FaSST: Fast, Scalable and Simple Distributed Transactions with Two-sided (RDMA) Datagram RPCs

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

Abstract

FaSST is an RDMA-based system that provides distributed in-memory transactions with serializability and durability. Existing RDMA-based transaction processing systems use one-sided RDMA primitives for their ability to bypass the remote CPU. This design choice brings several drawbacks. First, the limited flexibility of one-sided RDMA reduces performance and increases software complexity when designing distributed data stores. Second, deep-rooted technical limitations of RDMA hardware limit scalability in large clusters. FaSST eschews one-sided RDMA for fast RPCs using two-sided unreliable datagrams, which we show drop packets extremely rarely on modern RDMA networks. This approach provides better performance, scalability, and simplicity, without requiring expensive reliability mechanisms in software. In comparison with published numbers, FaSST outperforms FaRM on the TATP benchmark by almost 2x while using close to half the hardware resources, and it outperforms DrTM+R on the SmallBank benchmark by around 1.7x without making data locality assumptions.

1 Introduction

"Remote procedure calls (RPC) appear to be a useful paradigm."
— Birrel & Nelson, 1984

Serializable distributed transactions provide a powerful programming abstraction for designing distributed systems such as object stores and on-line transaction processing (OLTP) systems. Although earlier work in this space sacrificed strong transactional semantics for performance [9], recent systems have shown that transactions can be fast in the datacenter [12, 28, 7, 5].[1] The key enablers are high-speed networks and lightweight network stacks (i.e., kernel bypass). In addition, these systems exploit Remote Direct Memory Access (RDMA) for its low latency and CPU efficiency. A common thread in these systems is that they make extensive use of one-sided RDMA operations that bypass the remote CPU. The intent behind this decision is to harness one-sided RDMA's ability to save remote CPU cycles.

[1]We discuss only distributed transactions in this paper, so we use the more general but shorter term transactions.

In this paper, we explore whether one-sided RDMA is not the best choice for designing transaction processing systems. First, there is a gap between the paradigm of one-sided RDMA, and capabilities needed for efficient transactional access to remote data stores. One-sided RDMA provides only remote reads, writes, and atomic operations, whereas accessing data stores typically involves traversing data structures such as hash tables and B-Trees. In general, these structures consist of an index for fast lookup, and the actual data, requiring two or more RDMA reads to access data. This leads to lower throughput and higher latency, and reduces the net CPU savings from remote CPU bypass [15]. The key technique to overcome this gap is to *flatten* the data structure, by either ignoring the index [5], merging the data with the index [12], or caching the index [12, 28, 7] at all servers. Each of these variants has an associated cost in generality and system performance. Second, the connection-oriented nature of current one-sided RDMA implementations typically requires CPU cores to share local NIC queue pairs for scalability [11], reducing local per-core RDMA throughput by several factors, and the net benefit of remote CPU bypass.

We show that there is a better primitive for transactions: remote procedure calls (RPCs) over two-sided unreliable datagram messages. RPCs involve the remote CPU in message processing and are more flexible than one-sided RDMA, allowing data access in a single round trip [15]. However, previous RPC implementations over RDMA either performed poorly (e.g., up to 4x worse than one-sided RDMA in FaRM [12]), or were specialized (e.g., HERD's RPCs [15] deliver high performance for all-to-one communication where one server handles RPCs from many clients). A key contribution of this work is FaSST RPCs: an all-to-all RPC system that is fast, scalable, and CPU-efficient. This is made possible by using RDMA's datagram transport that provides scalability, and allows "Doorbell batching" which saves CPU cycles by reducing CPU-initiated PCIe bus transactions. We show that FaSST RPCs provide (1) up to 8x higher throughput, and 13.9x higher CPU efficiency than FaRM's RPCs (Section 4.5), and (2) 1.7–2.15x higher CPU efficiency, or higher throughput, than one-sided READs, depending on whether or not the READs scale to clusters with more than a few tens of nodes (Section 3.3).

