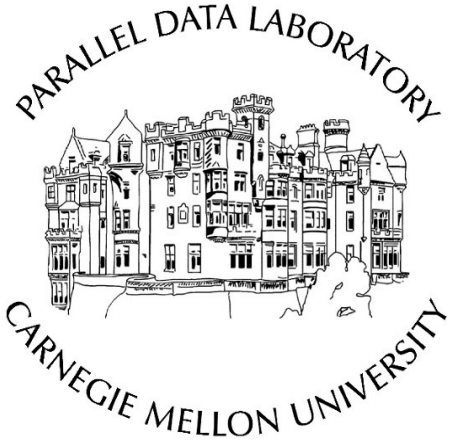


Using RDMA Efficiently for Key-Value Services

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

CMU-PDL-14-106
June 2014

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

This paper describes the design and implementation of HERD, a key-value system designed to make the best use of an RDMA network. Unlike prior RDMA-based key-value systems, HERD focuses its design on reducing network round trips while using efficient RDMA primitives; the result is substantially lower latency, and throughput that saturates modern, commodity RDMA hardware.
HERD has two unconventional decisions: First, it does not use RDMA reads, despite the allure of operations that bypass the remote CPU entirely. Second, it uses a mix of RDMA and messaging verbs, despite the conventional wisdom that the messaging primitives are slow. A HERD client writes its request into the server's memory; the server computes the reply. This design uses a single round trip for all requests and supports up to 19.8 million key-value operations per second with 11 µs average latency. Notably, for small key-value items, our full system throughput is similar to native RDMA read throughput and is over 2X higher than recent RDMA-based key-value systems. We believe that HERD further serves as an effective template for the construction of RDMA-based datacenter services.

Acknowledgements: This work was supported by funding from the National Science Foundation under awards CNS-1314721 and CCF-0964474, and Intel via the Intel Science and Technology Center for Cloud Computing (ISTC-CC). The PRObE cluster [11] used for many experiments is supported in part by NSF awards CNS-1042537 and CNS-1042543 (PRObE). We would like to thank the members and companies of the PDL Consortium (including Actifio, American Power Conversion, EMC Corporation, Emulex, Facebook, Fusion-io, Google, Hewlett-Packard Labs, Hitachi, Huawei Technologies Co., Intel Corporation, Microsoft Research, NEC Laboratories, NetApp, Inc., Oracle Corporation, Panasas, Riverbed, Samsung Information Systems America, Seagate Technology, STEC, Inc., Symantec Corporation, VMware, Inc., and Western Digital) for their interest, insights, feedback, and support.

Keywords: RDMA, InfiniBand, RoCE, Key-Value Stores

1 Introduction

This paper explores a question that has important implications for the design of modern clustered systems: What is the best method for using RDMA features to support remote hash-table access? To answer this question, we first evaluate the performance that, with sufficient attention to engineering, can be achieved by each of the RDMA communication primitives. Using this understanding, we show how to use an unexpected combination of methods and system architectures to achieve the maximum performance possible on a high-performance RDMA network.

Our work is motivated by the seeming contrast between the fundamental time requirements for cross-node traffic vs. CPU-to-memory lookups, and the designs that have recently emerged that use multiple RDMA (remote direct memory access) reads. On one hand, going between nodes takes roughly $1-3\mu$s, compared to $60-120ns$ for a memory lookup, suggesting that a multiple-RTT design as found in the recent Pilaf [20] and FaRM [8] systems should be fundamentally slower than a single-RTT design. But on the other hand, an RDMA read bypasses many potential sources of overhead, such as servicing interrupts and initiating control transfers, which involve the host CPU. In this paper, we show that there is a better path to taking advantage of RDMA to achieve high-throughput, low-latency key-value storage.

A challenge for both our and prior work lies in the lack of richness of RDMA operations. An RDMA operation can only read or write a remote memory location. It is not possible to do more sophisticated operations such as dereferencing and following a pointer in remote memory. Recent work in building key-value stores [20, 8] has exclusively focused on using RDMA reads to traverse remote data structures in a similar fashion to traversing the structure in local memory. This approach invariably requires multiple round trips across the network.

Consider an ideal RDMA read-based key-value store (or cache) where each `GET` request requires only 1 small RDMA read. Designing such a store is as hard as designing a hash-table in which each `GET` request requires only one random memory lookup. We instead provide a solution to a simpler problem: we design a key-value cache that provides performance identical to that of the ideal cache. However, our design does not use RDMA reads at all.

In this paper, we present HERD, a key-value cache that leverages RDMA features to deliver low latency and high throughput. As we demonstrate later, RDMA reads cannot use the full performance of the RDMA hardware whereas RDMA writes can. In HERD, clients write their request to the server's memory using an RDMA write. The server's CPU polls its memory for incoming requests. On receiving a new request, it executes the `GET` or `PUT` operation in its local data structures and sends the response back to the client. As RDMA write performance does not scale with the number of outbound connections, the response is sent as a SEND message over a datagram connection.

Our work makes three main contributions:

- We provide a thorough analysis of the performance of RDMA verbs and expose the various design options for key-value systems.
- We demonstrate that "two-sided" verbs are better than RDMA reads for key-value systems, refuting the previously held assumption [20, 8].
- We describe the design and implementation of HERD, a key-value cache that offers the maximum possible performance of RDMA hardware.

The rest of this paper is organized as follows. Section 2 provides a brief introduction to key-value stores and RDMA, and describes recent efforts in building key-value stores using RDMA. Section 3 discusses the rationale behind our design decisions and demonstrates that messaging verbs are a better choice than RDMA reads for key-value systems. Section 4 discusses the design and implementation of our key-value cache. In Section 5, we evaluate our system on a cluster of 18 nodes and compare it against FaRM [8] and Pilaf [20].

2 Background

This section provides background information on key-value stores and caches, which are at the heart of HERD. We then provide an overview of RDMA, as is relevant for the rest of the paper.

2.1 Key-Value stores

DRAM-based key-value stores and caches are widespread in large-scale Internet services. They are used both as primary stores (e.g., Redis [4] and RAMCloud [22]), and as caches in front of backend databases (e.g., Memcached [5]). At their most basic level, these systems export a traditional `GET`/`PUT`/`DELETE` interface. Internally, they use a variety of data structures to provide fast, memory-efficient access to their underlying data (e.g., hash table or tree-based indexes).

In this paper, we focus on the communication architecture to support both of these applications; we use a cache implementation for end-to-end validation of our resulting design.

Although recent in-memory object stores have used both tree and hash table-based designs, this paper focuses on hash tables as the basic indexing data structure. Hash table design has a long and rich history, and the particular flavor one chooses depends largely on the desired optimization goals. In recent years, several systems have used advanced hash table designs such as Cuckoo hashing [23, 16, 9] and Hopscotch hashing [12]. Cuckoo hash tables are an attractive choice for building fast key-value systems [9, 29, 16] because, with K hash functions (usually, K is 2 or 3), they require only K memory lookups for `GET` operations, plus an additional pointer dereference if the values are not stored in the table itself. In many workloads, `GET`s constitute over 95% of the operations [6, 21]. This property makes cuckoo hashing an attractive backend for an RDMA-based key-value store [20]. Cuckoo and Hopscotch-based designs often emphasize workloads that are read-intensive: `PUT` operations require moving values within the tables. We evaluate both balanced (50% `PUT`/`GET`) and read-intensive (95% `GET`) workloads in this paper.

To support both types of workloads without being limited by the performance of currently available data structure options, HERD internally uses a *cache* data structure that can evict items when it is full. Our focus, however, is on the network communication architecture—our results generalize across both caches and stores, so long as the implementation is fast enough that a high-performance communication architecture is needed. HERD's cache design is based on the recent MICA [17] system that provides both cache and store semantics. MICA's cache mode uses a lossy associative index to map keys to pointers, and stores the values in a circular log that is memory efficient, avoids fragmentation, and does not require expensive garbage collection. This design requires only 2 random memory accesses for both `GET` and `PUT` operations.

2.2 RDMA

Remote Direct Memory Access (RDMA) allows one computer to directly access the memory of a remote computer without involving the operating system at any host. This enables zero-copy transfers, reducing latency and CPU overhead. In this work, we focus on two types of RDMA-providing interconnects: InfiniBand and RoCE (RDMA over Converged Ethernet). However, we believe that our design is applicable to other RDMA providers such as iWARP, Quadrics, and Myrinet.

InfiniBand is a switched fabric network widely used in high-performance computing systems. RoCE is a relatively new network protocol that allows direct memory access over Ethernet. InfiniBand and RoCE NICs achieve low latency by implementing several layers of the network stack (transport layer through physical layer) in hardware, and by providing RDMA and kernel-bypass. In this section, we provide an overview of RDMA features and terminology that are used in the rest of this paper.

