Design Guidelines for High Performance RDMA Systems

Anuj Kalia, ***Carnegie Mellon University;*** **Michael Kaminsky,** ***Intel Labs;***
David G. Andersen, ***Carnegie Mellon University***

https://www.usenix.org/conference/atc16/technical-sessions/presentation/kalia

This paper is included in the Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC '16).

June 22–24, 2016 • Denver, CO, USA

978-1-931971-30-0

Open access to the Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC '16) is sponsored by USENIX.

Design Guidelines for High Performance RDMA Systems

Anuj Kalia Michael Kaminsky[†] David G. Andersen
Carnegie Mellon University *[†]Intel Labs*

Abstract

Modern RDMA hardware offers the potential for exceptional performance, but design choices including *which* RDMA operations to use and *how* to use them significantly affect observed performance. This paper lays out guidelines that can be used by system designers to navigate the RDMA design space. Our guidelines emphasize paying attention to low-level details such as individual PCIe transactions and NIC architecture. We empirically demonstrate how these guidelines can be used to improve the performance of RDMA-based systems: we design a networked sequencer that outperforms an existing design by 50x, and improve the CPU efficiency of a prior high-performance key-value store by 83%. We also present and evaluate several new RDMA optimizations and pitfalls, and discuss how they affect the design of RDMA systems.

1 Introduction

In recent years, Remote Direct Memory Access (RDMA)-capable networks have dropped in price and made substantial inroads into datacenters. Despite their newfound popularity, using their advanced capabilities to best effect remains challenging for software designers. This challenge arises because of the nearly-bewildering array of options a programmer has for using the NIC[1], and because the relative performance of these operations is determined by complex low-level factors such as PCIe bus transactions and the (proprietary and often confidential) details of the NIC architecture.

Unfortunately, finding an efficient match between RDMA capabilities and an application is important: As we show in Section 5, the best and worst choices of RDMA options vary by a factor of *seventy* in their overall throughput, and by a factor of 3.2 in the amount of host CPU they consume. Furthermore, there is no one-size-fits-all best approach. Small changes in application requirements significantly affect the relative performance of different designs. For example, using general-purpose RPCs over RDMA is the best design for a networked key-value store (Section 4), but this same design choice provides lower scalability and 16% lower throughput than the best choice for a networked "sequencer" (Section 4; the sequencer returns a monotonically increasing integer to client requests).

This paper helps system designers address this challenge in two ways. First, it provides guidelines, backed by an open-source set of measurement tools, for evaluating and optimizing the most important system factors that affect end-to-end throughput when using RDMA NICs. For each guideline (e.g., "Avoid NIC cache misses"), the paper provides insight on both how to determine whether this guideline is relevant (e.g., by using PCIe counter measurements to detect excess traffic between the NIC and the CPU), and a discussion of which modes of using the NICs are most likely to mitigate the problem.

Second, we evaluate the efficacy of these guidelines by applying them to both microbenchmarks and real systems, across three generations of RDMA hardware. Section 4.2 presents a novel design for a network sequencer that outperforms an existing design by 50x. Our best sequencer design handles 122 million operations/second using a single NIC and scales well. Section 4.3 applies the guidelines to improve the CPU efficiency and throughput of the HERD key-value cache [20] by 83% and 35% respectively. Finally, we show that today's RDMA NICs handle contention for atomic operations extremely slowly, rendering designs that use them [27, 30, 11] very slow.

A lesson from our work is that low-level details are surprisingly important for RDMA system design. Our underlying goal is to provide researchers and developers with a roadmap through these details without necessarily becoming RDMA gurus. We provide simple models of RDMA operations and their associated CPU and PCIe costs, plus open-source software to measure and analyze them (`https://github.com/efficient/rdma_bench`).

We begin our journey into high-performance RDMA-based systems with a review of the relevant capabilities of RDMA NICs, and the PCIe bus that frequently arises as a bottleneck.

[1]In this paper, we refer exclusively to RDMA-capable network interface cards, so we use the more generic but shorter term NIC throughout.

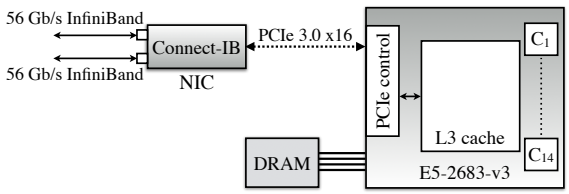


Figure 1: Hardware components of a node in an RDMA cluster

2 Background

Figure 1 shows the relevant hardware components of a machine in an RDMA cluster. A NIC with one or more ports connects to the PCIe controller of a multi-core CPU. The PCIe controller reads/writes the L3 cache to service the NIC's PCIe requests; on modern Intel servers [4], the L3 cache provides counters for PCIe events.

2.1 PCI Express

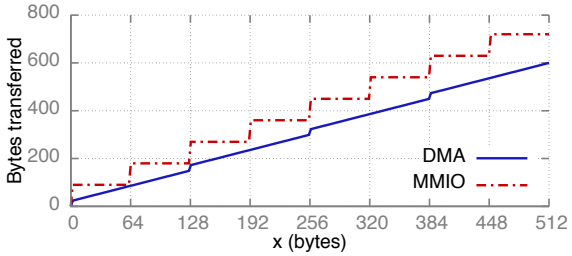
The current fastest PCIe link is PCIe "3.0 x16," the 3rd generation PCIe protocol, using 16 lanes. The bandwidth of a PCIe link is the per-lane bandwidth times the number of lanes. PCIe is a layered protocol, and the layer headers add overhead that is important to understand for efficiency. RDMA operations generate 3 types of PCIe transaction layer packets (TLPs): read requests, write requests, and read completions (there is no transaction-layer response for a write). Figure 2a lists the bandwidth and header overhead for the PCIe generations in our clusters. Note that the header overhead of 20–26 bytes is comparable to the common size of data items used in services such as memcached [25] and RPCs [15].

MMIO writes vs. DMA reads There are important differences between the two methods of transferring data from a CPU to a PCIe device. CPUs write to mapped device memory (MMIO) to initiate PCIe writes. To avoid generating a PCIe write for each store instruction, CPUs use an optimization called "write combining," which combines stores to generate cache line–sized PCIe transactions. PCIe devices have DMA engines and can read from DRAM using DMA. DMA reads are not restricted to cache lines, but a read response larger than the CPU's read completion combining size (C_{rc}) is split into multiple completions. C_{rc} is 128 bytes for the Intel CPUs used in our measurements (Table 2); we assume 128 bytes for the AMD CPU [4, 3]. A DMA read always uses less host-to-device PCIe bandwidth than an equal-sized MMIO; Figure 2b shows an analytical comparison. This is an important factor, and we show how it affects performance of higher-layer protocols in the subsequent sections.

PCIe counters Our contributions rely on understanding the PCIe interaction between NICs and CPUs. Although precise PCIe analysis requires expensive PCIe analyzers or proprietary/confidential NICs manuals, PCIe counters available on modern CPUs can provide several useful insights.[2] For each counter, the number of captured events per second is its *counter rate*. Our analysis primarily uses counters for DMA reads (`PCIeRdCur`) and DMA writes (`PCIeItoM`).

Gen	Bitrate	Per-lane b/w	Request	Completion
2.0	5 GT/s	500 MB/s	24 B	20 B
3.0	8 GT/s	984.6 MB/s	26 B	22 B

(a) *Speed and header sizes for PCIe generations. Lane bandwidth excludes physical layer encoding overhead.*



(b) *CPU-to-device PCIe traffic for an x-byte transfer with DMA and MMIO, assuming PCIe 3.0 and C_{rc} = 128 bytes.*

Figure 2: PCIe background

2.2 RDMA

RDMA is a network feature that allows direct access to the memory of a remote computer. RDMA-providing networks include InfiniBand, RoCE (RDMA over Converged Ethernet), and iWARP (Internet Wide Area RDMA Protocol). RDMA networks usually provide high bandwidth and low latency: NICs with 100 Gbps of per-port bandwidth and ~ 2µs round-trip latency are commercially available. The performance and scalability of an RDMA-based communication protocol depends on several factors including the operation (verb) type, transport, optimization flags, and operation initiation method.

