Storm: a fast transactional dataplane for remote data structures

Stanko Novakovic[1*] Yizhou Shan[3] Aasheesh Kolli[2,4] Michael Cui[2]
Yiying Zhang[3] Haggai Eran[5,6] Boris Pismenny[5] Liran Liss[5] Michael Wei[2]
Dan Tsafrir[2,6] Marcos Aguilera[2]

[1]*Microsoft Research* [2]*VMware* [3]*Purdue University* [4] *The Pennsylvania State University* [5] *Mellanox* [6] *Technion*

Abstract

RDMA technology enables a host to access the memory of a remote host without involving the remote CPU, improving the performance of distributed in-memory storage systems. Previous studies argued that RDMA suffers from scalability issues, because the NIC's limited resources are unable to simultaneously cache the state of all the concurrent network streams. These concerns led to various software-based proposals to reduce the size of this state by trading off performance.

We revisit these proposals and show that they no longer apply when using newer RDMA NICs in rack-scale environments. In particular, we find that one-sided remote memory primitives lead to better performance as compared to the previously proposed unreliable datagram and kernel-based stacks. Based on this observation, we design and implement Storm, a transactional dataplane utilizing one-sided read and write-based RPC primitives. We show that Storm outperforms eRPC, FaRM, and LITE by 3.3x, 3.6x, and 17.1x, respectively, on an InfiniBand cluster with Mellanox ConnectX-4 NICs.

CCS Concepts • **Networks** → *Network performance evaluation*; • **Software and its engineering** → *Distributed systems organizing principles*;

Keywords RDMA, RPC, data structures

1 Introduction

RDMA is coming to data centers [3, 12, 21, 28, 47]. While RDMA was previously limited to high-performance computing environments with specialized Infiniband networks, RDMA is now available in cheap Ethernet networks using technologies such as RoCE [2] or iWARP [36]. The main novelty of RDMA is *one-sided operations*, which permit an application to directly read and write the memory of a remote host without the involvement of the remote CPU. In theory, one-sided operations are supposed to lower latency, improve throughput, and reduce CPU consumption. However, prior work shows that one-sided operations suffer from scalability issues: with more than a few hosts, overheads in RDMA can overwhelm the benefits that it provides [10, 16].

Our first contribution is a study of multiple generations of RDMA NICs to understand how hardware evolution addresses (or not) its scalability concerns. The conventional wisdom is that one-sided RDMA performs poorly because of three issues (§3). First, it requires the use of reliable connections, which can exhaust the memory cache of the NIC. Second, one-sided RDMA typically demands virtual-to-physical address translation and memory-region protection metadata, which can also exhaust the NIC cache. Third, one-sided RDMA can incur many network round trips when an application wants to chase pointers remotely in dynamic data structures.

In this paper, we reexamine these problems in light of new and better hardware relative to prior work [10, 16, 42]. We find that some of the problems are mitigated; they are no longer a concern for rack-scale systems of up to 64 machines. Through experiments, we demonstrate that newer hardware efficiently supports a significantly larger number of connections than before, eschewing the scalability problem for rack-scale. Furthermore, we argue that connections actually *help* performance, as they permit delegating congestion control to the hardware and enable one-sided operations. Thus, systems should use reliable connections as the only transport for RDMA communication (§4). This is in stark contrast to some previous proposals, such as HERD [15], FaSST [16], and eRPC [14], which call for abandoning reliable connections with one-sided operations in favor of the unreliable transport with send/receive operations.

The second issue (virtual address translation and protection metadata) is mitigated in newer hardware but remains. While future hardware might solve this problem altogether (with larger NIC cache and better mechanisms to manage it), we must still address it today. Prior solutions suggest the use of huge pages to reduce region metadata [10] or access RDMA

* Work done while at VMware

SYSTOR '19, June 3–5, 2019, Haifa, Israel

ACM ISBN 978-1-4503-6749-3/19/06…$15.00
https://doi.org/10.1145/3319647.3325827

using physical addresses through a kernel interface [42]. These approaches are effective but have some drawbacks: huge pages are prone to fragmentation, while a kernel interface suffers from syscall overheads and lock contention issues. In this work, we propose enforcing contiguous memory allocation and leveraging the support for physical segments in user-space (§4): We find this approach to greatly reduce region metadata without the concerns of fragmentation or kernel overheads.

Third issue (round trips to chase pointers) is fundamental but arises only in certain workloads and data structures that require pointer chasing. Prior solutions fall in two categories: (1) replace one-sided operations with RPCs [14, 16], so that the RPC handler at the remote host can chase the pointers and send a reply in a single round trip, or (2) use data inlining and perform larger one-sided reads [10]. In this work, we adopt a new approach that performs better than prior solutions: the system dynamically determines whether to use one-sided operations or RPCs, depending on whether pointers need to be chased, and then uses the best mechanism. We refer to this hybrid scheme combining one-sided reads and write-based RPCs as *one-two-sided* operations (§4). When using RPCs, we employ one-sided write operations to transmit the RPC requests and replies.

Based on our insights, we design and implement a high-speed, transactional RDMA dataplane called Storm (§5). Storm can effectively use one-sided operations in a rack-scale system, despite prior concerns that they suffer from poor performance [14, 16]. We evaluate Storm and compare it against three state-of-the-art RDMA systems: FaSST/eRPC [14, 16], FaRM [10], and LITE [42].

eRPC is designed to avoid one-sided operations altogether. We show that Storm outperforms eRPC up to 3.3x by effectively using one-sided operations for direct reads and RPCs. Unlike two-sided reads, one-sided reads enable full-duplex *input-output operations per second*(IOPS) rates; no CPU-NIC interaction for processing replies. FaRM is designed and evaluated under an older generation of hardware and includes a locking mechanism to share connections. Our measurements show that this mechanism is no longer needed and produces overhead with newer hardware; we thus improve FaRM by removing the locking mechanism and our comparison refers to this improved design. Our evaluation shows that Storm outperforms the improved FaRM up to 3.6x. Our better performance comes primarily from avoiding large reads in FaRM and instead using fine-grained reads combined with our hybrid *one-two-sided* operations. For smaller key-value pairs, FaRM performs better compared to our FaRM measurements, which are based on 128-byte data items (1KB bucket neighborhood size). Smaller key-value pairs result in smaller bucket sizes, leading to higher IOPS rates. Finally, LITE is designed to work in the kernel; we improved LITE by extending it with support for asynchronous operations; our comparison refers to this improved scheme. Our evaluation shows that Storm outperforms the improved LITE up to 17.1x. Our better performance comes primarily from using user-space operations and a design that is free of dependencies, while we find that LITE is bottlenecked by the kernel overheads and sharing among the kernel and user-level threads (§6).

