Network Interface Architecture for Remote Indirect Memory Access (RIMA) in Datacenters

JIACHEN XUE, Nvidia
T. N. VIJAYKUMAR and MITHUNA THOTTETHODI, Purdue University

Remote Direct Memory Access (RDMA) fabrics such as InfiniBand and Converged Ethernet report latency shorter by a factor of 50 than TCP. As such, RDMA is a potential replacement for TCP in datacenters (DCs) running low-latency applications, such as Web search and memcached. InfiniBand's Shared Receive Queues (SRQs), which use two-sided send/recv verbs (i.e., *channel semantics*), reduce the amount of pre-allocated, pinned memory (despite optimizations such as InfiniBand's on-demand paging (ODP)) for message buffers. However, SRQs are limited fundamentally to a single message size per queue, which incurs either memory wastage or significant programmer burden for typical DC traffic of an arbitrary number (level of burstiness) of messages of arbitrary size.

We propose *remote indirect memory access (RIMA)*, which avoids these pitfalls by providing (1) network interface card (NIC) microarchitecture support for novel *queue semantics* and (2) a new "verb" called *append*. To append a sender's message to a shared queue, the receiver NIC atomically increments the queue's tail pointer by the incoming message's size and places the message in the newly created space. As in traditional RDMA, the NIC is responsible for pointer lookup, address translation, and enforcing virtual memory protections. This *indirection* of specifying a queue (and not its tail pointer, which remains hidden from senders) handles the typical DC traffic of an arbitrary sender sending an arbitrary number of messages of arbitrary size. Because RIMA's simple hardware adds only 1–2 ns to the multi-μs message latency, RIMA achieves the same message latency and throughput as InfiniBand SRQ with unlimited buffering. Running memcached traffic on a 30-node InfiniBand cluster, we show that at similar, low programmer effort, RIMA achieves significantly smaller memory footprint than SRQ. However, while SRQ can be crafted to minimize memory footprint by expending significant programming effort, RIMA provides those benefits with little programmer effort. For memcached traffic, a high-performance key-value cache (*FastKV*) using RIMA achieves either 3× lower 96th-percentile latency or significantly better throughput or memory footprint than *FastKV* using RDMA.

CCS Concepts: • **Networks → Datacenter networks**; • **Hardware → Networking hardware**;

Additional Key Words and Phrases: Network Interface, Indirection, remote append, receive queue management, remote direct memory access

13

This work was done when J. Xue was a graduate student at Purdue University.
This work was supported in part by the National Science Foundation under grant numbers 1633412-IIS and 1633318-IIS.
Authors' addresses: J. Xue, Nvidia, 2788 San Tomas Expressway, Santa Clara, CA, 95051; email: xuejiachen@gmail.com; T. N. Vijaykumar and M. Thottethodi, Purdue University, 465 Northwestern Ave, West Lafayette, IN, 47907; emails: vijay@ecn.purdue.edu, mithuna@purdue.edu.

1544-3566/2020/05-ART13
https://doi.org/10.1145/3374215

ACM Reference format:
Jiachen Xue, T. N. Vijaykumar, and Mithuna Thottethodi. 2020. Network Interface Architecture for Remote Indirect Memory Access (RIMA) in Datacenters. *ACM Trans. Archit. Code Optim.* 17, 2, Article 13 (May 2020), 22 pages.
https://doi.org/10.1145/3374215

1 INTRODUCTION

Datacenter (DC) network performance is critical for most modern web/cloud services whose backends run within DCs. While the dominant TCP/IP over Ethernet model has many advantages including scalability and stability, it suffers from significant latency (e.g., milliseconds). In contrast, RDMA over InfiniBand and RDMA over converged Ethernet (RoCE) are attractive for their low latency (e.g., microseconds) [13, 26, 28] as well as improved tail latency at datacenter scales [51]. Unfortunately, there remain several challenges in achieving high performance in an efficient manner.

RDMA's basic primitives, while well suited for high-performance computing (HPC), are a mismatch for the communication typical in today's DCs. Specifically, the request-response model of DCs must accommodate the following traffic constraints of typical DC workloads (e.g., key-value stores, Web search, dynamic advertisements, and news feeds): (1) arbitrary sources of traffic (*arb-src*): Because of sharding of content, objects may be widely distributed across many servers any of which may respond (e.g., data may be sharded across 5,000 servers in a modern DC) [8]; (2) arbitrary message sizes (*arb-size*): The key/values that are exchanged vary from small text updates (tens/hundreds of bytes) to large videos/images (MBs) [6]; and, (3) arbitrary burstiness (*arb-num*): Message arrival may be bursty—the well-known *incast problem* [2])—with the peak rate limited only by the line rate (e.g., 56 Gb/s for InfiniBand FDR). In contrast, HPC workloads typically have fewer sources and not as much variation in message sizes. Even in HPC, a large number of potential sources has been recognized to be a challenge [38].

Memory allocation to hold incoming messages is a key challenge in managing communication. TCP uses shared buffers to temporarily hold packets of multiple messages, which are then delivered by the operating system (OS) to the appropriate process. Because the buffers are shared across processes, the amount of memory allocated for the buffers can be small but OS involvement is necessary, which is slow. In contrast, under the RDMA read/write model (i.e., one-sided communication), memory for private buffers is proactively pre-allocated and pinned via memory registration. While Infiniband's On-Demand Paging (ODP) offers some relief from always pinning memory from different program phases, the near-term message receive buffers that we focus on cannot benefit from ODP, as we explain later in Section 2.2. The memory address is exchanged out-of-band enabling fast read/write from remote servers without OS involvement. This strategy effectively means that memory regions must be pre-allocated per sender; sharing a region among multiple senders is not possible, as one sender's data may be overwritten by another before being read (due to the *arb-src* constraint). Further, because of *arb-size*, and *arb-num* constraints, the baseline allocated memory must be large (even though there need not be any fragmentation). Effectively, handling the above constraints via proactive pre-allocation requires provisioning for the worst case of any possible communication, which results in large, grossly underutilized memory.

In contrast to RDMA's read/write model, the send/receive model (i.e., two-sided communication) requires the receiver to proactively post pre-allocated per-message memory buffers before the messages arrive. Because messages are copied to distinct memory buffers, the receive queue may be shared among the senders (unlike in RDMA read/write where writes to a given address do overwrite one another), which alleviates the *arb-src* constraint. The InfiniBand Shared Receive

Queue (SRQ) [22] achieves this purpose. Further, as long as enough memory is pre-allocated and posted to the receive queues to handle the line rate, incoming messages stay intact (satisfying the *arb-num* constraint). The send/receive semantics is that a newly received message is copied to the next available buffer, in the posted order, at the head of a receive queue. *Because the senders sharing a queue would not know the size of the next-available buffer at the receiver, each queue can fundamentally support only one buffer size.* Unfortunately, the *arb-size* requirement continues to be a problem due to this single-size constraint. *Separating and specializing for small and large messages to handle* arb-size*hurts tail latency or significantly degrades memory footprint or programmability (Section 5.2).*

We propose *Remote Indirect Memory Access (RIMA)* as a better-suited transport primitive for DCs. RIMA provides NIC microarchitecture support for novel *queue semantics* and a new "verb" called *append*, which allows programmers to append messages of arbitrary size to named, circular queues (i.e., one-sided communication with indirection). An append message looks up the tail pointer of the queue at the receiver and atomically increments the tail pointer by the message size (modulo the queue size) and copies the message into the newly created space in the circular queue (only the modulo increment is atomic; the copy is not). The tail pointer is in the receiver's virtual address space, and as in traditional RDMA, the NIC performs address translation and enforces protection. The head pointer handling is straightforward (Section 3.2). RIMA's simple and fast hardware—a small lookup table and an adder that adds 1–2 ns to the multi-μs message latency—can handle any number of messages of any size from any source. Further, the senders are aware only of the *indirect* address (the name of the queue) and not the value of the tail pointer (as is the case in RDMA).

The tail-pointer increment assumes a pre-allocated memory buffer. RIMA ensures that memory allocation overhead is hidden in the background and thus does not affect message latency. RIMA allocates, and keeps replenished, a *reserve pool* to accommodate enough space as per the line rate. To ensure adequate allocated memory, we use the traditional water-mark-based approach to manage the reserve pool. Whenever the reserve pool runs low, RIMA expands the reserve pool allocation in the background while continuing to accommodate message arrivals in the foreground and shrinks the allocation when message queues are drained. As such, queue sizes grow and shrink as needed without exposing the allocation latency to the messages.

In summary, the key contributions of this article are as follows:

- We observe that RDMA primitives do not satisfy the three key constraints (*arb-src*, *arb-size*, and *arb-num*) of typical request/response traffic in DCs. While subsets of these constraints may be handled without footprint problems (e.g., References [4, 13, 28]), DCs must handle all three.
- We propose RIMA, which simultaneously satisfies all the three constraints with little programmer effort. RIMA provides NIC microarchitecture support for novel *queue semantics*, i.e., one-sided communication with indirection, by combining indirect access and reactive tail-pointer manipulation using simple hardware.

