PRISM: Rethinking the RDMA Interface for Distributed Systems

Matthew Burke
Cornell University
United States
matthelb@cs.cornell.edu

Sowmya Dharanipragada
Cornell University
United States
sjd266@cornell.edu

Shannon Joyner
Cornell University
United States
saj9191@gmail.com

Adriana Szekeres
VMware Research
United States
aaasz@cs.washington.edu

Jacob Nelson
Microsoft Research
United States
jacob.nelson@microsoft.com

Irene Zhang
Microsoft Research
United States
irene.zhang@microsoft.com

Dan R. K. Ports
Microsoft Research
United States
dan@drkp.net

Abstract

Remote Direct Memory Access (RDMA) has been used to accelerate a variety of distributed systems, by providing low-latency, CPU-bypassing access to a remote host's memory. However, most of the distributed protocols used in these systems cannot easily be expressed in terms of the simple memory READs and WRITEs provided by RDMA. As a result, designers face a choice between introducing additional protocol complexity (e.g., additional round trips) or forgoing the benefits of RDMA entirely.

This paper argues that an extension to the RDMA interface can resolve this dilemma. We introduce the PRISM interface, which adds four new primitives: indirection, allocation, enhanced compare-and-swap, and operation chaining. These increase the expressivity of the RDMA interface, while still being implementable using the same underlying hardware features. We show their utility by designing three new applications using PRISM primitives, that require little to no server-side CPU involvement: (1) PRISM-KV, a key-value store; (2) PRISM-RS, a replicated block store; and (3) PRISM-TX, a distributed transaction protocol. Using a software-based implementation of the PRISM primitives, we show that these systems outperform prior RDMA-based equivalents.

The first three authors contributed equally to this work.

SOSP '21, October 26–29, 2021, Virtual Event, Germany

ACM ISBN 978-1-4503-8709-5/21/10…$15.00
https://doi.org/10.1145/3477132.3483587

CCS Concepts: • **Networks** → **Programming interfaces**; *Data center networks*; • **Software and its engineering** → **Distributed systems organizing principles**.

Keywords: RDMA, remote memory access, distributed systems

ACM Reference Format:
Matthew Burke, Sowmya Dharanipragada, Shannon Joyner, Adriana Szekeres, Jacob Nelson, Irene Zhang, and Dan R. K. Ports. 2021. PRISM: Rethinking the RDMA Interface for Distributed Systems. In *ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21), October 26–29, 2021, Virtual Event, Germany.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3477132.3483587

1 Introduction

Remote Direct Memory Access (RDMA) has quickly become one of the indispensable tools for achieving high throughput and low latency in datacenter systems. As network bandwidth continues to increase relative to CPU speed, it becomes critical to reduce the CPU cost of packet processing. This makes RDMA, which provides a standard, accelerated interface for one host to directly access another's memory, appealing: hardware RDMA implementations bypass the host CPU entirely [28], and even software implementations offer significant performance improvements by simplifying the network stack and reducing context-switching overhead [26].

A rich literature has explored how distributed systems can be redesigned to use RDMA communication [6, 10, 14, 31, 44]. A common theme is that adapting applications to run on RDMA requires complex—and costly—contortions. The source of this complexity is the RDMA interface itself: few distributed applications can readily express their logic in terms of simple remote memory READ and WRITE operations. Consequently, many RDMA applications are forced to add extra operations, i.e., extra network round trips, to their protocols, sacrificing some of the latency benefits [10, 31, 32]. Others use hybrid designs that require application CPU involvement on some operations [10, 31, 43] – or, in some

cases, just use RDMA to implement a faster message-passing protocol [15, 34] – negating the benefits of CPU bypass.

This paper argues that moving beyond the basic RDMA interface is necessary to achieve the full potential of network acceleration for building low-latency systems. The RDMA interface, originally designed to support parallel supercomputing applications, fails to meet the needs of today's distributed systems. We show that by extending the interface with a few additional primitives, it becomes possible to implement sophisticated distributed applications like replicated storage entirely using remote operations.

Our goal in this paper is to identify a set of generic (i.e., non-application specific) extensions to the RDMA interface that allows distributed systems to *better utilize the low latency and CPU offload capabilities* of RDMA hardware. Our proposal is the PRISM interface. PRISM extends the RDMA read/write interface with four additional primitives: indirection, allocation, enhanced compare-and-swap, and operation chaining. In combination, these support common remote access patterns such as data structure navigation, out-of-place updates, and concurrency control. We argue that the PRISM API is simple enough to implement in a RDMA NIC, as it reuses existing microarchitectural mechanisms. We also implement the PRISM interface in a prototype software-based networking stack that uses dedicated CPU cores to implement remote operations, inspired by the approach taken by Google's SNAP networking stack [26].

We demonstrate the PRISM API's benefits with three case studies of common distributed applications. The first, a key-value store, demonstrates that PRISM can be used to read and manipulate remote data structures, reducing read latency to 75% of Pilaf [31] despite our prototype emulating one-sided operations with software. The second, a replicated block store, provides fault tolerant storage by using compare-and-swap and indirect operations to implement the ABD quorum protocol [4], providing a 50% throughput improvement over a traditional lock-based approach [44], again despite using dedicated CPU cores to implement one-sided operations. The third, a transactional key-value store, provides strong atomicity guarantees over a sharded storage system. Its new protocol uses CAS and indirect operations to commit transactions using only two round trips. This yields a 20% improvement in throughput over FaRM [10] while reducing latency by 18%.

To summarize, this paper makes these contributions:

- We show that the existing RDMA interface leads to extra protocol complexity for distributed systems.
- We introduce the PRISM interface, which extends RDMA with additional primitives to support common patterns in distributed applications.
- We show that three sophisticated applications—key-value stores, replicated block storage, and distributed transactions— can be implemented entirely using the PRISM interface.
- We build a software-based prototype of the PRISM interface, and show that in spite of its additional performance overhead relative to a NIC, applications built atop PRISM achieve latency and throughput benefits.

2 Background and Motivation

RDMA is a widely-deployed [13, 26] network interface that allows a remote client to directly read or write memory on remote hosts, bypassing the remote CPU entirely.

2.1 RPCs vs Memory Accesses: The RDMA Dilemma

RDMA provides two types of operations. *Two-sided* operations have traditional message-passing semantics. A SEND operation transmits a message to a remote application that calls RECEIVE. *One-sided* operations allow a host to READ or WRITE memory (in a pre-registered region) on a remote host.

Considerable debate has ensued in the systems community about whether to use one-sided or two-sided operations [16, 19, 43]. One-sided operations are faster and more CPU efficient, but restricted to simple read and write operations. Two-sided message passing, because it allows processing at both ends, may yield a faster system overall even though the communications operations themselves are slower.

To see how this tradeoff plays out, consider, for example, Pilaf [31], an early RDMA-based key-value store. Pilaf stores pointers to key-value objects in a hash table, with the actual data stored in a separate extents structure. Both structures are exposed through RDMA, so a client can perform a key-value lookup by remotely reading the hash table, then using the pointer to perform a remote read into the extents store. This requires no server-side CPU involvement, but takes two round trips. A traditional key-value store implementation built using two-sided operations would require only one round trip but involve the CPU on every operation.

The dilemma for systems designers is, then, whether to build a more complex protocol out of read or write operations, or a simpler one with message passing? In other words, is it faster to do two (or more) one-sided RDMA operations, or a single RPC? In the early days of RDMA, the choice was clear: a RDMA operation was about 20× faster than a RPC [31]. As subsequent work has dramatically reduced the cost of two-sided RPC [16, 19] and RDMA has been deployed in larger-scale settings with higher latency [12, 13], the question of which to use is far more complicated.

To understand the current tradeoff, we measure the performance of one-sided RDMA operations vs. two-sided RPC implemented using eRPC [16] on two servers connected using 40 Gb Ethernet (see §4.3 for experimental details). Reading a 512-byte value using a one-sided read completes in about 3.2 μs, making it 43% faster than using a two-sided RPC (5.6 μs). But this implies that a system that does *two* one-sided reads, like the example above, is about 0.8 μs *slower* than a pure software implementation. Thus, one-sided RDMA

operations offer a performance benefit only when using them does not require a more complex protocol.

2.2 Principles for Post-RDMA Systems

Most work on RDMA systems assumes that we are limited to the current RDMA read/write interface. What if, instead, we could extend the RDMA interface? Hardware vendors [29] and software implementations [26] have added various new operations in an ad-hoc way. In this paper, we take a step back and ask what new functionality is needed to support distributed systems that run directly over RDMA, without requiring either CPU involvement or additional round trips.

