

Exposing RDMA Read Reordering on x86: An Empirical Litmus Test

1st Alla Lenchner
Software Architecture
NVIDIA Corporation
Yokneam, Israel

2nd Hagai Eran
Software Architecture
NVIDIA Corporation
Yokneam, Israel

3rd J.J. Vegas Olmos
Software Architecture
NVIDIA Corporation
Yokneam, Israel

4th Stefan Schmid
EECS (Faculty IV)
Technical University Berlin
Berlin, Germany

Abstract—RDMA has become a fabric of choice for AI training, where high-throughput, low-latency communication is essential. By allowing remote memory access while bypassing the CPU, RDMA improves performance but complicates reasoning about memory ordering. Defining precise memory-ordering semantics for RDMA remains challenging, because RDMA operations interact with the memory system through hardware paths that are not fully captured by conventional CPU-centric models.

Recent formal models of RDMA have proposed rich sets of possible execution outcomes. However, some of these outcomes have not been observed on real hardware, raising the question of whether they are physically realizable or merely theoretical artifacts.

In this work, we focus on one such case: the weak outcome of the message-passing litmus test. Unlike x86-TSO, where loads issued by the same thread are ordered, RDMA can allow concurrent remote reads to complete out of order. As a result, an RDMA reader may observe a synchronization flag as updated while still observing stale data.

We reproduce this outcome on real hardware and show that it arises from the NIC's interaction with the memory system through the PCIe interconnect. Its manifestation is sensitive to specific microarchitectural and system-level conditions.

This work offers an initial step toward empirically grounding RDMA memory-consistency models, suggesting that careful microarchitectural tuning can help bridge the gap between formally allowed outcomes and real-system behavior.

Index Terms—Program semantics, RDMA, x86-TSO, ccNUMA

I. INTRODUCTION

RDMA is widely used in high-performance distributed systems and has become a key building block for modern AI training workloads, where high-throughput, low-latency communication is essential. By enabling direct memory access across machines while bypassing the CPU, RDMA significantly improves performance, but it also complicates reasoning about correctness. In particular, the correctness of RDMA-based programs depends on understanding what memory-

ordering guarantees the hardware actually provides. Without a precise model, it is unclear which outcomes are possible, which can safely be assumed away, and where synchronization is required.

Recent work has made progress toward formalizing RDMA semantics. In particular, the RDMA-TSO model [1] provides both operational and declarative accounts of RDMA behavior on x86-TSO systems. A key insight of this line of work is that modeling RDMA requires rethinking the notion of a node: rather than treating a node as an entire multicore machine with a shared memory system, RDMA-TSO defines a node as a thread paired with its local memory. When communication crosses node boundaries, RDMA operations are executed by the NIC and traverse the end-to-end I/O path, including the RNICs, PCIe fabric, and root complex. Consequently, they are not constrained in the same way as CPU-issued memory operations.

This distinction gives rise to behaviors that do not occur in standard x86-TSO executions. In x86-TSO, loads issued by the same thread are observed in program order, and stores from a single thread are propagated in FIFO order. Therefore, in the classic message-passing litmus test, once a receiver observes a synchronization flag written by the sender, it cannot subsequently observe stale data from an earlier sender write. In the RDMA setting, however, the receiver's reads are remote reads issued by the NIC. If these reads are concurrent or overlapping in the I/O subsystem, they may complete out of order. Thus, even if the sender's writes become visible in order, the RDMA reader may observe the flag update while still observing stale data.

While such behaviors are permitted by formal models, it remains unclear whether they arise on real hardware or are merely artifacts of the model. In this work, we focus on the message-passing outcome and study it in practice. Our contribution is to demonstrate the previously predicted read-after-read reordering on real hardware, to show that it arises from subtle interactions between the NIC, the memory hierarchy, and the PCIe interconnect, and to identify techniques that can help reproduce similar weak behaviors in other litmus tests in the future.

Our results provide an initial step toward empirically

The authors would like to acknowledge the European Commission through the MSCA Doctoral Network QUARC with grant number 101073355, and the German Research Foundation (DFG), SPP 2378: Resilience in Connected Worlds: Mastering Failures, Overload, Attacks, and the Unexpected, project ReNO-2 (511099228), 2025-2029, for partly funding this research.