2.2.1 Comparison with classical Ethernet

To distinguish from RoCE, we refer to non-RDMA providing Ethernet networks as "classical Ethernet." Unlike classical Ethernet NICs, RDMA NICs (RNICs) provide reliable delivery to applications by employing hardware-based retransmission of lost packets. Further, RNICs provide kernel bypass for all communication.

These two factors reduce end-to-end latency as well as the CPU load on the communicating hosts. The typical end-to-end ($\frac{1}{2}$RTT) latency in InfiniBand/RoCE is 1 μs while that in modern classical Ethernet-based solutions [2, 17] is 10 μs. A large portion of this gap arises because of differing emphasis in the NIC design. RDMA is increasing its presence in datacenters as the hardware becomes cheaper [20]. A 40 Gbps ConnectX-3 RNIC from Mellanox costs about $500, while a 10 Gbps Ethernet adapter costs between $300 and $800. The introduction of RoCE will further boost RDMA's presence as it will allow sockets applications to run with RDMA applications on the same network.

2.2.2 Verbs and queue pairs

Userspace programs access RNICs directly using functions called *verbs*. There are several types of verbs. Those most relevant to this work are RDMA read (READ), RDMA write (WRITE), SEND, and RECEIVE. Verbs are posted by applications to queues that are maintained inside the RNIC. Queues always exist in pairs: a *send queue* and a *receive queue* form a *queue pair* (QP). Each queue pair has an associated *completion queue* (CQ), which the RNIC fills in upon completion of verb execution.

The verbs form a semantic definition of the interface provided by the RNIC. There are two types of verbs semantics: memory semantics and channel semantics.

Memory semantics: The RDMA verbs (READ and WRITE) have memory semantics: they specify the remote memory address to operate upon. These verbs are *one-sided*: the responder's CPU is unaware of the operation. This lack of CPU overhead at the responder makes one-sided verbs attractive. Furthermore, they have the lowest latency and highest throughput among all verbs.

Channel semantics: SEND and RECEIVE (RECV) have channel semantics, i.e., the SEND's payload is written to a remote memory address that is specified by the responder in a pre-posted RECV. An analogy for this would be an unbuffered sockets implementation that required `read()` to be called before the packet arrived. SEND and RECV are *two-sided* as the CPU at the responder needs to post a RECV in order for an incoming SEND to be processed. Unlike the memory verbs, the responder's CPU is involved. Two-sided verbs also have slightly higher latency and lower throughput than one sided verbs and have been regarded unfavorably for designing key-value systems [20, 8].

Although SEND and RECV verbs are technically RDMA verbs, we distinguish them from READ and WRITE. We refer to READ and WRITE as *RDMA verbs*, and refer to SEND and RECV as *messaging verbs*.

Verbs are usually posted to the send queue of a QP (except RECV, which is posted to the receive queue). To post a verb to the RNIC, an application calls into the userland RDMA driver. Then, the driver prepares a Work Queue Element (WQE) in the host's memory and rings a doorbell on the RNIC via Programmed IO (PIO). For ConnectX and newer RNICs, the doorbell contains the entire WQE [26]. For WRITE and SEND verbs, the WQE is associated with a payload that needs to be sent to the remote host. A payload up to the maximum PIO size (256 in our setup) can be *inlined* in the WQE, otherwise it can be fetched by the RNIC via a DMA read. An inlined post involves no DMA operations, reducing latency and increasing throughput for small payloads.

When the RNIC completes the network steps associated with the verb, it pushes a completion event to the queue pair's associated completion queue (CQ) via a DMA write. Using completion events adds extra overhead to the RNIC's PCIe bus. This overhead can be reduced by using *selective signaling*. When using a selectively signaled send queue of size S, up to $S-1$ consecutive verbs can be *unsignaled*, i.e., a completion event will not be pushed for these verbs. The receive queue cannot be selectively signaled. As S is large ($\sim$ 128), we use the terms "selective signaling" and "unsignaled" interchangeably.

2.2.3 Transport types

RDMA transports can be connected or unconnected. A connected transport requires a connection between two queue pairs that communicate exclusively with each other. Current RDMA implementations support two main types of connected transports: Reliable Connection (RC) and Unreliable Connection (UC). There is no

Verb	RC	UC	UD
SEND/RECV	✓	✓	✓
WRITE	✓	✓	✗
READ	✓	✗	✗

Table 1: Operations supported by each connection type. UC does not support READs, and UD does not support RDMA at all.

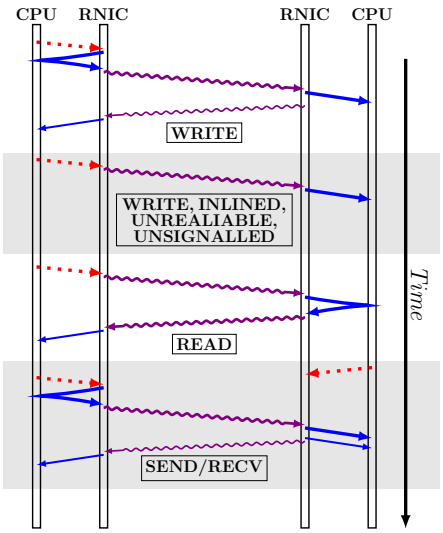


Figure 1: Steps involved in posting verbs. The dotted arrows are PCIe PIO operations. The solid, straight arrows are DMA operations: the thin ones are for writing the completion events. The thick wavy arrows are RDMA data packets and the thin ones are ACKs.

acknowledgement of packet reception in UC; packets can be lost and the affected message can be dropped. As UC does not generate ACK/NAK packets, it causes less network traffic than RC.

In an unconnected transport, one queue pair can communicate with any number of other queue pairs. Current implementations provide only one unconnected transport: Unreliable Datagram (UD). The RNIC maintains state for each active queue in its queue pair context cache, so datagram transport can scale better for applications with a one-to-many topology.

InfiniBand and RoCE employ lossless link-level flow control, namely, credit-based flow control and Priority Flow Control. *Even with unreliable transports (UC/UD), packets are never lost due to buffer overflows*. Reasons for packet loss include bit errors on the wire and hardware failures, which are extremely rare. Therefore, our design, similar to choices made by Facebook and others [21], sacrifices transport-level retransmission for fast common case performance at the cost of rare application-level retries.

Some transport types support only a subset of the available verbs. Table 1 lists the verbs supported by each transport type. Figure 1 shows the DMA and network steps involved in posting verbs.

2.3 Existing RDMA-based key-value stores

Pilaf [20] is a key-value store that aims for high performance and low CPU use. For `GET`s, clients access a cuckoo hash table at the server using READs, which requires 2.6 round trips on average for single `GET` request. For `PUT`s, clients send their requests to the server using a SEND message. To ensure consistent `GET`s in the presence of concurrent `PUT`s, Pilaf's data structures are *self-verifying*: each hash table entry is augmented with two 64-bit checksums.

The second key-value store we compare against is based upon the store designed in **FaRM** [8]. It is important to note that FaRM is a more general-purpose distributed computing platform that exposes memory of a cluster of machines as a shared address space; we compare *only* against a key-value store implemented on top of FaRM that we call FaRM-KV. Unlike the client-server design in Pilaf and HERD, FaRM is symmetric, befitting its design as a cluster architecture: each machine acts as both a server and client.

FaRM's design provides two components for comparison. First is its key-value store design, which uses a variant of Hopscotch hashing [12] to create a locality-aware hash table. For `GET`s, clients READ several consecutive Hopscotch slots, one of which contains the key with high probability. Another READ is required to fetch the value if it is not stored inside the hash table. For `PUT`s, clients WRITE their request to a circular buffer in the server's memory. The server polls this buffer to detect new requests. This design is not specific to FaRM—we use it merely as an extant alternative to Pilaf's Cuckoo-based design to provide a more in-depth comparison for HERD.

The second important aspect of FaRM is its symmetry; here it differs from both Pilaf and HERD. For small, fixed-size key-value pairs, FaRM can "inline" the value with the key. With inlining, FaRM's RDMA read-based design still achieves lower maximum *throughput* than HERD, but it uses less CPU. This tradeoff may be right for a cluster where all machines are also busy doing computation; we do not evaluate the symmetric use case here, but it is an important consideration for users of either design.

3 Design Decisions

Towards our goal of supporting key-value servers that achieve the highest possible throughput with RDMA, we explain in this section the reasons we choose to use—and not use—particular RDMA features and other design options. To begin with, we present an analysis of the performance of the RDMA verbs; we then craft a communication architecture using the fastest among them that can support our application needs.