RDMA's SRQ can also satisfy a subset of the constraints as it already handles single-size messages, large or small, from arbitrary sources at arbitrary rates in a shared hardware queue. RIMA's novelty is in enabling *arb-size* messages in a shared hardware queue to tackle the memory footprint problem *while* satisfying the *arb-src* and *arb-num* constraints. Some previous schemes have employed shared hardware queues for word-size [40] or cache-block-size [11]—*all single-size*—messages that are too small to cause footprint issues and are subsumed by SRQ. In one scheme [11], larger "bulk" transfers are fragmented into cache-block-sized packets requiring software TCP-like stack not present in RDMA. There is no shared hardware queue, in InfiniBand or elsewhere, that

can handle *arb-size* messages. Further, RIMA's contribution is in the network interface and not the network for which there are numerous high-performance schemes (e.g., References [1, 3, 10, 12, 32, 33, 49]), all orthogonal to RIMA.

Because RIMA adds little latency, it achieves the same message latency and throughput as SRQ with unlimited buffering. Running memcached traffic on a 30-node InfiniBand cluster, we show that at similar, low programmer effort for a single queue, RIMA achieves significantly smaller memory footprint than SRQ, reducing impractical, multi-GB footprints to small, few-hundred-MB ones. Further, multiple SRQs oblivious of the message-size distribution continue to yield impractical footprints for memcached. While SRQ's footprint can be shrunk by expending significant programming effort on carefully customizing multiple queues for the given distribution, RIMA provides those benefits without such effort, which must be repeated for each application in the case of SRQ. For evaluation using memcached traffic, we develop a HERD-like key-value cache called *FastKV* (described later in Section 5.2). *FastKV* using RIMA achieves either 3× lower 96th-percentile latency, or significantly better throughput or memory footprint than *FastKV* using RDMA.

2 CONTEXT AND SCOPE

We offer a brief background on InfiniBand RDMA and its operation to highlight the overheads in InfiniBand terms.

2.1 InfiniBand Memory Overhead

Queue Pairs (QPs) are a central primitive of InfiniBand communication, analogous to TCP/IP sockets. Each QP consists of a Send Queue and a Receive Queue. Both the source and destination nodes must create a QP to communicate.

The two most popular InfiniBand transportation modes are Reliable Connection (RC) and Unreliable Datagram (UD), which are roughly analogous to TCP and UDP. In the RC mode, QPs need to be connected before any data communication, which necessitates a source QP and destination QP for every communicating pair. UD, however, is connection-less, where each QP can communicate with any other QP.

There are two major sources of memory consumption for applications using InfiniBand.

InfiniBand Context Memory: This memory is allocated for creating InfiniBand communication context, such as Queue Pairs. A single QP context consumes as much as 68 KB of memory in MVAPICH 0.9.8 [35]. Context memory was a scalability bottleneck for traditional RC that needed per-connection QP context. However, with InfiniBand's Dynamic Connected Transport (DCT) [22], context memory is minimized even for RC as a single DCT QP can be used to communicate with many other QPs. Context memory does not pose a problem for UD, which is connection-less.

Data Buffer Memory: Data buffer memory stores data intended for communication. Such memory regions are required to be pre-allocated in physical memory before they can receive incoming messages. While on-demand paging [36] can allocate physical memory lazily, that approach is only meaningful for postponing physical memory allocation over program phases (and not for receive buffers that are expected to receive messages in the near-term). Each buffer consumes at least the size of an OS memory page. Because the problem of context memory has been addressed for both UD and RC, RIMA focuses on the data buffer memory overhead, which exists for both RC and UD.

In InfiniBand programming, read/write (i.e., one-sided) communication is called *memory semantics* where the receiver is not aware of the initiator's read or write operations. The data buffer in memory semantics is typically not shared due to potential data races among multiple senders writing to the same buffers. Avoiding races requires per-sender buffers, which can cause the memory footprint to grow, i.e., it is not scalable when there are a large number of potential senders (*arb-src*).

Table 1. Programmer Effort Comparison

Task No.	Task (SRQ)	Task (RIMA)
1	Determine number and buffer sizes of queues	No effort (only one queue)
2	Determine reserve pool size of all queues	Determine reserve pool size of one queue
3	Determine minimum-footprint configuration	No effort (only one queue
4	Co-ordinate across all clients	No effort (only one queue)
5	Coordinated performance tuning of servers and clients	Servers may be independently tuned

To lower the memory overhead of per-sender buffers, InfiniBand has proposed SRQ [22], which uses *send* and *receive* (i.e., two-sided) communication, also known as *channel semantics*. Instead of requiring each QP to physically allocate its own memory, the SRQ lets multiple QPs to work on a shared memory region. However, the receiver must pre-post "receives" with associated buffers large enough to hold the incoming messages. If the message size exceeds the size of the pre-posted receive buffer, then an error would be raised. Unfortunately, a single SRQ can handle only a single message size, because the senders sharing an SRQ would not know the size of the next-available memory buffer at the receiver, which consumes the buffers strictly in the posted order. *To restate this point in stark terms, allowing multiple sizes in a single SRQ requires the receiver to have perfect knowledge of the sequence of future message sizes so that receives may be pre-posted accordingly.* As we show below in Section 2.2, this *fundamental* single-size limitation poses a challenge in satisfying the *arb-size* constraint.

2.2 Alternatives to RIMA

RIMA requires hardware changes (albeit minimal). As such, it is worthwhile to consider software-only alternatives to managing the memory footprint problem.

Single SRQ: Despite sharing of data buffers across QPs, SRQ still incurs significant overhead because of its single size limitation (first row of Table 6). Because message size can span a wide range in DC traffic, SRQ can be used in one of two ways. A naive way is to use a single SRQ and pre-post receives of the maximum message size. This approach requires the same, little programmer effort as RIMA but leads to significant internal fragmentation (e.g., the average message size in Facebook traffic is about 231 B, despite object batching, whereas the maximum is 1 MB [6]). Consequently, the memory waste is unacceptably high (e.g., 13 GB per process as shown later in Section 5.1.1). We note that the receiver buffer is for the whole message, not for the maximum transfer unit (MTU). Thus, the memory waste would remain even if the MTU were 1 B.

Multi-SRQ: An alternative is to use multiple SRQs [15] of different buffer sizes and bin the message sizes into the queues (second row of Table 6). Unfortunately, this approach degrades programmability well beyond merely using a library to match messages to queues based on the message size. Given a message size distribution, determining the optimal number and buffer sizes, that minimizes the memory footprint, is not straightforward (e.g., memcached has as many as 42 distinct sizes [41]). There are two memory components: (1) the active part of the queue, which holds the messages before service and is sized for the average message arrival rate and service time. (Task 1 in Table 1) The other component is a reserve pool used when memory allocation is underway in case the active part fills up due to slow service. This pool is sized assuming the worst-case message arrival rate for a given message size during memory allocation (Task 2). We describe these size calculations for both RIMA (only one queue) and SRQ (single or multiple queues) in Section 3.2.1. With one SRQ using maximum-size buffers, the first component above is large, whereas with more queues using buffers closer in size to the actual messages, the second component increases. These opposing

Table 2. Impact of RDMA Verbs' *arb-∗* Constraints on Systems

Verb	Systems	Arbitrary Size	Arbitrary Source	Arbitrary Number	Comment
RWrite	HERD [28] (requests), FaRM [13], RFP [52]	Yes	No	Yes	RDMA Writes need per-sender buffers, which is impractical for DC scale systems
Send/recv	FaSST [30], scale-out ccNUMA [16]	No	Yes	Yes	SRQ is used as our baseline for comparisons.

trends imply a U-shaped frontier for the total memory as we add more queues (i.e., more queues do not necessarily reduce the memory footprint), which gives the minimum footprint (Task 3). We show this curve for memcached traffic in Section 5.1.1. Unfortunately, because each application's distribution is distinct, the entire optimization effort (Tasks 1–3) has to be repeated for each application, worsening programmer burden. However, multiple queues oblivious of the distribution (e.g., equi-spaced [15]) may yield impractical memory footprints (we show this result for memcached traffic in Section 5.1.1). Further, using multiple queues (irrespective of distribution awareness) also requires sender coordination in that messages must be binned according to their size and sent to the appropriate queue (Task 4). Finally, all communicating entities must stay synchronized on the bin sizes on an ongoing basis (Task 5). For example, changes in the distribution may require changes in the binning that must be pushed to all the clients and servers simultaneously.

Thus, SRQ incurs either a large memory footprint or high programming effort. In contrast, RIMA avoids both. Separating and specializing for small and large messages to handle *arb-size* hurts tail latency, or significantly degrades memory footprint or programmability, as we show in Section 5.2.

As an aside, we note that InfiniBand's on-demand paging (ODP) enables the applications to map only as much memory as needed in any program phase, instead of pinning large amounts of memory for the entire application duration, shrinking the application footprint. In our context, however, the memory needed to hold the incoming messages in the mere 1-ms allocate-and-map latency is large due to channel semantics (the single-message-size per SRQ restriction). This memory is the minimum needed to ensure that another allocation can occur before running out of memory. There is no bloat due to pinning memory unnecessarily for long duration. Consequently, ODP, including NIC TLB prefetch [36], cannot reduce the footprint.

