ and ROB to the I/O Hub would increase chip area by less than 0.9% and increase static power by less than 0.6%.

7 Related Work

Efficient Memory Consistency Models (MCMs). Efficient enforcement of MCMs has been the subject of extensive research. Our proposed interface builds on Release Consistency [13], which forms the foundation of prevalent modern MCMs found in architectures like ARM, RISC-V [3], and even in language-level specifications [18]. A key insight of our work is that there is no fundamental reason for high-performance interconnects like PCIe to operate with ordering semantics divorced from the well-established concepts and best practices developed within the MCM community.

Our techniques for enforcing memory ordering draw inspiration from classic works on MCM enforcement in processor pipelines. For example, our approach to speculative memory ordering is inspired by seminal work such as [12]. However, there are crucial differences between a Remote Load-Store Queue (RLSQ) and a traditional processor-centric Load-Store Queue (LSQ). By definition, a processor LSQ is local to a single core, whereas the RLSQ is shared by multiple independent thread contexts within the connected device. This fundamental difference underscores the criticality of incorporating our thread-context-based ordering optimization in the RLSQ. Furthermore, the invalidation-based tracking for speculative reads in the RLSQ presents another subtle difference as the RLSQ itself does not maintain a coherent cache. Despite these differences, our core contribution lies in identifying the significant ordering challenges in contemporary high-speed interconnects, formalizing this problem within the well-understood framework of memory consistency models, and consequently enabling the adaptation and application of these classic enforcement techniques to a new and increasingly important domain.

Non-coherent interconnects. While this work focuses on PCIe, our destination-based ordering concepts are equally applicable to other non-coherent fabrics. Here, we focus on two publicly available non-coherent interconnects. *CXL.io* [8] explicitly inherits PCIe's ordering rules, so our analysis transfers directly. In the case of *AMBA AXI* [2], another widely deployed non-coherent interconnect, the performance implications are even more significant. AXI does not guarantee ordering between transactions to different memory addresses—even if they share the same Transaction ID. Consequently, to implement a reliable Read→Read ordering (e.g., reading data then a flag), current AXI systems must enforce strict source-side serialization, waiting for the first response before issuing the second. Our proposed release/acquire semantics overcome this limitation by attaching explicit ordering attributes to requests; this allows the source to pipeline ordered reads efficiently, relying on the destination to enforce order locally via the RLSQ mechanism.

Coherent Interconnects. Recent research has explored cache-coherent I/O interconnects [30, 31] to achieve high bandwidth, low-latency CPU-NIC communication. CC-NIC

[31], a cache-coherent NIC interface, considered an MMIO-based PCIe baseline but dismissed it due to the perceived cost of ordering fences. Given MMIO's capability to achieve line rate and low latency without compromising ordering (through our proposed mechanisms), there is little need for coherence in CPU-NIC communication patterns. Instead, coherence protocols like CXL (a MESI variant) can hinder efficient CPU-NIC producer-consumer communication. These protocols typically require obtaining ownership and transitioning to an exclusive state on writes, so a CPU producer write pulls the cache block into an exclusive state. A subsequent consumer read by the NIC necessitates an indirection—the data must be fetched from the CPU's caches, thus increasing latency. Creative optimizations in software [31] or the protocol [30] can work around these inefficiencies, but we contend that these optimizations are effectively striving to achieve what our PCIe-based transmit path inherently provides: high throughput and low latency without the need for complex workarounds layered on top of coherence protocols. Our earlier work made a similar observation for MMIO writes and NIC transmission paths [22], yet it stopped short of proposing a holistic ordering framework for non-coherent interconnects.

Other Devices. While our optimizations target CPU-NIC interaction, the core concept of destination-based ordering also applies to other devices such as GPUs and FPGAs. Yu et al. [36] recently proposed a directory-based ordering mechanism tailored for efficient store→release ordering in CPU-GPU systems; they also propose a mechanism for efficiently enforcing ordering across multiple devices. In contrast, our approach introduces techniques to enforce both store→release and acquire→load orderings. Future work could integrate these ideas—CORD's distributed coordination and our bidirectional ordering support—to create a unified ordering framework for heterogeneous systems.