Using an unreliable transport layer requires handling packet loss. In RDMA networks such as InfiniBand, however, packet loss is extremely rare because the underlying link layer provides reliability. We did not observe any lost packets in our experiments that transmitted over 50 PB of network data on a real-world InfiniBand cluster with up to 69 nodes. Nevertheless, packet loss can occur during hardware failures, and corner cases of the link-layer's reliability protocol. We detect these losses using coarse-grained timeouts triggered at the RPC requester, and describe how they can be handled similarly to conventional machine failures.

FaSST is a new transaction processing system built on FaSST RPCs. It uses optimistic concurrency control, two-phase commit, and primary-backup replication. Our current implementation supports transactions on an unordered key-value store based on MICA [18], and maps 8-byte keys to opaque objects. We evaluate FaSST using three workloads: a transactional object store, a read-mostly OLTP benchmark called TATP, and a write-intensive OLTP benchmark called SmallBank. FaSST compares favorably against published per-machine throughput numbers. On TATP, FaSST outperforms FaRM [12] by 1.87x when using close to half the hardware (NIC and CPU) resources. On SmallBank, FaSST outperforms DrTM+R [7] by 1.68x with similar hardware without making data locality assumptions. The source code for FaSST and the experiments in this paper is available at `https://github.com/efficient/fasst`.

2 Background

2.1 Fast distributed transactions

This section outlines the environment that we target with FaSST. FaSST aims to provide distributed transactions inside a single datacenter where a single instance of the system can scale to a few hundred nodes. Each node in the system is responsible for a partition of the data based on a primary key, and nodes operate in the *symmetric model*, whereby each node acts both as a client and a server. For workloads with good data locality (e.g., transactions that only access data in one partition), the symmetric model can achieve higher performance by co-locating transactions with the data they access [11, 12].

FaSST targets high-speed, low-latency key-value transaction processing with throughputs of several million transactions/sec and average latencies around one hundred microseconds on common OLTP benchmarks with short transactions with up to a few tens of keys. Achieving this performance requires in-memory transaction processing, and fast userspace network I/O with polling (i.e., the overhead of a kernel network stack or interrupts is unacceptable). We assume commercially available network equipment: 10-100 Gbps of per-port bandwidth and ≈ 2 µs end-to-end latency.

Making data durable across machine failures requires logging transactions to persistent storage, and quick recovery requires maintaining multiple replicas of the data store. Keeping persistent storage such as disk or SSDs on the critical path of transactions limits performance. Similar to recent work, FaSST assumes that the transaction processing nodes are equipped with battery-backed DRAM [12], though future NVRAM technologies, if fast enough, would also work.

Finally, FaSST uses primary-backup replication to achieve fault tolerance. We assume that failures will be handled using a separate fault-tolerant configuration manager that is off of the critical path (the Vertical Paxos model [17]), similar to recent work on RDMA-based distributed transactions [12, 7]. We do not currently implement such a configuration manager.

2.2 RDMA

RDMA is a networking concept of which several implementations exist. The Virtual Interface Architecture (VIA) is a popular model for user-level, zero-copy networking [13], and forms the basis of current commodity RDMA implementations such as InfiniBand, RoCE (RDMA over Converged Ethernet), and iWARP (internet Wide Area RDMA Protocol). VIA NICs provide user processes with virtual interfaces to the network. VIA is fundamentally connection-oriented: a connection must be established between a pair of virtual interfaces before they are allowed to communicate. This design decision was made by VIA architects to simplify VIA implementations and reduce latency [13]. The discussion in this paper, and some of our contributions are specific to VIA-based RDMA implementations; we discuss other, non-commodity RDMA implementations in Section 7.1.