2.2.1 RDMA verbs and transports

RDMA hosts communicate using queue pairs (QPs); hosts create QPs consisting of a send queue and a receive queue, and post operations to these queues using the *verbs* API. We call the host initiating a verb the *requester* and the destination host the *responder*. For some verbs, the responder does not actually send a response. On completing a verb, the requester's NIC optionally signals completion by DMA-ing a completion entry (CQE) to a completion queue (CQ) associated with the QP. Verbs can be made *unsignaled* by setting a flag in the request; these verbs do not generate a CQE, and the application detects completion using application-specific methods.

The two types of verbs are memory verbs and messaging verbs. Memory verbs include RDMA reads, writes,

[2] The CPU intercepts cache line-level activity between the PCIe controller and the L3 cache, so the counters can miss some critical information. For example, the counters indicate 2 PCIe reads when the NIC reads a 4-byte chunk straddling 2 cache lines.

	SEND/RECV	WRITE	READ	WQE header
RC	✓	✓	✓	36 B
UC	✓	✓	✗	36 B
UD	✓	✗	✗	68 B

Table 1: Operations supported by each transport type, and their Mellanox WQE header size for SEND, WRITE, and READ. RECV WQE header size is 16 B for all transports.

and atomic operations. These verbs specify the remote address to operate on and bypass the responder's CPU. Messaging verbs include the send and receive verbs. These verbs involve the responder's CPU: the send's payload is written to an address specified by a receive that was posted previously by the responder's CPU. In this paper, we refer to RDMA read, write, send, and receive verbs as READ, WRITE, SEND, and RECV respectively.

RDMA transports are either reliable or unreliable, and either connected or unconnected (also called datagram). With reliable transports, the NIC uses acknowledgments to guarantee in-order delivery of messages. Unreliable transports do not provide this guarantee. However, modern RDMA implementations such as InfiniBand use a lossless link layer that prevents congestion-based losses using link layer flow control [1], and bit error–based losses using link layer retransmissions [8]. Therefore, unreliable transports drop packets very rarely. Connected transports require one-to-one connections between QPs, whereas a datagram QP can communicate with multiple QPs. We consider two types of connected transports in this paper: Reliable Connected (RC) and Unreliable Connected (UC). Current RDMA hardware provides only 1 datagram transport: Unreliable Datagram (UD). Different transports support different subsets of verbs: UC does not support RDMA reads, and UD does not support memory verbs. Table 1 summarizes this.

2.2.2 RDMA WQEs

To initiate RDMA operations, the user-mode NIC driver at the requester creates Work Queue Elements (WQEs) in host memory; typically, WQEs are created in a preallocated, contiguous memory region, and each WQE is individually cache line–aligned. (We discuss methods of transferring WQEs to the device in Section 3.1.) The WQE format is vendor-specific and is determined by the NIC hardware.

WQE size depends on several factors: the type of RDMA operation, the transport, and whether the payload is referenced by a pointer field or *inlined* in the WQE (i.e., the WQE buffer includes the payload). Table 1 shows the WQE header size for Mellanox NICs for three transports. For example, with a 36-byte WQE header, the size of a WRITE WQE with an x-byte inlined payload is $36 + x$ bytes. UD WQEs have larger, 68-byte headers to store additional routing information.

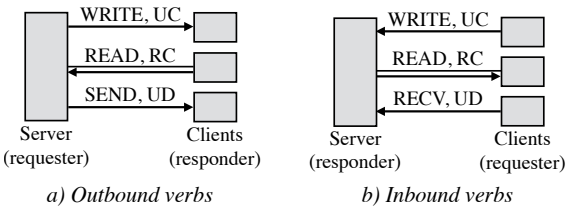


Figure 3: Inbound and outbound verbs at the server.

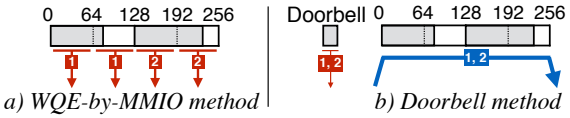


Figure 4: The WQE-by-MMIO and Doorbell methods for transferring two WQEs (shaded) spanning 2 cache lines. Arrows represent PCIe transactions. Red (thin) arrows are MMIO writes; the blue (thick) arrow is a DMA reads. Arrows are marked with WQE numbers; arrow width represents transaction size.

2.2.3 Terminology and default assumptions

We distinguish between *inbound* and *outbound* verbs because their performance differs significantly (Section 5): memory verbs and SENDs are outbound at the requester and inbound at the responder; RECVs are always inbound. Figure 3 summarizes this. As our study focuses on small messages, all WRITEs and SENDs are inlined by default. We define the padding-to-cache-line-alignment function $x' := \lceil x/64 \rceil * 64$. We denote the per-lane bandwidth, request header size, and completion header size of PCIe 3.0 by P_{bw}, P_r, and P_c, respectively.

3 RDMA design guidelines

We now present our design guidelines. Along with each guideline, we present new optimizations and briefly describe those presented in prior literature. We make two assumptions about the NIC hardware that are true for all currently available NICs. First, we assume that the NIC is PCIe-based device. Current network interfaces (NIs) are predominantly discrete PCIe cards; vendors are beginning to integrate NIs on-die [2, 5], or on-package [6], but these NIs still communicate with the PCIe controller using the PCIe protocol, and are less powerful than discrete NIs. Second, we assume that the NIC has internal parallelism using multiple processing units (PUs)—this is generally true of high-speed NIs [19]. As in conventional parallel programming, this parallelism provides both optimization opportunities (Section 3.3) and pitfalls (Section 3.4).

To discuss the impact on CPU and PCIe use of the optimizations below, we consider transferring N WQEs of size D bytes from the CPU to the NIC.

3.1 Reduce CPU-initiated MMIOs

Both CPU efficiency and RDMA throughput can improve if MMIOs are reduced or replaced with the more CPU-

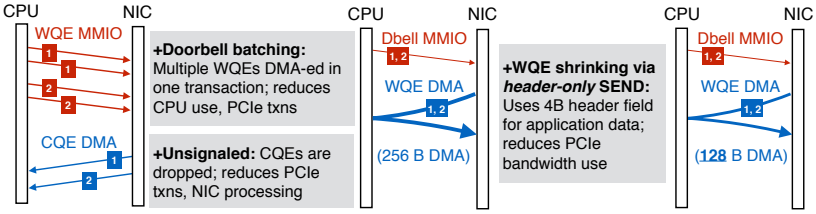


Figure 5: Optimizations for issuing two 4-byte UD SENDs. A UD SEND WQE spans 2 cache lines on Mellanox NICs because of the 68-byte header; we shrink it to 1 cache line by using a 4-byte header field for payload. Arrow notation follows Figure 4.

and bandwidth-efficient DMAs. CPUs initiate network operations by sending a message to the NIC via MMIO. The message can (1) contain the new work queue elements, or (2) it can refer to the new WQEs by using information such as the address of the last WQE. In the first case, the WQEs are transferred via 64-byte write-combined MMIOs. In the second case, the NIC reads the WQEs using one or more DMAs.[3] We refer to these methods as *WQE-by-MMIO* and *Doorbell* respectively. (Different technologies have different terms for these methods. Mellanox uses "BlueFlame" and "Doorbell," and Intel® Omni-Path Architecture uses "PIO send" and "SDMA," respectively.) Figure 4 summarizes this. The WQE-by-MMIO method optimizes for low latency and is typically the default. Two optimizations can improve performance by reducing MMIOs:

Doorbell batching If an application can issue multiple WQEs to a QP, it can use one Doorbell MMIO for the batch. *CPU:* Doorbell reduces CPU-generated MMIOs from $N * D' / 64$ with WQE-by-MMIO to 1. *PCIe:* For $N = 10$ and $D = 65$ and PCIe 3.0, Doorbell transfers 1534 bytes, whereas WQE-by-MMIO transfers 1800 bytes (Appendix A).

