Stop Taking the Scenic Route: the Shortest Distance Between the CPU and the NIC is MMIO

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

James McMahon
jamesm@cs.utah.edu
University of Utah
Salt Lake City, Utah, USA

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

What is the fastest way to transfer data from the CPU to a network interface card (NIC)? Conventional wisdom suggests that the answer is direct memory access (DMA), and recent works have advocated for leveraging cache-coherent I/O interconnects. However, in this paper we examine the arguments against memory-mapped I/O (MMIO) and show that high write throughput can be achieved with MMIO by relaxing ordering constraints. We also propose efficient hardware for recovering ordering at the NIC to ensure correctness while taking advantage of the performance benefits of unordered MMIO writes.

CCS Concepts

• **Hardware → Buses and high-speed links**; *Networking hardware.*

Keywords

Memory-mapped I/O, Network Interface Card, Memory Types

ACM Reference Format:
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 *Workshop on Hot Topics in Operating Systems (HOTOS '25), May 14–16, 2025, Banff, AB, Canada.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3713082.3730389

HOTOS '25, May 14–16, 2025, Banff, AB, Canada

ACM ISBN 979-8-4007-1475-7/2025/05
https://doi.org/10.1145/3713082.3730389

1 Introduction

Recent trends have increased interest in the interface between CPUs and network interface cards (NICs). Link speeds of 200 Gbps and sub-µs interhost communication have created tremendous pressure to build CPU-efficient NIC interfaces [2], and this pressure is increasing due to emerging 800 Gbps Ethernet standards [1]. In turn, these speeds have fueled similar growth in I/O interconnect performance with PCIe 6.0 16× reaching nearly 1000 Gbps [6], and it has helped lead to commodity low-latency coherent I/O interconnects like CXL [4].

Collectively, these changes have led industry and academia to aggressively optimize software interfaces to NICs. However, we assert that existing protocols are more complicated than needed while offering suboptimal performance. Most existing and new interfaces for PCIe are DMA-centric [20]. Alternatively, some recent research has begun to exploit cache coherent I/O interconnects [19, 21]. We show that perhaps neither is needed. In fact, a more basic approach that uses programmed I/O via memory mapped I/O (MMIO) on conventional PCIe may outperform existing approaches in both latency and bandwidth.

The most prevalent software-NIC interfaces today (Figure 1a) rely on (1) discontiguous buffers containing packet data for transmit and receive, (2) descriptors that point to those buffers, and (3) MMIO *doorbell* registers used to inform the NIC when software has enqueued new descriptors. A key benefit of this approach is that the CPU only initiates a small message to the NIC to inform it of new descriptors. The NIC uses coherent DMA to retrieve descriptors and buffer contents asynchronously without stalling the host CPU.

Recently, Ensō showed this classic three-stage approach (MMIO doorbell, then descriptor DMA, then buffer DMA) is highly inefficient and insufficient to transmit and receive small messages at line rate on today's 100 Gbps+ NICs [20]. This is because, though this design offloads the work of data

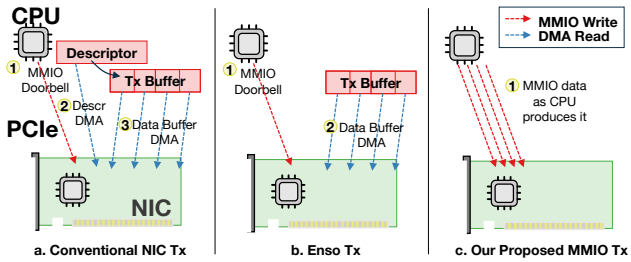

Figure 1: Software-to-NIC Interfaces for Transmit. ⓐ Most NICs use MMIO doorbells, descriptor DMA, and DMAs of packet data. ⓑ Ensō uses contiguous buffers to eliminate the need for descriptors. ⓒ We propose directly writing data to NIC registers via MMIO.