In VIA-based RDMA implementations, virtual interfaces are called queue pairs (QPs), each consisting of a send queue and a receive queue. Processes access QPs by posting *verbs* to these queues. Two-sided verbs—SEND and RECV—require involvement of the CPU at both the sender and receiver: a SEND generates a message whose data is written to a buffer specified by the receiver in a pre-posted RECV. One-sided verbs—READ, WRITE, and ATOMIC—bypass the remote CPU to operate directly on remote memory.

RDMA transports can be connected or connectionless. Connected transports offer one-to-one communication between two queue pairs: to communicate with N remote machines, a thread must create N QPs. These transports provide one-sided RDMA and end-to-end reliability, but do not scale well to large clusters. This is because NICs have limited memory to cache QP state, and exceeding

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

Table 1: Verbs supported by each transport type. RC, UC, and UD stand for Reliable Connected, Unreliable Connected, and Unreliable Datagram, respectively.

Name	**Hardware**
CX3	Mellanox ConnectX-3 (1x 56 Gb/s InfiniBand ports), PCIe 3.0 x8, Intel® Xeon® E5-2450 CPU (8 cores, 2.1 GHz), 16 GB DRAM
CIB	Mellanox Connect-IB (2x 56 Gb/s InfiniBand ports), PCIe 3.0 x16, Intel® Xeon® E5-2683-v3 CPU (14 cores, 2 GHz), 192 GB DRAM

Table 2: Measurement clusters

the size of this state by using too many QPs causes cache thrashing [11]. Connectionless (datagram) transports are extensions to the connection-oriented VIA, and support fewer features than connected transports: they do not provide one-sided RDMA or end-to-end reliability. However, they allow a QP to communicate with multiple other QPs, and have better scalability than connected transports as only one QP is needed per thread.

RDMA transports can further be either reliable or unreliable, although current commodity NICs do not provide a reliable datagram transport. Reliable transports provide in-order delivery of messages and return an error in case of failure. Unreliable transports achieve higher performance by avoiding acknowledgment packets, but do not provide reliability guarantees or return an error on network failures. Modern high-speed networks, including Mellanox's InfiniBand and Intel's OmniPath, also provide reliability below the transport layer [3, 6]. Their link layer uses flow control to prevent congestion-based losses, and retransmissions to prevent bit error-based losses. InfiniBand's physical layer uses Forward Error Correction to fix most bit errors, which themselves are rare. For example, the bit error rate of the InfiniBand cables used in our clusters is less than 10^{-15}. Therefore even unreliable transports, which lack end-to-end reliability, lose packets extremely rarely: we did not lose any packets in around 50 PB of unreliable data transfer (Section 3.4). Note that link-layer flow control in these networks can cause congestion collapse in rare scenarios, leading to low throughput, but not dropped packets.

Current RDMA implementations provide three main transports: Reliable Connected (RC), Unreliable Connected (UC), and Unreliable Datagram (UD). Table 1 shows the subset of verbs supported by implementations of each transport. Not all transport layers provide all types of verbs, so choosing a verb means accepting the limitations of the available transports. Note that only connected transports provide one-sided verbs, limiting the scalability of designs that use these verbs.

3 Choosing networking primitives

We now describe the rationale behind our decision to build an RPC layer using two-sided datagram verbs. We show that RPCs are:

1. **Fast**: Although READs can outperform similarly-sized RPCs on small clusters, RPCs perform better when accounting for the amplification in size or number of READs required to access real data stores.
2. **Scalable**: Datagram RPC throughput and CPU use remains stable as the cluster size increases, whereas READ performance degrades because READs must use connected transport with today's NICs.
3. **Simple**: RPCs reduce the software complexity required to design distributed data stores and transactions compared to one-sided RDMA-based systems.

3.1 Advantage of RPCs

Recent work on designing distributed data stores over RDMA-capable networks has largely focused on how to use one-sided RDMA primitives. In these designs, clients access remote data structures in servers' memory using one or more READs, similar to how one would access data in local memory. Various optimizations help reduce the number of READs needed; we discuss two such optimizations and their limitations below.