979-8-3195-4420-9/26/\$31.00 ©2026 IEEE

grounding RDMA memory-consistency models, demonstrating that behaviors predicted by formal semantics can emerge in practice under specific microarchitectural conditions.

II. BACKGROUND

A. RDMA

Remote Direct Memory Access (RDMA) enables a machine to access the memory of another machine directly, bypassing the remote CPU and operating system, and thus providing low latency and high bandwidth communication for distributed systems and HPC workloads. RDMA is now widely used in large-scale data centers and AI training clusters, where communication overhead is a dominant performance bottleneck. While several implementations of RDMA exist, we focus on devices implementing the InfiniBand or RoCE specifications [2].

B. Importance of formal models for correctness

The performance benefits of RDMA can make reasoning about correctness more challenging, because RDMA operations interact with local CPU memory models, NIC behavior, and the interconnect, affecting the ordering and visibility of memory accesses across machines.

Without precise formal models, programmers and tool builders cannot reliably determine which executions are allowed, which weak behaviors must be accounted for, and which synchronization patterns are sufficient to ensure correctness.

Formal memory models have been used in several other domains. For example, Lustig et al. [3] present a formal analysis of the NVIDIA PTX memory consistency model, clarifying the guarantees provided to GPU programmers and supporting systematic reasoning about concurrent GPU code.

C. Formal semantics for RDMA

Several works have proposed formal semantics for RDMA programs and systems. Marian et al. [4] present an axiomatic model for remote memory access programming, assuming sequential consistency for local concurrency. More recently, Ambal et al. [1] introduce RDMA-TSO, a set of operational and declarative models for RDMA on x86-TSO architectures, together with empirical validation on a range of litmus tests.

RDMA-TSO significantly advances our understanding of RDMA semantics, but it still leaves open questions about some weak behaviors that are difficult to observe in practice. In particular, some executions that are allowed by the model have not yet been observed on real hardware, either because they require very specific microarchitectural conditions or because existing validation efforts have not targeted them explicitly.

This gap can make practitioners think that current hardware provides stronger ordering guarantees than the existing models describe. Prior work on one-sided RDMA data structures

has shown that RDMA ordering assumptions affect both the correctness and performance of RDMA applications [5].

D. x86-TSO and RDMA-TSO

x86-TSO (Total Store Ordering) [6] is the memory model underlying modern Intel and AMD processors. It is weaker than sequential consistency because stores may be temporarily buffered before becoming visible to other cores. More specifically, TSO permits only one reordering: a store may be delayed past a subsequent load due to the per-core store buffer. All other orderings—load-load, load-store, and store-store—are preserved. In particular, two reads issued by the same thread in program order are performed in that order.

RDMA changes this picture. Two RDMA read operations issued in program order by the same initiator may observe remote memory in a different order, producing outcomes that would be impossible on a plain TSO machine.

RDMA-TSO [1] captures this distinction by generalizing the unit of reasoning from a *thread* to a **node**. In plain x86-TSO, a thread is a sequence of local reads and writes subject to TSO ordering. In RDMA-TSO, a node is a machine comprising one or more CPU threads, each with its own store buffer, together with a **NIC** that can independently issue and receive remote memory operations. Formally, each thread t is associated with a node $n(t)$, and when thread t on node n issues a remote operation toward node n' , the operation is delegated to the NIC subsystem of n and executed asynchronously—outside the CPU's store buffer and ordering guarantees.

Two new operation types are introduced on top of TSO's local reads and writes:

- **PUT (remote write)**: a thread on node n writes a value to a memory location on remote node n' .
- **GET (remote read)**: a thread on node n reads a memory location on remote node n' .

Both operations are carried out by the RDMA subsystem and involve the NICs on both nodes.

Two GETs issued on the same WQ carry no ordering guarantee between them: the InfiniBand specification [2] does not guarantee that RDMA reads are performed in issue order. This is the root cause of new weak behaviors: two GETs issued in program order may be reordered by the NIC, causing remote memory to be observed out of order, as in the message passing scenario examined in this work.