Navigating data structures. RDMA supports remote reads when the size and location are known. Most applications, like Pilaf, use more complex data structures to build indexes, to store variable-length objects, and to help handle concurrent updates. Traversing these structures requires multiple RDMA reads. Being able to perform indirect operations on pointers could eliminate some of these round trips.

Supporting out-of-place writes. Modifying remote data structures is particularly challenging because reads may happen concurrently. To avoid the ensuing consistency issues, many systems [10, 31, 32] perform writes only from the server CPU. We aim to build systems that can handle both reads and writes using RDMA operations. To do so, we advocate a design pattern where new data is written out-of-place into a separate buffer, then a pointer is atomically updated to swap from the old to new value – an approach similar to read-copy-update [27] in concurrent programming. Achieving this requires new RDMA support for writing data to a new buffer, and to atomically update a pointer to its location.

Optimistic concurrency control. Updates to complex data structures require synchronization. While it is possible to use RDMA to implement locks today [44, 45], the performance penalty can be substantial. Expanding RDMA's compare-and-swap functionality would allow us to implement sophisticated, version-based optimistic concurrency control [21], an approach that fits well with our out-of-place update method.

Chaining operations. A common theme is that applications need to perform compound operations, where the argument to one depends on the result of a previous one – reading a pointer then the value it points to, or writing an object then swapping a pointer. Today, this requires returning the intermediate result to the client and performing a new operation – and another network round trip. We could avoid this overhead if we had a way to chain operations so that one depends on the other, but still execute them in one round trip.

2.3 The Case for an Extended Interface

The principles of the previous section could be addressed by extending the RDMA interface in various ways. In this paper, we argue that a set of simple, generic extensions can be useful for a variety of applications. Using simple operations makes it feasible to implement and deploy these extensions.

An alternate approach is to allow applications to provide their own code that runs on the remote host, i.e., to deploy custom application logic to smart NICs [3, 37]. While powerful, this approach comes with considerable deployment challenges. From talking to cloud providers that have deployed smart NICs, rolling out smart NIC updates is a challenge even in a single-tenant environment because it involves downtime for everything running on the host. Allowing users in multi-tenant environments to provide their own code presents major security and performance isolation challenges [22, 41].

We instead argue for a set of simple, generic primitives. Such simple extensions are likely to be useful for more applications, both current and future. Simplicity here also aids implementation: we argue that our proposed primitives can be added either to software-based networking stacks, reconfigurable smart NICs, or even to future fixed-function NICs. The remainder of this paper proposes such general extensions, and demonstrates that they are useful for common applications.

3 PRISM Interface

To address the challenges inherent in building distributed systems with RDMA, we propose an extended network interface, PRISM (Primitives for Remote Interaction with System Memory). PRISM adds four additional features to the existing RDMA interface. These are designed to support common patterns we observe in implementing distributed protocols.

PRISM's interface is designed around three principles: (1) generality – they should not encode application-specific functionality; (2) minimal interface complexity; and (3) minimal implementation complexity, which enables fast, predictable performance and facilitates implementation on a variety of platforms, including future NIC ASICs.

Following these principles, we extend the RDMA interface in four ways. Table 1 provides a summary of the PRISM API.

3.1 Indirect Operations

Many RDMA applications need to traverse remote data structures. These structures use indirection for many purposes: to provide indexes, to support variable length data, etc. Currently, following a pointer requires an extra round trip.

PRISM allows READ, WRITE, and compare-and-swap (CAS) operations to take indirect arguments. The target address of these operations can instead be interpreted as the address of a pointer to the actual target. Additionally, the data for a WRITE or CAS operation can be read from a server-side memory location instead of from the RDMA request itself.

For READs and WRITEs, the target can optionally be interpreted as a $\langle ptr, bound \rangle$ struct. In that case, the operation length is limited to the smaller of *bound* or the client-requested length. This supports variable-length objects: a client can perform a read with large *length*, but only receive as much data as is actually stored.

Indirect Reads and Writes:

- READ(ptr *addr*, size *len*, bool *indirect*, bool *bounded*) → byte[]
  Returns the contents of the target address – either *addr* or, if *indirect* is set, **addr*.
  If *bounded* is set, reads min(*len*, *addr->bound*) bytes; otherwise reads *len* bytes.
- WRITE(ptr *addr*, byte[] *data*, size *len*, bool *addr_indirect*, bool *addr_bounded*, bool *data_indirect*)
  Writes data to a target address – either *addr* or, if *addr_indirect* is set, **addr*.
  If *addr_bounded* is set, writes min(*len*, *addr->bound*) bytes; otherwise writes *len* bytes.
  If *data_indirect* is set, *data* is interpreted as a server-side pointer and its target is used as the source data.

Allocation:

- ALLOCATE(qp *freelist*, byte[] *data*, size *len*) → ptr
  Pops the first buffer *buf* from the specified free list (represented as a RDMA queue pair). The specified *data* is written to *buf*, and the address of *buf* is returned.

Enhanced Compare-and-Swap:

- CAS(*mode*, ptr *target*, byte[] *data*, bitmask *compare_mask*, bitmask *swap_mask*, bool *target_indirect*, bool *data_indirect*) → byte[]
  Atomically compares (**target* & *compare_mask*) with (*data* & *compare_mask*) using the operator specified by *mode*, and, if successful, sets **target* = (**target* & *~swap_mask*) | (*data* & *swap_mask*). Returns the previous value of **target*.
  If *target_indirect* or *data_indirect* is set, the corresponding argument is first treated as a pointer and dereferenced; this is not guaranteed to be atomic.

Operation Chaining:

- CONDITIONAL: Executes operation only if previous operation was successful. Operations that generate NACKs or errors, or CAS operations that do not execute, are considered unsuccessful.
- REDIRECT(*addr*) Instead of returning operation's output to the client, write it to *addr* instead.

Table 1. PRISM Primitives

Indirect operations in PRISM reuse existing RDMA security mechanisms that ensure remote clients can only operate on regions of host memory to which they have been granted access. To access a memory region with an indirect operation, a client must include the *rkey* that was generated by the host when the region was first registered with NIC. The operation is rejected by the host if *either* the target address or the location pointed to by the target address is in a memory region with a different rkey (or that has not been registered at all).

3.2 Memory Allocation

Modifying data structures is particularly challenging with the existing RDMA interface: objects must be written into fixed, pre-allocated memory regions, making it difficult to handle variable-sized objects or out-of-place updates. What is needed is a memory allocation primitive. PRISM provides one, which allocates a buffer and returns a pointer to its location.

To use PRISM's allocation primitive, a server-side process registers ("posts", in RDMA parlance) a queue of buffers with the NIC. When the NIC receives an ALLOCATE request from a remote host, it pops a buffer from this free list, writes the provided data into the buffer, and responds with the address. This operation is especially powerful in combination with PRISM's request chaining mechanism, discussed below: a PRISM client can allocate a buffer, write into it, then install a pointer to it (via CAS) in another data structure.

PRISM performs memory allocations on the NIC (or software networking stack) data plane. Memory *registrations*, however, are done by the server CPU. This is necessary because registering memory requires interaction with the kernel to identify the corresponding physical addresses and pin the buffers. Because the server CPU is involved, it is essential for the correctness of applications reusing these buffers that recycled buffers only be added back to the free list when concurrent NIC operations are complete. While this simply shifts the burden of synchronizing the NIC and the server CPU to the implementation of the primitives, it importantly moves this synchronization off the regular path for applications.

Management of client-allocated memory can be challenging; this challenge is a fundamental one for applications that modify state through remote accesses. Specific memory management policies are left to the discretion of the application. The applications in this paper use clients to detect when objects are no longer used, e.g., when a previous version has been replaced. They report the unused buffer to a daemon running on the server (via traditional RPC), which re-registers it with the NIC's free list; batching can be employed at both client and server sides to minimize overhead. An alternate, garbage-collection-inspired approach would be for server-side application code to periodically scan data structures to identify buffers that can be reclaimed.

Our allocator design is intentionally simple, merely allocating the first available buffer from a particular queue. We choose this over a more complex allocator because (as discussed in §4.2) existing RDMA NICs already have the necessary hardware support to implement it. A consequence is that allocating memory using entire pre-allocated buffers introduces space overhead. Applications can minimize this effect by registering multiple queues containing buffers of different sizes, and selecting the appropriate one. For example, using buffers sized as powers of two guarantees a maximum

space overhead of 2×. A pure software implementation might choose to use a more sophisticated allocator.