Value-in-index: FaRM [11, 12] provides hash table access in ≈ 1 READ on average by using a specialized index that stores data adjacent to its index entry, allowing data to be READ with the index. However, doing so amplifies the size of the READ by a factor of 6–8x, reducing throughput [15]. This result highlights the importance of comparing the application-level capabilities of networking primitives: although micro-benchmarks suggest that READs can outperform similar-sized RPCs, READs require extra network traffic and/or round-trips due to their one-sided nature, tipping the scales in the other direction.

Caching the index: DrTM [28, 7] caches the index of its hash table at all servers in the cluster, allowing single-READ GETs; FaRM [12] uses a similar approach for its B-Tree. Although this approach works well when the workload has high locality or skew, it does not work in general because indexes can be large: Zhang et al. report that indexes occupy over 35% of memory for popular OLTP benchmarks in a single-node transaction processing system [30]; the percentage is similar in our implementation of distributed transaction processing benchmarks. In this case, caching even 10% of the index requires *each* machine to dedicate 3.5% of the *total cluster memory* *capacity* for the index, which is impossible if the cluster contains more than $100/3.5 \approx 29$ nodes. The Cell B-Tree [21] caches B-Tree nodes 4 levels above the leaf nodes to save memory and reduce churn, but requires multiple round trips (~ 4) when clients access the B-Tree using READs.

RPCs allow access to partitioned data stores with two messages: the request and the reply. They do not require message size amplification, multiple round trips, or caching. The simplicity of RPC-based programming reduces the software complexity required to take advantage of modern fast networks in transaction processing: to implement a partitioned, distributed data store, the user writes only short RPC handlers for a single-node data store. This approach eliminates the software complexity required for one-sided RDMA-based approaches [11, 21]. For example, in this paper, we use MICA's hash table design [18] for unordered key-value storage. We made only minor modifications to the MICA codebase to support distributed transactions. In the future, we plan to use Masstree [19] for ordered storage.

3.2 Advantage of datagram transport

Datagram transport allows each CPU core to create one datagram QP that can communicate with all remote cores. Since the number of QPs is relatively small (as many as the number of cores), providing each core exclusive access to QPs is possible without overflowing the NIC's cache. Providing exclusive access to QPs with connected transport, however, is not scalable: In a cluster with N machines and T threads per machine, doing so requires $N * T$ QPs at every machine, which may not fit in the NIC's queue pair cache. Threads can share QPs to reduce the QP memory footprint [11]. Sharing QPs reduces CPU efficiency because threads contend for locks, and the cache lines for QP buffers bounce between their CPU cores. The effect can be dramatic: in our experiments, QP sharing reduces the per-core throughput of one-sided READs by up to 5.4x (Section 3.3.2). Similarly, FaRM's RPCs that use one-sided WRITEs and QP sharing become CPU-bottlenecked at 5 million requests/sec (Mrps) per machine [12]. Our datagram-based RPCs, however, do not require QP sharing and achieve up to 40.9 Mrps/machine, and even then they are bottlenecked by the NIC, not CPU (Section 3.3.1).

In comparison with connected transports, datagram transport confers a second important advantage in addition to scalability: *Doorbell batching* reduces CPU use. We describe this feature in a simplified form here; a detailed discussion is available in our earlier paper [16]. User processes post operations to the NIC by writing to a per-QP Doorbell register on the NIC over the PCIe bus, specifying the number of new operations on that QP. This write is relatively expensive for the CPU because it requires flushing the write buffers, and using memory barriers for ordering. In transactional systems, however, applications can amortize this cost by issuing multiple RDMA work requests at a time. Examples include reading or validating multiple keys for multi-key transactions, or sending update messages to the replicas of a key. With a datagram QP, the process only needs to ring the Doorbell once per batch, regardless of the individual message destinations within the batch. With connected QPs, however, the process must ring multiple Doorbells—as many as the number of destinations appearing in the batch. Note that Doorbell batching does not coalesce packets at the RDMA layer (i.e., it does not put multiple application-level requests in a single RDMA packet); Doorbell batching also does not add latency because we do it opportunistically, i.e., we do not wait for a batch of messages to accumulate.