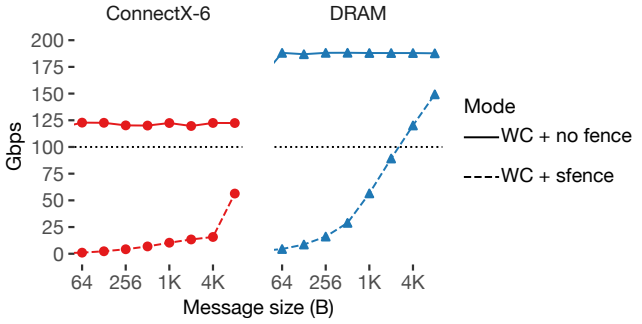

Figure 2: MMIO Goodput for Write Combined Stores to an NVIDIA ConnectX-6 Dx and to DRAM. WC MMIO exceeds the NIC line rate so long as `sfences` are avoided.

transfer to the NIC, it results in discontiguous memory accesses that bottleneck transfers from host memory to the NIC. Ensō demonstrates an alternative design where transmitted and received data is contiguous (Figure 1b), which makes DMA efficient, allowing it to saturate 100 Gbps links even with small messages. However, data transmission in Ensō is still a two-stage process: first the CPU must fill data into a buffer, then it must perform an MMIO to initiate DMA and transmission of the data.

In this paper, we ask: why not send data directly to the NIC via MMIO for transmission (Figure 1c)? It is the lowest latency approach, and it can avoid the costs of staging data in buffers for transmission. We are not the first to consider this [3, 7, 14, 16]; modern NICs like NVIDIA's ConnectX series *selectively* send data via MMIO to optimize latency [22]; however, past approaches have dismissed MMIO as the *primary* means for moving data from the CPU to the NIC. Three arguments against MMIO are: (1) programmed I/O has high overhead due to the CPU cycles for each byte transmitted [23], (2) PCIe MMIO messages have higher metadata overheads per byte transmitted than DMA [12], and (3) ordering restrictions prevent efficiently pipelining MMIOs to the NIC [21].

We investigate these arguments against MMIO and find that a simple approach of writing all data via MMIO directly to the NIC with write combined (WC) stores can move more than 100 Gbps of traffic to a device for transmission. Like others have pointed out [21], this creates the challenge that data can arrive out of order at the NIC; however, we show that the NIC can overcome this obstacle with a very simple hardware reorder unit that works at line rate, without requiring costly ordering operations at the CPU (e.g. `sfences`). Finally, we contrast our proposed design to state-of-the-art DMA-based, coherence-based, and commodity MMIO-optimized transmission paths. We argue our MMIO-centric design is likely to dominate existing approaches in transmission latency-throughput frontier for small messages sizes, and it may even work well for larger maximum transmission size messages.

2 Why not MMIO?

As Figure 1 shows, all existing approaches to triggering NIC data transmission require an MMIO write, generally to a doorbell register. Hence, for standard I/O interconnects a single MMIO write represents the minimum latency required for transmission. Since MMIO writes can carry data, clearly it is also possible to build a transmission path that relies solely on them. So, why do existing approaches rely on DMA or coherent I/O when they aren't strictly required? To answer this we examine common objections to using MMIO for transmission, and we look at state-of-the-art approaches to transferring data to NICs for transmission.

2.1 Can a CPU Drive MMIO Transmission?

Early computer systems with limited CPU processing power adopted DMA to enable I/O transfers to happen in parallel with computation [5]. In the context of CPU to NIC transmissions, DMA frees the CPU from having to issue instructions to perform programmed I/O for each word to transmit. Hence, one argument against using MMIO instructions for data transmission is that the CPU could bottleneck transmission. This is especially a concern since CPUs are often a bottleneck even with today's DMA-based transmission paths (Figure 1a) [20].