To summarize, we make the following contributions:

- We perform an experimental study of three generations of hardware to understand how its evolution addresses (or not) each problem facing one-sided operations.
- We build a fast RDMA dataplane called Storm, which incorporates the lessons we learned from our experimental study. Storm provides a well-understood transactional API for manipulating remote data structures and allows the developer to implement any such data structure using a callback mechanism.
- We evaluate Storm and compare it against eRPC, and improved versions of FaRM and LITE, dubbed Lock-free_FaRM and Async_LITE. We show that Storm performs well in a rack-scale setting with up to 64 servers and outperforms eRPC, lock-free FARM, and Asynchronous LITE by 3.3x, 3.6x, and 17.1x in throughput.

Ultimately, Storm refutes a widely held belief that one-sided operations—the main novelty of RDMA—are inefficient due to its scalability issues.

2 Background

2.1 Remote Direct Memory Access (RDMA)

RDMA allows applications to directly access memories of remote hosts, with user-level and zero-copy operations for efficiency. Moreover, RDMA offloads the network stack to the Network Interface Card (NIC), reducing CPU consumption. RDMA was originally designed for specialized InfiniBand (IB) networks used in high-performance computing [35]. More recently, the IB transport has been adapted for Ethernet networks, bringing RDMA to commodity datacenter networks [12, 47].

Memory management. To use RDMA, applications register memory regions with the NIC, making them available for remote access. During registration, the NIC driver pins the memory pages and stores their virtual-to-physical address translations in *Memory Translation Tables* (MTTs). The NIC driver also records the memory region permissions in *Memory Protection Tables* (MPTs). When serving remote memory requests, the NIC uses MTTs and MPTs to locate the pages and check permissions. The MTTs and MPTs reside in system memory, but the NIC caches them in SRAM. If the MTTs and MPTs overflow the cache, they are accessed from main memory via DMA/PCIe, which incurs overhead.

Queue pairs. Applications issue RDMA requests via the IB transport API, known as IB verbs. IB verbs use memory-mapped control structures called *Queue Pairs* (QPs). Each QP consists of a Send Queue (SQ) and a Receive Queue (RQ). Applications initiate RDMA operations by placing Work Queue Entries (WQEs) in the SQ; when operations complete, applications are notified through the Completion Queue (CQ). This asynchronous model allows applications to pipeline requests and do other work while operations complete.

RDMA supports two modes of communication: one-sided performs data transfers without the remote CPU; two-sided is the traditional send-receive paradigm, which requires the remote CPU to handle the requests. One-sided operations (`read`/`write`) deliver higher throughput (i.e., IOPS), while two-sided operations (`send`/`recv`) offer more flexibility as they involve the remote CPU.

Transports. RDMA supports different transports; we focus on two: Reliably Connected (RC) and Unreliable Datagram (UD). The RC transport requires endpoints to be connected and the connection to be associated with a QP. For each QP, the system must keep significant state: QP metadata, congestion control state [28, 47], in addition to WQEs, MTTs, and MPTs. QP state amounts to $\approx 375B$ per connection [14]. UD does not require connections; a single QP allows an endpoint to communicate with any target host. Thus, UD requires significantly fewer QPs, which saves transport state. But UD has some drawbacks: it is unreliable (requests can be lost), it does not support one-sided operations, and it requires receive buffers to be registered with the NIC, which impacts scalability.

2.2 Distributed in-memory systems using RDMA

Prior work shows how to build distributed in-memory storage systems using RDMA [7, 10, 16, 26, 27, 38, 46]. Such storage systems tend to have (i) high communication fan out; (ii) small data item size, and (iii) moderate computational overheads. Systems with these properties benefit from RDMA's low-latency and high IOPS rates. For example, in a transactional store, clients issue transactions with many read/writes on different objects, where data is partitioned across the servers [6]. Using RDMA, clients can read/write data using `read`s and `write`s or implement lightweight RPCs for that purpose, reducing the end-to-end latency and improving throughput.

3 Motivation

3.1 Problem statement

Our main goal is to use RDMA efficiently and scalably in a rack-scale setting. While some companies have deployments with thousands of machines, the vast bulk of enterprises use rack-scale deployments, consisting of one or a few racks with up to 64 machines in total; that is our target environment. Prior work has shown that RDMA-based distributed storage systems do not scale well in these settings [10, 14, 42]. As we add more machines and increase their memory, the amount of RDMA state increases. For good performance, the active RDMA state must be in the NIC's SRAM cache, but this cache is small and can be exhausted with a few remote peers [10]. When that happens, RDMA state spills to CPU caches and main memory, requiring expensive DMA operations over PCIe to access it. PCIe latency adds 300-400ns on unloaded systems to several microseconds on loaded systems [24, 32]. These DMA overheads are exacerbated with transaction processing workloads, which have high fan-out, fine-grained accesses.

3.2 Shortcomings of prior art

To mitigate this problem, several software solutions have been proposed. We focus on three systems trying to address RDMA scalability issues in software.

Systems using one-sided operations. Previously proposed FaRM [9] and LITE [42] use one-sided operations and try to reduce the number of QPs by sharing them across groups of threads. To share, these systems use locks, but locking degrades throughput [10]. Also, FaRM uses large reads to reduce the number of round-trips when performing lookups, limiting maximum throughput.

Unreliable datagram transport. Another way to reduce QP state is to use the UD transport, as in FaSST/eRPC [14, 16]. With UD, a thread uses just one QP to talk to all the machines in the cluster. However, UD precludes the efficient one-sided operations (`read`/`write`), requires application-level retransmission and congestion control, all of which limit maximum throughput (§6). Furthermore, we show that managing receive queues in UD impacts scalability.

Kernel-space RDMA stacks. LITE [42] provides a kernel interface for RPCs and remote memory mapping. Thus, LITE uses physical addressing and eliminates MTT/MPT overhead in the NIC, but it adds additional overhead due to frequent system calls which are now somewhat more costly due to recent kernel patches (i.e., KPTI, retpoline) [4]. Moreover, LITE operations are blocking, which limits concurrency and throughput. We extend LITE with asynchronous `read`s and RPCs to improve its throughput. This version achieves 2× higher throughput for a single thread (§6), but the maximum IOPS with multiple threads remains small compared to RDMA on a modern NIC. We find this occurs because of serialization and lock contention in LITE.