3.3 Enhanced Compare-And-Swap

Atomic compare-and-swap (CAS) is a classic primitive for updating data in parallel systems. The RDMA standard already offers an atomic CAS operation [30], but it is highly restricted: it does a single equality comparison and then swaps a 64-bit value. In our experience, this is insufficient to implement performant applications (including those in §6–8). Indeed, few applications today use RDMA atomic operations except to implement locks [33, 45]. While it is possible to implement arbitrarily complex atomic operations using locks, this requires multiple costly round trips and increases contention.

To address this, we extend the CAS primitive in three ways. First, we adopt the extended atomics interface currently provided by Mellanox RDMA devices [30], which allows CAS operations up to 32 bytes, and uses separate bitmasks for the compare and swap arguments to make it possible to compare one field of a structure and swap another. Second, we incorporate indirect addressing (§3.1) for either the target address or compare and swap values. We do not guarantee that dereferencing the indirect argument pointers is atomic – only the CAS itself is – but this allows us to load argument values from memory. Finally, we provide support for arithmetic comparison operators (greater/less than) in the compare phase, in addition to bitwise equality. This supports the common pattern of updating a versioned object: check whether the new version is greater than the existing one, and if so update both the version number and the object.

PRISM's CAS operations are atomic with respect to other PRISM operations. Like existing RDMA atomics, they are not guaranteed atomic with respect to concurrent CPU operations.

3.4 Operation Chaining

Distributed applications often need to do sequences of data-dependent operations to read or update remote data structures. For example, they may wish to allocate a buffer, write to it, then update a pointer to point to it. Currently, each operation must return to the client before it can issue the next. PRISM provides a chaining mechanism that allows compound operations like this to be executed in a single round trip.

Conditional operations. RDMA does not, in general, guarantee that operations execute in order. We add a *conditional* flag that delays execution of an operation until previous operations from the same client complete, and does not execute unless the previous operation was *successful*. A CAS operation whose comparison fails is treated as unsuccessful here.

Output redirection. We introduce another flag which specifies that the output of an operation (READ or ALLOCATE) should, rather than being sent to the client, be written to a specified memory location. That memory location will generally be a per-connection temporary buffer. For example, one could perform an ALLOCATE, redirect its output to a temporary location, then use a conditional WRITE to store a pointer to the newly allocated buffer elsewhere.

3.5 Discussion

The PRISM primitives together fulfill the goals from §2.2 and §2.3. Indirect operations reduce the number of round trips needed to navigate data structures. The ALLOCATE and enhanced CAS primitives, combined with chaining, support out-of-place updates: an application can ALLOCATE a new buffer, write data into it, and install a pointer to it into another structure using CAS, all within a single round trip. Finally, the flexibility of our CAS operation makes it possible to implement version-based concurrency control mechanisms.

4 PRISM Implementation

PRISM's API consists of simple primitives so that they can be easily added to a variety of RDMA implementations. To evaluate their effectiveness in building distributed applications, we have built a software-based implementation (§4.1). We also analyze the feasibility of implementing PRISM in a NIC (§4.2). We evaluate the performance of our software implementation and projected benefits of hardware implementation, along with a smart NIC approach. (§4.3).

4.1 Software PRISM Implementation

We use a software implementation, inspired by Google's Snap [26], to quantitatively evaluate PRISM's benefits for applications. This is a library used on both the client and server, that supports both traditional RDMA operations and the PRISM extensions. It implements PRISM extensions by communicating via eRPC [16] with a dedicated thread on the remote side, which implements the appropriate primitive.

Although software implementations use the server-side CPU, one-sided operations executed within the network stack provide performance benefits by avoiding the cost of context switching and application-level thread scheduling. Reports on the large-scale deployment of Snap at Google [26] have noted these benefits, along with the deployment benefits of easier upgradability and broader hardware support. A software approach also makes it easier to deploy new primitives; Snap already supports (and its applications use) indirect reads.

PRISM's limited interface makes an efficient implementation possible. While in principle it is possible to run arbitrarily complex, application-specific operations in a software networking stack, this poses deployment and security challenges. PRISM avoids this by providing a library of simple, common primitives. Each of PRISM's primitives can be run in short, bounded time, which is important to prevent starvation.

Smart NIC deployments. In principle, the software implementation could also be used on smart NICs; we have experimented with it on a Mellanox BlueField. However, as we show below (§4.3), this approach has lower performance

than a software implementation, so we do not advocate its use unless reducing CPU utilization is the primary goal.

4.2 Hardware NIC Feasibility

PRISM is designed to be implementable in future RDMA NICs. This part of our analysis is necessarily speculative, as RDMA-capable NIC ASICs are proprietary designs that we do not have the capability to extend. However, we argue that implementing the PRISM primitives is feasible because they reuses underlying mechanisms that already exist on today's NICs. For example, problems common to all primitives such as loss, corruption, and timeout would be handled using the same CRC and retransmission mechanisms that NICs already implement, and each primitive reuses existing mechanisms:

Indirection. Indirect reads or writes are conceptually identical to a direct read followed by a direct read or write, RDMA NICs already process memory accesses asynchronously, so indirection does not change the processing model fundamentally. However, it adds performance overhead, notably an extra PCIe round trip, which we evaluate in the next section.

Allocation. Although ALLOCATE is conceptually a new function, we observe that its behavior closely resembles traditional SEND/RECEIVE functionality, where the NIC allocates a buffer from a receive queue to write an incoming message; existing SRQ functionality allows multiple connections to share a receive queue. We represent the free list the same way as a *queue pair* – a standard RDMA structure containing a list of free buffers. When the NIC processes an ALLOCATE request, it pops the first buffer from the queue pair and returns the pointer to the buffer. We use separate, application-selected queue pairs for buffers of different sizes.

PRISM requires that buffers only be posted when all other operations on the NIC are complete. Normal operations acquire the read side of a reader-writer lock and posting a buffer acquires the write side. This type of synchronization mechanism already exists on NICs for processing CAS operations.

Enhanced CAS. RDMA NICs that support extended atomics already support masked operations up to 32 bytes [29]. We allow the equality comparison to be replaced with an inequality. The adder used to implement RDMA's FETCH-AND-ADD primitive can be used to compute this inequality.

Chaining. Additional flags enable conditional execution and output redirection. RDMA NICs already enforce ordering relationships between certain types of operations and stop processing on errors; our conditional flag just adds an additional constraint. If the NIC lacks sufficient memory to buffer a chained operation, it can, as always, reject the request with a Receiver Not Ready packet, the standard flow control mechanism.

Output redirection requires writing results to memory rather than the network. If the target address is in host memory, the operation whose output is being redirected would incur an additional PCIe round trip, a potentially significant performance hit. Fortunately, recent NIC designs provide a user-accessible on-NIC memory region (256 KB on our Mellanox ConnectX-5 NICs [28]). Applications using output redirection should redirect to this on-NIC memory when possible and only use it for temporary storage as part of a chain of operations. One possible concern is that our design requires per-connection temporary storage. Historically, per-connection state has limited the number of connections that a RDMA NIC can efficiently support [10, 18]. However, the amount of temporary space needed is small—32B/connection suffices for our applications—in comparison to the existing per-connection state (≈ 375B [16]). A 256 KB memory region provides 8192 32B locations, which exceeds the recommended concurrent connection limit for good performance [18].

Wire Protocol Extensions The PRISM API requires minimal modifications to the RDMA wire protocol. It requires an additional 5 flags in the RDMA header (the IB BTH). Three are used for indirection – two control whether two arguments of READ, WRITE, or CAS are treated as indirect, and a third indicates whether the target address of a READ or WRITE is a bounded pointer. Two flags are used for chaining, to control conditional and output redirection behavior respectively.

4.3 Performance Analysis

We compare the performance of our software implementation of PRISM with existing RDMA operations in Figure 1. These experiments use two machines (Intel Xeon Gold 6138 processors, 96GB RAM, Ubuntu 18.04) with Mellanox ConnectX-5 25 GbE RDMA NICs. To factor out the effect of network latency, we use a direct connection (no switch) between the two NICs. The baseline RDMA operations have 2.5 μs latency; PRISM's software prototype adds another 2.5–2.8 μs depending on operation.

We also compare against a performance model of a hypothetical ASIC-based PRISM NIC, computed by adding the costs of additional PCIe round-trips (using prior measurements [35]) to the latency of the corresponding RDMA operation – e.g., an indirect READ is modeled as a RDMA READ plus one extra pointer-sized PCIe read. We also model the cost of a smart NIC deployment by running our implementation on a Mellanox BlueField, which combines a RDMA network card with an ARM Cortex A72 CPU (8 cores, 800 MHz), and adding the measured overhead of accessing host memory. Perhaps surprisingly, this option is the slowest: the BlueField's slower CPU gives it higher processing latency, and it has high latency (∼3 μs) access to host memory.[1]

PRISM's performance depends not just on the execution cost but also network latency, because its performance benefit

[1]The BlueField is an *off-path* NIC [23]: its accesses to host memory must be performed as RDMA requests through an internal switch. Other smart NIC designs with lower latency to host memory (e.g., the Netronome NFP-6000 at 0.6 μs [35]) may yield better performance.