However, when we measure MMIO write throughput to modern devices, it is clear that MMIO is not a transmission bottleneck (Figure 2). To show this, we use a *single* CPU core to transfer buffers of varying sizes to a 100 Gbps NVIDIA ConnectX-6 Dx connected via PCIe 4.0 16× using write combined (WC) MMIO. With WC, the CPU combines stores that target the same cacheline, allowing 64 B PCIe transactions

instead of less efficient word-at-a-time transactions. This has ordering implications [17], which we address in §2.5; for now, ignore the results with `sfence`. The target address range on the NIC is 256 KB, ensuring address translation hits in the L1 TLB. Our server is equipped with 2×32-core 2.8 GHz AMD EPYC 7543 CPUs.

Figure 2 shows that even for minimum size 64 B Ethernet frames a single CPU core can consistently write about 120 Gbps of packet data via MMIO. Hence, MMIO exceeds the 100 Gbps line rate for this NIC, preventing it from being a bottleneck even for minimum-size packet transmission. Indeed, the CPU itself is capable of even higher MMIO rates (much higher than the NICs 100 Gbps line rate) via WC writes. The right side of Figure 2 shows the same experiment while writing data to host DRAM instead of MMIO ConnectX-6 Dx registers. It shows that a single CPU core can issue nearly 200 Gbps of traffic.

We also measured the latency of MMIO transmission by immediately reading the written value (with an `mfence` in between to ensure that the read value is not bypassed from the write buffer). On average the latency for performing an MMIO write followed by an MMIO read is 650 ns. Considering that an MMIO read requires a full roundtrip over PCIe, MMIO write latency amounts to about 217 ns. **In sum, today's CPUs *could* efficiently drive transmission purely via MMIO writes at high throughput and low latency.**

2.2 CPU Efficiency of MMIO

However, these results only show that MMIO writes can exceed NIC line rate, not that MMIO is CPU efficient. At first look, DMA seems to save the CPU cycles that would have been spent doing MMIO. But, this ignores the fact that to use DMA the application had to place data into buffers to begin with. While it is true that applications could transmit data from host memory without CPU involvement via DMA, in practice this rarely works. The first reason is that many applications must generate the data that needs to be transmitted. For example, applications will generally need to fill in destination addresses, request data, response status codes, and more. Since the CPU must produce these fields, the most efficient way for it to transmit that data is directly via MMIO rather than the two-step process of staging the generated data into a buffer in cache that is later fetched via DMA.

Second, and more consequentially, many applications struggle to benefit from such "zero-copy" DMAs. In practice, several studies have found that fetching small, discontiguous values via multiple DMA accesses is far less efficient than having a CPU copy those values into a contiguous buffer that can be described with a single descriptor and fetched with a single DMA [13, 15, 18, 20]. Therefore, applications that send and receive small messages are the ones that would benefit substantially from the latency improvements that MMIO would bring, and, for efficiency, these applications will already need to load all values that will be transmitted into CPU registers today. **Overall, the key point is that in today's applications that send small messages, the CPU cost of transmission is already proportional to the amount of transmitted data, so DMA is unlikely to have significant benefits.**

2.3 PCIe Bandwidth Efficiency of MMIO

One commonly cited reason for avoiding MMIO comes from Kalia et. al. [12]. Although PCIe supports larger MMIO writes, Intel CPUs only write up to 64 B per MMIO when using write combining. This results in extra metadata for each cacheline transferred by MMIO compared to DMA reads. On Intel CPUs, long DMA reads only generate extra PCIe metadata overhead for every 128 B transferred for read completion messages [12]. The implication is that with MMIO writes PCIe bandwidth may become a bottleneck sooner than with DMA.