In this paper, we refer to Doorbell batching as batching—a batched WQE transfer via WQE-by-MMIO is identical to a sequence of individual WQE-by-MMIOs, so batching is only useful for Doorbell.

WQE shrinking Reducing the number of cache lines used by a WQE can improve throughput drastically. For example, consider reducing WQE size by only 1 byte from 129 B to 128 B, and assume that WQE-by-MMIO is used. *CPU:* CPU-generated MMIOs decreases from 3 to 2. *PCIe:* Number of PCIe transactions decreases from 3 to 2. Shrinking mechanisms include compacting the application payload, or overloading unused WQE header fields with application data.

3.2 Reduce NIC-initiated DMAs

Reducing DMAs saves NIC processing power and PCIe bandwidth, improving RDMA throughput. Note that the batching optimization above *adds* a DMA read, but it avoids *multiple* MMIOs, which is usually a good tradeoff.

[3]In this paper, we assume that the NIC reads *all* new WQEs in one DMA, as is done by Mellanox's Connect-IB and newer NICs. Older Mellanox NICs read one or more WQEs per DMA, depending on the NIC's proprietary prefetching logic.

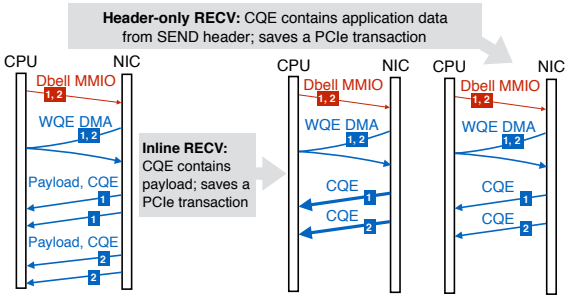


Figure 6: Optimizations for RECVs with small SENDs.

Known optimizations to reduce NIC-initiated DMAs include unsignaled verbs which avoid the completion DMA write (Section 2.2.1), and payload inlining which avoids the payload DMA read (Section 2.2.2). The two optimizations in Section 3.1 affect DMAs, too: batching with large N requires fewer DMA reads than smaller N; WQE shrinking further makes these reads smaller.

NICs must DMA a completion queue entry for completed RECVs [1]; this provides an additional optimization opportunity, as discussed below. Unlike CQEs of other verbs that only signal completion and are dispensible, RECV CQEs contain important metadata such as the size of received data. NICs typically generate two separate DMAs for payload and completion, writing them to application- and driver-owned memory respectively. We later show that the corresponding performance penalty explains the rule-of-thumb that messaging verbs are slower than memory verbs, and using the DMA-avoiding optimizations below challenges this rule-of-thumb for some payload sizes. We assume here that the corresponding SEND for a RECV carries an X-byte payload.

Inline RECV If X is small (~ 64 for Mellanox's NICs), the NIC encapsulates the payload in the CQE, which is later copied by the driver to the application-specified address. *CPU:* Minor overhead for copying the small payload. *PCIe:* Uses 1 DMA instead of 2.

Header-only RECV If $X = 0$ (i.e., the RDMA SEND packet consists of only a header and no payload), the payload DMA is not generated at the receiver. Some information from the packet's header is included in the DMA-ed CQE, which can be used to implement application protocols. We call SENDs and RECVs with $X = 0$ *header-only*, and *regular* otherwise. *PCIe:* Uses 1 DMA instead of 2.

Figure 5 and Figure 6 summarize these two guidelines for UD SENDs and RECVs, respectively. These two verbs are used extensively in our evaluation.

3.3 Engage multiple NIC PUs

Exploiting NIC parallelism is necessary for high performance, but requires explicit attention. A common RDMA programming decision is to use as few queue pairs as possible, but doing so limits NIC parallelism to the number of QPs. This is because operations on the same QP have ordering dependencies and are ideally handled by the same NIC processing unit to avoid cross-PU synchronization. For example, in datagram-based communication, one QP per CPU core is sufficient for communication with all remote cores. Using one QP consumes the least NIC SRAM to hold QP state, while avoiding QP sharing among CPU cores. However, it "binds" a CPU core to a PU and may limit core throughput to PU throughput. This is likely to happen when per-message application processing is small (e.g., the sequencer in Section 4.2) and a high-speed CPU core overwhelms a less powerful PU. In such cases, using multiple QPs per core increases CPU efficiency; we call this the *multi-queue* optimization.

3.4 Avoid contention among NIC PUs

RDMA operations that require cross-QP synchronization introduce contention among PUs, and can perform over an order of magnitude worse than uncontended operations. For example, RDMA provides atomic operations such as compare-and-swap and fetch-and-add on remote memory. To our knowledge, all NICs available at the time of writing (including the recently released ConnectX-4 [7]) use internal concurrency control for atomics: PUs acquire an internal lock for the target address and issue read-modify-write over PCIe. Note that atomic operations contend with non-atomic verbs too. Future NICs may use PCIe's atomic transactions for higher performing, cache coherence-based concurrency control.

Therefore, the NIC's internal locking *mechanism*, such as the number of locks and the mapping of atomic addresses to these locks, is important; we describe experiments to infer this in Section 5.4. Note that due to the limited SRAM in NICs, the number of available locks is small, which amplifies contention in the workload.

3.5 Avoid NIC cache misses

NICs cache several types of information; it is critical to maintain a high cache hit rate because a miss translates to a read over PCIe. Cached information includes (1) virtual to physical address translations for RDMA-registered memory, (2) QP state, and (3) a work queue element cache. While the first two are known [13], the third is undocumented and was discovered in our experiments. Address translation cache misses can be reduced by using large (e.g., 2 MB) pages, and QP state cache misses by using fewer QPs [13]. We make two new contributions in this context:

Name	Hardware
CX	ConnectX (1x 20 Gb/s InfiniBand ports), PCIe 2.0 x8, AMD Opteron 8354 (4 cores, 2.2 GHz)
CX3	ConnectX-3 (1x 56 Gb/s InfiniBand ports), PCIe 3.0 x8, Intel® Xeon® E5-2450 CPU (8 cores, 2.1 GHz)
CIB	Connect-IB (2x 56 Gb/s InfiniBand ports), PCIe 3.0 x16, Intel® Xeon® E5-2683-v3 CPU (14 cores, 2 GHz)

Table 2: Measurement clusters. CX is NSF PRObE's Nome cluster [17], CX3 is Emulab's Apt cluster [31], and CIB is a cluster at NetApp. CX uses PCIe 2.0 at 2.5 GT/s.

Detecting cache misses All types of NIC cache misses are transparent to the application and can be difficult to detect. We demonstrate how PCIe counters can be leveraged to accomplish this, by detecting and measuring WQE cache misses (Section 5.3.2). In general, subtracting the application's expected PCIe reads from the actual reads reported by PCIe counters gives an estimate of cache misses. Estimating expected PCIe reads in turn requires PCIe models of RDMA operations (Section 5.1).

WQE cache misses The initial work queue element transfer from CPU to NIC triggers an insertion of the WQE into the NIC's WQE cache. When the NIC eventually processes this WQE, a cache miss can occur if it was evicted by newer WQEs. In Section 5.3.2, we show how to measure and reduce these misses.

4 Improved system designs

We now demonstrate how these guidelines can be used to improve the design of whole systems. We consider two systems: networked sequencers, and key-value stores.