3.3 Revisiting RDMA hardware capabilities

Systems like FaRM or LITE were designed for older generations of NICs with very limited processing and memory resources. Their design choices (e.g., QP sharing and software address translation) improve performance on such NICs but underutilize the capabilities of newer hardware (CX4 and

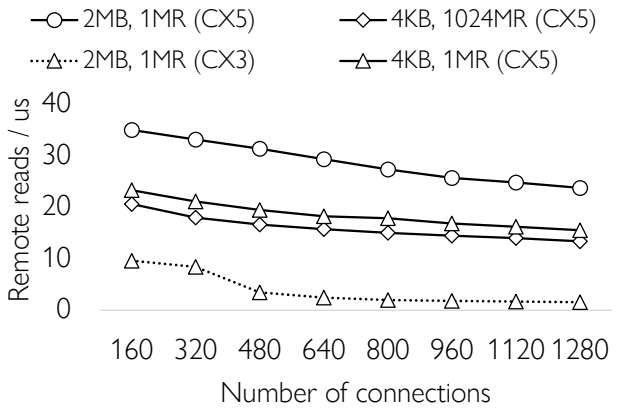

Figure 1. CX3 vs. CX5 comparison. CX5 provides better performance even when using 4KB pages (as opposed to 2MB) and a larger number of memory regions (1024MR). CX3 performs much worse even with 2MB pages and 1MR.

CX5). Figure 1 tries to capture a significant performance gap between CX3 and CX5 Mellanox RoCE NICs. It shows the throughput per machine for a workload that performs random 64-byte remote `reads` on 20GB of memory (2MB page sizes). For CX5, we show the performance when using 4KB pages and also when MPTs are larger; *4KB,1024MR (CX5)* uses 4KB pages and breaks the 20GB of buffer space into 1024 smaller RDMA memory regions (MR). On the X-axis we vary the number of connections between the source and destination servers. The RoCE hardware is described in Table 3.

Figure 1 shows that (i) CX5 significantly outperform CX3; (ii) CX5 scales better with the number of connections than CX3. We measure the throughput reductions going from 160 to 1280 connections to be: 83%, 42%, and 32% for CX3, CX4, CX5, respectively (CX4 results omitted for clarity); (iii) MTT and MPT remain a significant overhead with many memory regions and large page counts. Finally, we find that CX5 throughput becomes constant at around 10000 QP connections after reaching zero cache hit rate. The constant throughput that we measure is around 10 $reqs/\ \mu s$, which is equal to the maximum throughput a CX3 can provide (when there is no contention). Next, we list a number of factors that drive the better performance of modern NICs.

Larger caches, better cache management. CX4 and CX5 have larger caches (≈2MB) [14] for RDMA state, reducing the number of PCIe/DMA operations on system memory. Moreover, these NICs can better utilize their cache space, with improved prefetching, higher concurrency, and better cache management [24]. Such optimizations allow a modern RDMA NIC to deliver competitive throughput even when there are virtually no cache hits on the NIC.

More and improved processing units. Modern NICs are equipped with increasingly powerful Processing Units (PUs). This allows NICs to issue more requests in parallel, which in turn increases throughput and hides PCIe latency to fetch data on cache misses [24]. This challenges the need for various aggregation techniques and data layout optimizations used previously. For a sufficient number of active QP connections (each mapped to a single PU), a CX5 RoCE delivers close to 40 million reads per second (no contention).

The NIC's ability to support more active connections allows allocating exclusive connected QPs to threads, reducing QP sharing overheads (lock contention) and enabling one-sided operations.

Physical segment support. CX4 and CX5 support physical segments with bound checks. While LITE on CX3 requires kernel involvement for protection, we can now use physical segments from user space. This mechanism bypasses virtual-to-physical translation and reduces the MPT and MTT sizes. This is important for hosts with large persistent-memory systems with tens of TBs to a PB of memory [29, 40]. In such systems, even 1GB pages could lead to large MTTs (e.g., 100TB would require close to 1MB of MTT with 1GB pages). Physical segments support arbitrarily large memory regions with just one MPT entry and no MTTs.

Efficient transport protocols. QPs in RC consume 375B per connection [14], and RC requires many connections, which can overwhelm the NIC caches. Modern NICs provide a new Infiniband transport called Dynamically Connected Transport (DCT) [1], which can share a QP connection across multiple hosts, thereby reducing the amount of QP state. DC is not available for RoCE and suffers from frequent reconnects which diminish its purpose [16]. In this paper we focus on the RC transport. As we show, RC scales well on clusters with up to 64 hosts.

3.4 Revisiting prior work on improved RDMA

A key contribution of this paper is to show that on modern NICs, one-sided primitives can outperform alternatives for moderate cluster sizes (tens of machines), even when the NIC caches are being thrashed. For instance, on CX5 it takes on the order of at least 2500 to 3800 concurrent connections for `reads` performance to become as low as UD-based send/receive [14, 16]. We expect the break-even point to increase in the future through the improvements mentioned in §3.3, and argue that one-sided operations are best for building low-latency, high-throughput, and low-CPU utilization systems.

4 Design principles

We propose four design principles for RDMA-based in-memory rack-scale systems:

1. Leverage RC connections. As we mentioned, RC has a scalability cost: it consumes more transport-level state than UD, which can lead to NIC cache thrashing. We show that new hardware—with larger caches, better cache management, and more processing units—changes the trade-off in favor of RC in rack-scale deployments. That is, the cost is more than offset by the many benefits of RC: (1) RC allows lightweight one-sided primitives (`read`/`write`) that have lower CPU utilization and achieve higher IOPS; (2) RC offloads retransmissions from the CPU to the NIC, and (3) RC offloads congestion control as well. In addition to one-sided operations, we show the benefit of RC connections for RPCs over UD, by using RDMA `write`.

2. Minimize RDMA region metadata. While new hardware addresses NIC cache state concerns for RC, another issue remains: cache state for MPTs and MTTs. To address this problem, we use two techniques. First, we minimize the number of registered RDMA memory regions by using a contiguous memory allocator (CMA) [10]. Such an allocator requests large chunks of memory from the kernel and manages small object allocations. Thus, we only register a small number of large chunks that we expand and shrink dynamically as the application allocates/deallocates memory, minimizing MPTs. The system could then use on-demand paging to repurpose unused pages (though current support is limited to 4KB pages). Dynamically expanding/shrinking a region can be achieved with little downtime by leveraging Linux CMA. Once the region is expanded/shrunk, it has to be re-registered for RDMA.