2.3 Managing with a Fixed Memory Footprint Using Existing Verbs

While many recent papers argue for RDMA in DCs, the fact that they are constrained to use existing verbs implies that they run into one or more of the *arb-∗* constraints (see Table 2). This section examines the specific challenges imposed by existing verbs on recently proposed systems. For example, when using memory semantics (i.e., remote read/write), systems are forced to employ per-sender private receive buffers to avoid races. This approach, which is used for requests in HERD [28] and for both requests and responses in References [13, 52], is not scalable in typical datacenters with thousands of senders and receivers for the key-value size distribution seen at Facebook [46]. Similarly, with respect to the *arb-size* constraint (when using channel semantics with SRQ as in References [16, 29, 30]), the choice of limiting the memory footprint of the receive buffer in turn imposes message size limits (as we quantify later in Section 5). Such size limits will require external fragmentation/reassembly to handle larger messages or use rendezvous with two network round trips [26] (with corresponding latency/throughput penalties). There also exist non-RDMA networks that similarly limit message size (e.g., the StarT-Voyager [4] network interface uses messages no longer than 88 bytes), which violates the *arb-size* constraint. Attempting to limit

Table 3. RIMA API

ibv_create_aq(. . .);	Creating an append queue with initial attributes
ibv_destroy_aq(. . .);	Destroy an append queue
ibv_query_aq(. . .);	Query attributes of queue
ibv_modify_aq(. . .);	Modify attributes of queue

footprint by limiting the number of outstanding requests (i.e., *arb-num* violation; not shown in Table 2) will hurt throughput by introducing unnecessary inter-request gaps. We emphasize that these limitations flow from the verbs and are not attributable to the systems that are forced to work with deficiencies of existing RDMA verbs.

2.4 Scope

We focus on replacing TCP/IP with RDMA as a transport mechanism for two-sided requests/responses between servers (as also done in many previous systems [16, 29, 30]. Some proposals [13, 28, 52] use one-sided RDMA verbs to read/write directly from/to data structures on remote servers (e.g., FaRM [13]). As argued in HERD [28], we do not consider such approaches that require (a) the data structures to be pinned in memory, (b) multiple remote accesses for typical requests instead of more-efficient local accesses, and (c) cross-node synchronization for writes to shared state. Further, we do *not* intend for RIMA to replace RDMA universally. RIMA is better for request/response traffic. For large, non-request/response messages, RDMA memory semantics may remain the better option (e.g., using rendezvous protocols [55]).

3 REMOTE INDIRECT MEMORY ACCESS

Conceptually, RIMA achieves queue semantics, i.e., one-sided communication with indirection, by (1) using a layer of indirection to specify the destination memory address for messages and (2) reactively increasing the queue tail pointer to accommodate incoming messages. We describe RIMA's design in terms of the programmer interface (Section 3.1) and the NIC operation (Section 3.2).

3.1 RIMA Interface

The programmer may create and destroy RIMA queues at the receiver. The queue identifier (ID) must be communicated out-of-band (possibly over TCP/IP) to potential senders, like SRQ's Queue Pair information at setup. Each queue offers a single remote operation: sending a message of arbitrary size to a remote queue on the target host. With RIMA, senders are free to intermingle messages of any size (unlike in SRQ). Further, applications do not need any synchronization to coordinate with other senders who may be sending to the same queue (unlike in RDMA read/write). Although RIMA's independence from message size means that it does not need multiple queues, programmers can use multiple queues for logically distinct communication channels or for reducing contention. For example, applications may prefer to have control messages (e.g., heart beat) and data messages (e.g., database objects) in separate queues.

Interface Details: Table 3 lists the new API we propose to manage RIMA's "append queue" at the local node. RIMA's interface is based on InfiniBand's DCT framework. RIMA's append queue (AQ) is a substitute for the SRQ. A sender posts a send using the new IBV_WR_ APPEND instead of the old IBV_WR_SEND. Pre-posting receives is no longer required at the receiver's side.

3.2 RIMA NIC Microarchitecture

RIMA assumes that a sufficiently large, contiguous, virtual address region can be reserved for its queue buffers. The region is large enough to rule out the possibility of buffer overflow (e.g., a

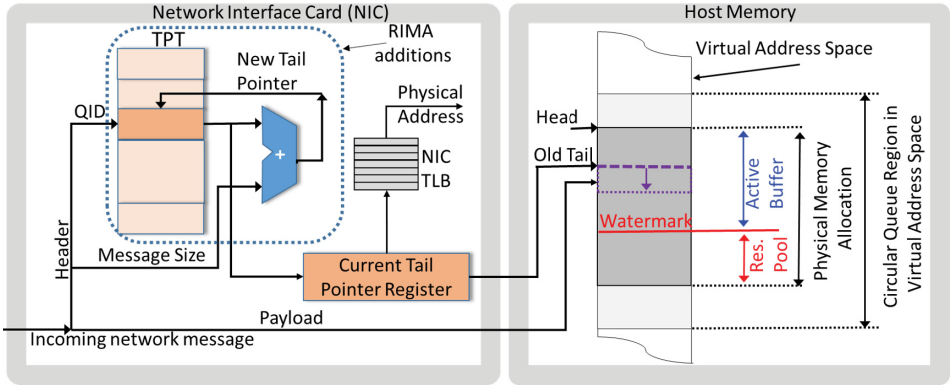

Fig. 1. RIMA network interface: Hardware organization.

significant fraction of physical memory size). Note that such a capacity constraint is not unique to RIMA. No messaging system allows for unbounded growth of received messages. Given the large 64-bit virtual address spaces in modern processors, such virtual address reservations have little cost. The contiguous region is managed as a circular queue to ensure continuous operation indefinitely (with a head-pointer where the oldest messages reside and a tail-pointer where new incoming messages are placed).

Figure 1 illustrates RIMA hardware. We assume that RIMA messages arrive with message size and queue-id in the packet header. RIMA messages look up the *tail pointer table (TPT)* to translate the queue ID to the tail pointer (a virtual address). Such lookups are not unique to RIMA; SRQ messages similarly look up Queue Pair information based on message headers. The tail pointer is atomically incremented by the message size (modulo the queue size), thereby re-actively creating buffer space at the receiver. In RIMA, the tail pointer is part of the queue's attributes and is held in the TPT on the NIC (Figure 1). As such, RIMA's atomic-increment of the tail pointer is fast. There is a TPT entry per queue, and typically there are only a few queues per receive process (Section 3.1), where each queue can be shared among multiple senders. Thus, the number of queues per node is independent of the number of senders or nodes in the cluster. Therefore, the TPT is a small SRAM table (say 1K entries); and the increment is a hardware add. Thus, RIMA adds only 1–2 ns (as per CACTI 6.0 [45]) to the multi-μs message latency. In contrast, RDMA's native atomic fetch-and-op operates on arbitrary locations in the host memory, which requires traversing the PCIe bus (in addition to traversing the DC network from the sender to the receiver). As we show later, this traversal is partly why emulating append using existing verbs degrades tail latency. With its simple hardware, RIMA avoids this penalty.

After the atomic tail-pointer modulo-increment, the message is copied to the host memory starting at the old tail pointer (i.e., in place in the circular queue); the NIC TLB holds the translation for the pointer, as shown in Figure 1 (similarly to RDMA). Thus, while SRQ's pre-posting of buffers can handle only one message size per SRQ (Section 2.1), RIMA's increment of the tail pointer *after* message arrival allows any message size in one queue, satisfying the key *arb-size* requirement. Upon completion of message copy to the queue, a message is posted to the completion queue, similarly to SRQ operation. The application's threads poll the completion queue and process messages after delivery. Upon completion, the head pointer, *which is maintained in the host application software*, is incremented to remove the processed messages from the queue and to auto-reclaim the space. For simplicity, we move the head pointer in queue order. Under typical operation, the small differences in processing times of concurrently processed messages (tens/hundreds of nanoseconds)

do not justify sophisticated out-of-order reclamation of space. Further, even in more challenging situations wherein a page-fault that is incurred when processing the message at the head of the queue can stall the head-pointer movement, sophisticated garbage collection is unwarranted as our RIMA's footprint is small (e.g., 11 MB per process). Thus, RIMA can handle such occurrences by expanding the queue size via physical memory allocation, which we discuss next.

While the functionality of the TPT (indirect lookup of tail pointer and atomic increment) could be implemented in firmware, implementing it in hardware, as described above, is better for performance similar to the NIC TLB implementation in hardware for performance reasons. Absent a hardware implementation, the firmware-based TPT lookup, which may incur instruction processing overheads and page faults, may not be fast enough to sustain the line rate.

3.2.1 Heavy-weight Allocation in Background. Though the virtual address region reserved for each queue is large, physical memory is pre-allocated minimally (in the background) to ensure that both buffered and future messages can be accommodated. RIMA relies on light-weight tail-pointer increment to create room re-actively for messages. Such simple tail-pointer increment works in RIMA, because we efficiently guarantee that adequate memory is allocated in the background to handle arbitrary message arrival rates. On one hand, RIMA requires a pre-allocated pool of physical memory to ensure that simple tail-pointer manipulation suffices. However, under this approach, the pre-allocated memory beyond the tail pointer must also be counted toward RIMA's physical memory footprint. On the other hand, the footprint minimization goal pushes the design in the opposite direction—it is better to grow and shrink the memory dynamically to minimize wasted memory.

RIMA resolves this dilemma by decoupling the heavy-weight memory allocation (and possibly OS-visible page mapping), which occurs in the background from light-weight tail-pointer manipulation in hardware (which is incurred for each message). RIMA uses a *reserve pool* of physical memory beyond the tail pointer to ensure that incoming messages can be accommodated without seeing the heavy-weight allocation delay. While this reserve pool does add to the physical memory footprint, the size of the reserve pool is manageable in absolute terms, especially in today's servers.