8 Conclusion

The PCIe specification has been a cornerstone of system interconnects for decades. However, its original design catered to an era of slow I/O devices optimized for bulk data transfers. The rise of heterogeneous computing and high-bandwidth, low-latency interconnects has fundamentally transformed the nature of host-device communication. In this context, we have identified crucial inefficiencies in how current systems order remote memory operations. To address these challenges, we have proposed a new PCIe interface designed to integrate effectively with contemporary host memory consistency models. Additionally, we have introduced novel and efficient microarchitectural techniques for enforcing remote memory ordering, drawing inspiration from established methods used to enforce memory consistency models in host processors. Ultimately, this work argues for a fundamental shift in interconnect design: treating ordering not as an implicit property of the fabric, but as an explicit, first-class semantic co-designed with the ISA, enabling fast and correct I/O for future systems. By establishing a high-performance baseline for non-coherent I/O, this work raises the question of whether the complexity of coherent interconnects (like CXL) is truly necessary for future host-device communication.

Acknowledgments

We thank the reviewers, and particularly our shepherd, Haggai Eran, for their detailed and helpful feedback. This material is based upon work supported by the National Science Foundation under Grant Nos. CCF-2525270 and CNS-2245999. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. This work was also supported in part by Arm, Google, VMware, Meta.

References

[1] 2016. *Semantics of MMIO mapping attributes across architectures.* Available at https://lwn.net/Articles/698014/.

[2] 2023. *AMBA® AXI Protocol Specification.* Technical Report ARM IHI 0022 Issue K. Arm Limited.

[3] 2025. The RISC-V Instruction Set Manual: Volume I: Unprivileged Architecture.

[4] Rajeev Balasubramonian, Andrew B. Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories. *ACM Trans. Archit. Code Optim.* 14, 2, Article 14 (June 2017), 25 pages. doi:10.1145/3085572

[5] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011), 1–7. doi:10.1145/2024716.2024718

[6] Matthew Burke, Sowmya Dharanipragada, Shannon Joyner, Adriana Szekeres, Jacob Nelson, Irene Zhang, and Dan R. K. Ports. 2021. PRISM: Rethinking the RDMA Interface for Distributed Systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 228–242. doi:10.1145/3477132.3483587

[7] Harold W Cain and Mikko H Lipasti. 2004. Memory ordering: A value-based approach. *ACM SIGARCH Computer Architecture News* 32, 2 (2004), 90.

[8] Compute Express Link Consortium, Inc. 2023. *Compute Express Link (CXL) Specification.* Compute Express Link Consortium, Inc. Available at https://www.computeexpresslink.org/download-the-specification.

[9] Mahesh Dananjaya, Vasilis Gavrielatos, Antonios Katsarakis, Nikos Ntarmos, and Vijay Nagarajan. 2025. Fast, Highly Available, and Recoverable Transactions on Disaggregated Data Stores. (2025).

[10] Debendra Das Sharma. 2009. Intel® 5520 chipset: An I/O hub chipset for server, workstation, and high end desktop . In *2009 IEEE Hot Chips 21 Symposium (HCS)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–18. doi:10.1109/HOTCHIPS.2009.7478355

[11] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. 2014. FaRM: fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (Seattle, WA) *(NSDI'14)*. USENIX Association, USA, 401–414.

[12] Kourosh Gharachorloo, Anoop Gupta, and John L Hennessy. 1991. *Two techniques to enhance the performance of memory consistency models*. Computer Systems Laboratory, Stanford University.

[13] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. 1998. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *25 Years of the International Symposia on Computer Architecture (Selected Papers)* (Barcelona, Spain) *(ISCA '98)*. Association for Computing Machinery, New York, NY, USA, 376–387. doi:10.1145/285930.285997

[14] Peripheral Component Interconnect Special Interest Group. 2017. *PCI Express® Base Specification Revision 4.0*. Available at https://pcisig.com/specifications/pciexpress/.

[15] Simon David Hammond, Karl Scott Hemmert, Michael J Levenhagen, Arun F Rodrigues, and Gwendolyn Renae Voskuilen. 2015. *Ember: Reference Communication Patterns for Exascale*. Technical Report. Sandia National Lab.(SNL-NM), Albuquerque, NM (United States).

[16] Matthias Jasny, Tobias Ziegler, Jacob Nelson-Slivon, Viktor Leis, and Carsten Binnig. 2025. Synchronizing Disaggregated Data Structures with One-Sided RDMA: Pitfalls, Experiments and Design Guidelines. *ACM Trans. Database Syst.* 50, 1, Article 4 (March 2025), 40 pages. doi:10.1145/3716377

[17] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2014. Using RDMA efficiently for key-value services. *SIGCOMM Comput. Commun. Rev.* 44, 4 (Aug. 2014), 295–306. doi:10.1145/2740070.2626299

[18] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing sequential consistency in C/C++11. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 618–632. doi:10.1145/3062341.3062352

[19] Steen Larsen and Ben Lee. 2011. Platform io dma transaction acceleration. *CACHES. ACM* (2011).