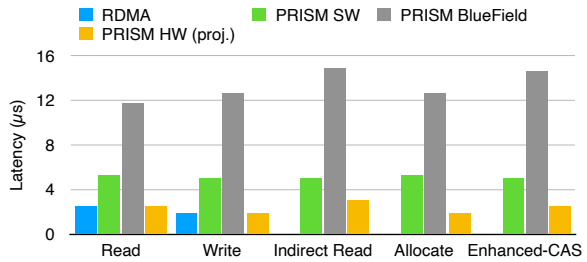

Figure 1. Microbenchmarks of PRISM software implementation, compared to hardware RDMA implementation. All operations are performed with 512 byte values (except CAS).

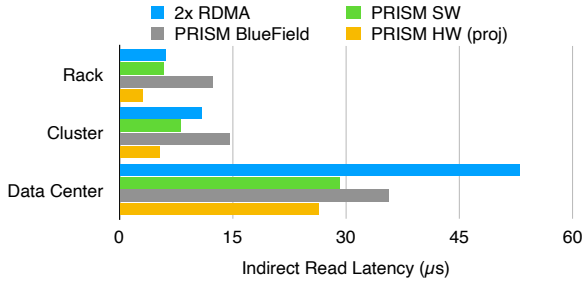

Figure 2. Comparison of indirect read latency using RDMA (two round trips) vs PRISM. We synthetically introduce network latency based on (a) a single ToR switch (0.6 μs), (b) a three-tier cluster network (3 μs), and (c) the reported RDMA latency from Microsoft data centers (24 μs [12])

comes from eliminating network round trips compared to RDMA. Figure 1, by using a direct network link, considers the worst case for PRISM. Figure 2 evaluates the impact of network latency by comparing the cost of an indirect read with PRISM against executing two RDMA reads. We consider a single-rack deployment with one switch, a cluster with a three-level switch hierarchy, and reported latency numbers from a real data center [12] which also reflect network congestion. In each case, PRISM's software implementation outperforms the RDMA baseline despite the additional cost of using the CPU.

5 Applications Overview

To demonstrate the potential benefits of PRISM, we use three case studies of representative distributed applications:

- PRISM-KV: a key-value store that implements both read and write operations over RDMA. (§6)
- PRISM-RS: a replicated storage system that implements the ABD [4] quorum replication protocol. (§7)
- PRISM-TX: a transactional storage system that implements a timestamp-based optimistic concurrency control protocol using PRISM's primitives. (§8)

Each of these classes of applications is widely used in practice and has been the subject of much research. In each of the following sections, we review existing RDMA implementations of one of the applications, show how the current RDMA interface leads to excess complexity or cost, and design a new system using the PRISM primitives.

Using our software-based PRISM prototype, we show that our PRISM applications outperform prior RDMA systems. We use a cluster of up to 12 machines (with specs as in §4.3) with 40 Gb Ethernet. Clients and servers are connected to one Arista 7050QX switch (0.6 μs added latency). This is a challenging configuration for PRISM, as a larger cluster or more congested network would have higher latency, and hence benefit more from reducing round trips.

6 PRISM-KV: Key-Value Storage

Key-value stores are a widely used piece of infrastructure, and are a natural opportunity for acceleration with RDMA. We begin by considering a simple remote, unreplicated key-value store, akin to memcached. It provides a GET/PUT interface, where both keys and values can be strings of arbitrary length.

As an example of the challenges of implementing a KV store with RDMA, consider again Pilaf [31]. Pilaf uses one-sided operations to implement GET operations only; PUT operations are sent through a two-sided RPC mechanism and executed by the server CPU. It stores a fixed size hash table that contains only a valid bit and a pointer to a key-value store, which is in a separate extents region. To perform a GET operation, a Pilaf client computes the hash of the key and performs a one-sided READ into the hash table, followed by a second READ to the data it points to. Hash collisions are resolved using linear probing or cuckoo hashing, which may require reading additional pointers.

Pilaf does not use one-sided operations to implement PUT requests. Doing so would be challenging using the RDMA interface: it supports variable-sized objects, so PUTs may need to allocate new space in the extents area, and any in-place modifications must be done in a way that does not conflict with other concurrent operations.

6.1 PRISM-KV Design

Our new key-value store, PRISM-KV, follows the same general design as Pilaf, but implements both GET and PUT operations using one-sided operations. It maintains a hash table index containing pointers to data items; these items are now in buffers allocated using PRISM's ALLOCATE primitive.

The PRISM-KV server initially allocates memory regions for the hash table and for the data, and registers both for RDMA access. It also posts a set of buffers to be used for ALLOCATE, and periodically checks if more buffers are needed.

A client performs a GET operation by probing for the correct hash table slot using a hash of the key. To do so, it performs a READ of a slot with the indirect bit set, retrieving the data (if any) pointed to by the slot. It then verifies that the key matches, retrying using linear probing in the case of a hash collision. Each access (or retry) requires only one PRISM operation, compared to 2 READS in Pilaf.

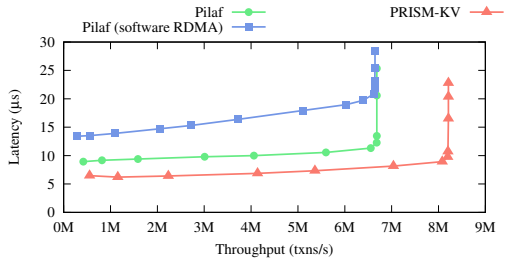

Figure 3. Throughput versus average latency comparison for PRISM-KV and Pilaf, 100% reads, uniform distribution.

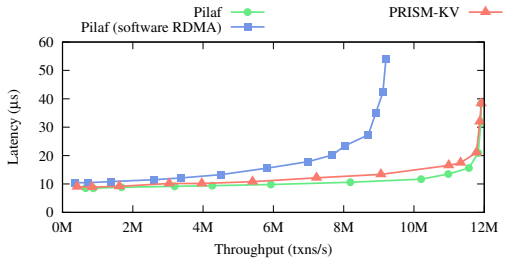

Figure 4. Throughput versus average latency for PRISM-KV and Pilaf, 50% reads, uniform distribution.

To perform a PUT, a client first determines the correct hash table slot as described above. The client then writes the new value and updates the hash table slot using a chain of PRISM primitives. First, the client writes the value to a new buffer using ALLOCATE and redirects the address to a temporary location. It then performs a CAS at the hash table slot such that the old address is replaced with the address stored at the temporary location if it has not changed since the client determined the correct slot.[2] Both the appropriate indirect bit and the conditional flag are set for this last operation. If the CAS fails, this indicates that a concurrent client subsequently overwrote the same key with a newer value. If it succeeds, the client asynchronously notifies the server to return the old version's buffer to the free list.

PRISM-KV ensures correctness during concurrent updates because objects are written to separate buffers and the pointers to these buffers are atomically installed into the appropriate hash table slot. Reads that are concurrent with updates also do not violate safety because an indirect read of a hash table slot is guaranteed to read a well-formed address and addresses fit within a cache line. Note that a client that just completed an update may attempt to free a buffer while an indirect read attempts to read it. This is not an issue for correctness because PRISM waits for concurrent NIC operations to complete before adding buffers back to the free list.

The entire chain of PRISM primitives takes a single round trip like Pilaf's two-sided PUT. However, unlike Pilaf, they can be executed without application CPU involvement.

6.2 Evaluation

We implement Pilaf and PRISM-KV in the same framework to evaluate the tradeoffs in using PRISM for a key-value store. We compare against both a version of Pilaf that uses RDMA hardware and one that uses the same software implementation that PRISM uses. Each experiment uses one server machine with the hardware described in Section 4.1 and up to 11 client machines. The server uses 16 dedicated cores to handle RPCs and implement the PRISM primitives, which is sufficient to achieve line rate for both systems.

We evaluate performance on YCSB [7] workloads A (50% R/50% W) and C (100% R). All workloads use 8 million 512 byte objects with 8 byte keys. The object access distribution is uniform, and we use a collisionless hash function.

Indirect reads reduce latency. The read-only workload (Figure 3) demonstrates that PRISM-KV achieves a latency improvement, principally because it can replace two RDMA READs with one indirect READ. As expected, when comparing PRISM-KV to Pilaf using the RPC-based read implementation, the difference is about 2× (6 vs 14 μs). The other 2 μs are CRC calculations that Pilaf uses to detect concurrent updates; PRISM-KV's atomic out-of-place update approach avoids the need for these. The hardware RDMA implementation reduces Pilaf's latency to 8 μs, still higher than PRISM-KV's.