To hide the heavy-weight allocation latency in the background, we employ a well-known strategy of triggering an allocation at an appropriately set low watermark to ensure that the reserve pool is replenished before exhaustion. Specifically, when the tail pointer crosses the watermark upon an append, the NIC interrupts the host to allocate memory in the background (the head pointer not involved). Note that messages continue to be received in the reserve pool below the watermark while the reserve pool is being replenished with a fresh allocation. In the unlikely case that a fresh allocation exceeds the large virtual circular queue region due to inordinate receiver delay, the allocation software throws an error (similarly to exceeding the swap space). In practice, however, senders will not continue to send requests to unresponsive receivers, irrespective of TCP, RDMA, or RIMA. Instead, senders will switch to another receiver (e.g., another memcache server) and resend their requests.

Computing the low watermark is a straightforward use of Little's Law. Assuming that the latency of allocating physical memory is L, a reserve pool that is larger than ($linerate \times L$) can accommodate any burst at line rate indefinitely if the newly allocated memory also equals (or exceeds) this size (see Table 4). To handle variability in memory allocation latency, we can use the 99th percentile latency. Because RIMA's buffers are small (e.g., just over 200 MB), the use of higher percentiles is an acceptable overhead to prevent far worse tree saturation due to buffers filling up. The reserve pool exists below the watermark, above which some space is needed to buffer the messages before being processed ("Host Memory" in Figure 1). This space, called the *active buffer*, can

Table 4. Parameters for Analytical Model

	Active Buffer	Reserve Pool
$SRQ_{min...max}$	λ = <measured> W = memcpy cost $s = max$	$msgrate = scaled(min)$ $L = 1ms$ $msgsize = max$
RIMA	λ =<same as SRQ> W = memcpy cost s =<actual size>	$linerate = 56Gbps$ $L = 1ms$

also be computed using Little's Law. Assuming messages arrive at λ (the line rate in messages/s), the *nominal* processing time of each incoming message is W (s/message), and the *nominal* message size is s (bytes), the size of the active buffer should be at least $\lambda \times W \times s$ (bytes) (see Table 4). Thus, each watermark crossing triggers an allocation of a chunk as large as the active buffer and reserve pool together.

As seen above, more memory is allocated when the low watermark is crossed. Conversely, an empty chunk is freed so the OS can reclaim the physical memory as per virtual memory policy (in the background without affecting message processing). Thus, the physical memory allocation sweeps through the virtual circular queue, expanding and contracting on demand. Such sweeping implies churn in the address translations and NIC TLB misses. An optimization is to reset the head and tail pointers to the top of the virtual circular queue, whenever the queue becomes empty. The host software resets head and tail pointers atomically to avoid intervening message arrival using atomic-compare-and-swap (for the tail pointer) and locks (for the head pointer). Such resetting encourages the reuse of the physical pages and translations, improving the TLB's performance.

The watermark strategy, with the active buffer and reserve pool components, can be applied to SRQ as well (single-queue or multi-queue option as described in Section 2.2). For SRQ's active buffer size, we assign λ and s corresponding to the nominal message size for each bin in $\lambda \times W \times s$ (bytes) (single-SRQ has only one bin) (see Table 4). For SRQ's reserve pools, we derive the sizes using ($msgrate \times L \times msgsize$). We assign $msgrate$ based on each bin's minimum message size (*arb-num* constraint) and $msgsize$ based on the bin's maximum message size (*arb-size* constraint) (see Table 4). In the worst case, the queue may receive minimum-sized messages (for the bin) that must be placed in entries that are capable of holding maximum-sized messages (for the bin). *The key difference between SRQ and RIMA is in the reserve pool size where the former is sized for maximum-size messages at the rate of minimum-size messages while the latter is sized for the line rate.* Note that the reserve pool is an extension of the active buffer. *Both* hold buffers of a specific message size to enable using the next-available buffer upon message arrival. Therefore, the reserve-pool cannot be shared across SRQs.

3.3 Other Issues

Congestion control is key to scalability in DCs. While TCP provides end-to-end congestion control, RDMA does not. However, congestion control is orthogonal to our memory footprint issue. RIMA can leverage RDMA congestion control [17, 18, 44, 60]. Similarly, baseline RDMA handles failures and MTU issues that RIMA can reuse.

4 EVALUATION METHODOLOGY

We evaluate RIMA in terms of memory footprint (active buffer + reserve pool), performance (throughput), and programmability. For memory footprint, we measure the active buffer size on a real cluster and calculate the reserve pool size that is anyway set analytically. For performance,

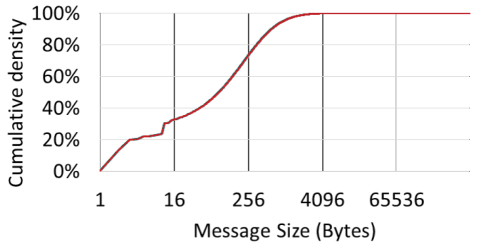

Fig. 2. Cumulative distribution of Memcached [6] tuple size.

we emulate RIMA by using SRQ, but accounting for the footprint without the internal fragmentation. *Such an emulation is valid, because RIMA does not affect SRQ message latency and throughput (nanoseconds overhead over microseconds, as discussed in Section 3.2) obviating detailed microarchitecture simulations.* To evaluate some of RIMA's alternatives that affect programming effort (multi-SRQ) and performance (large-message specialization), we use real-system runs.

4.1 Experimental Setup

Client/Server Setup (for Real Runs and Emulations): We evaluate RIMA on a 30-node test bed. The test bed's scale is not important, because RIMA's goal is to ensure that a single server can efficiently satisfy the *arb-size*, *arb-num*, *arb-src* constraints while serving traffic at the line rate, irrespective of whether the traffic is from one or one thousand other servers (i.e., what matters is the line rate, not how many servers generate the traffic). As such, we measure NIC message rates and memory footprint on a single randomly chosen server in our test bed serving at the line rate.

Each node consists of eight 4-core AMD Opteron processors (with two hardware threads per core) running at 2.8 GHz and 256 GB of memory. Each node is equipped with a Mellanox Connect X-3 HCA (56 Gb/s) with PCI-Ex Gen2 interface. All the nodes run RHEL 6.7 with kernel version 2.6.32, and Mellanox OFED 2.4-1.0.0. Fifteen nodes serve as clients (analogous to front-end servers in a DC) issuing requests to the remaining fifteen nodes, which act as servers. Clients send messages to randomly selected servers using send operations in the Reliable Connection (RC) transport mode. Client-to-server mappings are managed as permutations to ensure server load balance. Clients saturate the line rate at the server.

To avoid contention on a single queue (e.g., 32 receiver threads/processes on a single queue), we use multiple queues with the associated memory footprint (irrespective of multiple processes [5] or multiple threads [46]). We run 32 processes on each (32-core, 256-GB) node of our cluster (for reference, Amazon AWS recommends a similar large-memory configuration named "`r3.8xlarge`" with 32 cores and 244 GB memory [7] for memcaches). We report the memory footprint per process as each receive queue is private to a process.

Workload: We use memcache, which is a key-value store employed by hundreds of datacenter applications such as social networks, dynamic advertisements, and news feeds. Thus, our evaluation broadly captures many of these applications. We use Facebook's memcache message size distribution [6] (see Figure 2). Memcached's irregular and wide-ranged message-size distribution (*arb-size*), spanning 1 B to 1 MB, makes it challenging to determine the optimum number and buffer sizes for the SRQs in the multi-SRQ configuration (Section 2.2). We evaluate the multi-SRQ configurations with (1) equi-spaced [15] buffer-size bins (low effort), and (2) bin sizes as per memcached's carefully tuned slab allocator [46] as detailed in Section 5.1.2 (high effort).

Comparison Using an In-memory Key Value Cache: For performance comparison using an RDMA-based key-value cache, we implement *FastKV*, an in-memory key-value cache that uses

send/receive verbs. Because RIMA's focus is at the verb-level, we compare to *FastKV*'s verb choice of send/receive with SRQ, which is similar to that of FaSST [30] and Scale-out ccNUMA [16]. RIMA does not focus on higher-level system goals of transactional execution (as in Reference [30]) and/or consistent caching (as in Reference [16]) of other systems. We generate keys and values as per the memcached size distribution [6]. Though our runs use the 30-node cluster described above, we can isolate one of *FastKV*'s key performance challenges at larger cluster sizes. We experimentally found that the client throughput reaches 12.9 million requests/s, which translates to 23.8 Gbps of network bandwidth per client, a reasonable fraction of the 56-Gbps linerate considering that most messages are small. While server throughput measures cluster performance, we measure client throughput, which indicates how fast a Web page request is serviced given that each page typically accesses hundreds of objects [46].

Sizing of Active Buffers (Measured) and Reserve Pools (Calculated): Recall from Section 3.2 that the sizes of the active buffer and reserve pool for RIMA are, respectively, $\lambda \times W \times s$ (bytes) and $L \times linerate$ (Table 4). We use real-system measurements to determine the average number of messages in the active buffer, which is $\lambda \times W$, at full, steady-state throughput for SRQ, which is the same for RIMA. In this measurement, (1) we use the memcached message-size distribution and (2) the processing involves copying the message from the buffer to application memory (i.e., minimal computation for short active buffers in both SRQ and RIMA). See Table 4. s is the actual size of the messages seen in the measurement. For RIMA's reserve pool, we use 1 ms for L as the memory allocation latency (Section 5.2.1).