For some time this was true as Ethernet rates exploded while PCIe bandwidth remained flat for 7 years. However, since version 4.0 in 2017 [8], PCIe bandwidth has been doubling every two to three years, keeping it ahead of the explosive growth in network link rates. For example, NVIDIA's ConnectX-7 supports PCIe 4.0 and 5.0 16× and up to a 400 Gbps link rate. MMIO adds about an additional 20% PCIe bandwidth overhead over DMA's 128 B completions. Even after MMIO overheads, a 400 Gbps transmission rate would not use the full bandwidth of a 16× PCIe 5.0 link [9]. Ethernet is poised for standardization of 800 Gbps link rates, but PCIe 6.0 and 7.0 will double and quadruple its performance in the same timeframe [6, 10]. Also, since PCIe is full duplex, this overhead does not cut into the available bandwidth for the receive path. **Hence, while MMIO consumes more PCIe bandwidth than DMAs, those overheads fit within the bandwidth available in current PCIe generations.**

2.4 MMIO in Commodity NICs

NVIDIA ConnectX NICs include a set of latency optimizations for CPU-to-NIC communication. Normally, ConnectX NICs use a three-phase transmission path like the one in Figure 1a. To avoid the need for a separate DMA for each descriptor before each buffer, small messages (e.g. 512 B or smaller) can be *inlined* into the descriptor. Finally, descriptors themselves can avoid DMA using NVIDIA's *BlueFlame* optimization that uses MMIO writes to send the descriptor (including any inlined data) to the NIC rather than using DMA. Combining these two optimizations allows these NICs to initiate transmissions purely using MMIO [22].

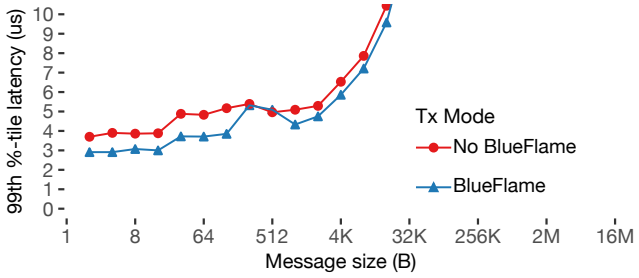

Figure 3: Single-threaded RDMA write latency between machines on a 100 Gbps ConnectX-6 Dx. BlueFlame reduces round trip time by eliminating DMA on the transmission path for small message sizes.

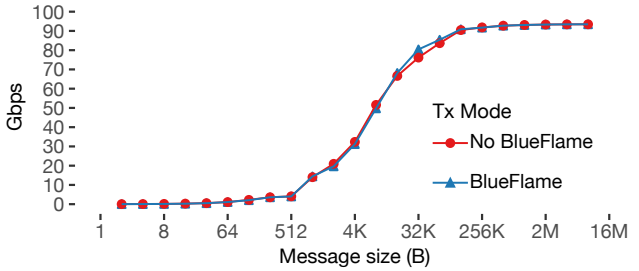

Figure 4: Single-threaded but pipelined RDMA write throughput between machines on a 100 Gbps ConnectX-6 Dx. Small messages use very little of the available 100 Gbps; messages of size 256 KB or more are needed to ensure the NIC isn't limited by DMA overheads.

However, these are optimizations on a fundamentally DMA-centric interface. For example, even when data is inlined and written via MMIO, applications must still fill in descriptors in memory since the NIC is allowed to fallback to using its DMA-based transmission path. The NVIDIA developer's guide also warns not to use this optimization when the NIC is under high load since it hurts peak transmission rates. In fact, NVIDIA's userspace drivers automatically disable these optimizations if message sizes grow large.

Figure 3 shows that BlueFlame significantly reduces transmission latency by eliminating some DMAs. This experiment uses NVIDIA's `ib_write_lat` benchmark, which performs RDMA writes between two machines connected via 100 Gbps ConnectX-6 Dx NICs using PCIe 4.0 16× with UC queue pairs. The client and server each use a single core on a 2.8 GHz AMD EPYC 7543. The two machines are separated by a switch. BlueFlame improves latency for small messages that can be transferred entirely using MMIO writes by about 800 ns.