3.3 Performance considerations

Two recent projects study the relative performance of RPCs and one-sided RDMA. In the asymmetric setting where multiple clients send requests to one server, HERD shows that RPCs perform similarly to READs [15]. In HERD, clients send requests to the server via WRITEs over UC; the server responds with SENDs over UD. This approach scales well with the number of clients because the number of *active* queue pairs at the server is small. The server's UC QPs are passive because the server's CPU does not access them; these passive QPs consume little memory in the NIC. The active UD QPs are few in number.

Unfortunately, as noted by Dragojevic et al. [12], HERD's RPC design does not scale well in the symmetric setting required for distributed transactions, where every machine issues requests and responses. This scenario requires many active UC QPs on each node for sending requests. In FaRM's experiments [12] in the symmetric setting, READs outperform *their* RPCs by 4x.

We now present experimental results showing that FaSST's RPCs are a better choice than one-sided RDMA for distributed transactions. The design and implementation of our RPC system is discussed in detail in Section 4; here, we use it to implement basic RPCs where both the request and reply are fixed-size buffers. We first compare the raw throughput of RPCs and one-sided READs by using a small cluster where READs do not require QP sharing. Next, we compare their performance on more realistic, medium-sized clusters.

Clusters used: To show that our results generalize to a range of RDMA hardware, we use two clusters with different NICs and CPU processing power (Table 2). The clusters are named after the initials of their NIC. CX3 is a shared Emulab [29] cluster with 192 nodes; our experiments used up to 69 nodes, depending on the availability of nodes. CX3 nodes have a ConnectX-3 NIC and an

Intel SandyBridge CPU with 8 cores. CIB is a private cluster with 11 nodes. CIB nodes have a more powerful Connect-IB NIC that provides 2x more bandwidth and around 4x higher message rate than a ConnectX-3 NIC. They also have a more powerful, 14-core Intel Haswell CPU.

Experiment setup: We use a cluster of machines in a symmetric setting, i.e., every machine issues requests (RPC requests or READs) to every other machine. For READs without QP sharing, each thread creates as many RC QPs as the number of machines, and issues READs to randomly chosen machines. We evaluate RPC performance for two request batch sizes (1 and 11) to show the effect of Doorbell batching for requests. We prevent RPC request coalescing (Section 4) by sending each request in a batch to a different machine; this restricts our maximum batch size on CIB to 11.

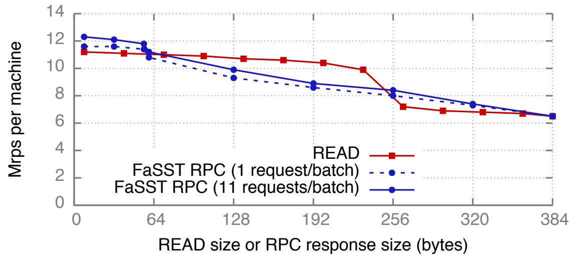
We compare RPC and READ performance for different response sizes; for RPCs, the request size is fixed at 32 bytes, which is sufficient to read from FaSST's data stores. We report millions of requests per second per machine (Mrps/machine). Note that for RPCs, each machine's CPU also serves responses to requests from other machines, so the number of messages sent by a machine is approximately twice the request rate that we report. Our results show that:

1. FaSST RPCs provide good raw throughput. For small messages up to 56 bytes, RPCs deliver a significant percentage of the maximum throughput of similar-sized READs on small clusters: 103–106% on CX3 and 68–80% on CIB, depending on the request batch size. When accounting for the amplification in READ size or number required to access data structures in real data stores, RPCs deliver higher raw throughput than READs.
2. On medium-sized clusters, if READs do not share QPs, RPCs provide 1.38x and 10.1x higher throughput on CIB and CX3, respectively. If READs do share QPs, their CPU efficiency drops by up to 5.4x, and RPCs provide 1.7–2.15x higher CPU efficiency.