Complexity increases per-request network usage. Each system can saturate the 40 Gbps network. However, PRISM-KV achieves 22% higher read throughput because the server transmits less data per request; Pilaf needs to send two read replies with their attendant headers, along with the application-level CRCs mentioned above.

Turning to the 50/50 read/write workload (Figure 4), Pilaf uses one two-sided RPC to process each PUT, averaging 6 μs. PRISM-KV uses two round trips, one indirect READ to identify the correct hash table slot and one to perform the chain of ALLOCATE, WRITE, and CAS. In our software prototype, this requires 12 μs. (Note that we are making the pessimal assumption that the hash table slot is not cached; a read-modify-write workload could avoid the first round trip on PUT).

Hardware implementations. Our software PRISM implementation outperforms the latency and throughput of a RDMA-enabled Pilaf on read-only workloads, and matches it for 50/50 mixed workloads. Per the analysis in §4.3, we anticipate that a hardware implementation of PRISM could further reduce latency by another 2 μs. Further, it frees up the server cores required to execute RPCs, improving efficiency.

[2]The CAS operation is used instead of a non-conditional WRITE to avoid overwriting the hash table entry in case the original object has been deleted and the slot reused by a different object. This is a simple but heavy-handed mechanism, in that it prevents *all* concurrent updates; a more sophisticated approach might instead compare against a generation number that is only incremented when a slot is reused for a *different* object.

7 PRISM-RS: Replicated Block Storage

As our next case study, we consider a replicated block store. A block store provides GET and PUT operations to objects with fixed identifiers, i.e., blocks. Such a system provides the foundation for reliably implementing more complex applications such as file systems [11, 39] or key-value stores [6, 10, 43, 44].

PRISM-RS is our block store design that guarantees linearizability, remains available as long as no more than f out of $n = 2f + 1$ replicas fail, and requires minimal CPU involvement at replicas. It does so by implementing a variant of the ABD [4] atomic register protocol with PRISM operations.

7.1 Background: ABD

The ABD protocol is a classic distributed algorithm that implements a fault tolerant, linearizable shared register [4]. Shared registers are simply objects that multiple clients may concurrently access with GET and PUT operations. We focus on Lynch and Shvartsman's multi-writer variant [25] because allowing multiple clients to make modifications is necessary for a practical storage system. We describe the protocol in terms of a single register, though it is natural to extend this to multiple registers by adding register identifiers to messages.

The protocol ensures fault-tolerance for up to f failures by replicating the value v of the register in memory at $n = 2f + 1$ replicas. Each replica associates a *tag* $t = (ts, id)$ with v where ts is a logical timestamp and id is the identifier of the client that wrote the value. Operations are ordered lexicographically by the tags that they observe (GETS) or produce (PUTS).

Clients execute GETS and PUTS using nearly identical two phase protocols. They first run a *read phase* to find the latest tag and value, then perform a *write phase*. For GETs, this phase writes the value read to ensure that replicas are up to date; for PUTs, it installs a new version with higher timestamp.

Read Phase. The client sends *Read* messages to all replicas. When a replica receives a *Read* message, it replies with its current value v and tag t. The client waits to receive a *ReadReply* from $f + 1$ replicas; then, it determines the most recent value and tag by choosing the value v_{max} associated with the maximum tag t_{max} returned in the $f + 1$ replies.

Write Phase. After the read phase, the client propagates a value v' and tag t' to $f + 1$ replicas based on the operation. For a GET, the client propagates v_{max} and t_{max}. For a PUT, the client propagates the new value v_{put} and forms a new tag t_{put} that is larger than $t_{max} = (ts_{max}, id_{max})$. It does so by choosing $t_{put} = (ts_{max} + 1, id_c)$ where id_c is the client identifier.

In either case, the client sends *Write* messages to all replicas that contain v' and t'. When a replica receives a $Write(v', t')$ message, it overwrites v with v' and t with t' if $t' > t$. Finally, a replica notifies the client that it updated its value and tag with a *WriteReply* message. The client completes the operation once it has received $f + 1$ *WriteReply* messages.

Takeaways. Fault tolerance is ensured when at most f out of $2f + 1$ replicas fail; every GET and PUT only needs responses from $f+1$ replicas to make progress. Linearizability is ensured because the tag observed or constructed by an operation is at least as large as the tag stored at $f + 1$ replicas. This is larger than the tag of any operation that previously completed because a complete operation propagates its tag to $f + 1$ replicas and these two sets of $f + 1$ replicas must intersect. All operations take two rounds of communication from the coordinating process to $f + 1$ replicas.

7.2 Block Store with Standard RDMA

In principle, multi-writer ABD seems amenable to an implementation that uses standard RDMA READ, WRITE, and CAS. A client reads the values and tags of a block, performs a local decision, and writes back new values and tags to replicas. The key challenge is in ensuring that these reads and writes happen atomically with respect to each other when multiple clients concurrently access the same block.

Existing work has also made this observation in the context of unreplicated storage [6, 43, 44]. In the DrTM family of systems [44], locks are used to mediate concurrent accesses to shared objects. A DrTM client may only read or write an object once it acquires the object's lock via a CAS. Once the client acquires the lock, it may read or write the object in place knowing that no other clients can concurrently write.

Our baseline that uses standard RDMA adapts this protocol in the context of multi-writer ABD. We assume that each block is of fixed size so that data may be stored in place.

The data for a block is stored in a known location. The first 8 bytes are the *lock* that holds the identifier of the client that currently has exclusive access to the block. The next 8 bytes are the tag t of the block. The remaining bytes are the value v.

Read Phase. Before a client reads the value and tag of the block from a replica, it acquires the lock at a majority of replicas by performing a CAS at each replica where its own identifier is swapped with *lock* only if the current value of *lock* is 0. Once a client succeeds in acquiring the lock at a majority, it reads t and v with a single READ. The rest of the read phase is performed locally at the client once it acquires locks and reads the tags and values from $f + 1$ replicas. If the client fails to acquire the lock at a majority, it releases any locks it acquired and retries after a backoff period.

Write Phase. The client determines the tag t' and value v' to propagate locally. Then, it performs a WRITE of this data to each replica on which it has acquired a lock. When the writes are acknowledged by $f + 1$ replicas, the client releases all locks that it owns with a CAS by swapping 0 with *lock* only if the current value of *lock* is the client's identifier.

Discussion. As we will see in the evaluation, using standard RDMA imposes a significant performance penalty: it

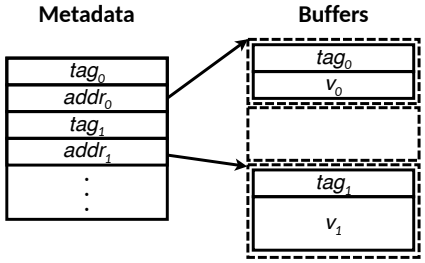

Figure 5. Layout of block data in memory for PRISM-RS. The addresses in the metadata array point to buffers registered for use with WRITE-ALLOCATE.

adds two additional round trips per GET/PUT for the CAS. Moreover, lock contention may delay operations.

There are additional issues with using this approach. There must be a protocol to force release locks if a client fails part way through an operation. Also, the system may enter a live-locked state if more than two clients attempt to acquire a lock for the same block from a majority or replicas. Furthermore, if replicas fail part way through an operation, clients must acquire additional locks at the remaining replicas.

7.3 PRISM-RS Design

The PRISM primitives enable an efficient implementation of the multi-writer ABD in RDMA. The ALLOCATE primitive and indirection allow multiple clients to write data to and read data from the same block atomically The CAS primitive guarantees that ordering updates at replicas takes a single round trip from clients even under contention.

PRISM-RS maintains an array of metadata in a known memory location. The array element at index i contains the metadata for block i: the tag tag_i and address $addr_i$. The tag_i is the version of block i that the replica currently stores and $addr_i$ is a pointer to a memory location that contains (1) the same tag_i and (2) the corresponding value. For simplicity, we assume fixed-size blocks, but it can be extended to variable-sized blocks by adding a len_i metadata field as in PRISM-KV.

An unusual feature of this representation is that the tag is intentionally duplicated in both the metadata array and the buffer it references. This ensures that (1) it is possible to read both the tag and value atomically with a single indirect READ to $addr_i$, and (2) that it is possible to make an update conditional on the current tag by performing a CAS on the $\langle tag_i, addr_i \rangle$ pair in the metadata array.