As hinted in Section 1, one of the core decisions to make is whether to use memory verbs (RDMA read and write) or messaging verbs (SEND and RECV). Recent work from the systems and networking communities, for example, has focused on RDMA reads as a building block, because they bypass the remote network stack and CPU entirely for `GET`s [20, 8]. In contrast, however, the HPC community has made wider use of messaging, both for key-value caches [14] and general communication [15]. These latter systems scaled to thousands of machines, but provided low throughput—less than one million operations per second in memcached [14]. The reason for low throughput in [14] is not clear, but we suspect application design that makes the system incapable of leveraging the full power of the RNICs.

There remains an important gap between these two lines of work, and to our knowledge, HERD is the first system to provide the best of both worlds: throughput even higher than that of the RDMA-based systems while scaling to several hundred clients.

HERD takes a hybrid approach, using both RDMA and messaging to best effect. RDMA reads, however, are unattractive because of the need for multiple round trips. In HERD, clients instead write their requests to the server using RDMA writes over an Unreliable Connection (UC). This write places the `PUT` or `GET` request into a per-client memory region in the server. The server polls these regions for new requests. Upon receiving one, the server process executes in conventional fashion using its local data structures. It then sends a reply to the client using messaging verbs: a SEND over an Unreliable Datagram.

To explain why we use this hybrid of RDMA and messaging, we describe the performance experiments and analysis that support it. Particularly, we describe why we prefer using RDMA writes instead of reads, not taking advantage of hardware retransmission by opting for unreliable transports, and using messaging verbs despite conventional wisdom that they are slower than RDMA.

3.1 Notation and experimental setup

In the rest of this paper, we refer to an RDMA read as READ and to an RDMA write as WRITE. The machines in our experiments are quad socket, 16 core AMD Opteron 6272 @2.4 GHz. The machines have

a single port Mellanox ConnectX-3 InfiniBand card and a ConnectX-3 RoCE card connected via PCIe 2.0 x8. We use only the CPU socket that is directly connected to the RNIC. We show the results of our microbenchmarks only for InfiniBand because the values for RoCE are very similar. We show the results for RoCE in our full system experiments.

These experiments use one server machine and several client machines. We denote the server machine by M_S and its RNIC by $RNIC_S$. Client machine i is denoted by C_i. The server and client machines may run multiple server and client processes respectively. We call a message from client to server a *request*, and the reply from server to client, a *response*. The host issuing a verb is the *requester* and the destination host *responder*. For unsignaled SEND and WRITE over UC, the destination host does not actually send a response, but we still call it a responder.

For throughput experiments, processes maintain a window of several outstanding verbs in their send queues. Using windows allows us to saturate our RNICs with fewer processes. In all of our throughput experiments, we manually tune the window size for maximum aggregate throughput.

3.2 Using WRITE instead of READ

There are several benefits to using WRITE instead of READ. WRITEs can be performed over the UC transport, which itself confers several performance advantages. Because the responder does not need to send packets back, its RNIC performs less processing, and thus can support higher throughput than with READs. The reduced network bandwidth similarly benefits both the server and client throughput. Finally, as one might expect, the latency of an unsignaled WRITE is about half that ($\frac{1}{2}$ RTT) of a READ. This makes it possible to replace one READ by two WRITEs, one client-to-server and one server-to-client (forming an application-level request-reply pair), without increasing latency significantly.

3.2.1 WRITEs have lower latency than READs

Measuring the latency of an unsignaled WRITE is not straightforward as the requester gets no indication of completion. Therefore, we measure it indirectly by measuring the latency of an *ECHO*. In an ECHO, a client transmits a message to a server and the server relays the same message back to the client. If the ECHO is realized by using unsignaled WRITEs, the latency of an unsignaled WRITE is at most one half of the ECHO's latency.

We also measure the latency of signaled READ and WRITE operations. As these operations are signaled, we use the completion event to measure latency. For WRITE, we also measure the latency with payload inlining.

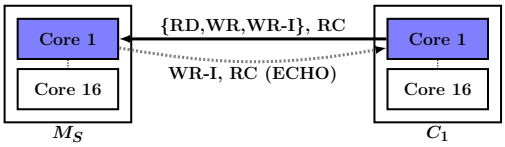
Figure 2 shows the average latency from these measurements. We use inlined and unsignaled WRITEs for ECHOs. On our RNICs, the maximum size of the inlined payload is 256 bytes. Therefore, the graphs for WR-INLINE and ECHO are only shown up to 256 bytes.

Unsignaled verbs: For payloads up to 64 bytes, the latency of ECHOs is close to READ latency, which confirms that the one-way WRITE latency is about half of the READ latency. For larger ECHOs, the latency increases because of the time spent in writing to the RNIC via PIO.

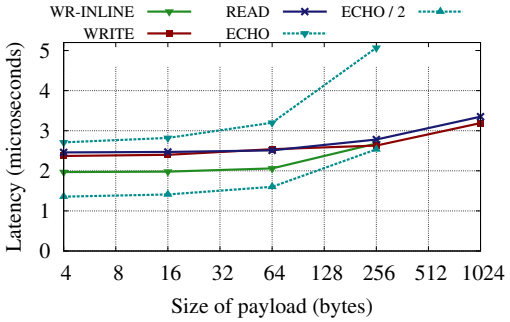
Signaled verbs: The solid lines in Figure 2 show the latencies for three signaled verbs—WRITE, READ, and WRITE with inlining (WR-INLINE). The latencies for READ and WRITE are similar because the length of the network/PCIe path travelled is identical. By avoiding one DMA operation, inlining reduces the latency of small WRITEs significantly.

3.2.2 WRITEs have higher throughput than READs

To evaluate throughput, it is first necessary to observe that with many client machines communicating with one server, different verbs perform very differently when used at the clients (talking to one server) and at the server (talking to many clients).

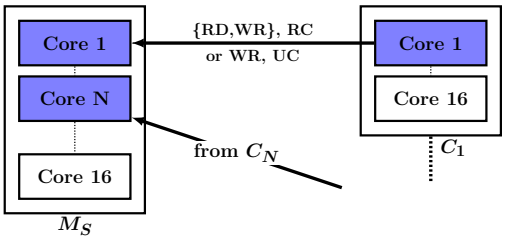


(a) Setup for measuring verbs and ECHO latency. We use one client process to issue operations to one server process

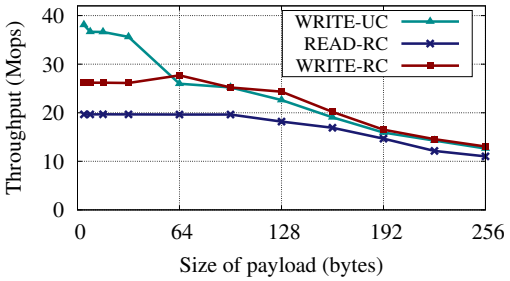


(b) The one-way latency of WRITE is half of the ECHO latency. ECHO operations used unsignaled verbs.

Figure 2: Latency of verbs and ECHO operations



(a) Setup for measuring inbound throughput. Each client process communicates with only one server process

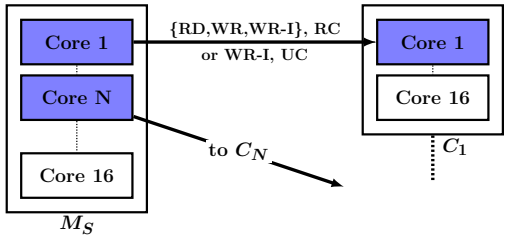


(b) For moderately sized payloads, WRITE has much higher inbound throughput than READs.

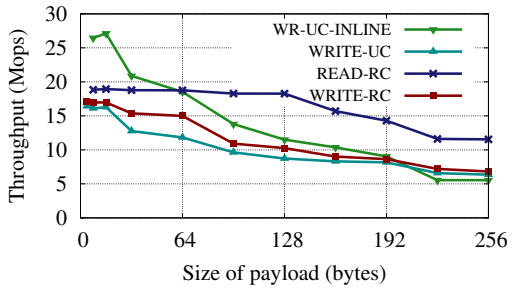
Figure 3: Comparison of inbound verbs throughput

Inbound throughput: First, we measured the throughput for *inbound* verbs, i.e., the number of verbs that multiple remote machines (the clients) can issue to one machine (the server). Using the notation introduced above, $C_1, ..., C_N$ issue operations to M_S as shown in Figure 3a. Figure 3b shows the cumulative throughput observed across the active machines. For small payloads (up to 32 bytes), WRITEs over UC achieve over 35 Mops, which is about 1.8 times higher than the maximum READ throughput. This is as expected: an inbound READ at $RNIC_S$ involves more work than an inbound WRITE because a read response must be sent back to the requester. For payloads up to 32 bytes, WRITEs over UC have significantly higher throughput than over RC because of the reduced ACK overhead. For payloads larger than 64 bytes, their throughputs are nearly identical, but using UC is still beneficial: It requires less processing at $RNIC_S$, and HERD uses this saved capacity to SEND responses.