Second, to reduce the memory translation table metadata (MTTs), we propose using *physical segments*, a feature available in newer RDMA NICs such as CX5 [25]. Physical segments export physical memory with user-defined bounds with no MTT overhead, and this feature is available in user-space.⋆ Physical segments were intended for single-tenant use; using them in a host with many tenants requires care to avoid security issues when exposing physical memory. We propose a solution to these issues, by mediating the registration of physical segments by the kernel. This approach is secure and imposes minimum overhead since kernel calls are off the data path. Moreover, this approach is more efficient than using huge pages (2MB or 1GB) to reduce the MTTs [10]. Huge pages lead to fragmentation and waste memory [18, 34], and may not suffice: for large memories with 100s of TBs, even 1GB pages result in large MTTs. It is important to limit the number of physical segments, as they are allocated using Linux CMA [23], and it may not be able to efficiently handle multiple growing regions that need to be physically contiguous.

⋆Unlike LITE, which enforces protection in the kernel.

3. Try `reads` first, then switch to RPCs. One-sided `reads` deliver high IOPS for simple lookups [10, 26, 27, 46]. However, they are less efficient to access data structures with cells and pointers, such as skip lists, trees, and graphs, which require pointer-chasing. Thus, prior work proposes two alternatives: (1) use RPCs implemented with `send`/`recv` verbs [14, 16] or (2) fetch more data at a time [10], arranging cells accordingly.

With new hardware, we show that the best approach is as follows. First, use one-sided `reads` to fetch one cell at a time. Our evaluation shows that combining cells and fetching more data results in lower throughput. Second, if the one-sided `read` reveals that we must chase pointers, switch to using RPCs. We call this hybrid scheme *one-two-sided* operations. Furthermore, we show that RPCs can be implemented efficiently using one-sided `writes`.

4. Resize and/or cache. For remote reads to be effective, data structure operations should require one round trip in the common case. Otherwise, RPCs are proven to be more effective [16]. One round trip per operation is hard to achieve, especially with pointer-linked data structures. This work proposes a simple approach, which is to trade abundant memory for fewer round trips with one-sided operations. There are two ways to achieve this trade. First, clients could cache item addresses for future use, as in DrTM+H [43]. Second, for hash tables, when RPC usage becomes excessive due to collision induced pointer chasing, one should resize the data structure to keep the occupancy low. We claim that the amount of consumed memory is not significant, especially in the face of high-density persistent memory technologies. For the latter, we find that keeping the occupancy below 60-70% is sufficient to emphasize the performance benefits of one-sided reads. Nevertheless, we are looking into ways to repurpose the unused portions of allocated memory.

5 Design and Implementation of Storm

Following our principles (§4), we design and implement Storm, a fast RDMA dataplane for remote data structures. Storm is designed to run at maximum IOPS rate of the NIC by using RDMA primitives and by minimizing the active protocol state. Storm exposes the familiar transactional API to the user.

Figure 2 shows the high-level design of Storm. Two independent data paths process remote requests coming from the local process: RPCs and one-sided reads (RR). The event loop processes inbound requests and all event completions. The Storm TX module provides a transactional API to the user by leveraging the data structure API and the RPC/RR data paths to execute transactions using a two-phase commit protocol.

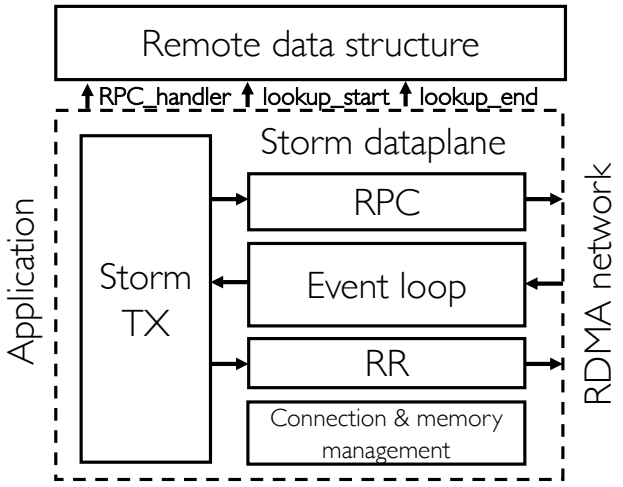

Figure 2. Storm high-level design. Independent pipelines for remote reads and RPCs. A single event loop processing all completions. Data structure completely independent of the data plane. The developer implements the data structure interface consisting of three callbacks.

5.1 Contiguous memory regions

To achieve best performance, we must manage memory efficiently in RDMA. Earlier, Figure 1 showed that large MTTs and MPTs leads to significant performance degradation, even on modern hardware. Thus, Storm aims to allocate virtually contiguous memory when possible to minimize the number of registered RDMA regions. By doing so, Storm minimizes the MPT state. In addition, Storm can allocate physically contiguous memory and expose it as one physical segment [25], requiring only a single MTT and one MPT entry. The physical segment support requires a trusted entity to perform memory registration, as this capability allows access to any part of the machine's physical memory. In Storm, we require from all applications to register physical segments through the OS. The overhead is negligible as registration is done off of the critical path. With sufficiently large pages, physical segments may not be necessary. Thus, in most of our experiments we do not use them. However, future storage-class memory systems with PB of memory will require support for physical segments.

5.2 Remote write-based RPCs

Storm leverages *RDMA write with immediate* operations to send and receive messages. This primitive allows the client to prepend a custom header to each message, which is useful for communicating additional information about the sender (e.g., process ID, coroutine ID, etc). More importantly, *writes with immediate* enables scalable polling on the receiver; the receiver receives a notification via a receive completion queue for each received message. In addition, the IB verbs interface also permits sharing a single completion queue across multiple senders. Thus, the receiver does not have to poll on multiple message buffers and multiple receive queues, improving scalability and throughput.

Algorithm 1 Processing a read-set item in Storm TX

1: **Input:** Data structure object ID, key, size
2: **Output:** Data item from remote memory
3: $success \leftarrow false$
4: $region\_id, offset, size \leftarrow lookup\_start(object\_id, key)$
5: **if** $region\_id \neq -1$ **then**
6: $\quad buffer \leftarrow remote\_read(region\_id, offset, size)$
7: $\quad success \leftarrow lookup\_end(buffer, object\_id, key)$
8: **if** $success \neq true$ **then**
9: $\quad buffer \leftarrow rpc\_send(object\_id, key, READ)$
10: $\quad success \leftarrow lookup\_end(buffer, object\_id, key)$

5.3 Storm remote data structure API

Storm exposes an intuitive and well-understood transactional API for manipulating remote data structures (Table 2). Clients add to the read/write sets and commit transactions at the end. Storm's event loop must be invoked periodically to process event completions and execute requests coming from the other nodes in the system.

Internally, Storm provides the following programming model for remote data structures: Developers implements three callback functions and register them with the Storm dataplane (Table 1). These functions are implemented as part of the remote data structure. *rpc_handler* is used for lookups on the owner (receiver) side. Locks and commits are also implemented in *rpc_handler*. *lookup_start* is the remote lookup handler for looking up a remote data structure's metadata on the client side. This metadata could be cached data structure addresses or simply a guess for an object's address based on a hash.