Read Phase. A client reads the value and tag of block i from each replica by performing a READ of the metadata array at the location where $\texttt{addr}_i$ is stored with the indirect bit set. Following the ABD protocol, it waits for responses from $f + 1$ replicas and determines the maximum tag and associated value.

Write Phase. After the client chooses a new tag t' and value v' to propagate, it updates the replicas using a chain of PRISM primitives. Each (except the first) has the conditional bit set:

1. WRITE t' to a temporary location `tmp`.
2. ALLOCATE and write $t'|v'$ to a new buffer, redirecting the output address to a temporary location `tmp_addr` immediately following the tag at `tmp` (i.e., redirect to $\texttt{tmp} + \texttt{sizeof}(t')$).
3. CAS with the indirect bit set. The CAS target address is the metadata array at the location where $\texttt{tag}_i|\texttt{addr}_i$ is stored; the comparand is `tmp`. The `CAS_GT` comparison is used and the bitmasks configured so that the tag and address stored at `tmp` are written only if $t' > \texttt{tag}_i$.

The client waits for acknowledgments to the CAS at $f + 1$ replicas before completing. Note that the CAS operation returns the old value of $\texttt{tag}_i|\texttt{addr}_i$. Since $\texttt{addr}_i$ is no longer in use, the client asynchronously notifies the replica so that the buffer can be returned to the free list.

Correctness. While the correctness of the protocol generally follows from the correctness of the MRMW ABD protocol itself [25], some attention to the atomicity of concurrent PRISM READs, WRITEs, and other operations is required. First, at any moment, both copies of the tag are consistent with each other and the value, because the CAS operation in the write phase atomically updates both the pointer and the tag in the metadata block. Second, in the read phase, the tag and value read from each node are guaranteed to be consistent with each other because they are stored together in a block that is never modified once it is ALLOCATEd and written to for the first time. This use of write-once, out-of-place updates avoids issues with concurrent writes, without the checksums or self-certifying data structures needed by other systems [31].

7.4 Evaluation

We compare PRISM-RS with the one-sided lock-based ABD variant described above (ABDLOCK, §7.2). As before, we also compare against ABDLOCK using the same software RDMA implementation as PRISM. We use 8 million 512-byte objects, replicated on 3 servers, and a 50% write workload.

On a uniform access distribution (Figure 6) where there is low contention, PRISM-RS outperforms even the one-sided RDMA ABDLOCK variant in both latency and throughput. The lower message complexity enabled by the PRISM primitives allows PRISM-RS to be about 2 μs faster than ABDLOCK and reach $\sim$ 4 million more ops/sec before saturating the network. The benefits are even more dramatic on workloads where there is contention on popular keys. Figure 7 considers a workload where 100 closed-loop clients access keys with a Zipf distribution. PRISM-RS remains as responsive for any contention level, while the performance of ABDLOCK degrades significantly due to lock contention.

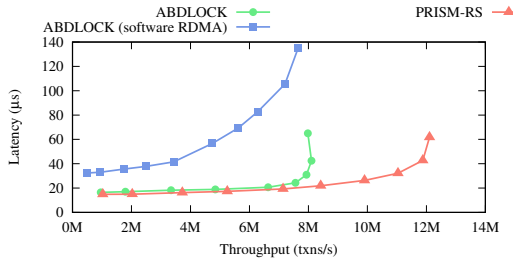

Figure 6. Throughput-latency comparison between PRISM-RS and the two variants of lock-based ABD.

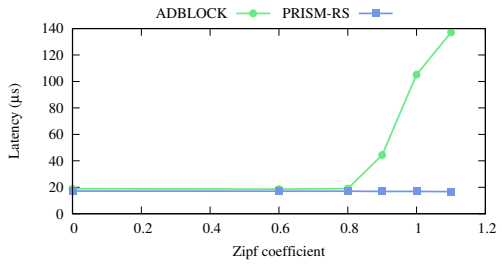

Figure 7. Latency comparison between PRISM-RS and ABDLOCK for various degrees of contention.

8 PRISM-TX: Distributed Transactions

We next move to the problem of providing distributed transactional storage. Specifically, we consider a storage system where data is partitioned among multiple servers, and clients group operations into transactions. During the execution phase of a transaction, they read or write data from different servers. Subsequently, clients request to commit a transaction, and the system either commits it atomically or aborts it. All committed transactions are serializable. Providing these transactional semantics has long been a classic problem for distributed databases [5]; more recent systems have used RDMA to accelerate transactional storage [6, 10, 19, 43, 44].

8.1 Background: FaRM

FaRM is a representative example of the state of the art in transaction processing over RDMA [10]. In FaRM, data is stored in a hash table that is accessible via RDMA and each object is associated with a version number and a lock. During the execution of a transaction, clients perform reads by accessing server memory using one-sided RDMA READ operations, and buffer writes locally until the commit phase. For the key-value store variant of FaRM, each access can require two READs, as in Pilaf. The commit process is a three-phase protocol requiring CPU involvement. Clients first lock all objects in the write set. Once they have done so, they reread all objects in the read set to verify that they have not been concurrently modified. Finally, they update the objects that have been written, and unlock them. The second phase can be implemented using READs; the other two require RPCs.

FaRM benefits from RDMA's low latency transport, yet it still requires significant server CPU involvement. Although reads are implemented using one-sided RDMA operations, the more complex logic of the commit phase cannot be.

8.2 PRISM-TX Design

Can we build a distributed transaction protocol that implements both its execution and commit phases using remote operations? Using PRISM's new primitives, notably the enhanced CAS operation, we implement optimistic concurrency control checks in a protocol called PRISM-TX.

The design of PRISM-TX draws inspiration from Meerkat [38], a recent distributed optimistic concurrency control protocol. Meerkat serves as an excellent starting point for our PRISM-TX design, because it partitions concurrency control metadata by key, augmenting a hash table with additional metadata for each object (in contrast to traditional OCC designs that maintain and scan lists of active transactions [21]). While Meerkat uses this to avoid multi-core bottlenecks, we leverage it to provide remote access: the OCC metadata is stored in well-defined, per-key locations.

In PRISM-TX, as in Meerkat, each transaction has a timestamp. These timestamps are selected at the client using a loosely synchronized clock, a strategy used in many prior systems [1, 8, 38, 40, 46]. Clients execute reads using RDMA operations, and buffer writes locally. To commit a transaction requires two phases. First, in the prepare phase, a client checks whether a conflicting transaction has prepared or committed, and records its intent to commit. It does so by updating timestamps associated with the records it read and wrote using PRISM's CAS operation. If this succeeds, it then commits the transaction by installing its writes.

Memory layout. PRISM-TX builds on both PRISM-KV and PRISM-RS. Like PRISM-KV, data is arranged in a hash table at each server and keys are accessed by probing for the correct hash table slot. Like PRISM-RS, each slot contains concurrency control metadata and a pointer to the location where committed data is stored. Figure 8 depicts the hash table and metadata. In particular, for each key, the following information is maintained:

PR – the timestamp of the most recent transaction that read the key and has prepared to commit.
PW – the timestamp of the most recent transaction that needs to write the key and has prepared to commit.
C – the timestamp of the most recent committed transaction that wrote this key.

Execution phase. The client executes the transaction by performing its reads using a mechanism similar to PRISM-KV's GETS (§6.1) at the corresponding partition servers, and by buffering its writes. The execution phase results in a *ReadSet*, a set containing a tuple $\langle key, RC \rangle$ for every key the transaction read, where *RC* is the version of the key (i.e., its value of *C*) that was read, and a *WriteSet*, a set containing

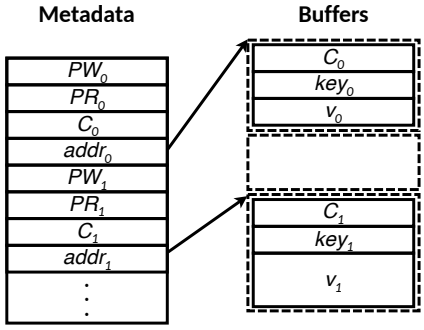

Figure 8. Memory layout for PRISM-TX. The addresses in the metadata array point to buffers registered for use with ALLOCATE.

a tuple $\langle key,\ value \rangle$ for every key the transaction needs to write.

Prepare phase. After the execution phase, the client selects a logical commit timestamp *TS* for the transaction using loosely synchronized logical clocks [1, 40], in the same way as Meerkat: it chooses a tuple $\langle clock\_time, cid \rangle$. The *clock_time* is initially set to the local clock time of the client, but it is adjusted such that $TS > RC$ for all *RC*s of the read keys. *cid* is the client's id, appended to ensure timestamps are unique.