However, Figure 4 shows that throughput between the machines suffers when message size is small. With 64 B messages, a single thread issuing pipelined RDMA writes only reaches about 1 Gbps. Scaling this across all 64 cores of the machine only raises the 64 B transmission rate to about 30 Gbps. **Overall, this shows that, while commodity NICs can transfer data via MMIO as a latency optimization, they generally use DMA for larger message sizes. Furthermore, even with DMA, NICs struggle to use any substantial fraction of their available line rate with small message sizes.**

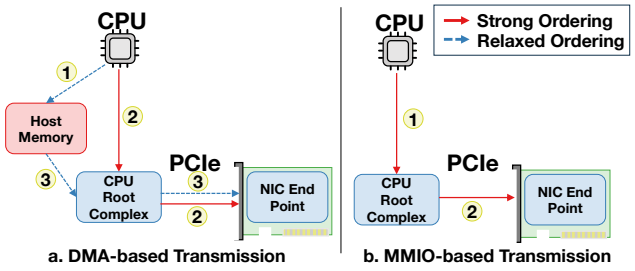

Figure 5: Ordering ⓐ using DMA versus ⓑ MMIO. Dashed lines represent relaxed ordering and solid lines represent strong ordering.

2.5 Ordering: MMIO's Achilles Heel

Data needs to be delivered to the NIC in the correct order. In the following, we discuss how end-to-end ordering can be achieved using DMA vs MMIO (Figure 5).

Suppose that a chunk of data (contiguous addresses, spanning multiple packets) needs to be delivered to the NIC *in order*. Recall that with the conventional DMA-based approach, ① the CPU writes the data to the CPU buffers and then ② rings the doorbell via MMIO. Here, the dashed line refers to relaxed ordering and the solid line refers to strong ordering. It is okay for the writes in the first step to be relaxed; all that matters is that once all of these writes to the host memory have been performed, the MMIO write needs to take place. Once the MMIO doorbell reaches the NIC, ③ the NIC pulls these data packets via multiple DMA reads (PCIe reads); once all of these packets have reached the NIC, the transmission is said to be completed, and the NIC is alerted. Because the NIC is alerted only after all of the packets have arrived, ordering is ensured end-to-end. And crucially, the DMA/PCIe reads in step ③ can potentially overlap with each other without introducing any ordering violations.

Achieving ordering using MMIO is potentially more expensive. Recall that there are two logical hops in the MMIO path from the CPU to the device: ① From the CPU to the CPU's PCIe root complex; and ② From the CPU's root complex to the NIC's PCIe end point. Enforcing ordering in the first hop requires a memory fence instruction after every store from the CPU (which disables write combining [17]),

and enforcing ordering in the second hop requires ordered PCIe writes. We find that the combination of the two can significantly reduce the achieved MMIO throughput: dropping from 120 to 1 Gbps for small messages. In other words, the peak throughput of 120 Gbps does not ensure ordering. The bottom lines of Figure 2 show that inserting `sfences` in between messages for ordering destroys MMIO throughput. **Overall, the key point is that while MMIO can achieve high transmit throughput, it compromises ordering.**

3 Enforcing Data Order at the NIC

Figure 2 showed that high MMIO throughput can be achieved at the cost of relaxing ordering. In practice, we need to ensure that data is delivered to the NIC in the correct order. How does one enforce ordered MMIO writes efficiently?

In this section we present an efficient hardware-based approach that works at the NIC endpoint. Our approach leverages the end-to-end principle and allows for reordering ① between the CPU and the PCIe root complex (so, no fences) and ② between the PCIe root complex and the PCIe end point (allowing for relaxed ordered PCIe writes) — but, it *recovers* ordering at the NIC endpoint.