To restore ordering when needed, the model introduces two fence variants:

- **fence**: a global barrier that blocks the thread on both the CPU and the NIC toward all nodes.
- **rfence(n')**: a directed barrier that blocks the thread only on the NIC toward a specific node n' , leaving other operations unaffected.

These two primitives let programmers selectively restore ordering: fence enforces a global order on all local and remote

operations, while $\text{rfence}(n')$ enforces order only toward a chosen remote node, avoiding unnecessary stalls on unrelated communication.

E. Challenges and Scope of Message Passing scenario violation

The message passing test checks whether one thread can use a flag variable to signal that data is ready, and whether another thread, upon seeing the flag, is guaranteed to also see the data.

$x = y = 0$	
$x := 1$	$a := y$
$y := 1$	$b := x$
$a = 1, b = 0 \times$	

Fig. 1. Message Passing in x86-TSO: The left thread writes x then y — both go into the store buffer in order, and the store buffer flushes in FIFO order. So if $y=1$ is visible to the right thread, then $x=1$ must already be visible too. TSO's store buffer preserves write order, which is precisely what makes message passing safe without extra fences on x86, making $(a,b)=(1,0)$ outcome being forbidden.

In the message-passing scenario in Figure 1, the outcome $a = 1, b = 0$ is forbidden on x86-TSO. Under TSO, each core's store buffer flushes in FIFO order, so writes become globally visible in the same order they were issued. If the right thread observes $y = 1$, then $x = 1$ must already be visible as well, since the left thread wrote x before y . A later write cannot overtake an earlier one from the same thread, so the usual message-passing violation never arises.

In the RDMA scenario in Figure 2, the local behavior of each CPU still follows TSO: the left node writes x and then y , both passing through its store buffer in FIFO order, so their global visibility order is preserved. What changes is how the remote reads are issued and ordered by the RNIC. The right node reads y and x via two remote GETs, and the responder RNIC may issue $\text{GET}(x)$ before $\text{GET}(y)$, even though the program issues them in the opposite order. $\text{GET}(x)$ can therefore reach memory earlier—before the client's write to x has been flushed—and return $x = 0$, while $\text{GET}(y)$ reaches memory later and returns $y = 1$. As a result, the right node can observe $x = 0, y = 1$.

MP4	
$x = y = 0$	
$x := 1$	$a := y^1$
$y := 1$	$b := x^1$
$a = 1, b = 0 \checkmark$	

Fig. 2. MP4, one of the message-passing variants from the RDMA-TSO paper: Under RDMA-TSO the outcome $a = 1, b = 0$ can be allowed. The right node issues $\text{GET}(y)$ then $\text{GET}(x)$ in program order, but the system may reorder them, executing $\text{GET}(x)$ first. $\text{GET}(x)$ then reaches remote memory before the left node has flushed $x=1$, returning $x=0$, while $\text{GET}(y)$ executes later and returns $y=1$. The global visibility ordering that TSO provides locally is no longer guaranteed across nodes.

While RDMA-TSO formally permits the message passing violation ($a = 1, b = 0$), this outcome was not observed in practice across extensive empirical testing — the original paper itself notes this gap. The responder NIC, PCIe and the CPU seemingly preserve read order between two consecutive GETs in common configurations, even though the model does not require it.

This gap between formal permission and empirical observability is precisely the challenge this work addresses. Rather than accepting unobserved outcomes as mere theoretical artifacts, our goal is to identify the microarchitectural conditions — NIC behavior, NUMA (Non uniform memory access) topology, timing, and memory configuration — under which such violations can actually be triggered on real hardware, thereby bridging the gap between the formal model and physical execution.

F. Where reordering can happen

In a typical RDMA deployment over a reliable transport, two remote reads issued in program order can still be observed in the opposite order at the requester, but not because the network fabric reorders packets within a flow.

The RDMA specification requires a responder RNIC to process RDMA operations in the order defined by the transport layer. One may assume this prevents reordering from occurring in the network. However, there could be rare cases where reordering in the network can indeed occur. The requester in this scenario sends two RDMA read operations, the responder NIC receives them and sends two responses. The network can delay and drop some packets, as it is unreliable, so it may drop the first response, and delay the second one. After a timeout, the requester retransmits its two requests, as it assumes both of them have been lost. The responder then sends two additional responses. Assume the first of these arrives at the requester NIC, now we have a race between the two responses to the second request, and if the delayed one arrives first, the requester NIC sees the results of the RDMA reads reordered.