As described in Section 3.2.1, the sizes of SRQ's active buffer and reserve pool are, respectively, $\lambda \times W \times s$ (bytes) and $msgrate \times L \times msgsize$. We measure $\lambda \times W$ as stated above. For RIMA, s is the measured size of the messages sent to each bin. See Table 4. For SRQ's reserve pools, we compute $msgrate$ corresponding to the per-bin minimum message size and use 1 ms for L and the per-bin maximum message size for $msgsize$. For computing $msgrate$, we use 137 million messages per second for messages with 8-byte (or smaller) payload [23], because small-message throughput is typically lower than the line rate. To account for the fact that larger messages incur lower bandwidth loss, we scale this 8-byte throughput linearly as a function of the message size so that the loss of bandwidth for smaller messages compared to the line rate reduces with larger messages and vanishes for the largest message.

5 RESULTS

The key results of our evaluation show that (1) under similar performance and effort, RIMA avoids SRQ's high memory inefficiency (Section 5.1.1), (2) SRQ requires significantly higher effort to achieve similar performance and memory efficiency as RIMA (Section 5.1.2), and (3) *FastKV* using RDMA incurs either worse tail latency or larger footprint than *FastKV* using RIMA (Section 5.2.1).

5.1 Memory Footprint

5.1.1 RIMA Versus SRQ under Comparable (Low) Effort.

Single SRQ: Using a single SRQ results in extremely large memory footprint for Facebook's memcached message size distribution, because a single queue must accommodate the largest possible message (1 MB) even though the average message size is only 231 B. Such internal fragmentation results in the single SRQ message buffering bloating to 161 GB in contrast to RIMA's minimal 11 MB of message buffers. Because the single-SRQ configuration represents a naive basecase, we limit discussion of the configuration within the early part of this section. Later, we focus on an alternative, simple low-programmer-effort multi-SRQ configuration.

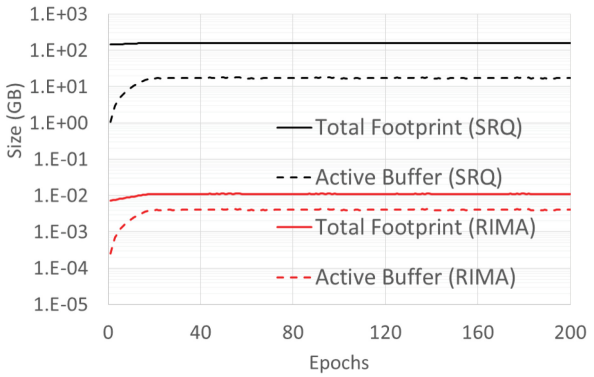

Fig. 3. Single SRQ versus RIMA.

For Facebook's memcached message size distribution (Figure 2), a single SRQ allocates 1 MB (the maximum message size) for all messages to satisfy the *arb-size* constraint. We do not claim that using 1 MB to hold 231 B, on average, is viable. This comparison is to highlight RIMA's programmability advantage that RIMA requires little effort to avoid the wastage. Later, we also consider a simple, distribution-oblivious way to use multiple SRQs.

We measure the footprint after every 2,048 messages, called *epochs*. Figure 3 shows the active buffer size (dashed lines) and the total footprint (solid lines, Y-axis, log-scale) in epochs (X-axis) for SRQ (black lines) and RIMA (pink lines). Beyond the transient behavior during the first 40 epochs, the active buffer size reaches steady state for both RIMA and SRQ at different levels. The reserve pool size does not change with time. As expected, both systems achieve equal, high throughput (4.3 million messages/s for memcache message size distribution with an average of 231 B).

In each configuration, the queues grow to (approximately) 16,800 messages per process at steady state. Though the number of messages is the same for both RIMA and SRQ, SRQ requires 17 GB in active buffers due to severe internal fragmentation. In contrast, RIMA's active buffer grows as per the actual message sizes to an average of 4 MB. Because RIMA's reserve pool size is dictated only by the true line rate (Section 3.2.1) whereas SRQ must provision for the worst case, RIMA's reserve pool is much smaller (143 GB versus 7 MB). To prevent the SRQ reserve pool from exhausting our system memory, we keep the 8-byte-message rate well under the maximum (137 million messages/s in Section 4). Thus, under similar, little effort of using a single queue, SRQ remains unviable—more than 161 GB footprint—whereas RIMA remains efficient (11 MB footprint).

Distribution-oblivious Multi-SRQ Approach: We use a message-size-distribution-oblivious approach that uses multiple SRQs, each covering an equi-sized sub-range of message sizes. This simple approach avoids the effort of customizing the sub-ranges to the distribution (Table 1) and can be pre-implemented in a library. Still, the approach requires the significant tuning effort of finding the optimal number of queues, which must be repeated for each application. As such, single SRQ may be the only SRQ configuration requiring as little effort as RIMA. Figure 4 plots the total footprint (including both active buffers and reserve pools) (Y-axis, in GB) with varying number of SRQs using this approach (X-axis). As expected from Section 2.2, the U-shaped curve indicates the initial steep fall off in overheads as we add more queues, with an eventual (slower) increase in total capacity, because each SRQ incurs reserve pool overhead. First, because the optimal number of queues depends on the message size distribution, the distribution-oblivious approach is insufficient despite using multiple SRQs. Second, even at the lowest point (200 queues), the footprint of 1.62 GB per-process is more than two orders of magnitude higher than that of RIMA. For

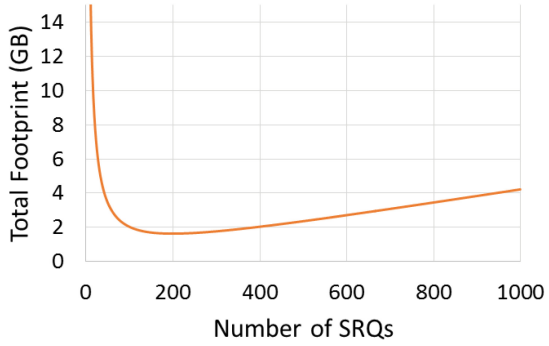

Fig. 4. Distribution-oblivious approach.

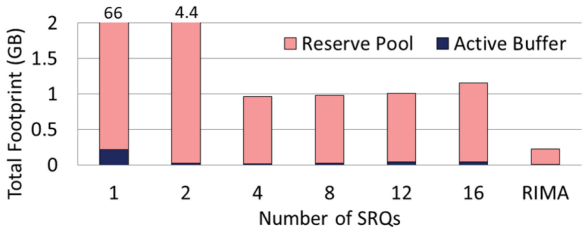

Fig. 5. Distribution-aware approach (upto 16-KB messages).

a 32-process server, the absolute overhead would be unacceptably large at over 50 GB. In contrast, RIMA's footprint is only 11 MB per process (= 7 MB (reserve pool) + 4 MB (active buffer). Even when considering a 32-process server configuration, the aggregate footprint is a modest 228 MB ($= 7 * 32 + 1 * 4$, because only the reserve pool scales with number of processes). The single-SRQ and multi-SRQ results above highlight our first key claim that under similar, little effort, RIMA is much more memory-efficient than SRQ.

5.1.2 RIMA versus SRQ under Comparable Memory. To address the high overhead of single SRQ and distribution-oblivious multi-SRQs, we explore message-size-distribution-aware multi-SRQ configurations. Instead of directly solving this non-linear optimization problem (Table 1), we leverage memcached's internal memory *slab allocator*, which is already tuned to reduce fragmentation in memory allocation [46]. We use the allocator's 42 slab granularity as message-size bins corresponding to queues in our multi-SRQ configurations. For configurations with fewer than 42 queues, we manually merge an appropriate number of adjacent bins to balance the queue sizes. We perform two experiments.

The first experiment uses real-hardware runs that limit message sizes to 16 KB to avoid exhausting our cluster's memory (Section 4). Figure 5 plots the number of SRQs and RIMA on the X-axis and the total (steady-state) footprint for our 32-process server on the Y-axis. The footprint for memcached traffic is large for a single SRQ (leftmost bar) but the optimal configuration (4 SRQs) is close to RIMA. This optimal configuration is the result of careful tuning to find the appropriate message buffer sizes guided by memcached's slab allocator and the number of SRQs. However, RIMA does not require any tuning at all to achieve high memory efficiency.

Our second experiment extends the above analysis to memcached's full message-size distribution. Because the first experiment confirms that the reserve pool overwhelmingly dominates the overall footprint, we restrict the second experiment to only the reserve pool, which fortunately

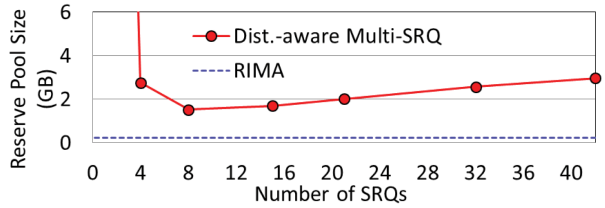

Fig. 6. Distribution-aware approach (all sizes).