Outbound throughput: We next measured the throughput for *outbound* verbs. Here, M_S issues operations to $C_1, ..., C_N$. As shown in Figure 4a, there are N processes on M_S; the i^{th} process communicates with C_i only (the scalability problems associated with all-to-all communication are explained in Section 3.3). For WRITEs, we also measure the throughput when the underlying transport is UC and when the payload is inlined. Figure 4b plots the throughput achieved by M_S for different payload sizes. For small sizes, inlined WRITEs have significantly higher outbound throughput than READs. For large sizes, the throughput of all WRITE variants is less than for READs, but it is never less than 50% of the READ throughput. Thus, even for these larger items, using a single WRITE (or SEND) for responses remains a better choice than using multiple READs for key-value items.



(a) Setup for measuring outbound throughput. Each server process communicates with only one client process.



(b) For small payloads, WRITE with inlining has a higher outbound throughput than READ.

Figure 4: Comparison of outbound verbs throughput

ECHO throughput is interesting for two reasons. First, it provides an upper bound on the throughput of a key-value cache based on one round trip of communication. Second, ECHOs help characterize the processing power of the RNIC: The advertised message rate of ConnectX-3 cards is 35 Mops but only small inbound WRITEs come close to this value.

An ECHO consists of a request message and a response message. Varying the verbs and transport types yield several different implementations of ECHO. Figure 5 shows the throughput for some of the possible combinations and for 32 byte payloads. The figure also shows that using inlinining, selective signaling, and UC transport increases the performance significantly.

ECHOs achieve maximum throughput (20 Mops) when both the request and the response are done as RDMA writes. However, as shown in Section 3.3, this approach does not scale with the number of connections. HERD uses RDMA writes (over UC) for requests and SENDs (over UD) for responses. *An ECHO server using this hybrid also achieves 20 Mops—it gives the performance of WRITE-based ECHOs, but with much better scalability.*

By avoiding the overhead of posting RECVs at the server, our method of WRITE based requests and SEND-based responses provides better throughput than purely SEND-based ECHOs. Interestingly, however, after enabling all optimizations, the throughput of purely SEND-based ECHOs (with no RDMA operations) is 14.6 Mops, which is more than three-fourths of the peak READ throughput (19.6 Mops). Both Pilaf and FaRM have noted that RDMA reads vastly outperform SEND-based ECHOs, which our results agree with *if our optimizations are removed*. With these optimizations, however, SENDs significantly outperform READs in cases where a single SEND-based ECHO can be used in place of multiple READs per request.

Our experiments show that several ECHO designs, with varying degrees of scalability, can perform better than multiple-READ designs. *From a network-centric perspective, this is fortunate: it also means that designs that use only one cross-datacenter RTT can potentially outperform multiple-RTT designs both in throughput and in latency.*

Discussion of verbs throughput: The ConnectX-3 card is advertised to support 35 million messages per second. Unidirectionally, our experiments show that it can slightly exceed this rate, but only for inbound WRITEs of 32 bytes or less (Figure 3b). All other verbs are slower than 30 Mops regardless of operation size, and no one sender can generate such a high outbound write rate (Figure 4b). While the manufacturer does not specify bidirectional message throughput, we know empirically that $RNIC_S$ can service 20 million ECHOs per second, or at least 40 million total Mops of inbound WRITEs and outbound SENDs.

Examining 32 byte payloads, the reduced throughputs can be attributed to several factors:

- For inbound WRITEs, the peak throughput is approximately 35 Mops and the bottleneck is the message rate of the RNIC. Outbound WRITEs use PIO and have a lower peak throughput of 21 Mops.

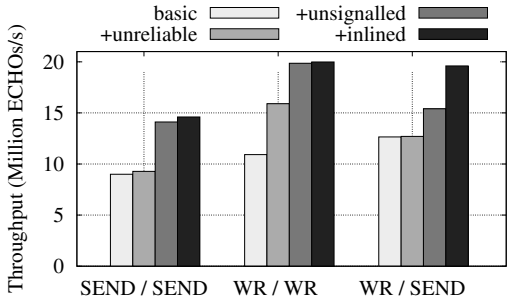


Figure 5: Throughput of ECHOs with 32 byte messages. In WR-SEND, the response is sent over UD.

- The maximum throughput for both inbound and outbound reads is around 19.5 Mops, which is approximately one half of the RNIC's message rate. Unlike WRITEs, READs are bottlenecked by the RNIC's processing power. This is as expected. Outbound READs involve a PIO operation, a packet transmission, a packet reception, and a DMA write, whereas outbound (inlined) WRITEs avoid the last two step. Inbound READs require a DMA read by the RNIC followed by a packet transmission, whereas inbound WRITEs only require a DMA write.

3.3 Using UD for responses

Our previous experiments did not show that as the number of connections increases, connected transports begin to slow down. To reduce hardware cost, power consumption, and design complexity, RNICs have very little on-chip memory (SRAM) to cache address translation tables and queue pair contexts [25]. A miss in this cache requires a PCIe transaction to fetch the data from host memory. When the communication fan-in or fan-out exceeds the capacity of this cache, performance begins to suffer. This is a potentially important effect to avoid both for cluster scaling, but also because it interacts with the cache or store architectural decisions. For example, the cache design we build on in HERD partitions the keyspace between several server processes in order to achieve efficient CPU and memory utilization. Such partitioning further increases the fan-in and fan out of connections to a single machine.

To evaluate this effect, we modified our throughput experiments to enable *all-to-all* communication. We use N client processes (one process each at $C_1, ..., C_N$) and N server processes at M_S. For measuring inbound throughput, client processes select a server process at random and issue a WRITE to it. For outbound throughput, a server process selects a client at random and issues a WRITE to it. The results of these experiments for 32 byte messages are presented in Figure 6. Several results stand out:

Outbound WRITEs scale poorly: for $N = 16$, there are 256 active queue pairs at $RNIC_S$ and the server-to-clients throughput degrades to 15% of the maximum outbound WRITE throughput (Figure 4b). With many active queue pairs, each posted verb can cause a cache miss, severely degrading performance.

Inbound WRITEs scale well: Clients-to-server throughput is high even for $N = 16$. The reason for this is that queueing of outstanding verbs operations is performed at the requesting RNIC and very little state is maintained at the responding RNIC. Therefore, the responding RNIC can support a much larger number of active queue pairs without incurring cache misses. The higher requester overhead is amortized because the clients outnumber the server.

In a different experiment, we used 1600 client processes spread over 16 machines to issue WRITEs over UC to one server process. HERD uses this many-to-one configuration to reduce the number of active connections at the server (Section 4.2). This configuration also achieves 30 Mops.

Outbound WRITEs scale poorly only because $RNIC_S$ must manage many connected queue pairs. This problem cannot be solved if we use connected transports (RC/UC/XRC) because they require at least as many queue pairs at M_S as the number of client machines. Scaling outbound communication therefore mandates

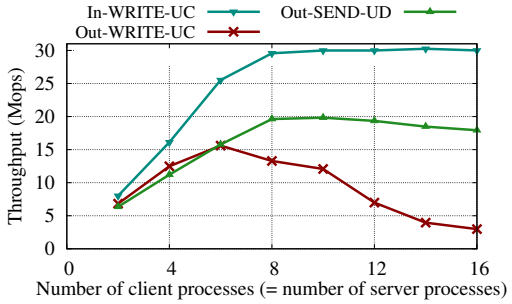


Figure 6: Comparison of UD and UC for all-to-all communication with 32 byte payloads. Inbound WRITEs over UC and outbound SENDs over UD scale well up to 256 queue pairs. Outbound WRITEs over UC scale poorly. All operations are inlined and unsignaled.

using datagrams. UD transport supports one-to-many communication, i.e., a single UD queue can be used to issue operations to multiple remote UD queues. The main problem with using UD in a high performance application is that it only supports messaging verbs and not RDMA verbs.

Fortunately, messaging verbs only impose high overhead at the receiver. Senders can directly transmit their requests; only the receiver must pre-post a RECV before the SEND can be handled. For the sender, the work done to issue a SEND is identical to that required to post a WRITE. Figure 6 shows that, when performed over Unreliable Datagram transport, SEND side throughput is high and scales well with the number of connected clients. The throughput of outbound SENDs nearly equals that of outbound WRITEs (Figure 4b).

The slight degradation of SEND throughput beyond 10 connected clients happens because the SENDs are unsignaled, i.e., server processes get no indication of verb completion. This leads to the server processes overwhelming $RNIC_S$ with too many outstanding operations, causing cache misses inside the RNIC. As HERD uses SENDs for responding to requests, it can use new requests as an indication of the completion of old SENDs, thereby avoiding this problem.