The client then performs a series of *validation checks* to establish if the transaction can be ordered at *TS*. For every key in the *ReadSet*, the client performs a read validation check: it checks that no concurrent transaction has prepared to write to that key, i.e., the transaction read the latest version. If not, it updates *PR* to record its read and block conflicting writes:

- **Read Validation**: for each key i, RC in *ReadSet*,
  if $RC = PW_i$, update $PR_i = \max(TS, PR_i)$

Note that the condition is equivalent to $RC \geq PW$, because *PW* never decreases and is always larger than *C*. As a result, this can be expressed as a single CAS operation that checks if $RC|TS$ is greater than $PW|PR$ (where | is the concatenation of the two values). This performs the update atomically. The CAS can fail if the check fails ($RC < PW$) *or* if the update was unnecessary (*PR* was already greater than *TS*); the client can distinguish the two using the value returned by the CAS.

If all read validation checks succeed, the client moves on to validate the writes. For every key in the *WriteSet*, the client performs a write validation check: it checks that writing this key at *TS* does not invalidate concurrent reads (transactions that read might appear not to have read the latest version). It does so by checking that $TS > PR$. The client also checks that it is the most recent write by checking that $TS > PW$.

- **Write Validation**: for each key i in *WriteSet*,
  if $TS > PR_i$ and $TS > PW_i$, update $PW_i = TS$.

This cannot be performed as a single CAS operation. However, note that the update does not need to be performed atomically with *both* conditions; it suffices for it to be performed atomically with the second condition, as long as the first condition is checked after. Intuitively, this is because it is always safe to increment *PW*, as this only prevents concurrent readers from committing. In other words, it is safe to optimistically record the transaction's intent to write the key (even if it does not ultimately succeed). Doing so may cause unnecessary aborts, but does not violate correctness. Thus, PRISM-TX performs the $TS > PW$ check and updates $PW = TS$ with a CAS operation, then if it succeeds, separately checks that $TS > PR$.

Commit phase. If all validation checks are successful, the client commits the transaction and applies each write $\langle key, value \rangle$ in *WriteSet* using a sequence of PRISM primitives. It follows the same ALLOCATE/WRITE/CAS pattern in PRISM-RS's write phase (§7.3), allocating a new buffer containing $TS \mid key \mid value$ and installing a pointer to it in the appropriate slot as long as $TS > C_i$.

If any validation check fails, the client aborts the transaction. Typical transaction protocols (including Meerkat) would then undo the metadata updates during the prepare phase; PRISM-TX cannot do this because it tracks only the *latest* timestamps for *PR* and *PW*. Instead, it leaves these timestamps as is. As noted above, it is always safe to use a higher (i.e., conservative) value for *PR* and *PW*. However, this can block other transactions from committing. To reduce this effect, the client updates *C* to *TS* if $TS > C$, for every key it successfully completed the write check for.

Correctness. PRISM-TX ensures serializability: transactions appear to execute in timestamp order. We cannot provide a detailed proof due to space constraints, but note that the proof follows as an extension of Meerkat's protocol [38]. Intuitively, PRISM-TX prevents anomalies by forbidding a transaction *T* from committing if any transaction with an earlier timestamp modified the data *T* read, or if any transaction with a later timestamp read or modified the earlier version of data *T* modified. PRISM-TX follows essentially the same timestamp assignment and validation protocol as Meerkat, except PRISM-TX's read and write validation rules are strictly more conservative than the ones used in Meerkat, as Meerkat tracks the full list of prepared transactions that read or wrote a key. PRISM-TX tracks only the *highest* timestamps of prepared transactions that read (*PR*) or modified (*PW*) a key.

8.3 Evaluation

We compare PRISM-TX with our implementation of the FaRM protocol, again using both hardware and software RDMA implementations for FaRM. Due to the limited size of our testbed, we use only a single shard (but still run the full distributed commit protocol). We use a YCSB-T [9] workload, consisting of short read-modify-write transactions, with 8 million 512-byte objects. Figure 9 shows the performance

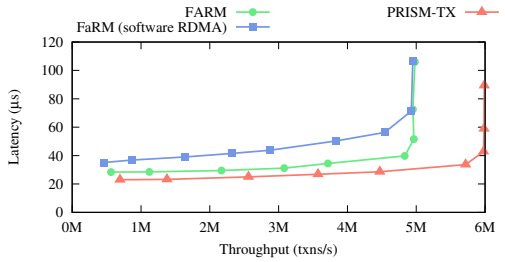

Figure 9. Throughput-latency comparison between PRISM-TX and FaRM for YCSB-T workload with low contention.

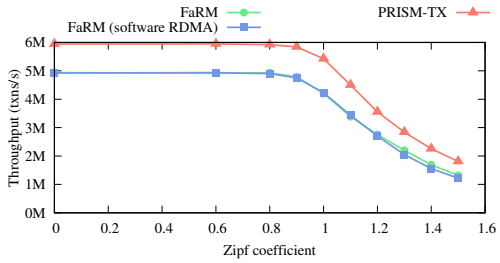

Figure 10. Peak throughput comparison between PRISM-TX and FaRM for YCSB-T workload with varying contention.

results for YCSB-T using a uniform access pattern. PRISM-TX outperforms FaRM in both latency and throughput. With a lower message complexity, enabled by the PRISM primitives, PRISM-TX is 5.5 μs faster than FaRM and reaches 1 million more transactions per second before saturating the network.

Because PRISM-TX uses a different concurrency control protocol, the performance tradeoff may vary with different workloads. Optimistic protocols can suffer under high contention. Figure 10 evaluates the peak throughput under different levels of workload skew (Zipf distributions). PRISM-TX maintains its performance benefit under high contention.

9 Related Work

RDMA Systems. Many storage systems have been built around the RDMA interface; we do not aim to provide a complete list, but merely to highlight relevant trends. Pilaf [31]'s hash table uses indirection, so it requires two reads to perform a GET, and uses a RPC for a PUT. Cell [32] implements a B-tree, which requires even more round trips to perform a read (though caching can be effective). XStore [42] replaces the tree with a learned index structure to search with fewer RDMA reads, but still requires indirection. PRISM's indirection primitives can help many of these systems. Among transaction systems, FaRM [10] uses RDMA to read data, but RPCs to commit updates. DrTM builds a lock-based protocol [44] from one-sided operations. PRISM broadens the design space by enabling one-sided OCC protocols.

Other work argues that two-sided RPCs can outperform one-sided RDMA operations. HERD [17] eschews one-sided reads for a mix of one-sided writes and two-sided operations, and FaSST [19] implements transactions using a RPC mechanism. DrTM+H[43] improves DrTM's performance using a hybrid one-sided/two-sided approach. Octopus uses a similar hybrid approach for a file system [24]. PRISM, by offering more complex remote operations, offers a middle ground.

RDMA Extensions. Implementers have occasionally defined new extensions to the RDMA model. Mellanox's extended atomics API [30] allows CAS operations to compare and swap on parts of a larger operand vs. one 8-byte value. Snap's software RDMA stack [26] supports indirect operations as well as a pattern-search primitive, and these are used within Google, though not publicly described in detail. Simple primitives for far-memory data structures [2] have also been proposed, including indirect addressing.

StRoM [37] and RMC [3] propose to allow applications to install their own one-off primitives on FPGA and multi-core NICs respectively. This shares our goal of adding server-side processing to avoid extra network round trips – and indeed supports more complex server-side processing. However, as discussed in §2.3, running custom application logic presents deployment challenges, and supporting a small library of generic primitives affords more implementation possibilities. PRISM demonstrates that such an API can be useful.

HyperLoop [20] implements a different form of chaining than PRISM: it shows that, with a specially crafted RDMA request, a sender can cause a receiver to initiate a RDMA request to a *third* machine. This could be used in combination with PRISM to implement other communication patterns. RedN [36] takes this approach further, using self-modifying RDMA chains as Turing machines, a potential approach to implementing PRISM primitives on existing hardware.

10 Conclusion

We have presented PRISM, an extended API for access to remote memory. PRISM offers a middle ground between the RDMA READ/WRITE interface and the full generality of RPC communication. PRISM significantly expands the design space for network-accelerated applications, making it possible to build new types of applications with minimal server-side CPU involvement – as demonstrated by our key-value store, replicated storage, and distributed transaction examples.

Using a software implementation of PRISM, we showed that the more advanced interface allows building more efficient protocols – often outperforming ones that use hardware-accelerated RDMA. Looking forward, a hardware implementation of PRISM is feasible and would permit even higher performance with better CPU efficiency – enabling new deployment options such as network-attached memory nodes.