Evaluation setup We perform our evaluation on the three clusters described in Table 2. We name the clusters with the initials of their NICs, which is the main hardware component governing performance. CX3 and CIB run Ubuntu 14.04 with Mellanox OFED 2.4; CX runs Ubuntu 12.04 with Mellanox OFED 2.3. Throughout the paper, we use WQE-by-MMIO for non-batched operations and Doorbell for batched operations. However, when batching is enabled but the available batch size is one, WQE-by-MMIO is used. (Doorbell provides little CPU savings for transferring a single small WQE, and uses an extra PCIe transaction.) For brevity, we primarily use the state-of-the-art CIB cluster in this section; Section 5 evaluates our optimizations on all clusters.

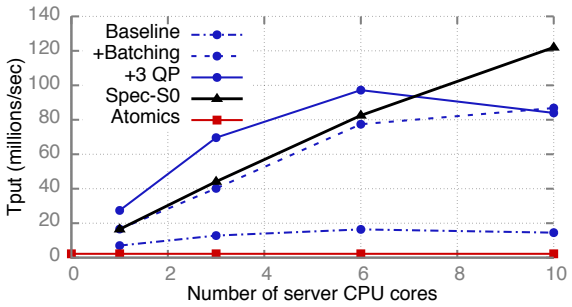


Figure 7: Impact of optimizations on HERD RPC-based sequencer (blue lines with circular dots), and throughput of `Spec-S0` and the atomics-based sequencer

4.1 Overview of HERD RPCs

We use HERD's RPC protocol for communication between clients and the sequencer/key-value server. HERD RPCs have low overhead at the server and high number-of-clients scalability. Protocol clients use unreliable WRITEs to write requests to a request memory region at the server. Before doing so, they post a RECV to an unreliable datagram QP for the server's response. A server thread (a *worker*) detects a new request by polling on the request memory region. Then, it performs application processing and responds using a UD SEND posted via WQE-by-MMIO.

We apply the following two optimizations to HERD RPCs in general; we present specific optimizations for each system later.

- **Batching** Instead of responding after detecting one request, the worker checks for one request from each of the C clients, collecting $N \leq C$ requests. Then, it SENDs N responses using a batched Doorbell.
- **Multi-queue** Each worker alternates among a tuneable number of UD queue pairs across the batched SENDs.

Note that batching does not add significant latency because we do it opportunistically [23, 21]; we do not wait for a number of requests to accumulate. We briefly discuss the latency added by batching in Section 4.2.2.

4.2 Networked sequencers

Centralized sequencers are useful building blocks for a variety of network applications, such as ordering operations in distributed systems via logical or real timestamps [11], or providing increasing offsets into a linearly growing memory region [10]. A centralized sequencer can be the bottleneck in high-performance distributed systems, so building a fast sequencer is an important step to improving whole-system performance.

Our sequence server runs on a single machine and provides an increasing 8-byte integer to client processes running on remote machines. The baseline design uses HERD RPCs. The worker threads at the server share an 8-byte counter; each client can send a sequencer request to any worker. The worker's application processing consists of atomically incrementing the shared counter by one. When Doorbell batching is enabled, we use an additional application-level optimization to reduce contention for the shared counter: after collecting N requests, a worker atomically increments the shared counter by N, thereby claiming ownership of a sequence of N consecutive integers. It then sends these N integers to the clients using a batched Doorbell (one integer per client).

	Baseline	+RPC opts	`Spec-S0`	Atomics
Throughput	26	97.2	122	2.24
Bottleneck	CPU	DMA bw	NIC	PCIe RTT

Table 3: Sequencer throughput (Mrps) and bottlenecks on CIB

Figure 7 shows the effect of batching and multi-queue on the HERD RPC-based sequencer's throughput with an increasing number of server CPU cores. We run 1 worker thread per core and use 70 client processes on 5 client machines. Batching increases single-core throughput from 7.0 million requests per second (Mrps) to 16.6 Mrps. In this mode, each core still uses 2 response UD queue pairs—one for each NIC port—and is bottlenecked by the NIC processing units handling the QPs; engaging more PUs with multi-queue (3 per-port QPs per core) increases core throughput to 27.4 Mrps. With 6 cores and both optimizations, throughput increases to 97.2 Mrps and is bottlenecked by DMA bandwidth: The DMA bandwidth limit for the batched UD SENDs used by our sequencer is 101.6 million operations/s (Section 5.2.1). At 97.2 Mrps, the sequencer is within 5% of this limit; we attribute the gap to PCIe link- and physical-layer overheads in the DMA-ed requests, which are absent in our SEND-only benchmark. When more than 6 cores are used, throughput drops because the response batch size are smaller: With 6 cores (97.2 Mrps), there are 15.9 responses per batch; with 10 cores (84 Mrps), there are 4.4 responses per batch.

4.2.1 Sequencer-specific optimizations

The above design is a straightforward adoption of general-purpose RPCs for a sequencer, and inherits the limitations of the RPC protocol. First, the connected QPs used for writing requests require state in the server's NIC and limit scalability to a few hundred RPC clients [20]. Higher scalability necessitates exclusive use of datagram transport which only supports SEND/RECV verbs. The challenge then is to use SEND/RECV instead of WRITEs for sequencer *requests* without sacrificing server performance. Second, it uses PCIe inefficiently: UD SEND work queue elements on Mellanox's NICs span $\geq$ 2 cache lines because of their 68-byte header (Table 1); sending 8 bytes of useful sequencer data requires 128 bytes (2 cache lines) to be DMA-ed by the NIC.

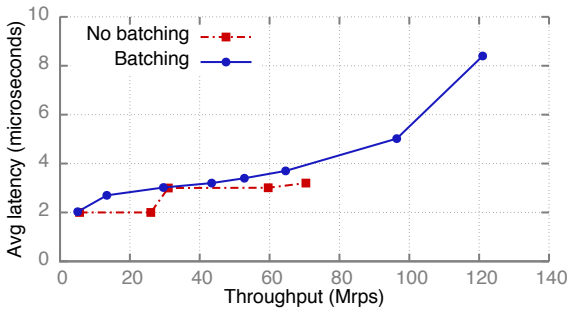


Figure 8: Impact of response batching on `Spec-S0` latency

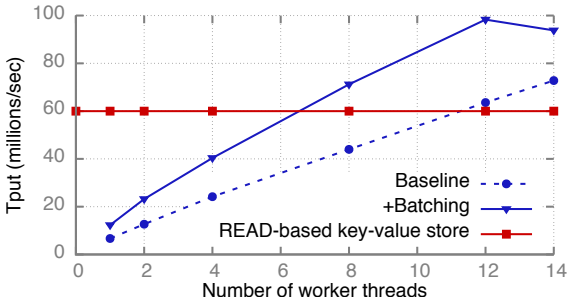


Figure 9: Improvement in HERD's throughput with 5% PUTs

We exploit the specific requirements of the sequencer to overcome these limitations. We use header-only SENDs for both requests and responses to solve both problems:

1. The client's header-only request SENDs generate header-only, single-DMA RECVs at the server (Figure 6), which are as fast as the WRITEs used earlier.
2. The server's header-only response SEND WQEs use a header field for application payload and fit in 1 cache line (Figure 5), reducing the data DMA-ed per response by 50% to 64 bytes.

Using header-only SENDs requires encoding application information in the SEND packet header; we use the 4-byte *immediate integer* field of RDMA packets [1]. Our 8-byte sequencer works around the 4-byte limit as follows: Clients speculate the 4 higher bytes of the counter and send it in a header-only SEND. If the client's guess is correct, the server sends the 4 lower bytes in a header-only SEND, else it sends the entire 8-byte value in a regular, non header-only SEND which later triggers an update of the client's guess. Only a tiny fraction ($\leq C/2^{32}$ with C clients) of SENDs are regular. We discuss this speculation technique further in Section 5.