In practice, such reorderings are rare, so we focus on reorderings that may occur within the responder's machine.

Reordering can occur at three points along the path shown in figure 3. First, at the **responder RNIC**, multiple outstanding read requests may be launched toward host memory out of order due to internal scheduling, buffering, or caching, so the RNIC can issue PCIe read TLPs in a different order than it received the higher-level RDMA requests.

Second, along the **PCIe request path**, the responder RNIC sends PCIe memory-read transactions toward the root complex. Because PCIe ordering rules do not require read requests to remain ordered with respect to one another, the corresponding PCIe transactions may reach the root complex in a different order than the RDMA reads were issued.

Third, at the **root complex and in the coherent memory system**, the root complex can issue host-side read transactions

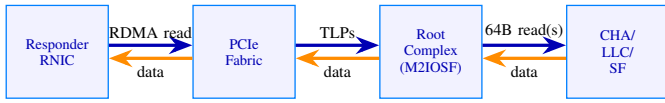


Fig. 3. The Path of an RDMA read

into the coherence fabric in program order, yet the fabric and memory hierarchy may satisfy those reads out of order (for example, one hits in a cache or fast path while another goes to DRAM), so the CPU or a polling NIC can observe Read2's data before Read1's even though the root complex initiated Read1 first.

Contributions

This work makes the following contributions:

- We empirically demonstrate the RDMA message-passing violation (MP4) on real hardware, confirming that a behavior previously only predicted by RDMA-TSO is physically realizable.
- We identify concrete microarchitectural conditions—NIC address-translation caching, PCIe/NUMA placement, and RDMA read noise—that are sufficient to trigger GET/GET reordering in practice.
- We provide a reusable methodology and litmus-test setup for exposing similar weak RDMA behaviors, helping to empirically ground and refine RDMA memory consistency models.

III. DESIGN: CONDITIONS FOR TRIGGERING THE MP4 REORDERING

A. The RDMA Read Path: DDIO, M2IOSF, and the CHA

When the responder NIC receives an RDMA Read, the request travels through several fabric components [7] before the data is returned. Understanding this path is central to understanding how the MP4 reordering arises.

The full RDMA read as seen by the fabric is shown in Figure 4.

The inbound read request is first received by the **M2IOSF** (Mesh-to-I/O Sub-Fabric), which serves as the bridge between the PCIe domain and the processor's mesh interconnect.

Within the M2IOSF, the IRP sub-block performs address translation (if necessary) and divides the transaction into 64-byte cache-line chunks before forwarding each chunk to the appropriate **Cache Home Agent (CHA)**, determined by a hash of the destination physical address [7]. The CHA then looks up the data in the Last Level Cache (LLC) via Intel **DDIO** (Data Direct I/O), which allows inbound I/O requests to access the LLC directly rather than going to main memory [7].

Two read requests targeting different addresses are routed to their respective CHAs independently. There is no ordering guarantee between them at the CHA level—each is serviced according to the cache state at the time of arrival. If one request