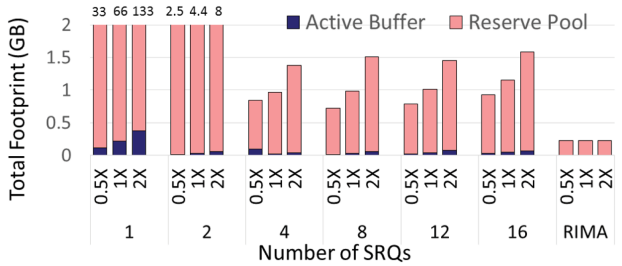

Fig. 7. Sensitivity to message-size distribution.

can be studied analytically as our cluster cannot accommodate messages larger than 16 KB. Figure 6 compares SRQ and RIMA on the reserve pool size for our 32-process server on the Y-axis as we vary the number of SRQs from 1 to 42 queues (X-axis). A single SRQ incurs extremely large footprint for the full 32-process configuration(>4,500 GB, beyond range). With more SRQs, we see the expected U-shaped curve. As with Figure 5, at fewer SRQs, the benefit of avoiding internal fragmentation overpowers the cost of additional reserve pools. However, beyond 8 SRQs, the reserve pool begins to grow. So, blindly using as many SRQs as the allocator slabs (42) is sub-optimal. The optimum is 8 SRQs, with a reasonable 1.5-GB reserve pool. The reserve pool sizes in Figure 6 and Figure 5 are different due to message size differences. We note that this tuning would be even harder without the slab allocator's message buffer sizes. RIMA achieves even smaller footprints without such tuning, which is our second key claim.

5.1.3 Sensitivity to Message-size Distribution. Our results so far have relied on memcached's message size distribution [6] (with or without truncation at 16KB). To verify that RIMA's memory savings hold for other message-size distributions, we artificially scale up and scale down the message sizes of the truncated memcached distribution by a factor of 2 without changing the fraction of messages at a given size. This scaling corresponds to trends toward half as small and twice as large messages.

Figure 7 plots the total footprint (including reserve pools) for our 32-process server at steady state (Y-axis) for the various multi-SRQ configurations (X-axis) and RIMA (rightmost set of bars) for three distributions truncated at 16 KB: scaled-down (0.5×), original (1×), and scaled-up (1×). The key point here is that RIMA's footprint does not change with the distribution, because the footprint is dominated by the reserve pool whose size depends largely on the line rate and not message sizes. In contrast, SRQ's footprint increases for larger messages as both the active buffers and the reserve pools grow. Further, RIMA's advantage over SRQ remains for all the distributions.

We do not study RIMA's sensitivity to the memory allocation-and-pinning latency of 1 ms (Section 5.2.1). This latency linearly affects the reserve pool size (Section 4), which is so much larger for

Table 5. *FastKV*-RDMA versus *FastKV*-RIMA

Num. Servers	64	128	256	512	1024	2048
Equal footprint comparison (256 MB)						
FastKV-RDMA Norm. Throughput	21%	11%	5.2%	infeasible		
FastKV-RIMA Norm. Throughput	100%					
FastKV-RIMA Throughput relative. to *FastKV*-RDMA	4.8×	9×	19×	-	-	-

Table 6. Design Space and Challenges

	Design choice	Impact
SRQ	(1) Single SRQ	Large footprint
	(2) Multi-SRQ	Programming/tuning effort
Large-message specialization	(3) On-demand allocate	High tail latency
	(4) Rendezvous using RDMA's fetch-and-op	High tail latency
Per-sender queue	(5) Pre-allocate	Large footprint
RIMA	(6) Hardware support	Small footprint and high performance

SRQ than for RIMA that even a 100% change in the latency will not qualitatively change RIMA's advantage, because both schemes either gain or lose.

5.2 Performance

5.2.1 Performance Comparison Using FastKV. Broadly, *FastKV* can be configured to work in a *footprint-priority* mode wherein it ensures that it operates within a fixed memory footprint, even if it means throttling requests (and thus performance) to ensure staying within the footprint. In our first experiment, we consider a *FastKV* that is configured to use SRQ, but one that ensures that it never exceeds the 256-MB memory footprint. To ensure that the footprint is not exceeded, this variant imposes limits on concurrent requests. Not surprisingly, there is a steep performance penalty for such request throttling as shown in Table 5. The combination of fixed footprint (256 MB) and maximum message size (1-MB) forces at most 256 outstanding requests in the whole system. As shown in Table 5, this limits scaling as it is impossible to have more than 256 servers/clients as that could potentially exceed the maximum footprint. Further, there is significant throughput throttling even at smaller scales as the number of outstanding requests can at most be 4, 2, and 1 when the number of servers/clients are 64, 128, and 256, respectively.

Given the serious throughput penalty of using a fixed memory footprint, we now examine more flexible variants of *FastKV* that separate and specialize the handling of small and large messages (e.g., similar to file system i-nodes for small and large files). For instance, messages under 1 KB use a single SRQ with 1-KB buffers and larger messages are handled separately (1-KB messages are at the 95.6th-percentile in Figure 2). The 1-KB SRQ imposes a much lower overhead than all messages using 1-MB buffers. However, such specialization raises two issues.

First, the 1-KB cutoff uses message-size distribution knowledge, which is a programmability issue that RIMA avoids. Without the knowledge, the cutoff may not work (e.g., if an application's average is 10 KB then using 1-KB cutoff would incur high overheads). Second, allocating memory for the large (>1 KB) messages has three options all of which are problematic. Table 6 summarizes

these alternatives to RIMA. Row 1 in the table shows the naive single-SRQ approach, which results in a large footprint due to internal fragmentation.

The first option is to pre-allocate 1-MB buffer for each large message. This option is the same as using two SRQs with message size distribution knowledge (row 2 in Table 6). Even so, this option incurs memory footprints of around 37.8 GB for our 32-process server the vast majority (33.6 GB) of which is for the reserve pool for messages larger than 1 KB. Recall from Section 3.2.1 that the reserve pool must be sized for the message arrival rate of the smallest possible message (1 KB + 1 B) but the largest possible message buffer (1 MB). The remaining overhead is from the reserve pool for messages smaller than 1 KB and the active buffer. RIMA's 228 MB (Section 5.1.1) is much smaller than this 37.8 GB. Though this bloat can be eliminated by using multiple SRQs optimized to match the message size distribution, doing so would come at the cost of significant programming effort (Section 2.2).

The second option is to allocate memory on demand for each large message by using the recently introduced InfiniBand feature of on-demand paging [36] (row 3 in Table 6). By allocating memory of the exact size of the message, this option avoids the first option's memory bloat. To explore this option, we measured the time taken to allocate and map (`kalloc()` and `mlock()`) some memory on our system. While the latency varies with allocation size, we found the sweet spot to be 16 MB with median and the 90th percentile latencies of 0.97 ms and 1 ms over 4000 runs, respectively. Even though the allocation latency is worse for smaller and larger allocations, we conservatively assume a 1-ms latency for all sizes. However, the 95.6th-percentile 1-KB messages incur 0.15-μs bandwidth-limit delay at 56 Gbps plus around 50 μs of one-way RDMA-based network latency [60] = 50.15 μs total one-way latency, amounting to 20× allocation overhead to the corresponding tail latency. This tail latency issue relevant to datacenter applications is not applicable to file system i-nodes where such specialization may be effective. If the on-demand paging feature is not available, then remote allocation would require three network traversals instead of one (e.g., two extra traversals using rendezvous protocol [55]—one to make an allocation request and another to return the buffer pointer after allocation). On-demand paging avoids the extra traversals but not the allocation latency.

The third option is to allocate on-demand some large buffer (e.g., 100 MB) to be shared among several messages larger than 1 KB (row 4 in Table 6). This option amortizes the 1-ms allocation overhead of the second option over several messages. However, this option requires creating space in this buffer for every large message *before* placing the message there. The space creation requires a remote RDMA fetch-and-add, which involves rendezvous-like two-step messaging between the sender and receiver, and a *local* RDMA atomic fetch-and-add at the receiver. Unfortunately, local RDMA atomic operations access the host memory over the I/O bus, which is slow (e.g., PCIe). Thus, the 1-KB messages incur around 100 μs extra RDMA-based network latency plus 2 μs for an unloaded, local RDMA fetch-and-add. Other work [42] reports PCIe RTT of 2.1 μs without any fetch-and-add and explains in great detail that PCIe's narrow links exacerbate latency well beyond serialization alone. Consequently, the 1-KB messages incur a total of 102 μs extra latency, which, compared to 50.15-μs true latency, is around 3× slower. In contrast, RIMA *locally* creates the required space *after* the message is received at the receiver's NIC. RIMA's atomic operation is on the tail pointer in the queue's context memory held in the NIC. Thus, RIMA entirely avoids the I/O bus as well as the extra network traversals, and therefore incurs little overhead. Though the third option's 3× tail latency overhead is better than the second option's 20× overhead, both options degrade programmability by requiring message size distribution-dependent 1-KB cutoff.

The tail latency overheads of the second and third options can be decreased from the 95.6th-percentile 1-KB messages to the less frequent 99.9th-percentile 5.4-KB messages (Figure 2).

However, doing so would increase the small-message buffers from 1 KB to 5.4 KB causing the reserve pool to expand to 22.4 GB for our 32-process server.