4 Design of HERD

To evaluate whether these network-driven architectural decisions work for a real key-value application, we designed and implemented an RDMA-based KV cache, called HERD, based upon recent high-performance key-value designs. Our HERD setup consists of one server machine and several client machines. The server machine runs N_S server processes. N_C client processes are uniformly spread across the client machines.

4.1 Key-Value cache

The fundamental goal of this work is to evaluate our networking and architectural decisions in the context of key-value systems. We do not focus on building better back-end key-value data structures but rather borrow existing designs from MICA [17].

MICA is a near line-rate key-value cache and store for classical Ethernet. We restrict our discussion of MICA to its cache mode. MICA uses a lossy index to map keys to pointers, and stores the actual values in a circular log. On insertion, items can be evicted from the index (thereby making the index lossy), or from the log in a FIFO order. In HERD, each server process creates an index for 64 Mi keys, and a 4 GB circular log. We use MICA's algorithm for both `GET`s and `PUT`s: each `GET` requires up to two random memory lookups, and each `PUT` requires one.

MICA shards the key space into several partitions based on a keyhash. In its "EREW" mode, each server core has exclusive read and write access to one partition. MICA uses the Flow Director [3] feature of modern Ethernet NICs to direct request packets to the core responsible for the given key. HERD achieves the same

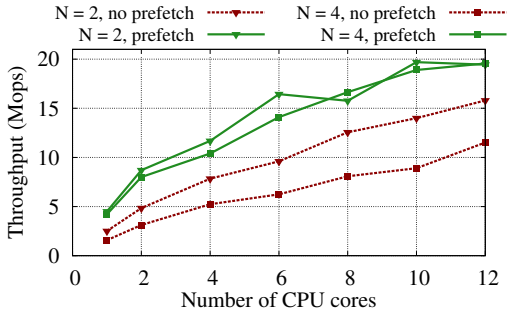


Figure 7: Effect of prefetching on throughput

effect by allocating per-core request memory at the server, and allowing clients to WRITE their requests directly to the appropriate core.

4.1.1 Masking DRAM latency with prefetching

To service a `GET`, a HERD server must perform two random memory lookups, prepare the SEND response (with the key's value inlined in the WQE), and then post the SEND verb using the `post_send()` function. The memory lookups and the `post_send()` function are the two main sources of latency at the server. Each random memory access takes 60-120 *ns* and the `post_send()` function takes about 200 *ns*. While the latter is unavoidable, we can mask the memory access latency by overlapping memory accesses of one request with computation of another request.

MICA and CuckooSwitch [17, 29] mask latency by overlapping memory fetches and prefetches, or request decoding and prefetches. HERD takes a different approach: we overlap prefetches with the `post_send()` function used to transmit replies. To process multiple requests simultaneously in the absence of a driver that itself handles batches of packets [2, 17, 29]), HERD creates a pipeline of requests at the application level.

In HERD, the maximum number of memory lookups for each request is two. Therefore, we create a request pipeline with two stages. When a request is in stage i of the pipeline, it performs the i-th memory access for the request and issues a prefetch for the next memory address. In this way, requests only access memory for which a prefetch has already been issued. On detecting a new request, the server issues a prefetch for the request's index bucket, advances the old requests in the pipeline, pushes in the new request, and finally calls `post_send()` to SEND a reply for the pipeline's completed request. The server process expects the issued prefetches to finish by the time `post_send()` returns.

Figure 7 shows the effectiveness of prefetching. We use a WRITE/SEND-based ECHO server but this time the server performs N random memory accesses before sending the response. Prefetching allows fewer cores to deliver higher throughput: 12 cores can deliver the peak throughput even with $N = 4$. We conclude that there is significant headroom to implement more complex key-value applications, for instance, key-value stores, on top of HERD's request-reply communication mechanism.

With a large number of server processes, this pipelining scheme can lead to a deadlock. A server does not advance its pipeline until it receives a new request, and a client does not advance its request window until it gets a response. We avoid this deadlock as follows. While polling for new requests, if a server fails for 100 iterations consecutively, it pushes a no-op into the pipeline.

4.2 Requests

Clients WRITE their `GET` and `PUT` requests to a contiguous memory region on the server machine which is allocated during initialization. This memory region is called the *request region* and is shared among all the server processes by mapping it using `shmget()`. The request region is logically divided into 1 KB slots (the maximum size of a key-value item in HERD is 1 KB).

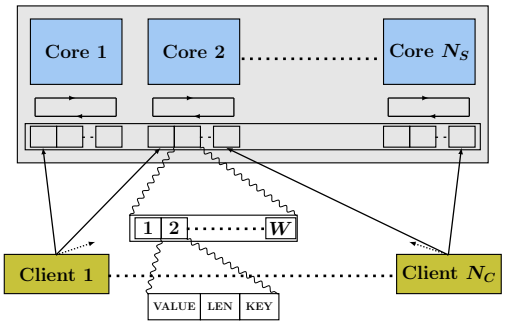


Figure 8: Layout of the request region at the server

Requests are formatted as follows. A `GET` request consists only of a 16-byte keyhash. A `PUT` request contains a 16-byte keyhash, a 2-byte LEN field (specifying the value's length), and up to 1000 bytes for the value. To poll for incoming requests, we use the left-to-right ordering of the RNIC's DMA writes [15, 8]. We use the keyhash field to poll for new requests; therefore, the key is written to the rightmost 16 bytes of the 1 KB slot. A non-zero keyhash indicates a new request, so we do not allow the clients to use a zero keyhash. The server zeroes out the keyhash field of the slot after sending a response, freeing it up for a new request.

Figure 8 shows the layout of the request region at the server machine. It consists of separate chunks for each server process which are further sub-divided into per-client chunks. Each per-client chunk consists of W slots, i.e., each client can have up to W pending requests to each server process (in our current implementation, W is also the total number of outstanding requests maintained by each client). The size of the request region is $N_S \cdot N_C \cdot W$ KB. With $N_C = 200$, $N_S = 16$ and $W = 2$, this is approximately 6 MB and fits inside the server's L3 cache. Each server process polls the per-client chunks for new requests in a round robin fashion. If server process s has seen r requests from client number c, it polls the request region at the request slot number $s \cdot (W \cdot N_c) + (c \cdot W) + r \ mod \ W$.

A network configuration using bidirectional, all-to-all, communication with connected transports would require $N_C \cdot N_S$ queue pairs at the server. HERD, however, uses connected transports for only the request side of communication, and thus requires only N_C connected queue pairs. The configuration works as follows. An initializer process creates the request region, registers it with $RNIC_S$, establishes a UC connection with each client, and goes to sleep. The N_S server processes then map the request region into their address space via `shmget()` and do not create any connections for receiving requests.

4.3 Responses

In HERD, responses are sent as SENDs over UD. Each client creates N_S UD queue pairs (QPs) whereas each server process uses only one UD QP. Before writing a new request to server process s, a client posts a RECV to its s-th UD QP. This RECV specifies the memory area on the client where the server's response will be written. Each client allocates a *response region* containing $W \cdot N_S$ response slots: this region is used for the target addresses in the RECVs. After writing out W requests, the client starts checking for responses by polling for RECV completions. On each successful completion, it posts another request.

The design outlined thus far deliberately shifts work from the server's RNIC to the client's, with the assumption that client machines often perform enough other work that saturating 40 or 56 gigabits of network bandwidth is not their primary concern. The servers, however, in an application such as Memcached, are often dedicated machines, and achieving high bandwidth is important.

5 Evaluation

We evaluate HERD on an 18-node InfiniBand/RoCE cluster. Our evaluation shows that:

- HERD uses the full processing power of the RNIC. A single HERD server can process up to 19.8 million requests per second. For item size up to 32 bytes, HERD's request throughput is *greater than native READ throughput* and is much greater than that of READ-based key-value services: it is over 2X higher than FaRM-KV and Pilaf.
- HERD delivers up to 18 Mops with less than 10 μs latency. Its latency is over 40% lower than Pilaf and FaRM-KV at their peak throughput respectively.
- HERD scales to the moderately sized Apt cluster, sustaining peak throughput with over 250 connected client processes.

We conclude the evaluation by examining the seeming drawback of the HERD design relative to READ-based designs—its higher server CPU use—and put this in context with the *total* (client + server) CPU required by all systems.

5.1 Experimental setup

We run our experiments run on a cluster of 18 machines. The 17 client machines run up to 3 client processes each. The machine configuration is described in Section 3.1. The machines run Ubuntu 12.04 with Mellanox's `mlx4` RDMA drivers.