We call this datagram-only sequencer `Spec-S0` (speculation with header-only SENDs). Figure 7 shows its throughput with increasing server CPU cores. `Spec-S0`'s DMA bandwidth limit is higher than HERD RPCs because of smaller response WQEs; it achieves 122 Mrps and is limited by the NIC's processing power instead of PCIe bandwidth. `Spec-S0` has lower single-core throughput than the HERD RPC-based sequencer because of the additional CPU overhead of posting RECVs.

4.2.2 Latency

Figure 8 shows the average end-to-end latency of `Spec-S0` with and without response batching. Both modes receive a batch of requests from the NIC; the two modes differ only in the method used to send responses. The non-batched mode sends responses one-by-one using WQE-by-MMIO whereas the batched mode uses Doorbell when multiple responses are available to send. We batch atomic increments to the shared counter in both modes. We use 10 server CPU cores, which is the minimum required to achieve peak throughput. We measure throughput with increasing client load by adding more clients, and by increasing the number of outstanding requests per client. Batching adds up to 1 µs of latency because of the additional DMA read required with the Doorbell method. We believe that the small additional latency is acceptable because of the large throughput and CPU efficiency gains from batching.

4.2.3 Atomics-based sequencers

Atomic fetch-and-add over RDMA is an appealing method to implement a sequencer: Binnig et al. [11] use this design for the timestamp server in their distributed transaction protocol. However, lock contention for the counter among the NIC's PUs results in poor performance. The effects of contention are exacerbated by the *duration* for which locks are held—several hundred nanoseconds for PCIe round trips. Our RPC-based sequencers have lower contention and shorter lock duration: the programmability of general-purpose CPUs allows us to batch updates to the counter which reduces cache line contention, and proximity to the counter's storage (i.e., core caches) makes these updates fast. Figure 7 shows the throughput of our atomics-based sequencer: it achieves only 2.24 Mrps, which is 50x worse than our optimized design, and 12.2x worse than our single-core throughput. Table 3 summarizes the performance of our sequencers.

4.3 Key-value stores

Several designs have been proposed for RDMA-based key-value storage. The HERD key-value cache uses HERD RPCs for all requests and does not bypass the remote CPU; other key-value designs bypass the remote CPU for key-value GETs (Pilaf [24] and FaRM-KV [13]), or for both GETs and PUTs (DrTM-KV [30] and Nessie [27]). Our goal here is to demonstrate how our guidelines can be used to optimize or find flaws in RDMA system designs in general; we do not compare across different key-value systems. We first present performance improvements for HERD. Then, we show how the performance of atomics-based key-value stores is adversely affected by lock contention inside NICs.

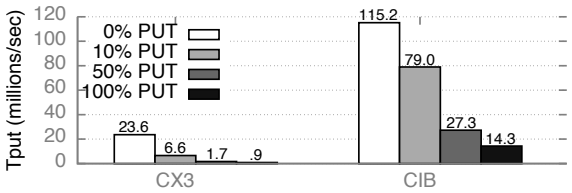


Figure 10: Throughput of emulated DrTM-KV with increasing updates.

4.3.1 Improving HERD's performance

We apply batching to HERD as follows. After collecting $N \leq C$ requests, the server-side worker performs the GET or PUT operations on the backing datastore. It uses prefetching to hide the memory latency of accesses to the storage data structures [32, 23]. Then, the worker sends the responses either one-by-one using WQE-by-MMIO, or as a single batch using Doorbell.

For evaluation, we run a HERD server with a variable number of workers on a CIB machine; we use 128 client threads running on eight client machines to issue requests. We pre-populate the key space partition owned by each worker with 8 million key-value pairs, which map 16-byte keys to 32-byte values. The workload consists of 95% GET and 5% PUT operations, with keys chosen uniformly at random from the inserted keys.

Figure 9 shows the throughput achieved in the above experiment. We also include the maximum throughput achievable by a READ-based key-value store such as Pilaf or FaRM-KV that uses ≥ 2 small READs per GET (one READ for fetching the index entry, and one for fetching the value). We compute this analytically by halving CIB's peak inbound READ throughput (Section 5.3.1). We make three observations:

- Batching improves HERD's per-core throughput by 83% from 6.7 Mrps to 12.3 Mrps. This improvement is smaller than for the sequencer because the CPU processing time saved from avoiding MMIOs is smaller relative to per-request processing in HERD than in the sequencer.
- Batching improves peak throughput by 35% from 72.8 Mrps to 98.3 Mrps. Batched throughput is bottlenecked by PCIe DMA bandwidth.
- With batching, HERD's throughput is up to 63% higher than a READ-based key-value store. While HERD's original non-batched design requires 12 cores to outperform a READ-based design, only 7 cores are needed with batching. This highlights the importance of including low-level factors such as batching when comparing RDMA system designs.

4.3.2 Atomics-based key-value stores

DrTM-KV [30] and Nessie [27, 28] use RDMA atomics to bypass the remote CPU for both GETs and PUTs. However, these projects do not consider the impact of the NIC's concurrency control on performance, and present performance for either GET-only (DrTM-KV) or GET-mostly workloads (Nessie). We now show that locking inside the NIC results in low PUT throughput, and degrades throughput even when only a small fraction of key-value operations are PUTs.

We discuss DrTM-KV here because of its simplicity, but similar observations apply to Nessie. DrTM-KV caches some fields of its key-value index at all clients; GETs for cached keys use one READ. PUT operations lock, update, and unlock key-value items; locking and unlocking is done using atomics. Running DrTM-KV's codebase on CIB requires significant modification because CIB's dual-port NIC are connected in a way that does not allow cross-port communication. To overcome this, we wrote a simplified emulated version of DrTM-KV: we emulate GETs with 1 READ and PUTs with 2 atomics, and assume a 100% cache hit rate.

Figure 10 shows the throughput of our emulated DrTM-KV server with different fractions of PUT operations in the workload. The server hosts 16 million items with 16-byte keys and 32-byte values. Clients use randomly chosen keys and we use as many clients as required to maximize throughput. Although throughput for a 100% GET workload is high, adding only 10% PUTs degrades it by 72% on CX3 and 31% on CIB. Throughput with 100% PUTs is a tiny fraction of GET-only throughput: 4% on CX3 and 12% on CIB. Note that the degradation for CIB is more gradual than for CX3 because CIB has a better locking mechanism, as shown in Section 5.

5 Low-level factors in RDMA

Our guidelines and system designs are based on an improved understanding of low-level factors that affect RDMA performance, including I/O initiation mechanisms, PCIe, and NIC architecture. These factors are complicated and there is little existing literature describing them or studying their relevance to networked systems. We attempt to fill this void by presenting clarifying performance measurements, experiments, and models; Table 4 shows a partial summary for CIB. Additionally, we discuss the importance of these factors to general RDMA system design beyond the two systems in Section 4.

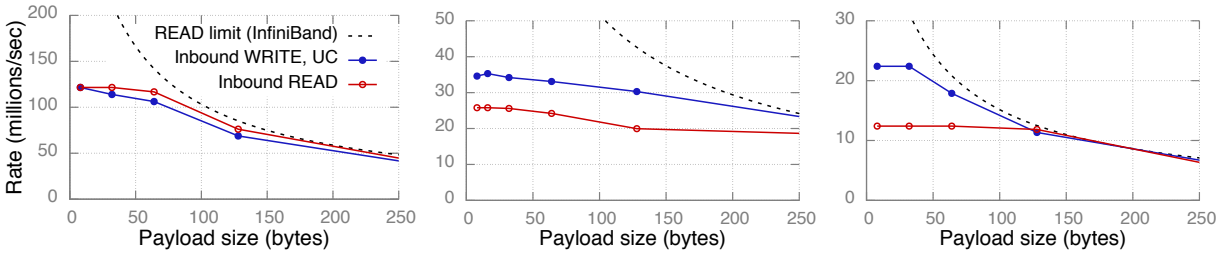
We divide our discussion into three common use cases that highlight different low-level factors: (1) batched operations, (2) non-batched operations, and (3) atomic operations. For each case, we present a performance analysis focusing on hardware bottlenecks and optimizations, and discuss implications on RDMA system design.