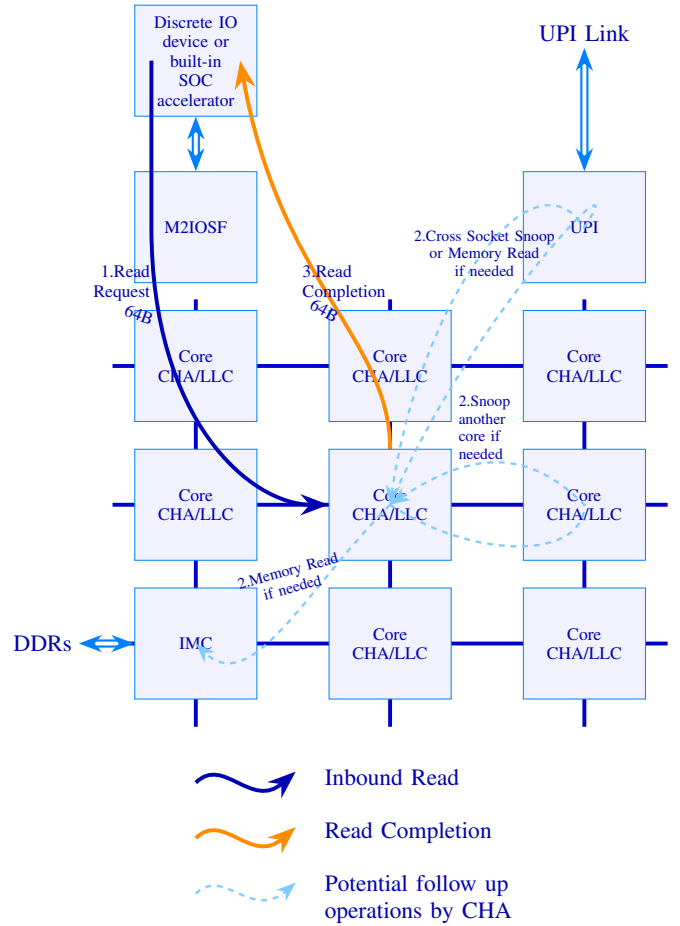


Fig. 4. Inbound Read transaction flow

arrives earlier due to faster address resolution at the M2IOSF, it will be serviced first, regardless of the original issue order.

B. Interleaving that allows reordering

The outcome $(a, b) = (1, 0)$ i.e., $y = 1, x = 0$ can occur only if the operations of the writer and the remote reader are interleaved as:

```

GET(x)           ← reads x = 0
Store(x := 1)
Store(y := 1)
GET(y)           ← reads y = 1

```

We assume the writer CPU is TSO: its stores to x and y become visible in program order, so the write side cannot reorder x and y . Therefore, the only way to obtain $(x = 0, y = 1)$ is for the system to (a) issue and service $\text{GET}(x)$ early enough to see the old value $x = 0$, and (b) delay $\text{GET}(y)$ until after $\text{Store}(y := 1)$ has become visible.

Any other relative ordering of these four operations breaks the outcome:

- If $\text{GET}(x)$ occurs after $\text{Store}(x := 1)$, it returns $x = 1$.

- If $\text{GET}(y)$ occurs before $\text{Store}(y := 1)$, it returns $y = 0$.

In other words, given locally TSO stores, a necessary pattern for the message-passing violation is that the responder NIC and fabric effectively run $\text{GET}(x)$ ahead of $\text{GET}(y)$ and keep $\text{GET}(y)$ delayed until both stores have committed.

Toward this end, we explored several microarchitectural techniques:

- **Responder NIC-level concurrency** Issuing additional parallel RDMA reads and writes through the NIC alongside the test operations, increasing contention within the NIC’s internal queues and PCIe transaction pipeline — creating variability in the order and timing with which requests are forwarded to M2IOSF.
- **NUMA placement** The NIC is attached to NUMA node 0, and we systematically varied thread and data placement: the writer CPU thread ran on either node 0 or node 1, x and y were placed on either node 0 or node 1, and the noise client/server threads were likewise pinned to node 0 or node 1. We always place x and y on different pages (and thus different cache lines) to avoid false sharing. The strongest reordering effects were observed when the writer thread and (x, y) resided on node 1, while the NIC’s work-completion thread executed on node 0.
- **Pre-caching x translation** We issue a warm-up RDMA Read of x before the test so that the responder NIC resolves x ’s virtual address and caches the resulting translation in its on-NIC Memory Translation Table (MTT) cache [8], while the corresponding DMA mapping can also be cached in the IOMMU’s translation cache (IOTLB).

On the subsequent test iteration, the NIC can issue the request for x using these warm translation caches (MTT on the NIC and IOTLB in the IOMMU), while y ’s virtual address may still incur translation misses, creating a systematic latency asymmetry between the two GETs. Without this translation pre-caching, we never observed the reordering of the GETs.

Importantly, this method only caches the *translation*, not the data itself: the RNIC is non-coherent, so two consecutive RDMA reads to x still require two full round trips to the memory coherence fabric. Additionally, we always place x and y on different pages (and thus different cache lines) to avoid false sharing and unintended cache-line coupling between the two variables.