Comparison against stripped-down alternatives: In keeping with our focus on understanding the effects of network-related decisions, we compare our (full) HERD implementation against simplified implementations of Pilaf and FaRM-KV. These simplified implementations use the same communication methods as the originals, but *omit* the actual key-value storage, instead returning a result instantly. We made this decision for two reasons. First, while working with Pilaf's code, we observed several optimization opportunities; we did not want our evaluation to depend on the relative performance tuning of the systems. Second, we did not have access to the FaRM source code, and we could not run Windows Server on our cluster. Instead, we created and evaluated emulated versions of the two systems which do not include their backing data structures. This approach gives these systems the maximum performance advantage possible, so the throughput we report for both Pilaf and FaRM-KV may be higher than is actually achievable by those systems.

Pilaf is based on 2-level lookups: a hash-table maps keys to pointers. The pointer is used to find the value associated with the key from flat memory regions called *extents*. FaRM-KV, in its default operating mode, uses single-level lookups. It achieves this by inlining the value in the hash-table. It also has a two-level mode, where the value is stored "out-of-table." Because the out-of-table mode is necessary for memory efficiency with variable length keys, we compare HERD against both modes. In the following two subsections, we denote the size of a key, value, and pointer by S_K, S_V, and S_P respectively.

5.1.1 Emulating Pilaf

In K-B cuckoo hashing, every key can be found in K different buckets, determined by K orthogonal hash functions. For associativity, each bucket contains B slots. Pilaf uses 3-1 cuckoo hashing with 75% memory efficiency and 1.6 average probes per `GET` (higher memory efficiency with fewer, but slightly larger, average probes is possible with 2-4 cuckoo hashing [9]). When reading the hash index via RDMA, the smallest unit that must be read is a bucket. A bucket in Pilaf has only one slot that contains a 4 byte pointer, two 8 byte checksums, and a few other fields. We assume the bucket size in Pilaf to be 32 bytes for alignment.

`GET`: A `GET` in Pilaf consists of 1.6 bucket READs (on average) to find the value pointer, followed by a S_V byte READ to fetch the value. It is possible to reduce Pilaf's latency by issuing concurrent READs for both cuckoo buckets. As this comes at the cost of decreased throughput, we wait for the first READ to complete and issue the second READ only if it is required.

`PUT`: For a `PUT`, a client SENDs a $S_K + S_V$ byte message containing the new key-value item to the server. This request may require relocating entries in the cuckoo hash-table, but we ignore that as our evaluation focuses on the network communication only.

In emulating Pilaf, we enable all of our RDMA optimizations for both request types; we call the resulting system Pilaf-em-OPT.

5.1.2 Emulating FaRM-KV

FaRM-KV uses a variant of Hopscotch hashing to locate a key in approximately one READ. Its algorithm guarantees that a key-value pair is stored in a small neighborhood of the bucket that the key hashes to. The size of the neighborhood is tunable, but its authors set it to 6 to balance good space utilization and performance for items smaller than 128 bytes. FaRM-KV can inline the values in the buckets, or it can store them separately and only store pointers in the buckets. We call our version of FaRM-KV with inlined values FaRM-em and without inlining FaRM-em-VAR (for variable length values).

GET: A GET in FaRM-em requires a 6 * ($S_K + S_V$) byte READ. In FaRM-em-VAR, a GET requires a 6 * ($S_K + S_P$) byte READ followed by a S_V byte READ.

PUT: FaRM-KV handles PUTs by sending messages to the server via WRITEs, similar to HERD. The server notifies the client of PUT completion using another WRITE. Therefore, a PUT in FaRM-em (and FaRM-em-VAR) consists of one $S_K + S_V$ byte WRITE from a client to the server, and one WRITE from the server to the client. For higher throughput, we perform these WRITEs over UC unlike the original FaRM paper that used RC (Figure 5).

5.2 Workloads

Three main workload parameters affect the throughput and latency of a key-value system: relative frequency of PUTs and GETs, item size, and skew.

We use two types of workloads: *read-intensive* (95% GET, 5% PUT) and *write-intensive* (50% GET, 50% PUT). Our workload can either be *uniform* or *skewed*. Under a uniform workload, the keys are chosen uniformly at random from the 16 byte keyhash space. The skewed workload draws keys from a Zipf distribution with parameter .99. This workload is generated offline using YCSB [7]. We generated 480 million keys once and assigned 8 million keys to each of the 51 client processes.

5.3 Throughput comparison

We now compare the end-to-end throughput of HERD against the emulated versions of Pilaf and FaRM. We present our evaluation for both InfiniBand and RoCE. We do not seek to compare the two, rather we wish to show that our design generalizes to different RDMA-providing networks.

Figure 9 plots the throughput of these system for read-intensive and write-intensive workloads for 48-byte items (S_K = 16, S_V = 32). We chose this item size because it is representative of real-life workloads: an analysis of Facebook's general-purpose key-value store [6] showed that the 50-th percentile of key sizes is approximately 30 bytes, and that of value sizes is 20 bytes. To compare the READ-based GETs of Pilaf and FaRM with Pilaf's SEND/RECV-based PUTs, we also plot the throughput when the workload consists of 100% PUTs.

In HERD, both read-intensive and write-intensive workloads achieve about 18.5 Mops, which is close to the throughput of native RDMA reads of a similar size (Figure 3b). For small key-value items, there is very little difference between PUT and GET requests at the RDMA layer because both types of requests fit inside one cacheline. Therefore, the throughput does not depend on the workload composition.

The GET throughput of Pilaf-em-OPT and FaRM-em(-VAR) is directly determined by the throughput of RDMA READs. A GET in Pilaf-em-OPT involves 2.6 READs (on average). Its GET throughput is 7.5 Mops, which is about 2.6X smaller than the maximum READ throughput. For GETs, FaRM-em requires a single 288 byte READ and delivers 9.2 Mops. FaRM-em-VAR requires a second READ and has throughput of 7.5 Mops for GETs.

Surprisingly, the PUT throughput in our emulated systems is much larger than their GET throughput. This is explained as follows. In FaRM-em(-VAR), PUTs use small WRITEs over UC that outperform the

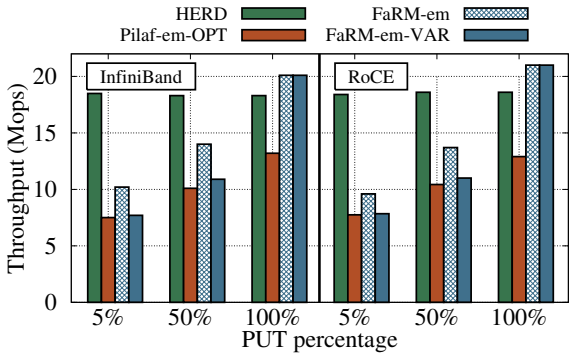


Figure 9: End-to-end throughput comparison for 48 byte key-value items

large READs required for GETs. Pilaf-em-OPT uses SEND/RECV-based requests and replies for PUT. Both Pilaf and FaRM assume that messaging-based ECHOs are much more expensive than READs. (Pilaf reports that for 17 byte messages, the throughput of RDMA reads is 2.449 Mops whereas the throughput of SEND/RECV-based ECHOs is only 0.668 Mops.) If SEND/RECV can provide only one fourth the throughput of READ, it makes sense to use multiple READs for GET.

However, we believe that these systems do not achieve the full capacity of SEND/RECV. After optimizing SENDs by using unreliable transport, payload inlining, and selective signaling, SEND/RECV based ECHOs provide respectable throughput. As shown in Figure 5, our implementation of SEND/RECV-based ECHOs achieves 14.6 Mops (for 32-byte messages, unlike Figure 9 which uses 48-byte messages), which is considerably more than half of our READ throughput (19.6 Mops). Therefore, we conclude that SEND/RECV-based communication, when used effectively, is more efficient than using multiple READs per request.

Figure 10 shows the throughput of the three systems with 16 byte keys and different value sizes for a read-intensive workload. For up to 32-byte items, HERD delivers over 19.8 Mops, which is greater than the peak READ throughput. With 4-byte values, FaRM-em also delivers high throughput because it only requires one 120-byte READ. However, its throughput declines quickly with increasing value size because the size of FaRM-em's READs grow rapidly (as $6 * (S_V + 16)$). This problem is fundamental to the Hopscotch-based KV design which amplifies the READ size to reduce round trips. FaRM-KV quickly saturates link bandwidths (PCIe or InfiniBand/RoCE) with smaller items than HERD, which conserves network bandwidth by transmitting only essential data. Figure 10 illustrates this effect; while FaRM-em saturates the PCIe 2.0 bandwidth with 4 byte values, HERD achieves high performance up to 32 byte values, where it saturates the smaller PCIe PIO bandwidth.

With large values (larger than 192) bytes, HERD switches to using non-inlined SENDs for responses. The outbound throughput of large inlined messages is less than non-inlined messages because DMA outperforms PIO for large payloads (Figure 4b). For large values, the performance of HERD, FaRM-em, and Pilaf-em-OPT are within 10% of each other.