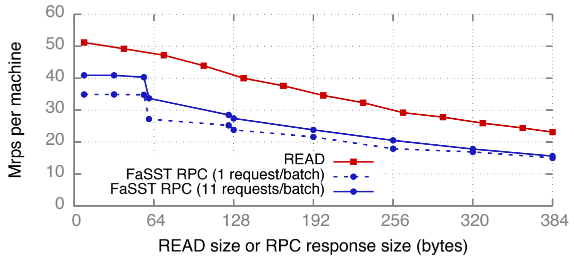
These experiments highlight the sometimes dramatic difference in performance between micro-benchmarks and more realistic settings.

3.3.1 On small clusters

To measure the maximum raw throughput of READs, we use 6 nodes so that only a small number of QPs are needed even for READs: each node on CX3 (8 cores) and CIB (14 cores) uses 48 and 84 QPs, respectively. We use 11 nodes for RPCs to measure performance with a large request batch size—using only 6 nodes for RPCs would restrict the maximum non-coalesced request batch size to 6. (As shown in Section 3.3.2, using 11 nodes for READs gives lower throughput due to cache misses in the NIC, so we use fewer nodes to measure their peak throughput.) Figure 1 shows Mrps/machine for READs and RPCs on the two clusters.



(a) CX3 cluster (ConnectX-3 NIC)



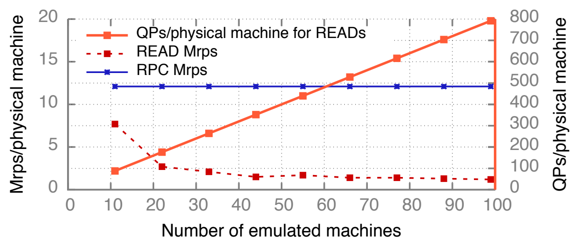
(b) CIB cluster (Connect-IB NIC)

Figure 1: Small clusters: Throughput comparison of FaSST RPCs (11 nodes) and READs (6 nodes). Note that the two graphs use a different Y scale.

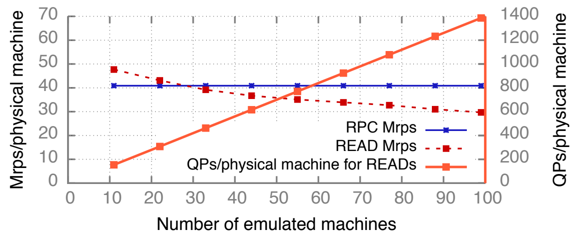
Raw throughput: Depending on the request batch size, FaSST RPCs deliver up to 11.6–12.3 Mrps on CX3, and 34.9–40.9 Mrps on CIB. READs deliver up to 11.2 Mrps on CX3, and 51.2 Mrps on CIB. The throughput of both RPCs and READs is bottlenecked by the NIC: although our experiment used all cores on both clusters, fewer cores can achieve similar throughput, indicating that the CPU is not the bottleneck.

Comparison with READs: Although RPCs usually deliver lower throughput than READs, the difference is small. For response sizes up to 56 bytes, which are common in OLTP, RPC throughput is within 103–106% of READ throughput on CX3, and 68–80% of READ throughput on CIB, depending on the request batch size. For larger responses, READs usually outperform our RPCs, but the difference is smaller than 4x, as is the case for FaRM's one-sided RPCs. This is because FaSST's RPCs are bottlenecked by the NIC on both clusters, whereas FaRM's RPCs become CPU-bottlenecked due to QP sharing (Section 3.3.2). As noted above, these "raw" results are only baseline micro-benchmarks; the following paragraphs consider the numbers in the context of "real-world" settings.