IV. EVALUATION AND DISCUSSION

A. Experimental Setup

Experiments were conducted on an x86 dual-socket NUMA system, with the NIC physically connected to node 0. The MP4

test was configured as a loopback, with the NIC communicating with itself over the local fabric. Table I summarizes the experimental platform.

TABLE I
EXPERIMENTAL PLATFORM

Component	Specification
<i>Host Machine</i>	
CPU	2× Intel Xeon Gold 6438M
Cores	64 physical (32/socket), 128 logical
L1d / L1i	48 KiB / 32 KiB per core
L2	2 MiB per core (128 MiB total)
L3	60 MiB per socket (120 MiB total)
Memory	128 GB
<i>NIC – NVIDIA BlueField-3 (MT43244)</i>	
NIC Model	ConnectX-7 integrated BlueField-3
NIC CPU	16× Armv8.2+ A78 Hercules cores
NIC Memory	up to 32 GB DDR5
Bandwidth	400 Gb/s
PCIe	Gen 5.0

In the MP4 scenario, the **client** acts as the *writer*—executing $\text{Store}(x:=1)$ followed by $\text{Store}(y:=1)$ —while the **server** acts as the *reader*, issuing $\text{GET}(y)$ followed by $\text{GET}(x)$ remotely via RDMA. The server process was bound to a core on NUMA node 0 (`physcpubind=0`), and the client process to a core on NUMA node 1 (`physcpubind=1`).

The variables x and y were allocated on NUMA node 1 using `membind=1`, and placed on different pages to avoid false sharing and any cache-line coupling between them.

Before each run, we issue a warm-up RDMA read to the page containing x , so that subsequent RDMA reads of x use a cached address translation rather than incurring a full page-table walk in the IO translation path. When the responder NIC subsequently issues reads for both y and x , x ’s address and thus x ’s request is resolved faster and reaches the coherent fabric first. The two requests are forwarded to CHA_y and CHA_x independently, where they are serviced at different times—creating the window in which the MP4 reordering occurs.

Automatic NUMA page migration was disabled to prevent the OS from silently migrating memory pages during the test.

```
echo 0 > /proc/sys/kernel/numa_balancing
```

We explored several placement configurations for the `ib_read_bw` client and server: bound to NUMA node 0 or node 1 on the x86 host, or running on the ARM core embedded inside the NIC - the latter isolating the effect of purely incoming or outgoing DMA reads/writes. In the below pinning, only x86 core was in use and traffic was pinned to the respective NUMA nodes and ran continuously in parallel with the MP4 test:

```
# ib_read_bw noise Server (node 1):
numactl --physcpubind=1 --membind=0 \
  ib_read_bw \
  --run_infinitely -F --qp 25 -s 4096 \
  -t 512 -o 512
```

```
# ib_read_bw noise Client (node 1):
numactl --physcpubind=1 --membind=0 \
  ib_read_bw \
  --run_infinitely -F --qp 25 -s 4096 \
  -t 512 -o 512 192.168.0.1
```

Additionally, we insert a randomized delay followed by a fixed 500ns delay to probabilistically align the client's stores with the server's GET operations, increasing the chance that both $\text{Store}(x:=1)$ and $\text{Store}(y:=1)$ complete within the reordering window.

B. Results

The outcome $(a,b)=(1,0)$ was successfully observed on real hardware. To assess sensitivity, we varied four dimensions across configurations: (1) NUMA binding of the MP4 client and server processes, (2) allocation of x and y across NUMA nodes, (3) background `ib_read_bw` noise, with the noise client or server located either on the x86 host or on the BlueField ARM core, and (4) client-side synchronization delay, tested at several values. Each configuration was executed 10 times with 1,000 iterations per run. Configurations where the synchronization delay was too short yielded a 0% violation rate across all runs and were excluded from the table.

Table II reports 12 configurations. The four NUMA placements vary the location of the writer, the shared variables x and y , and the reader across the two NUMA nodes: Exps. 1 and 2 place the writer and x, y on node 1 (remote from the NIC), differing only in whether the reader is on node 0 or node 1; Exp. 3 keeps the writer on node 1 but moves x, y to node 0; Exp. 4 places all processes and variables on node 0, the NIC's local node. Each placement was tested under three noise settings—both `ib_read_bw` endpoints on x86, noise client on the BlueField ARM core, and noise server on the ARM core.