Algorithm 1 shows how the Storm dataplane on the client side processes each request from the read set. It first invokes *lookup_start* to get the RDMA region ID and offset where the requested item may reside. If successful, the client looks up the data at the returned address using a remote read.

When a lookup is finished, the client invokes *lookup_end* to validate the returned data. If the data is not valid (e.g., the read key does not match the requested key), the client issues an RPC. *lookup_end* may decide to cache the address of the returned object for future use. This depends on the remote data structure implementation. Current RDMA technology does not allow additional remote reads without hurting performance, but future faster interconnects may change this trade-off. Invoking *lookup_end* is necessary for lookups using remote reads, but it

Table 1. Storm remote data structure API

API	Description
rpc_handler	local RPC handler
lookup_start	get data item region ID and offset
lookup_end	check if successful and cache

Table 2. Storm TX API

API	Description
storm_eventloop	process requests and completions
storm_start_tx	start a new transaction
storm_add_to_read_set	add an item to read set
storm_add_to_write_set	add an item to write set
storm_tx_commit	commit a transaction
storm_register_handler	register a callback handler (Table 3)

is also invoked after every RPC lookup, so that the data structure can store the returned address for future use. *lookup_end* may return false even after the RPC call if, for example, the item does not exist.

5.4 Storm transactional protocol and API

Storm is capable of executing serializable transactions efficiently. It implements a typical variation of the two-phase commit protocol that is optimized for RDMA; throughout the execution phase, Storm copies objects to local memory and modifies them locally. Before committing, Storm client validates that no concurrent transaction has modified the read set. This is done using remote reads, as Storm keeps track of the remote offsets of each individual object in the read set. Finally, Storm uses write-based RPCs to update the objects from the write set and unlock them. Using RPCs for writes is a widely accepted approach in RDMA-based transactional systems, as it reduces implementation complexity [9] [43]. Storm uses optimistic concurrency control [17], but locks the objects that the transaction intends to write in the execution phase.

5.5 Example remote data structure: hash table

We use a hash table as a classical remote data structure example. We modified the MICA hash table [20] to accommodate for zero-copy transfers and extended it with handlers from Table 1. Zero-copy is achieved through inlining of the required metadata, including: key, lock and version. The *rpc_handler* is compatible with Storm transactions and implements lookups, lock acquisition, updates, inserts and deletes. To lookup remote items, the clients call into *lookup_start* to get the address based on the hash. The MICA hash table allows us to change buffer allocation and specify the bucket size, which we leverage to reduce hash collisions. Besides hash tables, Storm allows users to implement other data structures, such as queues and trees, and adjust the caching strategy accordingly.

5.6 Concurrency

Asynchronous scheduling of remote reads and RPCs is a difficult task. One could use callback continuations to pipeline multiple remote operations concurrently. While this approach has low overhead, prior work preferred using user-level threads (i.e., coroutines) [16, 43]. Storm leverages coroutines to provide concurrency within individual threads, while offering blocking semantics to the developers, reducing the complexity of building applications on top of the Storm TX API.

6 Evaluation

6.1 Methodology

We use InfiniBand EDR to evaluate key design benefits of Storm and point out the downsides of the previous proposals. We first briefly explain our experimental methodology.

RDMA test-bed. We deployed and evaluated Storm on a 32-node InfiniBand EDR (100Gbps) cluster. Each machine features a Mellanox ConnectX-4 NIC, which has similar performance characteristics to ConnectX-5. In addition, we have access to three pairs of servers, a pair for each of the three most recent ConnectX generations (CX3, CX4, CX5), all based on RoCE (Table 3). In addition to Storm, we also deploy and run eRPC and our emulated and improved version of FaRM. We were not able to deploy LITE on this cluster, as we were not allowed to patch the kernel. Instead, we ported and deployed LITE on our CX5(RoCE) servers and projected the results to our CX4(IB) platform.

Emulation. With Storm we are able to emulate RDMA clusters larger than 32 nodes. To achieve that, Storm allocates the same amount of resources that would exist in a real environment, including connections and registered RDMA buffers. For example, each thread maintains a connection to each of its "siblings" (i.e., threads with the same local ID) on the other servers. By varying the number of QP connections and the amount of message buffers used per pair of threads, we can accurately emulate clusters of 3-4x larger sizes. The maximum size is limited because of the amount of compute that is fixed.

Workloads. We use two workloads, described next.
• *Key-value lookups* uses Storm to look up random keys in the Storm distributed hash table. Each bucket has a configurable number of slots for data. Colliding items are kept in a linked list when the bucket capacity is exceeded. When the hash table is highly occupied, linked list traversals are needed to find individual keys. Each data transfer, including the application-level and RPC-level headers, is 128 bytes in size.
• *TATP* is a popular benchmark that simulates accesses to the Home Location Register database used by a mobile carrier; it is often used to compare the performance of in-memory

Table 3. Different evaluation platforms used in this work

Platform:	CPU/memory	RDMA network	Max. Machines
CX3 (RoCE)	Intel Xeon Gold 5120, 192GB DRAM	Mellanox ConnectX-3 Pro 40Gbps	2
CX4 (RoCE)		Mellanox ConnectX-4 VPI 100Gbps	
CX5 (RoCE)		Mellanox ConnectX-5 VPI 100Gbps	
CX4 (IB)	Intel Xeon E5-2660, 128GB DRAM	Mellanox ConnectX-4 IB EDR 100Gbps	32

transaction processing systems. TATP uses Storm transactions to commit its operations.

Baselines. We compare Storm to three different baseline systems: (i) eRPC, which is a system based on Unreliable Datagrams (UD); (ii) FaRM, a system that leverages the Hopscotch hashtable algorithm to minimize the number of round trips; and (iii) LITE, a kernel-based RDMA system that onloads the protection functionality to improve scalability. eRPC does not allow for one-sided reads and is an RPC-only system. It relies on UD, which is an unreliable InfiniBand transport requiring onloaded congestion control and retransmissions. We emulate FaRM by configuring Storm with FaRM parameters and by rewriting the hash table algorithm. Also, to provide a fair comparison, we do not share QPs using locks as our NICs scale better compared to the CX3, which have been used to evaluate FaRM. Finally, we improved LITE by extending it with support for asynchronous remote operations. Asynchronous operations are important for IOPS-bound applications, such as transactions.

6.2 Performance at rack-scale

We first evaluate Storm in isolation using the *Key-value lookups* workload. Then, we compare Storm to the previously proposed systems using the same workload, and finally we evaluate TATP running on Storm.