5.1 PCIe models

We have demonstrated that understanding the PCIe behavior is critical for improving RDMA performance. How-

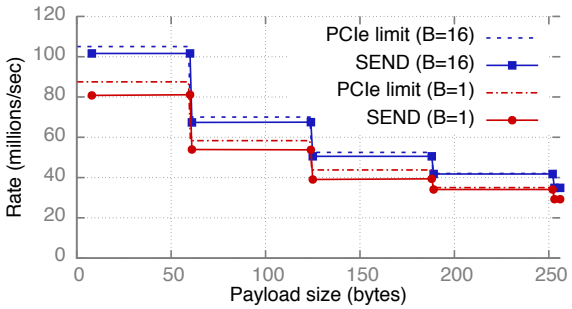
	Outbound verbs			Inbound verbs					
	UD SENDs			UD RECVs		READs		Atomics	
	Non-batch	Batch	Batch + HO	Batch	Batch + HO	≤ 64 B	128 B	$Z = 1$	$Z \geq 4096$
Rate (Mops)	80	101.6	157	82	122	121.6	76.2	2.24	52
Bottleneck	MMIO bw	DMA bw	NIC	NIC	NIC	NIC	IB bw	PCIe RTT	NIC

Table 4: Throughput and bottleneck of different modes of RDMA verbs on CIB. HO denotes the header-only optimization.

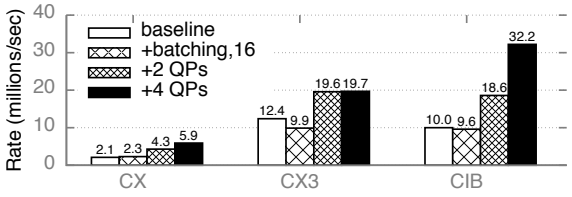


(a) CIB *(b) CX3* *(c) CX*

Figure 12: Inbound READ and UC WRITE throughput, and the *InfiniBand* limit for READs. Note the different scales for Y axes.



(a) PCIe limits for UD SEND with batch size B



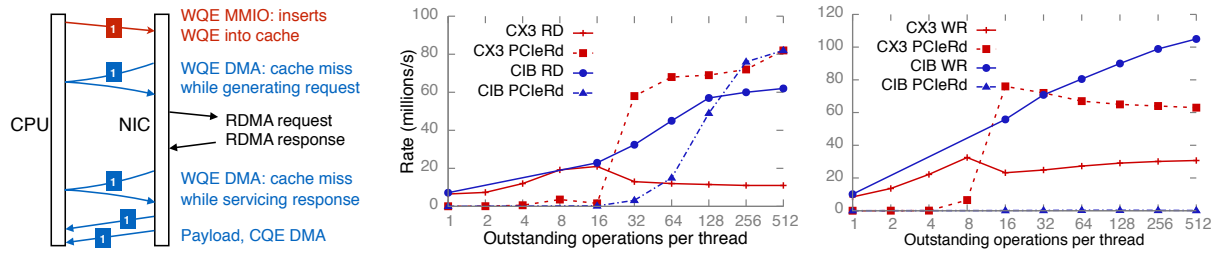
(b) Optimizations for UD SEND

Figure 11: *(a)* Peak batched and non-batched UD SEND throughput on CIB, with batch size B; dotted lines show corresponding PCIe limits. *(b)* Effect of optimizations on single-core UD SEND throughput with 60-byte payload (128-byte WQE).

ever, deriving analytical models of PCIe behavior without access to proprietary/confidential NIC manuals and our limited resources—per–cache line PCIe counters and undocumented driver software—required extensive experimentation and analysis. Our derived models are presented in a slightly simplified form at several points in this paper (Figures 5, 6, 13). The exact analytical models are complicated and depend on several factors such as the NIC, its PCIe capability, the verb and transport, the level of Doorbell batching, etc. To make our models easily accessible, we instrumented the datapath of two Mellanox drivers (ConnectX-3 and Connect-IB) to provide statistics about PCIe bandwidth use (https://github.com/efficient/rdma_bench). Our models and drivers are restricted to requester-side PCIe behavior. We omit responder-side PCIe behavior because it is the same as described in our previous work [20]: inbound READs and WRITEs generate one PCIe read and write, respectively; inbound SENDs trigger a RECV completion—we discuss the PCIe transactions for the RECV.

5.2 Batched operations

A limitation of batching on current hardware makes it useful mainly for datagram transport: all operations in a batch must use the same queue pair because Doorbells are per QP. This limitation seems fundamental to the parallel architecture of NICs: In a hypothetical NIC design where Doorbells contained information relevant for multiple queue pairs (e.g., a compact encoding of "2 and 1 new WQEs for QP 1 and QP 2, respectively"), sending the Doorbell to the NIC processing units handling these QPs would require an expensive selective broadcast inside the NIC. These PUs would then issue separate DMAs for WQEs, losing the coalescing advantage of batching. This limitation makes batching less useful for connected QPs, which provide only one-to-one communication between two machines: the chances that a process has multiple messages for the same remote machine are low in large deployments. We therefore discuss batching for UD transport only.



(a) PCIe model for reliable verbs *(b) WQE cache misses for READs* *(c) WQE cache misses for RC WRITEs*

Figure 13: PCIe model showing possible WQE cache misses, and measurement of WQE cache misses for READs and RC WRITEs.

5.2.1 UD SENDs

Figure 11 shows the throughput and PCIe bandwidth limit of batched and non-batched UD SENDs on CIB. We use one server to issue SENDs to multiple client machines. With batching, we use batches of size 16 (i.e., the NIC DMAs 16 work queue elements per Doorbell). Otherwise, the CPU writes WQEs by the WQE-by-MMIO method. We use as many cores as required to maximize throughput. Batching improves peak SEND throughput by 27% from 80 million operations/s (Mops) to 101.6 Mops.

Bottlenecks Batched throughput is limited by DMA bandwidth. For every DMA completion of size C_{rc} bytes, there is header overhead of P_c bytes (Section 2.1), leading to 13443 MB/s of useful DMA read bandwidth on CIB. As UD WQEs span at least 2 cache lines, the maximum WQE transfer rate is 13443/128 = 105 million/s, which is within 5% of our achieved throughput; we attribute the difference to link- and physical-layer PCIe overheads.

Non-batched throughput is limited by MMIO bandwidth. The write-combining MMIO rate on CIB is $(16 * P_{bw})/(64 + P_r) = 175$ million cache lines/s. UD SEND WQEs with non-zero payload span at least 2 cache lines (Table 1), and achieve up to 80 Mops. This is within 10% of the 87.5 Mops bandwidth limit.

Multi-queue optimization Figure 11b shows single-core throughput for batched and non-batched 60-byte UD SENDs—the largest payload size for which the WQEs fit in 2 cache lines. Interestingly, batching *decreases* core throughput if only one QP is used: with one QP, a core is coupled to a NIC processing unit (Section 3.3), so throughput depends on how the PU handles batched and non-batched operations. Batching has the expected effect when we break this coupling by using multiple QPs. Batched throughput increases by ~ 2x on all clusters with 2 QPs, and between 2–3.2x with 4 QPs. Non-batched (WQE-by-MMIO) throughput *does not* increase with multiple QPs (not shown in graph), showing that it is CPU-limited.