Acknowledgments

We thank Lorenzo Alvisi and Anirudh Badam, along with the anonymous reviewers from OSDI and SOSP and our shepherd Amin Vahdat, for their valuable feedback.

References

[1] Atul Adya, Robert Gruber, Barbara Liskov, and Umesh Maheshwari. 1995. Efficient Optimistic Concurrency Control Using Loosely Synchronized Clocks. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*. ACM, San Jose, CA, USA.

[2] Marcos K. Aguilera, Kimberly Keeton, Stanko Novakovic, and Sharad Singhal. 2019. Designing Far Memory Data Structures: Think Outside the Box. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*.

[3] Emmanuel Amaro, Zhihong Luo, Amy Ousterhout, Arvind Krishnamurthy, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. 2020. Remote Memory Calls. In *Proceedings of the 16th Workshop on Hot Topics in Networks (HotNets '20)*. ACM, Chicago, IL, USA.

[4] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. 1990. Sharing memory robustly in message-passing systems. In *Proceedings of the 9th ACM Symposium on Principles of Distributed Computing (PODC '90)*. ACM, Quebec City, QC, Canada.

[5] Philip Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley.

[6] Yanzhe Chen, Xinda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. 2016. Fast and General Distributed Transactions Using RDMA and HTM. In *Proceedings of the 11th ACM SIGOPS EuroSys (EuroSys '16)*. ACM, London, United Kingdom.

[7] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of SOCC 2010*.

[8] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2012. Spanner: Google's Globally-Distributed Database. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*. USENIX, Hollywood, CA, USA.

[9] Akon Dey, Alan Fekete, Raghunath Nambiar, and Uwe Rohm. 2014. YCSB+T: Benchmarking web-scale transactional databases. In *Proceedings of the 30th International Conference on Data Engineering Workshops (ICDEW)*.

[10] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. 2014. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14)*. USENIX, Seattle, WA, USA.

[11] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*. ACM, Bolton Landing, NY, USA.

[12] Chuanxiong Guo. 2017. RDMA in Data Centers: Looking Back and Looking Forward. Keynote at APNet.

[13] Chuanxiong Guo, Haitao Wu, Zhong Deng, Jianxi Ye Gaurav Soni, Jitendra Padhye, and Marina Lipshteyn. 2016. RDMA over Commodity Ethernet at Scale. In *Proceedings of ACM SIGCOMM 2016*. ACM, Florianopolis, Brazil.

[14] Sagar Jha, Jonathan Behrens, Theo Gkountouvas, Matthew Milano, Weijia Song, Edward Tremel, Robbert van Renesse, Sydney Zink, and Kenneth P. Birman. 2019. Derecho: Fast State Machine Replication for Cloud Services. *ACM Trans. Comput. Syst.* 36, 2 (2019), 4:1–4:49.

[15] 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 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2012)*. IEEE, Ottawa, ON, Canada.

[16] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter RPCs can be general and fast. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19)*. USENIX, Boston, MA, USA.

[17] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2014. Using RDMA Efficiently for Key-Value Services. In *Proceedings of ACM SIGCOMM 2014*. ACM, Chicago, IL, USA.

[18] 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, Denver, CO, USA.

[19] 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, Savannah, GA, USA.

[20] Daehyeok Kim, Amirsaman Memaripour, Anirudh Badam, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Shachar Raindel, Steven Swanson, Vyas Sekar, and Srinivasan Seshan. 2018. Hyperloop: group-based NIC-offloading to accelerate replicated transactions in multi-tenant storage systems. In *Proceedings of ACM SIGCOMM 2018*. ACM, Budapest, Hungary.

[21] H. T. Kung and John T. Robinson. 1981. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems* 6, 2 (June 1981), 213–226.

[22] Jiaxin Lin, Kiran Patel, Brent E. Stephens, Anirudh Sivaraman, and Aditya Akella. 2020. PANIC: A High-Performance Programmable NIC for Multi-tenant Networks. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*. USENIX, Banff, AL, Canada.

[23] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. 2019. Offloading distributed applications onto SmartNICs using iPipe. In *Proceedings of ACM SIGCOMM 2019*. ACM, Beijing, China.

[24] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. 2017. Octopus: an RDMA-enabled distributed persistent memory file system. In *Proceedings of the 2017 USENIX Annual Technical Conference*. USENIX, Santa Clara, CA, USA.

[25] Nancy Lynch and Alex Shvartsman. 1997. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proceedings of the 27th Annual International Symposium on Fault-Tolerant Computing (FTCS' 97)*. IEEE, Seattle, WA, USA, 272–281.

[26] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. 2019. Snap: A Microkernel Approach to Host Networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. ACM, Shanghai, China.

[27] Paul E. McKenney, Jonathan Appavoo, Andi Kleen, Orran Krieger, Rusty Russell, Dipankar Sarma, and Maneesh Soni. 2002. Read-Copy Update. In *Proceedings of the 2002 Ottawa Linux Symposium*. Ottawa, ON, CA, 336–367.

[28] Mellanox Technologies. [n.d.]. ConnectX Ethernet Adapters. https://www.mellanox.com/products/ethernet/connectx-smartnic.

[29] Mellanox Technologies. [n.d.]. RDMA Extended Atomics. https://docs.mellanox.com/display/rdmacore50/Extended%20Atomics.

[30] Mellanox Technologies 2015. *RDMA Aware Networks Programming User Manual*. Mellanox Technologies. Revision 1.7.

[31] 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 2013 USENIX Annual Technical Conference*.

USENIX, San Jose, CA, USA.

[32] Christopher Mitchell, Kate Montgomery, Lamont Nelson, Siddhartha Sen, and Jinyang Li. 2016. Balancing CPU and Network in the Cell Distributed B-Tree Store. In *Proceedings of the 2016 USENIX Annual Technical Conference*. USENIX, Denver, CO, USA.

[33] S. Narravula, A. Marnidala, A. Vishnu, K. Vaidyanathan, and D. K. Panda. 2007. High Performance Distributed Lock Management Services using Network-based Remote Atomic Operations. In *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2007)*. IEEE, Rio de Janeiro, Brazil.

[34] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. 2015. Latency-Tolerant Software Distributed Shared Memory. In *Proceedings of the 2015 USENIX Annual Technical Conference*. USENIX, Santa Clara, CA, USA.

[35] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. 2018. Understanding PCIe Performance for End Host Networking. In *Proceedings of the 2018 ACM SIGCOMM* (Budapest, Hungary).

[36] Waleed Reda, Marco Canini, Dejan Kostić, and Simon Peter. 2022. RDMA is Turing complete, we just did not know it yet!. In *Proceedings of NSDI '22*.

[37] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. 2020. StRoM: Smart Remote Memory. In *Proceedings of the 15th ACM SIGOPS EuroSys (EuroSys '20)*. ACM, Heraklion, Crete, Greece.

[38] Adriana Szekeres, Michael Whittaker, Naveen Kr. Sharma, Jialin Li, Arvind Krishnamurthy, Irene Zhang, and Dan R. K. Ports. 2020. Meerkat: Scalable Replicated Transactions Following the Zero-Coordination Principle. In *Proceedings of the 15th ACM SIGOPS EuroSys (EuroSys '20)*. ACM, Heraklion, Crete, Greece.

[39] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. 1997. Frangipani: A Scalable Distributed File System. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*. ACM, Saint-Malo, France.

[40] Robert H. Thomas. 1979. A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases. *ACM Transactions on Database Systems* 4, 2 (June 1979), 180–209.

[41] Tao Wang, Hang Zhu, Fabian Ruffy, Xin Jin, Anirudh Sivaraman, Dan R. K. Ports, and Aurojit Panda. 2020. Multitenancy for fast and programmable networks in the cloud. In *Proceedings of the 11th Hot Topics in Cloud Computing (HotCloud '20)*. Boston, MA, USA.

[42] Xingda Wei, Rong Chen, and Haibo Chen. 2020. Fast RDMA-based Ordered Key-Value Store using Remote Learned Cache. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*. USENIX, Banff, AL, Canada.

[43] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. 2018. Deconstructing RDMA-enabled Distributed Transactions: Hybrid is Better!. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*. USENIX, Carlsbad, CA USA.

[44] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. 2015. Fast In-memory Transaction Processing using RDMA and HTM. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*. ACM, Monterey, CA, USA.

[45] Dong Young Yoon, Mosharaf Chowdhury, and Barzan Mozafari. 2018. Distributed Lock Management with RDMA: Decentralization without Starvation. In *Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data*. ACM, Houston, TX, USA.

[46] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. 2015. Building Consistent Transactions with Inconsistent Replication. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*. ACM, Monterey, CA, USA.