5.4 Latency comparison

Unlike FaRM-KV and Pilaf, HERD uses only one network round trip for any request. FaRM-KV and Pilaf use one round trip for PUT requests but require multiple round trips for GETs (except when FaRM-KV inlines values in the hash-table). This causes their GET latency to be higher than the latency of a single RDMA READ.

Figure 11 compares the average latencies of the three systems for a read-intensive workload; the error bars indicate the 5th and 95th percentile latency. To understand the dependency of latency on throughput, we increase the load on the server by adding more clients until the server is saturated. When using 10

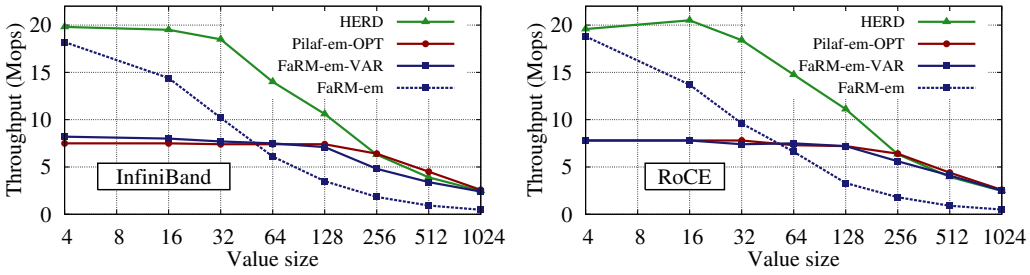


Figure 10: End-to-end throughput comparison with different value sizes

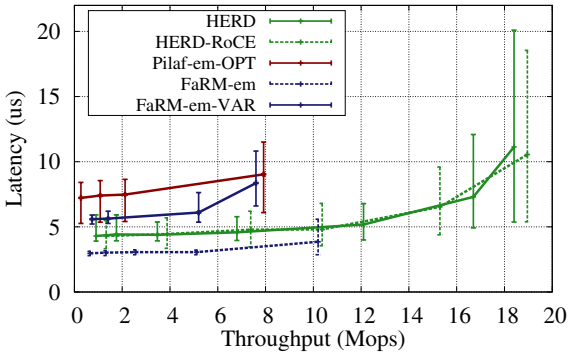


Figure 11: End-to-end latency with 48 byte items and read-intensive workload

CPU cores at the server, HERD is able to deliver 18 million requests per second with approximately 10 μs average latency. For fixed-length key-value items, FaRM-em provides the lowest latency among the three systems because it requires only one network round trip (unlike Pilaf-em-OPT) and no computation at the server (unlike HERD). For variable length values, however, FaRM's variable length mode requires two RTTs, yielding worse latency than HERD.

The `PUT` latency for all three systems (not shown) is similar because the network path traversed is the same. The measured latency for HERD was slightly higher than that of the emulated systems because it performed actual hash table and memory manipulation for inserts, but this is an artifact of the performance advantage we give Pilaf-em and FaRM-em.

With one partition, HERD's minimum average latency is 3.6 μs which is significantly smaller than the minimum average latency with 10 partitions. This is explained as follows. Each server tries to keep its pipeline of requests full—it tries to postpone calling `post_send()` till it can overlap it with memory prefetches. If it does not detect new requests, it waits for several hundred CPU cycles before flushing the pipeline. When a large number of cores work under a light load, the wait adds to the overall latency.

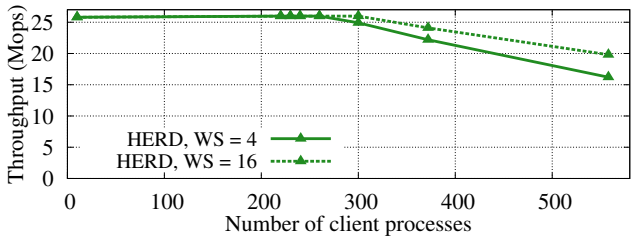


Figure 12: Throughput with variable number of client processes and different window sizes

5.5 Scalability

To understand HERD's number-of-clients scalability, we conduct experiments on a larger cluster with 187 machines. Each machine has a single Intel E5-2450 CPU (8 cores, 16 hyperthreads), an a ConnectX-3 354A card connected via PCIe 3.0 x8. We used one machine to run 6 server processes and the remaining 186 machines for client processes. The experiment uses 16 byte keys and 32 byte values.

Figure 12 shows the results from this experiment. HERD delivers its maximum throughput for up to 260 client processes. With even more clients, HERD's throughput starts decreasing almost linearly. The rate of decrease can be reduced by increasing the number of outstanding requests maintained by each client, at the cost of higher request latency. Figure 12 shows the results for two window sizes: 4 (HERD's default) and 16. This observation suggests that the decline is due to cache misses in $RNIC_S$, as more outstanding verbs in a queue can reduce cache pressure. We expect this scalability limit to be resolved with the introduction of Dynamically Connected Transport in the new Connect-IB cards [1, 8],

Another likely scalability limit of our current HERD design is the round-robin polling at the server for requests. With thousands of clients, using WRITEs for inbound requests may incur too much CPU overhead; mitigating this effect may necessitate switching to a SEND/SEND architecture over Unreliable Datagram transport. Figure 5 shows there is a 4-5 Mops decrease to this change, but once made, the system should scale up to many thousands of clients, while still outperforming an RDMA READ-based architecture.[1] We expect the performance of the SEND/SEND architecture relative to WRITE-SEND to increase with the introduction of inlined RECVs in Connect-IB cards. This will reduce the load on RNICs by encapsulating the RECV payload in the RECV completion.

5.6 HERD CPU Use

The primary drawback of not using READs in HERD is that `GET` operations require the server CPU to execute requests, in exchange for saving one cross-datacenter RTT. While at first glance, it might seem that HERD's CPU usage should be higher than Pilaf and FaRM-KV, we show that in practice these two systems also have significant sources of CPU usage that reduce the extent of the difference.

First, issuing extra READs adds CPU overhead at the Pilaf and FaRM-KV clients. To issue the second READ, the clients must poll for the first READ to complete. HERD shifts this overhead to the server's CPU, making more room for application processing at the clients.

Second, handling `PUT` requests requires CPU involvement at the server. Achieving low-latency `PUT`s requires dedicating server CPU cores that poll for incoming requests. Therefore, the exact CPU use depends on the fraction of `PUT` throughput that server is *provisioned* for, because this determines the CPU resources that must be allocated to it, not the dynamic amount actually used. For example, our experiments show that, even ignoring the cost of updating data structures, provisioning for 100% `PUT` throughput in Pilaf requires at least 9 CPU cores just to post the SENDs and RECVs. Figure 13 shows Pilaf-em-OPT's `PUT` throughput for different numbers of CPU cores at the server.

In Figure 13, we also plot HERD's throughput for a workload with 48-byte items by varying the number of server CPU cores. HERD is able to deliver its maximum throughput of 18.5 Mops with 10 CPU cores. The modest gap to Pilaf arises because the HERD server in this experiment is handling hash table lookups and updates, whereas the emulated Pilaf is handling only the network traffic.

We believe, therefore, that HERD's higher throughput and lower latency, along with the significant CPU utilization in Pilaf and FaRM-KV, justifies the architectural decision to have the CPU involved on the `GET` path for small key-value items. For a 50% `PUT` workload, for example, the moderate extra cost of adding a few more cores—or using the already-idle cycles on the cores—is likely worthwhile for many applications.

[1] Figure 5 uses SENDs over UC, but we have verified that similar throughput is possible using SENDs over UD.

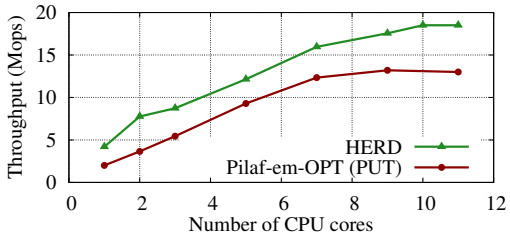


Figure 13: Throughput as a function of server CPU cores

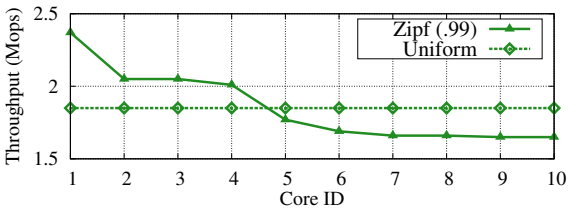


Figure 14: Per-core throughput under skewed and uniform workloads. Note that the y-axis does not begin at 0.

5.7 Resistance to skew

To understand how HERD's behavior is impacted by skew, we tested it with a workload where the keys are drawn from a Zipf distribution. HERD adapts well to skew, delivering its maximum performance even when the Zipf parameter is .99. HERD's resistance to skew comes from two factors. First, the back-end MICA architecture [17] that we use in HERD performs well under skew; a skewed workload spread across several partitions produces little variation in the partitions' load compared to the skew in the workload's distribution. Under our Zipf-distributed workload, with 6 partitions, the most loaded CPU core is only 36% more so than the least loaded core, even though the most popular key is over 10^5 times more popular than the average.