The x86-only baseline of Exp. 1 achieved 1.54% (95% CI: [1.30, 1.78]), roughly 15 reorderings per 1000 iterations. Exp. 4 yielded the lowest x86-only rate (0.41%), consistent with the shorter RDMA path offering a narrower reordering window. Across all four NUMA configurations, moving the noise client to the ARM core raised the rate relative to the x86-only baseline, peaking at 1.92% (Exp. 3). Moving the noise server to the ARM core suppressed the rate in every case, falling as low as 0.13% (Exp. 4). Both effects are consistent and are discussed in Section IV-C.

C. Discussion

a) Role of background noise: In `ib_read_bw`, the server registers a memory region and the client repeatedly issues RDMA reads from it. To service each read, the responder—side NIC issues a DMA read from the server's host memory across PCIe $\rightarrow$ M2IOSF, and the requester-side NIC then deposits the returned data into the client's receive buffer via a DMA write.

TABLE II
REORDERING RATE $(a,b)=(1,0)$ ACROSS NUMA AND NOISE CONFIGURATIONS ($n=10, 1000$ ITER/RUN). “Wr” = WRITER (CLIENT) CPU NODE; “ x, y ” = MEMORY NODE OF SHARED VARIABLES; “Rd” = READER (SERVER) CPU NODE; BW-CLI/BW-SRV = NUMA NODE OF `IB_READ_BW` CLIENT/SERVER, OR ARM FOR BLUEFIELD CORE.

Wr	x, y	Rd	BW-cli	BW-srv	Mean	95% CI
<i>Exp. 1 — writer and x, y on node 1, reader on node 0</i>						
1	1	0	x86-1	x86-1	1.54%	[1.30, 1.78]
1	1	0	ARM	x86-1	1.91%	[1.65, 2.17]
1	1	0	x86-1	ARM	0.76%	[0.62, 0.90]
<i>Exp. 2 — writer and x, y on node 1, reader on node 1</i>						
1	1	1	x86-1	x86-1	1.47%	[1.16, 1.78]
1	1	1	ARM	x86-1	1.87%	[1.64, 2.10]
1	1	1	x86-1	ARM	0.21%	[0.13, 0.29]
<i>Exp. 3 — writer on node 1, x, y on node 0, reader on node 1</i>						
1	0	1	x86-0	x86-0	1.27%	[0.98, 1.56]
1	0	1	ARM	x86-0	1.92%	[1.62, 2.22]
1	0	1	x86-0	ARM	0.46%	[0.33, 0.59]
<i>Exp. 4 — all processes and memory on node 0</i>						
0	0	0	x86-0	x86-0	0.41%	[0.31, 0.51]
0	0	0	ARM	x86-0	0.81%	[0.63, 0.99]
0	0	0	x86-0	ARM	0.13%	[0.04, 0.22]

When the noise client runs on the BlueField ARM core and the server on x86, the noise traffic consists of DMA reads from x86 memory across PCIe $\rightarrow$ M2IOSF—the same direction and transaction type as the MP4 GET (x) and GET (y) operations. When the noise client runs on x86 and the server on the ARM core, the traffic instead consists mainly of DMA writes into the x86 client buffer.

In both cases the background traffic affects the violation rate, but DMA reads have a stronger effect than DMA writes, consistent with reads being more contention-inducing on the shared PCIe path.

b) Role of address-translation caching: The pre-read of x before the test begins was strictly necessary. The server issues an RDMA READ of x and waits for its completion before the test starts, ensuring that x 's address translation is cached either in the NIC's Memory Translation Table (MTT) cache and/or in the IOMMU's translation cache (IOTLB). Omitting this step eliminates the latency asymmetry between the two GET requests, and the reordering is never triggered.

c) Role of timing and synchronization: We insert a randomized delay followed by a fixed 500ns delay on the client side to align the execution timelines of the client and server. This tuning is critical: too little overlap between the server's GET operations and the client's stores reduces the reordering window, causing the server to consistently observe $(a,b) = (1,1)$; too much serialization eliminates it entirely, yielding $(a,b) = (0,0)$.

d) Role of NUMA topology: Placing x and y on NUMA node 1 — remote from the NIC on node 0 — consistently produced higher violation rates than local allocation. The key asymmetry is not between the two GETs, but between

the server and the client: the server's GET operations must traverse the inter—socket interconnect to reach node 1, while the client's stores are local to node 1 and complete quickly. This mismatch increases the probability that both $\text{Store}(x:=1)$ and $\text{Store}(y:=1)$ complete before the delayed $\text{GET}(x)$ reaches memory.