	RECV 0	RECV ≥ 1	SEND 0	SEND ≥ 1
CIB	122.0	82.0	157.0	101.6
CX3	34.0	21.8	32.1	26.0
CX	15.3	9.6	11.9	11.9

Table 5: Per-NIC rate (millions/s) for header-only (0) and regular (≥ 1) SENDs and RECVs

Design implications RDMA-based systems can often choose between CPU-bypassing and CPU-involving designs. For example, clients can access a key-value store either by READing directly from the server's memory [24, 13, 30, 28], or via RPCs as in HERD [20]. Our results show that achieving peak performance on even the most powerful NICs *does not* require a prohibitive amount of CPU power: only 4 cores are needed to saturate the fastest PCIe links. Therefore, CPU-involving designs will not be limited by CPU processing power, provided that their application-level processing permits so.

5.2.2 UD RECVs

Table 5 compares the throughput of header-only and payload-carrying regular RECVs (Figure 6). In our experiment, multiple client machines issue SENDs to one server machine that posts RECVs. On CIB, avoiding the payload DMA with header-only RECVs increases throughput by 49% from 82 Mops to 122 Mops, and makes them as fast as inbound WRITEs (Figure 12a).[4] Table 5 also compares header-only and regular SENDs (Figure 5). Header-only SENDs use single-cache line WQEs and achieve 54% higher throughput.

Design implications Developers avoid RECVs at performance critical machines as a rule of thumb, favoring the faster READ/WRITE verbs [24, 20]. Our work provides the exact reason: RECVs are slow due to the CQE DMA; they are as fast as inbound WRITEs if it is avoided.

Speculation Current RDMA implementations allow 4 bytes of application data in the packet header of header-

[4]Inline-receive improves regular RECV throughput from 22 Mops to 26 Mops on CX3, but is not yet supported for UD on CIB.

only SENDs. For applications that require larger messages, header-only SEND/RECV can be used if speculation is possible; we demonstrated such a design for an 8-byte sequencer in Section 4.2. In general, speculation works as follows: clients transmit their expected response along with requests, and get a small confirmation response in the common case. For example, in a key-value store with client-side caching, clients can send GET requests with the key and its cached version number (using a WRITE or regular SEND). The server replies with a header-only "OK" SEND if the version is valid.

There are applications for which 4 bytes of per-message data suffices. For example, some database tables in the TPC-C [29] benchmark have primary key size between 2 and 3 bytes. A table access request can be sent using a header-only SEND (using the remaining 1 byte to specify the table ID), while the response may need a larger SEND.

5.3 Non-batched operations

5.3.1 Inbound READs and WRITEs

Figure 12 shows the measured throughput of inbound READs and UC WRITEs, and the InfiniBand bandwidth limit of inbound READs. We do not show the InfiniBand limit for WRITEs and the PCIe limits as they are higher.

Bottlenecks On our clusters, inbound READs and WRITEs are initially bottlenecked by the NIC's processing power, and then by InfiniBand bandwidth. The payload size at which bandwidth becomes a bottleneck depends on the NIC's processing power relative to bandwidth. For READs, the transition point is approximately 128 bytes, 256 bytes, and 64 bytes for CX, CX3, and CIB, respectively. CIB NICs are powerful enough to saturate 112 Gbps with 64-byte READs, whereas CX3 NICs require 256-byte READs to saturate 56 Gbps.

Implications The transition point is an important factor for systems that make tradeoffs between the size and number of READs: For key-value lookups of small items (~32 bytes), FaRM's key-value store [13] can use one large (~256-byte) READ. In a client-server design where inbound READs determine GET performance, this design performs well on CX3 because 32- and 256-byte READs have similar throughput; other designs such as DrTM-KV [30] and Pilaf [24] that instead use 2–3 small READs may provide higher throughput on CIB.

5.3.2 Outbound READs and WRITEs

For brevity, we only present a summary of the performance of non-batched outbound operations on CIB. Outbound UC WRITEs larger than 28 bytes, i.e., WRITEs with WQEs spanning more than one cache line (Table 1), achieve up to 80 Mops and are bottlenecked by PCIe MMIO throughput, similar to non-batched outbound SENDs (Figure 11a). READs achieve up to 88 Mops and

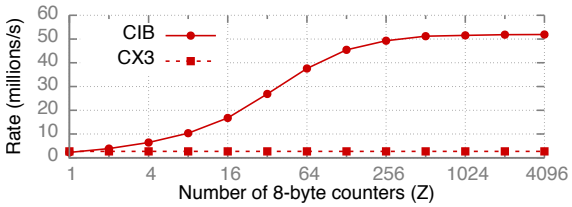


Figure 14: Atomics throughput with increasing concurrency

are bottlenecked by NIC processing power.

Achieving high outbound throughput requires maintaining multiple outstanding requests via pipelining. When the CPU initiates an RDMA operation, the work queue element is inserted into the NIC's WQE cache. However, if the CPU injects new WQEs faster than the NIC's processing speed, this WQE can be evicted by newer WQEs. This can cause cache misses when the NIC eventually processes this WQE while generating its RDMA request packets, while servicing its RDMA response, or both. Figure 13a summarizes this model.

To quantify this effect, we conduct the following experiment on CIB: 14 requester threads on a server issue windows of N 8-byte READs or WRITEs over reliable transport to 14 remote processes. In Figures 13b and 13c, we show the cumulative RDMA request rate, and the extent of WQE cache misses using the `PCIeRdCur` counter rate. Each thread waits for the N requests to complete before issuing the next window. We use all 14 cores on the server to generate the maximum possible request rate, and RC transport to include cache misses generated while processing ACKs for WRITEs. We make the following observations, showing the importance of the WQE cache in improving and understanding RDMA throughput:

- The optimal window size for maximum throughput is not obvious: throughput does not always increase with increasing window size, and is dependent on the NIC. For example, $N = 16$ and $N = 512$ maximize READ throughput on CX3 and CIB respectively.
- Higher RDMA throughput may be obtained at the cost of PCIe reads. For example, on CIB, both READ throughput and PCIe read rate increases as N increases. Although the largest N is optimal for a machine that only issues outbound READs, it may be suboptimal if it also serves other operations.
- CIB's NIC can handle the CPU's peak WQE injection rate for WRITEs and never suffers cache misses. This is not true for READs, indicating that they require more NIC processing than reliable WRITEs.

5.4 Atomic operations

NIC processing units contend for locks during atomic operations (Section 3.4). The performance of atomics depends on the amount of parallelism in the workload with respect to the NIC's internal locking scheme. To vary

the amount of parallelism, we create an array of Z 8-byte counters in a server's memory, and multiple remote client processes issue atomic operations on counters chosen randomly at each iteration. Figure 14 shows the total client throughput in this experiment. For CX3, it remains 2.7 Mops irrespective of Z; for CIB, it rises to 52 Mops.

Inferring the locking mechanism The flatness of CX3's throughput graph indicates that it serializes all atomic operations. For CIB, we measured performance with randomly chosen pairs of addresses and observed lower performance for pairs where both addresses have the same 12 LSBs. This strongly suggests that CIB uses 4096 buckets to slot atomic operations by address—a new operation waits until its slot is empty.

Bottlenecks and implications Throughput on CX3 is limited by PCIe latency because of serialization. For CIB, buffering and computation needed for PCIe read-modify-write makes NIC processing power the bottleneck.

The abysmal throughput for $Z = 1$ on both NICs reaffirms that atomics are a poor choice for a sequencer; our optimized sequencer in Section 4 provides 12.2x higher performance with a *single* server CPU core. A lock service for data stores, however, might use a larger Z. Atomics could perform well if such an application used CIB, but they are very slow with CX3, which is the NIC used in prior work [27, 30]. With CIB, careful lock placement is still necessary. For example, if page-aligned data records have their lock variables at the same offset in the record, all lock requests will have the same 12 LSBs and will get serialized. A deterministic scheme that places the lock at different offsets in different records, or a scheme that keeps locks separate from the data will perform better.