6.2.1 Key-value lookups

Figure 3 shows the performance for three different Storm setups: (i) *Storm* uses only RPCs to perform lookups. We observe that the throughput stabilizes with the node count; more nodes amortizes the polling overhead on the receiver. (ii) *Storm(oversub)* enforces lower collision rate by allocating a larger hash table. With 32 nodes the throughput is 1.7x higher compared to *Storm*. The throughput is not stable as we scale because the collision rate is not the same for different node counts, which translates to a higher or fewer number of reads followed by RPCs (*one-two-sided*), impacting throughput. Finally, (iii) *Storm (perfect)* assumes no RPCs on the data path. Using only remote reads in Storm is possible through a combination of memory oversubscription and caching of the addresses of pointer-linked items. At 32 nodes, *Storm (perfect)* outperforms *Storm* by 2.2x.

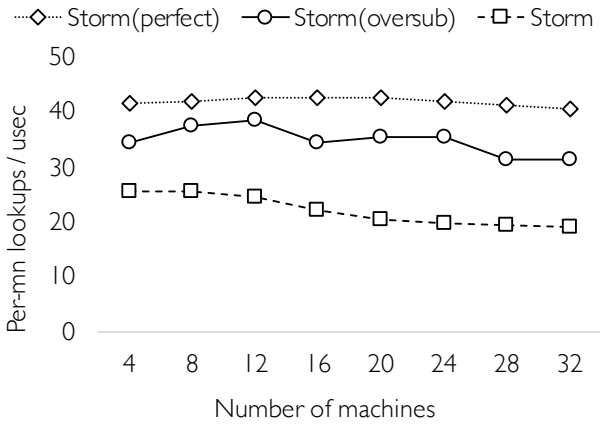

Figure 3. Comparison of Storm configurations for a read-only key-value workload. Average per-machine throughput on the Y-axis.

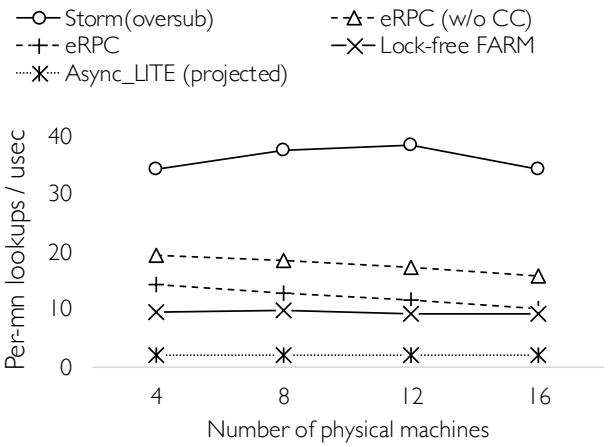

Figure 4. Comparison of Storm, eRPC, FaRM, and LITE. eRPC includes two versions, with and without congestion control.

6.2.2 Key-value lookups (comparison)

In this section we compare the performance of Storm, eRPC, FaRM, and LITE using the *Key-value lookups* workload. Figure 4 presents the performance of all the systems running on a real cluster with sizes varying from 4 to 16 machines. We were not able to deploy eRPC on more than 16 nodes (hence X-axis goes up to 16), as our NICs do not support sufficiently

large receive queues. eRPC relies on a large-enough number of registered receive buffers to prevent receiver-side packet loss. For Storm, we only plot *Storm(oversub)*. For eRPC, we study a version with and one without congestion control, whereas *Storm(oversub)* has hardware congestion control always enabled. For FaRM, we use our improved emulated version that does not require QP locks, unlike the original FaRM implementation [10]. We emulate FaRM by configuring Storm with the same parameters from the original FaRM paper [10]. A key difference is that we use 128B items, which increases the bucket size in FaRM and affects throughput. Finally, we use our improved version of LITE that enables asynchronous remote reads and RPCs (Async_LITE).

The key takeaways are: (1) Storm significantly outperforms previous systems. This gap is mainly due to Storm's ability to take advantage of fine-grain remote reads. (2) Even though eRPC does not use a reliable transport (no connections), the throughput decreases with node count due to the increasing overhead of posting onto the receive queue. This issue can be fixed using "strided" RQ, which unfortunately is not available on our infrastructure. Strided RQ enables posting a single RQ descriptor for a set of virtually contiguous buffers. The lack of this feature also limits us to 16 nodes. This limit holds only for eRPC and not for other evaluated systems. (3) eRPC with no congestion control performs 1.53x better at 16 nodes than eRPC with application-level congestion control enabled [14], indicating that relying on the implicit congestion control provided by RC rather than the custom congestion control at the application level may be beneficial. For larger message sizes, the overhead of software-managed congestion control may be less of an issue, as reported in eRPC (20% bandwidth degradation for 8MB message size). The overhead of onloaded congestion control will become more problematic with decreasing network latencies and increasingly higher IOPS rates (4) FaRM with its coarse-grained reads performs worse than eRPC, suggesting that trading larger network transfers (8x) per lookup for fewer network round trips comes with performance penalty. For items smaller than 128 bytes, FaRM achieves higher throughput, as this results in smaller bucket transfers. Finally, (5) LITE performs the worst due to the kernel complexity. We measured the throughput on two CX5 nodes only and projected these measurements to 16 nodes. LITE is compute-bound and does not suffer from NIC cache thrashing. Hence, we expect the throughput to be similar when running on clusters with CX4.

6.2.3 TATP performance

On Figure 5 we study TATP for two Storm configurations. Both configurations allocate the same amount of memory for the data. The configurations are as follows: (1) *Storm (oversub)* uses an oversized hash table with bucket width of one,

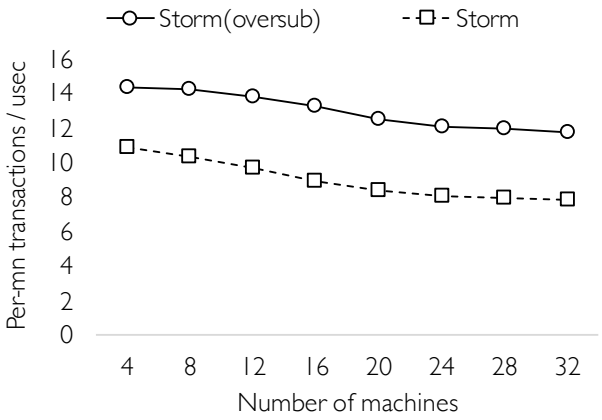

Figure 5. TATP running on Storm. Lower occupancy of TATP hash tables leads to better performance.

where each unsuccessful remote read lookup is followed by an RPC (*one-two-sided*) to traverse the overflow chain. To perform read-for-update and commit, Storm uses RPCs. The oversized hash table results in fewer collisions and the ability to successfully leverage remote reads most of the time; (2) *Storm* always uses RPC to execute all application requests, independent of the bucket size.