Second, because the CPU cores share the RNIC, the highly loaded cores are able to benefit from the idle time provided by the less-used cores. Figure 13 demonstrates this effect: with a *uniform* workload and using only a single core, HERD can deliver 4.2 Mops. When the system is configured to use 10 cores—the minimum required by HERD to deliver its peak throughput—the system delivers 1.8 Mops *per core*. The per-core performance reduction is not because of a CPU bottleneck, but because the server processes saturate the PCIe PIO throughput. Therefore, even if the workload is skewed, there is ample CPU headroom on a given core to handle the extra requests.

Figure 14 shows the per-core throughput of HERD for a skewed workload. The experimental configuration is: 48-byte items, read-intensive, skewed workload, 10 total CPU cores. The per-core throughput for a uniform workload is included for comparison.

6 Related Work

RDMA-based key-value stores: Other than Pilaf and FaRM, several projects have designed memcached-like systems over RDMA. Panda et al. [14] describe a memcached implementation using a hybrid of UD and RC transports. It uses SEND/RECV messages for all requests and avoids the overhead of UD transport (caused by a larger header size than RC) by actively switching connections between RC and UD. Although their cluster (ConnectX, 32 Gbps) is comparable to ours (ConnectX-3, 40 Gbps), their request rate is less than 1.5 Mops. Stuedi et al. [24] describe a SoftiWARP [27] based version of memcached targeting CPU savings in wimpy nodes with 10GbE.

Accelerating systems with RDMA: Several projects have used verbs to improve the performance of systems such as HBase, Hadoop RPC, PVFS [28, 13, 19]. Most of these use only SEND/RECV verbs as a fast alternative to socket-based communication. In a PVFS implementation over InfiniBand [28], `read()`

and `write()` operations in the filesystem use both RDMA and SEND/RECV. They favor WRITEs over READs for the same reasons as in our work, *implying that the performance gap has existed over several generations of InfiniBand hardware*. There have been several versions of MPI over InfiniBand [15, 18]. In MPICH2, RDMA writes are used for one-sided messaging: the server polls the head of a circular buffer that is written to by a client. HERD extends this messaging in a scalable fashion for all-to-all request-reply communication. While [28, 13, 19, 15, 18] have benchmarked verbs performance before, it has been for large messages in the context of applications like NFS and MPI. Our work exploits the performance differences that appear only for small messages and are relevant for message rate-bound applications like key-value stores.

User level networking: Taken together, we believe that one conclusion to draw from the union of HERD, Pilaf, FaRM, and MICA [17] is that the biggest boost to throughput comes from bypassing the network stack and avoiding CPU interrupts, *not* necessarily from bypassing the CPU entirely. All four of these systems use mechanisms to allow user-level programs to directly receive requests or packets from the NIC: the userlevel RDMA drivers for HERD, Pilaf, and FaRM, and the Intel DPDK library for MICA. As we discuss below, the *throughput* of these systems is similar, but the batching required by the DPDK-based systems confers a latency advantage to the hardware-supported InfiniBand systems. These lessons suggest profitable future work in making user-level classical Ethernet systems more portable, easier to use, and lower-latency. One ongoing effort is NIQ [10], an FPGA-based low-latency NIC which uses cacheline-sized PIOs (without any DMA) to transmit and receive small packets. Inlined WRITEs in RDMA use the same mechanism at the requesters's side.

General key-value stores: MICA [17] is a recent key-value system for classical Ethernet. It assigns exclusive partitions to server cores to minimize memory contention, and exploits the NIC's capability to steer requests to the responsible core [3]. A MICA server is capable of delivering 77 Mops with 4 dual-port 10 Gbps NICs with 50μs average latency. Although the systems are not directly comparable, HERD delivers 26 Mops with 5 μs average latency with 56 Gbps InfiniBand. This suggests that the major benefit of using RDMA in key-value services is low latency and not necessarily high throughput. RAMCloud [22] is a RAM-based, persistent key-value store that uses messaging verbs for low latency communication.

7 Conclusion

This paper explored the options for implementing fast, low-latency key-value systems atop RDMA, arriving at an unexpected and novel combination that outperforms prior designs and uses fewer network round-trips. Our work shows that, contrary to widely held beliefs about engineering for RDMA, single-RTT designs with server CPU involvement can outperform the "optimization" of CPU-bypassing remote memory access when the RDMA approaches require multiple RTTs. These results contribute not just a practical artifact—the HERD low-latency, high-performance key-value cache—but an improved understanding of how to use RDMA to construct future DRAM-based storage services.

References

[1] Connect-IB: Architecture for Scalable High Performance Computing. URL http://www.mellanox.com/related-docs/applications/SB_Connect-IB.pdf.

[2] Intel DPDK: Data Plane Development Kit. URL http://dpdk.org.

[3] Intel 82599 10 Gigabit Ethernet Controller: Datasheet. URL http://www.intel.com/content/www/us/en/ethernet-controllers/82599-10-gbe-controller-datasheet.html.

[4] Redis: An Advanced Key-Value Store. URL http://redis.io.

[5] memcached: A Distributed Memory Object Caching System, 2011. URL http://memcached.org.

[6] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of a Large-Scale Key-Value Store. In *SIGMETRICS*, 2012.

[7] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *SoCC*, 2010.

[8] A. Dragojevic, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast Remote Memory. In *USENIX NSDI*, 2014.

[9] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *USENIX NSDI*, 2013.

[10] M. Flajslik and M. Rosenblum. Network Interface Design for Low Latency Request-Response Protocols. In *USENIX ATC*, 2013.

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

[12] M. Herlihy, N. Shavit, and M. Tzafrir. Hopscotch Hashing. In *DISC*, 2008.

[13] J. Huang, X. Ouyang, J. Jose, M. W. ur Rahman, H. Wang, M. Luo, H. Subramoni, C. Murthy, and D. K. Panda. High-Performance Design of HBase with RDMA over InfiniBand. In *IPDPS*, 2012.

[14] J. Jose, H. Subramoni, K. C. Kandalla, M. W. ur Rahman, H. Wang, S. Narravula, and D. K. Panda. Scalable Memcached Design for InfiniBand Clusters Using Hybrid Transports. In *CCGRID*. IEEE, 2012.

[15] J. Li, J. Wu, and D. K. Panda. High Performance RDMA-Based MPI Implementation over InfiniBand. *International Journal of Parallel Programming*, 2004.

[16] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A Memory-efficient, High-performance Key-value Store. In *SOSP*, 2011.

[17] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *USENIX NSDI*, 2014.

[18] J. Liu, W. Jiang, P. Wyckoff, D. K. Panda, D. Ashton, D. Buntinas, W. Gropp, and B. Toonen. Design and Implementation of MPICH2 over InfiniBand with RDMA Support. In *IPDPD*, 2004.

[19] X. Lu, N. S. Islam, M. W. ur Rahman, J. Jose, H. Subramoni, H. Wang, and D. K. Panda. High-Performance Design of Hadoop RPC with RDMA over InfiniBand. In *ICPP*, 2013.

[20] C. Mitchell, Y. Geng, and J. Li. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *USENIX ATC*, 2013.

[21] 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 *USENIX NSDI*, 2013.

[22] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast Crash Recovery in RAMCloud. In *SOSP*, 2011.

[23] R. Pagh and F. F. Rodler. Cuckoo Hashing. *J. Algorithms*, 2004.

[24] P. Stuedi, A. Trivedi, and B. Metzler. Wimpy Nodes with 10GbE: Leveraging One-Sided Operations in Soft-RDMA to Boost Memcached. In *USENIX ATC*, 2012.

[25] S. Sur, A. Vishnu, H.-W. Jin, W. Huang, and D. K. Panda. Can Memory-Less Network Adapters Benefit Next-Generation InfiniBand Systems? In *HOTI*, 2005.

[26] S. Sur, M. J. Koop, L. Chai, and D. K. Panda. Performance Analysis and Evaluation of Mellanox ConnectX Infiniband Architecture with Multi-Core Platforms. In *HOTI*, 2007.

[27] A. Trivedi, B. Metzler, and P. Stuedi. A Case for RDMA in Clouds: Turning Supercomputer Networking into Commodity. In *APSys*, 2011.

[28] J. Wu, P. Wyckoff, and D. K. Panda. PVFS over InfiniBand: Design and Performance Evaluation. In *Ohio State University Tech Report*, 2003.

[29] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen. Scalable, High Performance Ethernet Forwarding with CuckooSwitch. In *CoNEXT*, 2013.