Notably, even Exp. 4 — where all processes and variables reside on node 0, the NIC's local node, with no inter—socket traversal — produced a non—zero violation rate (0.41%, 95 % CI: [0.31, 0.51]). This shows that NUMA distance amplifies the effect but is not a necessary condition: some reordering occurs even when the memory topology offers no additional latency asymmetry.

V. CONCLUSION

This work demonstrates that the message—passing reordering $(a, b)=(1, 0)$ is physically realizable on commodity RDMA hardware. We identify two necessary conditions for observing GET/GET reordering in our setup: pre-caching the address translation for x before the test begins, and the presence of background NIC traffic. Neither condition alone was sufficient; both were required to reproduce the violation consistently.

Beyond these necessary conditions, NUMA placement further amplifies the effect: configurations where the writer and shared variables reside on the node remote from the NIC consistently produced higher violation rates than local placement, though reordering was observed across all tested NUMA configurations.

Together, these observations suggest that the reordering window is shaped by two distinct components of the read path: address translation latency at the CPU-side IOMMU, which delays the first GET relative to the writer's stores, and PCIe read contention at the responder NIC, which introduces timing asymmetry between the two GETs.

This work thus offers an initial step toward closing the gap between formal RDMA memory models and observed hardware behavior.

ACKNOWLEDGMENT

The authors would like to acknowledge Liran Liss for fruitful discussions on this topic.

This work was funded by the QUARC project of the European Union (EU) Horizon Europe research and innovation programme within the framework of Marie Skłodowska-Curie Actions under grant number 101073355. Generative AI tools were used to assist with literature review, drafting, and editing of this manuscript; all technical content was verified by the authors.

REFERENCES

- [1] G. Ambal, B. Dongol, H. Eran, V. Klimis, O. Lahav, and A. Raad, "Semantics of remote direct memory access: Operational and declarative

- models of rdma on tso architectures," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA2, pp. 1982–2009, 2024.
- [2] InfiniBand Trade Association (IBTA), "About infiniband," 2018, accessed: Apr. 2026. [Online]. Available: <https://www.infinibandta.org/about-infiniband/>
- [3] D. Lustig, S. Sahasrabudde, and O. Giroux, "A formal analysis of the nvidia ptx memory consistency model," in *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. ACM, Apr 2019, pp. 257–270.
- [4] D. A. Marian, P. Lam, T. Hoefler, and M. Vechev, "Modeling and analysis of remote memory access programming," in *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, Oct 2016, pp. 129–144.
- [5] M. Jasny, T. Ziegler, J. Nelson-Slivon, V. Leis, and C. Binnig, "Synchronizing disaggregated data structures with one-sided rdma: Pitfalls, experiments and design guidelines," *ACM Transactions on Database Systems*, vol. 50, no. 1, p. Article 4, Mar 2025.
- [6] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen, "x86-tso: A rigorous and usable programmer's model for x86 multiprocessors," *Communications of the ACM*, vol. 53, no. 7, pp. 89–97, Jul 2010.
- [7] Intel Corporation, "Intel ddio analysis and performance monitoring," accessed: Apr. 2026. [Online]. Available: <https://www.intel.com/content/www/us/en/developer/articles/technical/ddio-analysis-performance-monitoring.html>
- [8] S. Novakovic, Y. Shan, A. Kolli, M. Cui, Y. Zhang, H. Eran, B. Pismenny, L. Liss, M. Wei, D. Tsafir, and M. K. Aguilera, "Storm: A fast transactional dataplane for remote data structures," in *Proceedings of the 12th ACM International Conference on Systems and Storage (SYSTOR)*, Haifa, Israel, 2019, pp. 97–108.