3.1 The Hardware Reorder Unit

We assume that a message's data are written by the CPU to sequential addresses that are in a range specified by the NIC's PCI BARs. PCIe write transactions include both message data and the target physical address of the MMIO, so ordering is ensured as long as: (a) the (out-of-order) received data is written into NIC-side buffers in an order consistent with the physical address associated with each transaction; and (b) we can determine *when* all of the data that makes up a message has been received. We implement this with a simple NIC hardware reorder unit that holds the following metadata.

Data Buffer temporarily holds received MMIO data. The size of this buffer need only be large enough to tolerate the maximum amount of reordering the CPU may do for its WC MMIO writes. Though this is microarchitecture-specific, we expect this to be bounded to a few cachelines in practice.

Valid Vector stores a valid bit for each byte in the Data Buffer to indicate whether the byte has been received. All valid bits are initialized to zero.

Length Address Register holds the address of the byte indicating length for the earliest message held in the Data Buffer. We assume that every message contains a special byte at fixed offset p from the start of the message that specifies the length of the message in bytes.

When a PCIe MMIO is received, its data is written into both the NIC memory and the Data Buffer in parallel. The bits in the Valid Vector corresponding to the received bytes are also set. Once the Valid Vector bit corresponding to location

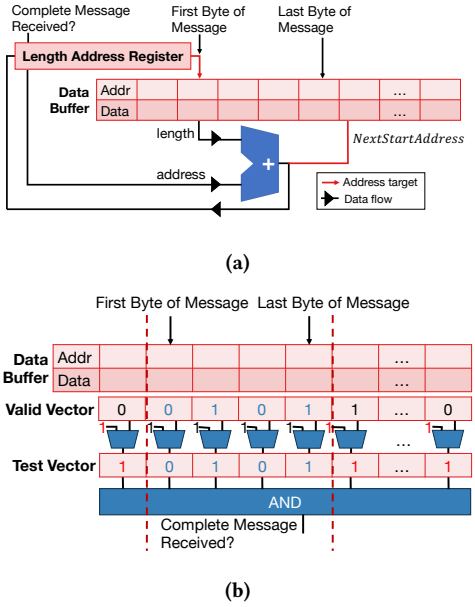

Figure 6: (a) computes *NextStartAddress* assuming that the length is in the first byte of the message ($p = 0$). In this case Length Address Register can be updated to *NextStartAddress* if the complete message is received and (b) selecting the values for the Test Vector using the start and end bytes of the earliest message to determine that a message is fully received.

holding the length has been set, we can retrieve the length of the message. Once the length is available for the earliest message in the Data Buffer, we compute (Figure 6a): (a) the first address of the next earliest message, *NextStartAddress*, and (b) the last address of the earliest message.

Once we know the length of the earliest message, we check the corresponding bits of the Valid Vector (using the parallel AND) to determine whether the earliest message has been received (Figure 6b). If the earliest message has been received, the corresponding bits in the Valid Vector are cleared so that the entries in the Data Buffer can be reused. *NextStartAddress* is provided to the NIC to communicate that all of the data corresponding to lower addresses have been fully received.

3.2 Can the Unit Operate at Line Rate?

We have synthesized a prototype using an AMD XCVU9P Field Programmable Gate Array using the Vivado 2018.2 accessed through an Amazon AWS EC2 instance. The main overhead of the unit is the delay between receiving the last packet for the earliest message and delivering the value of *NextStartAddress* to the NIC. In our prototype, we achieved a clock frequency of 250 MHz, which is the peak for the device. Since our design is able to deliver $\frac{250\times10^6}{2}$ = 125 million 64 B messages to the NIC per second, it would be able to deliver

$64 \times 125 \times 10^6 \times 8 = 64$ Gbps. If the design was implemented on an ASIC with the NIC, it should be possible to achieve a clock frequency of at least 500 MHz which would allow a throughput of 128 Gbps, achieving line rate.