6 Related work

High-performance RDMA systems Designing high-performance RDMA systems is an active area of research. Recent advances include several key-value storage systems [24, 13, 20, 30, 28] and distributed transaction processing systems [30, 12, 14, 11]. A key design decision in each of these systems is the choice of verbs, made using a microbenchmark-based performance comparison. Our work shows that there are more dimensions to these comparisons than these projects explore: two verbs cannot be exhaustively compared without exploring the space of low-level factors and optimizations, each of which can offset verb performance by several factors.

Low-level factors in network I/O Although there is a large body of work that measures the throughput and CPU utilization of network communication [18, 16, 26, 13, 20], there is less existing literature on understanding the low-level behavior of network cards. NIQ [15] presents a high-level picture of the PCIe interactions between an Ethernet NIC and CPUs, but does not discuss the more subtle interactions that occur during batched transfers. Lee et al. [22] study the PCIe behavior of Ethernet cards using a PCIe protocol analyzer, and divide the PCIe traffic into Doorbell traffic, Ethernet descriptor traffic, and actual data traffic. Similarly, analyzing RDMA NICs using a PCIe analyzer may reveal more insights into their behavior than what is achievable using PCIe counters.

7 Conclusion

Designing high-performance RDMA systems requires a deep understanding of low-level RDMA details such as PCIe behavior and NIC architecture: our best sequencer is ~50x faster than an existing design and scales perfectly, our optimized HERD key-value store is up to 83% faster than the original, and our fastest transmission method is up to 3.2x faster than the commonly-used baseline. We believe that by presenting clear guidelines, significant optimizations based on these guidelines, and tools and experiments for low-level measurements on their hardware, our work will encourage researchers and developers to develop a better understanding of RDMA hardware before using it in high-performance systems.

Acknowledgments We are tremendously grateful to Joseph Moore and NetApp for providing access to the CIB cluster. We thank Hyeontaek Lim and Sol Boucher for providing feedback, and Liuba Shrira for shepherding. Emulab [31] and PRObE [17] resources were used in our experiments. PRObE is supported in part by NSF awards CNS-1042537 and CNS-1042543 (PRObE). This work was supported by funding from the National Science Foundation under awards 1345305 and 1314721, and by Intel via the Intel Science and Technology Center for Cloud Computing (ISTC-CC).

Appendix A. WQE-by-MMIO and Doorbell PCIe use

We denote the doorbell size by d. The total data transmitted from CPU to NIC with the WQE-by-MMIO method is $T_{bf} = 10 * (\lceil 65/64 \rceil * (64 + P_r))$ bytes. With cache line padding, 65-byte WQEs are laid out in 128-byte slots in host memory; assuming $C_{rc} = 128$, $T_{db} = (d+P_r)+(10*(128+P_c))$ bytes. We ignore the PCIe link-layer traffic since it is small compared to transaction-layer traffic: it is common to assume 2 link-layer packets (1 flow control update and 1 acknowledgment, both 8 bytes) per 4-5 TLPs [9], making the link-layer overhead < 5%. Substituting $d = 8$ gives $T_{bf} = 1800$, and $T_{db} = 1534$.

References

[1] Infiniband architecture specification volume 1. https://cw.infinibandta.org/document/dl/7859.

[2] Intel Atom Processor C2000 Product Family for Microserver. http://www.intel.in/content/dam/www/public/us/en/documents/datasheets/atom-c2000-microserver-datasheet.pdf.

[3] Intel Xeon Processor E5-1600/2400/2600/4600 v3 Product Families. http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xeon-e5-v3-datasheet-vol-2.pdf.

[4] Intel Xeon Processor E5-1600/2400/2600/4600 (E5-Product Family) Product Families. http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xeon-e5-1600-2600-vol-2-datasheet.pdf.

[5] Intel Xeon Processor D-1500 Product Family. http://www.intel.in/content/dam/www/public/us/en/documents/product-briefs/xeon-processor-d-brief.pdf.

[6] Intel Xeon Phi Processor Knights Landing Architectural Overview. https://www.nersc.gov/assets/Uploads/KNL-ISC-2015-Workshop-Keynote.pdf.

[7] Mellanox ConnectX-4 product brief. http://www.mellanox.com/related-docs/prod_silicon/PB_ConnectX-4_VPI_Card.pdf.

[8] Mellanox OFED for linux user manual. http://www.mellanox.com/related-docs/prod_software/Mellanox_OFED_Linux_User_Manual_v2.2-1.0.1.pdf.

[9] Understanding Performance of PCI Express Systems. http://www.xilinx.com/support/documentation/white_papers/wp350.pdf.

[10] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobber, M. Wei, and J. D. Davis. CORFU: a shared log design for flash clusters. In *Proc. 9th USENIX NSDI*, Apr. 2012.

[11] C. Binnig, U. Çetintemel, A. Crotty, A. Galakatos, T. Kraska, E. Zamanian, and S. B. Zdonik. The end of slow networks: It's time for a redesign. *CoRR*, abs/1504.01048, 2015. URL http://arxiv.org/abs/1504.01048.

[12] Y. Chen, X. Wei, J. Shi, R. Chen, and H. Chen. Fast and general distributed transactions using RDMA and HTM. In *Proc. 11th ACM European Conference on Computer Systems (EuroSys)*, Apr. 2016.

[13] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast remote memory. In *Proc. 11th USENIX NSDI*, Apr. 2014.

[14] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Proc. 25th ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2015.

[15] M. Flajslik and M. Rosenblum. Network interface design for low latency request-response protocols. In *Proc. USENIX Annual Technical Conference*, June 2013.

[16] S. Gallenmüller, P. Emmerich, F. Wohlfart, D. Raumer, and G. Carle. Comparison of frameworks for high-performance packet io. In *ANCS*, 2015.

[17] G. Gibson, G. Grider, A. Jacobson, and W. Lloyd. PRObE: A Thousand-Node Experimental Cluster for Computer Systems Research.

[18] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: a GPU-accelerated software router. In *Proc. ACM SIGCOMM*, Aug. 2010.

[19] S. Hauger, T. Wild, A. Mutter, A. Kirstaedter, K. Karras, R. Ohlendorf, F. Feller, and J. Scharf. Packet processing at 100 Gbps and beyond - challenges and perspectives. In *Photonic Networks, 2009 ITG Symposium on*, 2009.

[20] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA efficiently for key-value services. In *Proc. ACM SIGCOMM*, Aug. 2014.

[21] A. Kalia, D. Zhou, M. Kaminsky, and D. G. Andersen. Raising the bar for using GPUs in software packet processing. In *Proc. 12th USENIX NSDI*, May 2015.

[22] S. Larsen and B. Lee. Platform io dma transaction acceleration. In *CACHES*. ACM, 2011.

[23] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *Proc. 11th USENIX NSDI*, Apr. 2014.

[24] C. Mitchell, Y. Geng, and J. Li. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *Proc. USENIX Annual Technical Conference*, June 2013.

[25] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *Proc. 10th USENIX NSDI*, Apr. 2013.

[26] L. Rizzo. netmap: a novel framework for fast packet I/O. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, June 2012.

[27] T. Szepesi, B. Wong, B. Cassell, , and T. Brecht.

Designing a low-latency cuckoo hash table for write-intensive workloads. In *WSRC*, 2014.

[28] T. Szepesi, B. Cassell, B. Wong, T. Brecht, and X. Liu. Nessie: A decoupled, client-driven, key-value store using RDMA. Technical Report CS-2015-09, University of Waterloo, David R. Cheriton School of Computer Science, Waterloo, Canada, June 2015.

[29] TPC-C. TPC benchmark C. http://www.tpc.org/tpcc/.

[30] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, 2015.

[31] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. 5th USENIX OSDI*, pages 255–270, Dec. 2002.

[32] D. Zhou, B. Fan, H. Lim, D. G. Andersen, and M. Kaminsky. Scalable, High Performance Ethernet Forwarding with CuckooSwitch. In *Proc. 9th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, Dec. 2013.