At 32 nodes, *Storm (oversub)* outperforms *Storm* by 1.49x. The TATP workload has 16% of writes and 4% of inserts and deletes. Writes, inserts and deletes require RPCs and thus the improvement is not as significant as in the *Key-value lookups* workload. Also, with increasing node count, the throughput trend is similar to that of *Storm* in the *Key-value lookups* workload, and this is because of a larger fraction of RPCs. Again, as we add nodes into the system, fewer cycles are wasted as the event loop becomes more efficient in processing inbound requests and managing the queues. Similar to eRPC, strided RQ could be used in Storm to minimize the overheads associated with managing the receive queue.

6.2.4 Impact on latency

Table 4 shows the unloaded round trip latencies of the evaluated systems on two of our CX4 platforms, InfiniBand and RoCE. RoCE is generally known to have slightly higher latency compared to InfiniBand. RPC latency for Storm and eRPC is similar; both are optimized zero-copy implementations. FaRM requires transferring eight times larger blocks, hence higher latency. LITE has the highest latency due to the kernel overheads.

6.2.5 Physical segments

With the advent of extremely dense persistent memory technologies, we anticipate that future servers will be hosting hundreds of TBs memory. For such large memory machines, the

Platform	Storm (RR)	Storm (RPC)	eRPC	FaRM	LITE
CX4 (IB)	1.8us	2.7us	2.7us	2.1us	5.8us
CX4 (RoCE)	2.8us	3.9us	3.6us	3us	6.4us

Table 4. Round-trip latencies for the various baselines and Storm.

RDMA region metadata can overwhelm the NIC caches, especially due to the MTT size. We added support for physical segments in Storm and enforce kernel-level segment registration for security reasons. We use 4KB page sizes and compare them to using Storm to export application memory as a physical segment. By using 4KB pages, we emulate a PB-scale storage class memory with 1GB page size. Using physical segments vs 4KB pages leads to 32% higher throughput.

6.3 Discussion: beyond rack-scale

In this section, we emulate larger clusters using our 32-node CX4(IB) cluster by creating additional connections and allocating additional buffers between each pair of machines [43]. Figure 6 shows the throughput as we scale the system from 32 to 128 virtual nodes. At 96 nodes and 20 threads per (physical) node, the throughput drops by 1.57x when the NIC cache is overwhelmed with state. Most of this state consists of connections, as we minimized the amount of MTT and MPT through larger (2MB) pages and contiguous memory allocation.

We observe the following: (i) Up to 64 nodes, the throughput is stable. 64 or fewer nodes is enough for most rack-scale deployments, which are most common. (ii) by reducing the number of threads to 10 per server, the throughput is stable even at 128 nodes. A smaller number of threads leads to fewer initiated connections, which minimizes the amount of transport-level state. If an application requires more than 10 threads per node, we envision a low-overhead, lock-free connection sharing mechanism, where RDMA is exposed to only half of the threads, which connect to their sibling threads on the other nodes as usual. In addition, each such thread executes RDMA requests on behalf of another thread on the same host. This forwarding of requests can be achieved by establishing a virtual channel (connection) between each pair of threads. Finally, we are looking into memory management techniques for Storm to reduce memory footprint.

7 Related Work

Other than the systems discussed in the previous sections, there is a large body of work on RDMA-based key-value and transaction processing systems [8, 13, 19, 26, 27, 39, 44], distributed lock management [30, 45], DSM systems [31], PM systems [5, 22, 37, 38, 41], and resource disaggregation [11, 33]. We discuss only a few below.

Other one-sided RDMA storage systems. Pilaf [26] uses a self-verifying data structure to detect races and enforce synchronization. This mechanism is directly applicable to Storm.

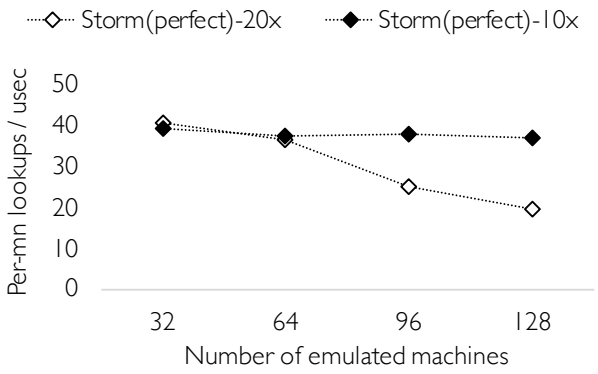

Figure 6. Emulation of larger clusters using a 32-node cluster. 128 emulated machines requires 4x more connections and RDMA buffers. Comparison of Storm(perfect) with 20 and 10 threads per machine.

NAM-DB [7, 46] leverages multi-versioning to minimize the overhead of running distributed transactions. Storm does not focus on optimizing the commit protocol and instead focuses on improving the datapath. Crail [38] is based on Java but provides competitive performance by cutting through the Java stack (e.g., bypasses serialization). However, Crail is better suited to data processing systems, unlike Storm, which is optimized for fine-grain one-sided transfers.

Hybrid RDMA systems. RTX [43] provides key insights about the choice of RDMA primitive for each phase of a two-phase commit protocol using both UD and RC transports. Unlike RTX, Storm's focus is on scalability. Storm uses RC only and takes advantage of high-throughput one-sided primitives (even for RPC). RTX validates our conclusion that one-sided operations achieve significantly higher IOPS compared to UD-based RPC for messages larger than 64 bytes and still opts to use UD for RPC due to scalability concerns. In this work, contrary to common wisdom, we show this is not necessarily a concern. UD-based systems can achieve higher than usual transfer rates for smaller transfer sizes (below 64 bytes) because of inlining. For example, on our testbed eRPC achieves close to 30 million operations per second for 32-byte message size. Our workloads require 128-byte message size.

8 Conclusion

Our analysis of multiple generations of RDMA hardware shows that modern RDMA hardware scales well on rack-scale clusters. We leverage these hardware improvements in Storm, a high-performance and transactional RDMA dataplane using one-sided reads and write-based RPCs. Our detailed evaluation of Storm compares it to FaSST/eRPC and improved versions of FaRM and LITE, one-sided operations are effective for rack-scale systems.

References

[1] Openfabrics. dynamically connected transport. https://www.openfabrics.org/images/eventpresos/workshops2014/DevWorkshop/presos/Monday/pdf/05_DC_Verbs.pdf, 2014.