3.3 Scalability and NIC Memory

Modern NICs often support thousands of transmission queues. In our approach, if each queue was represented by 256 KB of MMIO addresses, for example, it might seem that this would require a prohibitive amount of on-NIC memory. Fortunately, this is not the case. Each transmission queue would need separate MMIO addresses, but not separate memory. We envision that the reorder unit would be coupled with a unit that dynamically manages on-NIC memory, similar to many switches. As contiguous chunks of data arrived for a specific transmission queue, the unit would copy the data into a dynamically-managed buffer which would be freed after the NIC was able to transmit the buffer. Hence, the only data that the NIC would have hold in memory is any discontiguous data in the reorder unit and any data awaiting transmission, which is the same as what is required today.

3.4 MMIO Address Reuse

Since MMIO address space is finite, it would eventually be necessary to reuse MMIO addresses to transmit more data. It would be a problem if an address was reused before the data that was previously stored to it were flushed. This could be avoided if the NIC address space is at least as large as the total capacity of WC buffers so that all WC buffers must be filled and flushed before reusing addresses. On modern Intel processors, this capacity is 512 B since modern processors have up to eight 64 B WC buffers [11], though this is microarchitecture specific. Another simple solution is to perform one `sfence` whenever the CPU exhausts the entire MMIO space for that queue before reusing addresses.

A related concern is that the CPU might overrun the NIC if it uses MMIO writes to send the NIC data faster than the NIC can transmit. Fortunately, PCIe has flow control that prevents this; if a NIC is slow to transmit, the CPU's MMIO writes will stall, naturally slowing down transmission to match the NIC.

4 Discussion

4.1 What About Receive?

While we advocate for MMIO on the transmit path, the right approach on the receive path is whatever pushes data directly from the NIC to the CPU. Interestingly, in the NIC-to-CPU direction the most similar operation is DMA, not MMIO. Hence, Ensō's DMA-based receive path seems near optimal with no clear way to improve it using MMIO [20].

4.2 What About Coherent I/O such as CXL?

Recent research has begun to exploit cache coherent I/O interconnects [19, 21] in an effort to enable both high bandwidth as well as low latency CPU-NIC communication. CC-NIC [21], a cache-coherent NIC interface, considered an MMIO baseline but ruled it out due to the cost of ordering fences. Given that MMIO can achieve line rate and low latency without compromising ordering with our approach, we argue that there is little need for a coherence-based solution.

In fact, protocols such as CXL (a MESI variant) complicate efficient CPU-NIC producer-consumer communication, since these protocols obtain ownership and transition to exclusive state on writes; therefore, a CPU producer write obtains the cache block in exclusive state. This means that a consumer read from the NIC requires an indirection—wherein the data has to be pulled from the CPU's caches, thereby increasing the latency. Of course these fundamental inefficiencies can be worked around using creative optimizations in the software [21] or the protocol [19]. What these optimizations are working to achieve is, however, what is already achieved via our MMIO-based transmit path. **Overall, a simple MMIO transmit path provides high throughput and low latency without the need for work arounds on top of coherence protocols.**

5 Conclusion

Programmed I/O is the oldest and simplest form of moving data to devices; even so, it is fast enough to saturate the 100 Gbps+ line rate of modern NICs, provided the NIC can tolerate the bounded reordering that write combining allows. Since it is also the lowest latency path from the CPU to transmission, we contend that it is worth investigating software interfaces for NICs that are designed from the ground up around MMIO.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. CNS-2245999. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. This work was also supported in part by VMware and ARM.

References

[1] 2020. *800G Specification.* Available at https://ethernettechnologyconsortium.org/wp-content/uploads/2020/03/800G-Specification_r1.0.pdf.

[2] 2021. *NVIDIA ConnectX-6 Dx Ethernet SmartNIC.* Available at https://www.nvidia.com/content/dam/en-zz/Solutions/networking/ethernet-adapters/ConnectX-6-Dx-Datasheet.pdf.