Another way to avoid the tail latency overheads of the second and third options is by having each sender pre-allocate buffers at the receiver (row 5 in Table 6). Such pre-allocation ensures that each sender has space available before sending any messages. In a 5,000-server DC with 2,000 front-end servers (say), each receiver must allocate a reserve buffer for each of the 2,000 senders, negating the benefit of large-message specialization. Such allocation is replicated for each process running on the receiving server; with a modest 1-MB reservation per sender and 32-processes per receiver, the receiver memory footprint is around 64 GB (2000×1 MB $\times 32$). TCP does not face this 2,000× bloat, because the receivers share the buffers under OS control.

6 RELATED WORK

RDMA has its roots in earlier low-latency, user-space messaging such as Fast Messages [48], U-Net [56], Shrimp [9], and VIA [14].

Scalability in InfiniBand-based MPI Implementations: MPI implementations first discussed InfiniBand's memory scalability issue (e.g., Reference [38]). Several improvements followed: adaptive connection setup [59], unreliable datagram (UD) [35], eXtended Reliable Connection (XRC) for all-to-all communication [34], and DCT for reliable connection (RC) where each process can use just one QP to communicate with any other process. MPI implementations use DCT [53] and SRQ [54] for QP and memory efficiency.

InfiniBand for Datacenter Applications: As DCs move to leverage RDMA's low latency, high scalability and high performance must both be achieved. Scaleout NUMA [47] performs RDMA-like remote direct memory accesses using on-chip network interface. Many studies examine replacing TCP/IP with RDMA. Examples include RDMA-based Hadoop [39], HBase [21], and memcached-like key-value stores [25, 27, 28, 43]. FaRM [13] provides an InfiniBand-based programming framework for DCs, whereas other papers use InfiniBand to accelerate file and storage systems [24, 50, 58]. Though these studies confirm the need for fast communication in DCs and provide a strong motivation for RIMA, they incur RDMA's programmability, memory, and performance inefficiencies that RIMA eliminates.

Append in Other Systems: Portals [19] and MPI use legacy verbs to implement append functionality in upper software layers. Because of using legacy verbs, they incur the memory footprint problem that RIMA addresses.

Other NICs: Others have proposed (1) programmable NICs to offload protocol processing [31, 37, 57], and (2) to leverage coherence to improve communication performance [20].

7 CONCLUSION

RDMA's low latency makes it a strong contender to replace TCP/IP in DCs. However, InfiniBand's SRQs are limited fundamentally to a single message size per queue and incur memory wastage or programmer burden for typical DC traffic of an arbitrary number (level of burstiness) of messages of arbitrary size from an arbitrary source.

To avoid this limitation, we proposed *RIMA*, which provides NIC hardware support for novel *queue semantics* and a new "verb" called *append*. To append a sender's message to a shared queue, the receiver network interface card (NIC) atomically increments the queue's tail pointer by the incoming message's size and places the message in the newly created space. This *indirection* of specifying a queue, while its tail pointer remains hidden from senders, handles all the three constraints of typical DC traffic. To ensure that there is always space for an arriving message, RIMA employs

the traditional watermark scheme to add memory to the queue in the background while messages continue to be received in the reserve space below the watermark. RIMA uses simple hardware to achieve the same message latency and throughput as SRQ with unlimited buffering. Running memcached traffic on a 30-node InfiniBand cluster, we showed that at similar, low programmer effort, RIMA achieves small, practical memory footprints in contrast to SRQ's impractically large footprints. Though SRQ's memory footprint can be shrunk by expending significant programming effort, which must be repeated for each application, RIMA provides those benefits with little effort. We developed a high-performance in-memory key-value cache (*FastKV*) for evaluation of RIMA using memcached traffic. *FastKV* using RIMA achieves either 3× lower 96th-percentile latency, or significantly better throughput or memory footprint than *FastKV* using RDMA. RIMA's high memory efficiency, low programmer effort, and simple hardware bode well for widespread adoption in DCs.

REFERENCES

[1] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. 2008. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication (SIGCOMM'08)*. ACM, New York, NY, 63–74. DOI: https://doi.org/10.1145/1402958.1402967

[2] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM Conference on Data Communication (SIGCOMM'10)*. 63–74. DOI: https://doi.org/10.1145/1851182.1851192

[3] R. Alverson, D. Roweth, and L. Kaplan. 2010. The gemini system interconnect. In *Proceedings of the 2010 18th IEEE Symposium on High Performance Interconnects*. 83–87. DOI: https://doi.org/10.1109/HOTI.2010.23

[4] Boon S. Ang, Derek Chiou, D. L. Rosenband, M. Ehrlich, L. Rudolph, and Arvind. 1998. StarT-Voyager: A flexible platform for exploring scalable SMP issues. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (Supercomputing'98)*. 26–26. DOI: https://doi.org/10.1109/SC.1998.10042

[5] apache [n.d.]. Apache Performance Tuning. Retrieved from https://httpd.apache.org/docs/2.4/misc/perf-tuning.html.

[6] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'12)*. 53–64. DOI: https://doi.org/10.1145/2254756.2254766

[7] aws [n.d.]. Amazon EC2 Instance Types. Retrieved rom https://aws.amazon.com/ec2/instance-types/.

[8] Luiz André Barroso, Jeffrey Dean, and Urs Hölzle. 2003. Web search for a planet: The Google cluster architecture. *IEEE Micro* 23, 2 (Mar. 2003), 22–28. DOI: https://doi.org/10.1109/MM.2003.1196112

[9] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg. 1994. Virtual memory mapped network interface for the SHRIMP multicomputer. In *Proceedings of the International Symposium on Computer Architecture (ISCA'94)*. 142–153. DOI: https://doi.org/10.1145/191995.192024

[10] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, and J. N. Seizovic and. 1995. Myrinet: A gigabit-per-second local area network. *IEEE Micro* 15, 1 (Feb. 1995), 29–36. DOI: https://doi.org/10.1109/40.342015

[11] Eric A. Brewer, Frederic T. Chong, Lok T. Liu, Shamik D. Sharma, and John D. Kubiatowicz. 1995. Remote queues: Exposing message queues for optimization and atomicity. In *Proceedings of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'95)*. ACM, New York, NY, 42–53. DOI: https://doi.org/10.1145/215399.215416

[12] H. J. Casier, E. Laes, and E. Schutz. 1992. Analog-digital technologies for mixed-signal processing: The driving force to success for the European industry. *IEEE Micro* 25, 04 (Jul. 1992), 34–42. DOI: https://doi.org/10.1109/40.149734

[13] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast remote memory. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI'14)*. 401–414. https://www.usenix.org/conference/nsdi14/technical-sessions/dragojević.

[14] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. M. Merritt, E. Gronke, and C. Dodd. 1998. The virtual interface architecture. *IEEE Micro* 18, 2 (Mar. 1998), 66–76. DOI: https://doi.org/10.1109/40.671404

[15] Vasilis Gavrielatos, Antonios Katsarakis, Arpit Joshi, Nicolai Oswald, Boris Grot, and Vijay Nagarajan. 2018. Scale-out ccNUMA: Exploiting skew with strongly consistent caching. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys'18)*. ACM, New York, NY, USA, Article 21, 15 pages. https://doi.org/10.1145/3190508.3190550

[16] Vasilis Gavrielatos, Antonios Katsarakis, Arpit Joshi, Nicolai Oswald, Boris Grot, and Vijay Nagarajan. 2018. Scale-out ccNUMA: Exploiting skew with strongly consistent caching. In *Proceedings of the 30th European Conference on Computer Systems (EuroSys'18)*. ACM, New York, NY, Article 21, 15 pages. DOI: https://doi.org/10.1145/3190508.3190550

[17] E. G. Gran, M. Eimot, S. A. Reinemo, T. Skeie, O. Lysne, L. P. Huse, and G. Shainer. 2010. First experiences with congestion control in InfiniBand hardware. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'10)*. 1–12. DOI: https://doi.org/10.1109/IPDPS.2010.5470419

[18] E. G. Gran, S. A. Reinemo, O. Lysne, T. Skeie, E. Zahavi, and G. Shainer. 2012. Exploring the scope of the InfiniBand congestion control mechanism. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'12)*. 1131–1143. DOI: https://doi.org/10.1109/IPDPS.2012.104

[19] Amin Hassani, Anthony Skjellum, Ron Brightwell, and Brian W. Barrett. 2013. Design, implementation, and performance evaluation of MPI 3.0 on Portals 4.0. In *Proceedings of the European Conference on Message Passing Interface (EuroMPI'13)*. 55–60. DOI: https://doi.org/10.1145/2488551.2488563

[20] M. D. Hill, B. Falsafi, D. A. Wood, and S. S. Mukherjee. 1996. Coherent network interfaces for fine-grain communication. In *Proceedings of the International Symposium on Computer Architecture (ISCA'96)*. 247–247. DOI: https://doi.org/10.1109/ISCA.1996.10009

[21] J. Huang, X. Ouyang, J. Jose, W. Rahman, H. Wang, M. Luo, H. Subramoni, C. Murthy, and D. Panda. 2012. High-performance design of HBase with RDMA over InfiniBand. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'12)*.

[22] ibspec [n.d.]. *Infiniband Specification*. Technical Report.

[23] infiniband [n.d.]. Infiniband Performance. Retrieved from http://www.mellanox.com/page/performance_infiniband.

[24] N. S. Islam, M. W. Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy, and D. K. Panda. 2012. High performance RDMA-based design of HDFS over InfiniBand. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'12)*. Article 35, 35 pages.