[2] Supplement to infiniband architecture specification volume 1 release 1.2.2 annex a17: Rocev2 (ip routable roce). https://cw.infinibandta.org/document/dl/7781, 2014.

[3] Amazon elastic fabric adapter (efa). https://lwn.net/Articles/773973/, 2018.

[4] Speculative Execution Exploit Performance Impact. https://access.redhat.com/articles/3307751, 2019.

[5] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, S. Novaković, A. Ramanathan, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, and M. Wei. Remote regions: a simple abstraction for remote memory. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, 2018.

[6] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of a Large-Scale Key-Value Store. In *Proceedings of the 2012 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 2012.

[7] C. Binnig, A. Crotty, A. Galakatos, T. Kraska, and E. Zamanian. The end of slow networks: It's time for a redesign. In *Proc. VLDB Endow.*, 2016.

[8] Y. Chen, X. Wei, J. Shi, R. Chen, and H. Chen. Fast and general distributed transactions using rdma and htm. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys)*, 2016.

[9] A. Dragojevic, D. Narayanan, and M. Castro. RDMA Reads: To Use or Not to Use? In *IEEE Data Eng. Bull.*, 2017.

[10] A. Dragojevic, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast Remote Memory. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[11] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[12] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn. RDMA over Commodity Ethernet at Scale. In *Proceedings of the ACM SIGCOMM 2016 Conference*, 2016.

[13] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. W. ur Rahman, N. S. Islam, X. Ouyang, H. Wang, S. Sur, and D. K. Panda. Memcached design on high performance rdma capable interconnects. In *Proceedings of the 2011 International Conference on Parallel Processing (ICPP)*, 2011.

[14] A. Kalia, M. Kaminsky, and D. Andersen. Datacenter rpcs can be general and fast. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[15] A. Kalia, M. Kaminsky, and D. G. Andersen. Design guidelines for high performance rdma systems. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference (ATC)*, 2016.

[16] A. Kalia, M. Kaminsky, and D. G. Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *Proceedings of the 12th Symposium on Operating System Design and Implementation (OSDI)*, 2016.

[17] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. In *Proceedings of the ACM Transactions on Database Systems.*, 1981.

[18] Y. Kwon, H. Yu, S. Peter, C. J. Rossbach, and E. Witchel. Coordinated and efficient huge page management with ingens. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[19] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, 2017.

[20] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[21] Y. Lu, G. Chen, B. Li, K. Tan, Y. Xiong, P. Cheng, J. Zhang, E. Chen, and T. Moscibroda. Multi-path transport for RDMA in datacenters. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[22] Y. Lu, J. Shu, Y. Chen, and T. Li. Octopus: an rdma-enabled distributed persistent memory file system. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, 2017.

[23] LWN. A deep dive into CMA. https://lwn.net/Articles/486301/, 2012.

[24] Mellanox. Personal communication. 2018.

[25] Mellanox. Physical Address Memory Region. https://community.mellanox.com/docs/DOC-2480, 2019.

[26] C. Mitchell, Y. Geng, and J. Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *USENIX Conference on Annual Technical Conference (ATC)*, 2013.

[27] C. Mitchell, K. Montgomery, L. Nelson, S. Sen, and J. Li. Balancing CPU and network in the cell distributed b-tree store. In *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)*, 2016.

[28] R. Mittal, V. T. Lam, N. Dukkipati, E. R. Blem, H. M. G. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, and D. Zats. TIMELY: RTT-based Congestion Control for the Datacenter. In *Proceedings of the ACM SIGCOMM 2015 Conference*, 2015.

[29] T. P. Morgan. Intel shows off 3D XPoint memory performance. https://searchstorage.techtarget.com/definition/3D-XPoint, 2017.

[30] S. Narravula, A. Marnidala, A. Vishnu, K. Vaidyanathan, and D. K. Panda. High performance distributed lock management services using network-based remote atomic operations. In *Proceedings of the 7th IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, 2007.

[31] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Latency-tolerant software distributed shared memory. In *Proceedings of the 2015 USENIX Annual Technical Conference (ATC)*, 2015.

[32] R. Neugebauer, G. Antichi, J. Zazo, Y. Audvevich, S. López-Buedo, and A. Moore. Understanding pcie performance for end host networking. In *Proceedings of the ACM SIGCOMM 2018 Conference*, 2018.

[33] V. Nitu, B. Teabe, A. Tchana, C. Isci, and D. Hagimont. Welcome to zombieland: Practical and energy-efficient memory disaggregation in a datacenter. In *Proceedings of the 13th EuroSys Conference (EuroSys)*, 2018.

[34] A. Panwar, A. Prasad, and K. Gopinath. Making huge pages actually useful. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[35] G. F. Pfister. An introduction to the infiniband architecture. *High Performance Mass Storage and Parallel I/O*, 2001.

[36] R. Recio, B. Metzler, P. Culley, J. Hilland, and D. Garcia. A remote direct memory access protocol specification (rfc 5040). 2007.

[37] Y. Shan, S.-Y. Tsai, and Y. Zhang. Distributed shared persistent memory. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC)*, 2017.

[38] P. Stuedi, A. Trivedi, J. Pfefferle, R. Stoica, B. Metzler, N. Ioannou, and I. Koltsidas. Crail: A high-performance i/o architecture for distributed data processing. In *IEEE Data Eng. Bull.*, 2017.

[39] M. Su, M. Zhang, K. Chen, Z. Guo, and Y. Wu. Rfp: When rpc is faster than server-bypass with rdma. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*, 2017.

[40] M. M. Swift. Towards o(1) memory. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS)*, 2017.

[41] A. Tavakkol, A. Kolli, S. Novakovic, K. Razavi, J. Gomez-Luna, H. Hassan, C. Barthels, Y. Wang, M. Sadrosadati, S. Ghose, et al. Enabling efficient rdma-based synchronous mirroring of persistent memory transactions. In *arXiv preprint arXiv:1810.09360*, 2018.

[42] S.-Y. Tsai and Y. Zhang. LITE Kernel RDMA Support for Datacenter Applications. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[43] X. Wei, Z. Dong, R. Chen, and H. Chen. Deconstructing rdma-enabled distributed transactions: Hybrid is better! In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

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

[45] D. Y. Yoon, M. Chowdhury, and B. Mozafari. Distributed lock management with rdma: Decentralization without starvation. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD)*, 2018.

[46] E. Zamanian, C. Binnig, T. Harris, and T. Kraska. The end of a myth: Distributed transactions can scale. In *Proc. VLDB Endow.*, 2017.

[47] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang. Congestion Control for Large-Scale RDMA Deployments. In *Proceedings of the ACM SIGCOMM 2015 Conference*, 2015.