[3] Mahesh Chaudhari, Kedar Kulkarni, Shreeya Badhe, and Vandana Inamdar. 2017. Evaluating Effect of Write Combining on PCIe Throughput to Improve HPC Interconnect Performance. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 639–640.

[4] 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.

[5] IBM Corporation. 1960. *IBM General Information Manual 709-7090 Data Processing System*.

[6] Debendra Das Sharma. 2021. PCI Express 6.0 Specification: A Low-Latency, High-Bandwidth, High-Reliability, and Cost-Effective Interconnect With 64.0 GT/s PAM-4 Signaling. *IEEE Micro* 41, 1 (2021), 23–29. https://doi.org/10.1109/MM.2020.3039925

[7] Mario Flajslik and Mendel Rosenblum. 2013. Network Interface Design for Low Latency Request-Response Protocols. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. USENIX Association, San Jose, CA, 333–346. https://www.usenix.org/conference/atc13/technical-sessions/presentation/flajslik

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

[9] Peripheral Component Interconnect Special Interest Group. 2018. *PCI Express® Base Specification Revision 5.0*. Available at https://pcisig.com/doubling-bandwidth-under-two-years-pci-express%C2%AE-base-specification-revision-50-version-09-now.

[10] Peripheral Component Interconnect Special Interest Group. 2024. *PCI Express® Base Specification Revision 7.0*. Available at https://pcisig.com/blog/pcie%C2%AE-70-specification-version-05-now-available-full-draft-available-members.

[11] Intel. 2024. *Intel® 64 and ia-32 architectures software developer's manual*.

[12] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. Design Guidelines for High Performance RDMA Systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO, 437–450. https://www.usenix.org/conference/atc16/technical-sessions/presentation/kalia

[13] Aniraj Kesavan, Robert Ricci, and Ryan Stutsman. 2016. To Copy or Not to Copy: Making In-memory Databases Fast on Modern NICs. In *International Workshop on In-Memory Data Management and Analytics*. Springer, 79–94.

[14] 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.

[15] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, et al. 2015. The RAMCloud storage system. *ACM Transactions on Computer Systems (TOCS)* 33, 3 (2015), 1–55.

[16] Boris Pismenny, Liran Liss, Adam Morrison, and Dan Tsafrir. 2022. The benefits of general-purpose on-NIC memory. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 1130–1147. https://doi.org/10.1145/3503222.3507711

[17] Azalea Raad, Luc Maranget, and Viktor Vafeiadis. 2022. Extending Intel-x86 consistency and persistency: formalising the semantics of Intel-x86 memory types and non-temporal stores. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–31. https://doi.org/10.1145/3498683

[18] Deepti Raghavan, Shreya Ravi, Gina Yuan, Pratiksha Thaker, Sanjari Srivastava, Micah Murray, Pedro Henrique Penna, Amy Ousterhout, Philip Levis, Matei Zaharia, et al. 2023. Cornflakes: Zero-copy serialization for microsecond-scale networking. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 200–215.

[19] Anastasiia Ruzhanskaia, Pengcheng Xu, David Cock, and Timothy Roscoe. 2024. Rethinking Programmed I/O for Fast Devices, Cheap Cores, and Coherent Interconnects. arXiv:2409.08141 [cs.AR] https://arxiv.org/abs/2409.08141

[20] 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

[21] 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. https://doi.org/10.1145/3617232.3624868

[22] Mellanox Technologies. 2016. *Mellanox Adapters Programmer's Reference Manual (PRM)*. https://network.nvidia.com/files/doc-2020/ethernet-adapters-programming-manual.pdf.

[23] Xiuxiu Wang, Yipei Niu, Fangming Liu, and Zichen Xu. 2022. When FPGA Meets Cloud: A First Look at Performance. *IEEE Transactions on Cloud Computing* 10, 2 (2022), 1344–1357. https://doi.org/10.1109/TCC.2020.2992548