[25] Jithin Jose, Hari Subramoni, Krishna Kandalla, Md. Wasi-ur Rahman, Hao Wang, Sundeep Narravula, and Dhabaleswar K. Panda. 2012. Scalable memcached design for InfiniBand clusters using hybrid transports. In *Proceedings of the IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID'12)*. 236–243. DOI: https://doi.org/10.1109/CCGrid.2012.141

[26] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. Wasi ur Rahman, N. S. Islam, X. Ouyang, H. Wang, S. Sur, and D. K. Panda. 2011. Memcached design on high performance RDMA capable interconnects. In *Proceedings of the International Conference on Parallel Processing (ICPP'11)*. 743–752. DOI: https://doi.org/10.1109/ICPP.2011.37

[27] Jithin Jose, Hari Subramoni, Miao Luo, Minjia Zhang, Jian Huang, Md. Wasi-ur Rahman, Nusrat S. Islam, Xiangyong Ouyang, Hao Wang, Sayantan Sur, and Dhabaleswar K. Panda. 2011. Memcached design on high performance RDMA capable interconnects. In *Proceedings of the International Conference on Parallel Processing (ICPP'11)*. 743–752. DOI: https://doi.org/10.1109/ICPP.2011.37

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

[29] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. Design guidelines for high performance RDMA systems. In *Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC'16)*. USENIX Association, 437–450. https://www.usenix.org/conference/atc16/technical-sessions/presentation/kalia.

[30] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, 185–201. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/kalia.

[31] Hyong-youb Kim and Scott Rixner. 2006. Connection handoff policies for TCP offload network interfaces. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'06)*. 293–306. http://www.usenix.org/events/osdi06/tech/kim.html.

[32] J. Kim, W. J. Dally, S. Scott, and D. Abts. 2008. Technology-driven, highly-scalable dragonfly topology. In *Proceedings of the 2008 International Symposium on Computer Architecture*. 77–88. DOI: https://doi.org/10.1109/ISCA.2008.19

[33] J. Kim, W. J. Dally, B. Towles, and A. K. Gupta. 2005. Microarchitecture of a high radix router. In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA'05)*. 420–431. DOI: https://doi.org/10.1109/ISCA.2005.35

[34] Matthew J. Koop, K. Sridhar Jaidev, and Dhabaleswar K. Panda. 2014. Scalable MPI design over infiniband using extended reliable connection. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'14)*. 278–295. DOI: https://doi.org/10.1007/978-3-319-07518-1_18

[35] Matthew J. Koop, Sayantan Sur, Qi Gao, and Dhabaleswar K. Panda. 2007. High performance MPI design using unreliable datagram for ultra-scale InfiniBand clusters. In *Proceedings of the 21st Annual International Conference on Supercomputing*. 180–189. DOI: https://doi.org/10.1145/1274971.1274997

[36] Mingzhe Li, Khaled Hamidouche, Xiaoyi Lu, Hari Subramoni, Jie Zhang, and Dhabaleswar K. Panda. 2016. Designing MPI library with on-demand paging (ODP) of infiniband: Challenges and benefits. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'16)*. Article 37, 11 pages. http://dl.acm.org/citation.cfm?id=3014904.3014954.

[37] Kevin Lim, David Meisner, Ali G. Saidi, Parthasarathy Ranganathan, and Thomas F. Wenisch. 2013. Thin servers with smart pipes: Designing SoC accelerators for memcached. In *Proceedings of the International Symposium on Computer Architecture (ISCA'13)*. 36–47. DOI: https://doi.org/10.1145/2485922.2485926

[38] Jiuxing Liu, Balasubramanian Chandrasekaran, Jiesheng Wu, Weihang Jiang, Sushmitha Kini, Weikuan Yu, Darius Buntinas, Peter Wyckoff, and D. K. Panda. 2003. Performance comparison of MPI implementations over InfiniBand, myrinet and quadrics. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'03)*. 58–. DOI: https://doi.org/10.1145/1048935.1050208

[39] X. Lu, N. Islam, W. Rahman, J. Jose, H. Subramoni, H. Wang, and D. Panda. 2013. High-performance design of Hadoop RPC with RDMA over InfiniBand. In *Proceedings of the International Conference on Parallel Processing (ICPP'13)*.

[40] Evangelos P. Markatos, Manolis G. H. Katevenis, and Penny Vatsolaki. 1998. The remote enqueue operation on networks of workstations. In *Network-Based Parallel Computing Communication, Architecture, and Applications*, Dhabaleswar K. Panda and Craig B. Stunkel (Eds.). Springer, Berlin, 1–14.

[41] memcached [n.d.]. Memcached. Retrieved from http://memcached.org/.

[42] David J. Miller, Philip M. Watts, and Andrew W. Moore. 2009. Motivating future interconnects: A differential measurement analysis of PCI latency. In *Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS'09)*. 94–103. DOI: https://doi.org/10.1145/1882486.1882513

[43] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using one-sided RDMA reads to build a fast, CPU-Efficient key-value store. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'13)*. 103–114. https://www.usenix.org/conference/atc13/technical-sessions/presentation/mitchell.

[44] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. 2015. TIMELY: RTT-based congestion control for the datacenter. In *Proceedings of the SIGCOMM Conference on Data Communication (SIGCOMM'15)*. 537–550. DOI: https://doi.org/10.1145/2785956.2787510

[45] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. [n.d.]. *CACTI 6.0: A Tool to Model Large Caches*. Technical Report.

[46] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. 2013. Scaling memcache at Facebook. In *Proceedings of the Conference on Networked Systems Design and Implementation (NSDI'13)*. 385–398.

[47] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. 2014. Scale-out NUMA. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*. 3–18. DOI: https://doi.org/10.1145/2541940.2541965

[48] Scott Pakin, Mario Lauria, and Andrew Chien. 1995. High performance messaging on workstations: Illinois fast messages (FM) for myrinet. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (Supercomputing'95)*. Article 55. DOI: https://doi.org/10.1145/224170.224360

[49] F. Petrini, A. Hoisie, S. Coll, and E. Frachtenberg. 2001. The quadrics network (QsNet): High-performance clustering technology. In *Proceedings of the Symposium on High Performance Interconnects (HOT 9 Interconnects'01)*. 125–130. DOI: https://doi.org/10.1109/HIS.2001.946704

[50] D. Shankar, X. Lu, W. Rahman, N. Islam, and D. Panda. 2015. Benchmarking key-value stores on high-performance storage and interconnects for web-scale workloads. In *Proceedings of the 2015 IEEE International Conference on Big Data*.

[51] Akshitha Sriraman, Sihang Liu, Sinan Gunbay, Shan Su, and Thomas F. Wenisch. 2016. Deconstructing the tail at scale effect across network protocols. In *Proceedings of the Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD'16)*. http://arxiv.org/abs/1701.03100.

[52] Maomeng Su, Mingxing Zhang, Kang Chen, Zhenyu Guo, and Yongwei Wu. 2017. RFP: When RPC is faster than server-bypass with RDMA. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys'17)*. ACM, New York, NY, USA, 1–15. DOI: https://doi.org/10.1145/3064176.3064189

[53] Hari Subramoni, Khaled Hamidouche, Akshey Venkatesh, Sourav Chakraborty, and Dhabaleswar K. Panda. 2008. Designing MPI library with dynamic connected transport DCT of InfiniBand: Early experiences. In *Proceedings of the IEEE Cluster*. 203–212. DOI: https://doi.org/10.1109/CLUSTR.2008.4663773

[54] Sayantan Sur, Lei Chai, Hyun-Wook Jin, and Dhabaleswar K. Panda. 2006. Shared receive queue based scalable MPI design for infiniband clusters. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'06)*. 101–101. http://dl.acm.org/citation.cfm?id=1898953.1899033.

[55] Sayantan Sur, Hyun-Wook Jin, Lei Chai, and Dhabaleswar K. Panda. 2006. RDMA read based rendezvous protocol for MPI over InfiniBand: Design alternatives and benefits. In *Proceedings of the Annual Conference on Principles and Practice of Parallel Programming (PPoPP'06)*. 32–39. DOI: https://doi.org/10.1145/1122971.1122978

[56] T. von Eicken, A. Basu, V. Buch, and W. Vogels. 1995. U-Net: A user-level network interface for parallel and distributed computing. In *Proceedings of the Symposium on Operating Systems Principles (SOSP'95)*. 40–53. DOI: https://doi.org/10.1145/224056.224061

[57] Paul Willmann, Hyong-youb Kim, Scott Rixner, and Vijay S. Pai. 2005. An efficient programmable 10 gigabit ethernet network interface card. In *Proceedings of the (HPCA-11)*. 96–107. DOI: https://doi.org/10.1109/HPCA.2005.6

[58] Jiesheng Wu, Pete Wyckoff, and Dhabaleswar K. Panda. 2003. PVFS over InfiniBand: Design and performance evaluation. In *Proceedings of the International Conference on Parallel Processing (ICPP'03)*. 125–132.

[59] Weikuan Yu, Qi Gao, and Dhabaleswar K. Panda. 2006. Adaptive connection management for scalable MPI over InfiniBand. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'06)*. 102–102. http://dl.acm.org/citation.cfm?id=1898953.1899034.

[60] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. 2015. Congestion control for large-scale RDMA deployments. In *Proceedings of the SIGCOMM Conference on Data Communication (SIGCOMM'15)*. 523–536. DOI: https://doi.org/10.1145/2785956.2787484

Received April 2019; revised September 2019; accepted November 2019