[20] Steen Larsen and Ben Lee. 2014. Chapter Two - Survey on System I/O Hardware Transactions and Impact on Latency, Throughput, and Other Factors. Advances in Computers, Vol. 92. Elsevier, 67–104. doi:10.1016/B978-0-12-420232-0.00002-7

[21] Steen Larsen, Ben Lee, et al. 2015. Reevaluation of programmed I/O with write-combining buffers to improve I/O performance on cluster systems.. In *NAS*. 345–346.

[22] Wei Siew Liew, Md Ashfaqur Rahaman, James McMahon, Ryan Stutsman, and Vijay Nagarajan. 2025. Stop Taking the Scenic Route: the Shortest Distance Between the CPU and the NIC is MMIO. In *Proceedings of the 2025 Workshop on Hot Topics in Operating Systems* (Banff, AB, Canada) *(HotOS '25)*. Association for Computing Machinery, New York, NY, USA, 144–150. doi:10.1145/3713082.3730389

[23] Haodi Lu, Haikun Liu, Yujian Zhang, Zhuohui Duan, Xiaofei Liao, Hai Jin, and Yu Zhang. 2025. Fast Distributed Transactions for RDMA-based Disaggregated Memory. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*. 943–958.

[24] Nick McKeown, Adisak Mekkittikul, Venkat Anantharam, and Jean C. Walrand. 1999. Achieving 100% throughput in an input-queued switch. *IEEE Trans. Commun.* 47, 8 (1999), 1260–1267. doi:10.1109/26.780463

[25] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. USENIX Association, San Jose, CA, 103–114. https://www.usenix.org/conference/atc13/technical-sessions/presentation/mitchell

[26] Christopher Mitchell, Kate Montgomery, Lamont Nelson, Siddhartha Sen, and Jinyang Li. 2016. Balancing CPU and Network in the Cell Distributed B-Tree Store. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO, 451–464. https://www.usenix.org/conference/atc16/technical-sessions/presentation/mitchell

[27] 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 Conference of the ACM Special Interest Group on Data Communication* (Budapest, Hungary) *(SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 327–341. doi:10.1145/3230543.3230560

[28] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. 2013. Scaling Memcache at Facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX Association, Lombard, IL, 385–398. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala

[29] Waleed Reda, Marco Canini, Dejan Kostić, and Simon Peter. 2022. RDMA is Turing complete, we just did not know it yet!. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 71–85. https://www.usenix.org/conference/nsdi22/presentation/reda

[30] 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 *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 1005–1025. https://www.usenix.org/conference/osdi23/presentation/sadok

[31] 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, Volume 1* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 52–68. doi:10.1145/3617232.3624868

[32] Gurbir Singh, Robert Safranek, Nilesh Bhagat, Rob Blankenship, Ken Creta, Debendra Das Sharma, David L Hill, David Johnson, and Robert A Maddox. 2010. The Feeding Of High-Performance Processor Cores–QuickPath Interconnects and the New I/O Hubs. *Intel Technology Journal* 14, 3 (2010).

[33] Arjun Singhvi, Aditya Akella, Maggie Anderson, Rob Cauble, Harshad Deshmukh, Dan Gibson, Milo MK Martin, Amanda Strominger, Thomas F Wenisch, and Amin Vahdat. 2021. CliqueMap: productionizing an RMA-based distributed caching system. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 93–105.

[34] Qing Wang, Youyou Lu, and Jiwu Shu. 2022. Sherman: A Write-Optimized Distributed B+Tree Index on Disaggregated Memory. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1033–1048. doi:10.1145/3514221.3517824

[35] Xingda Wei, Rong Chen, Haibo Chen, and Binyu Zang. 2021. XStore: Fast RDMA-Based Ordered Key-Value Store Using Remote Learned Cache. *ACM Transactions on Storage (TOS)* 17, 3 (2021), 1–32.

[36] Yanpeng Yu, Nicolai Oswald, and Anurag Khandelwal. 2025. CORD: Low-Latency, Bandwidth-Efficient and Scalable Release Consistency via Directory Ordering. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25)*. Association for Computing Machinery, New York, NY, USA, 1311–1326. doi:10.1145/3695053.3731074

[37] Ming Zhang, Yu Hua, Pengfei Zuo, and Lurong Liu. 2022. FORD: Fast One-sided RDMA-based Distributed Transactions for Disaggregated Persistent Memory. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. USENIX Association, Santa Clara, CA, 51–68. https://www.usenix.org/conference/fast22/presentation/zhang-ming

[38] Tobias Ziegler, Sumukha Tumkur Vani, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. Designing Distributed Tree-based Index Structures for Fast RDMA-capable Networks. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 741–758. doi:10.1145/3299869.3300081