Effect of multiple READs: In all cases (i.e., regardless of cluster used, response size, and request batch size), RPCs provide higher throughput than using 2 READs.



(a) CX3 cluster (ConnectX-3 NIC)



(b) CIB cluster (Connect-IB NIC)

Figure 2: Comparison of FaSST RPC and READ throughput, and the number of QPs used for READs with increasing *emulated* cluster size.

Thus, for any data store/data structure that requires two or more READs, RPCs provide strictly higher throughput.

Effect of larger READs: Consider, for example, a hash table that maps 8-byte keys to 40-byte values (this configuration is used in one of the database tables in the TATP benchmark in Section 6) on CX3. For this hash table, FaRM's single-READ GETs require approximately 384-byte READs (8x amplification) and can achieve up to 6.5 Mrps/machine on CX3. With FaSST RPCs, these key-value requests can be handled in one RPC with an 8-byte request and a 40-byte response (excluding header overheads), and can achieve 11.4–11.8 Mrps/machine (over 75% higher) before the ConnectX-3 NIC becomes the bottleneck. On CIB, 384-byte READs achieve 23.1 Mrps, whereas FaSST RPCs achieve 34.9–40.9 Mrps (over 51% higher).

3.3.2 On medium-sized clusters

Measuring the impact of one-sided RDMA's poor scalability requires more nodes. As the CIB cluster has only 11 physical machines, we emulate the effect of a larger cluster by creating as many QPs on each machine as would be used in the larger cluster. With N physical nodes, we emulate clusters of $N * M$ nodes for different values of M. Instead of creating N QPs, each worker thread creates $N * M$ QPs, and connects them to QPs on other nodes. Note that we only do so for READs because for FaSST's RPCs, the number of local QPs does not depend on the number of machines in the cluster.

Figure 2 compares READ and RPC throughput for increasing emulated cluster sizes. We use 32-byte READs and RPC requests and responses. Note that the peak READ throughput in this graph is lower than Figure 1 that used 6 nodes. This is because NIC cache misses occur with as few as 11 nodes. On CX3, READ throughput drops to 24% of its peak with as few as 22 emulated nodes. On CIB, READs lose their throughput advantage over RPCs on clusters with 33 or more nodes. The decline with Connect-IB NICs is more gradual than with ConnectX-3 NICs. This may be due to a larger cache or better cache miss pipelining [10] in the Connect-IB NIC. Section 7.2 discusses the possible impact of future NIC and CPU hardware on queue pair scalability.

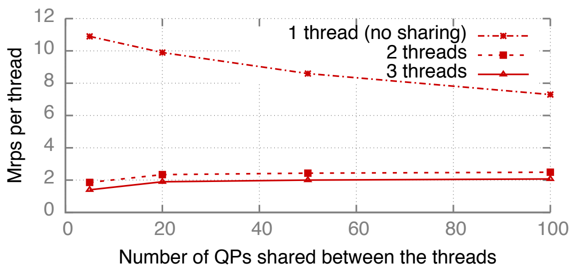


Figure 3: Per-thread READ throughput with QP sharing (CIB)

Sharing QPs: Fewer QPs are required if they are shared between worker threads, but doing so drastically reduces the CPU efficiency of one-sided RDMA. QP sharing is typically implemented by creating several sets of N QPs, where each set is connected to the N machines [11]. A machine's threads are also grouped into sets, and threads in a set share a QP set.

We measure the loss in CPU efficiency as follows. We use one server machine that creates a tuneable number of QPs and connects them to QPs spread across 5 client machines (this is large enough to prevent the clients from becoming a bottleneck). We run a tuneable number of worker threads on the server that share these QPs, issuing READs on QPs chosen uniformly at random.

We choose the number of QPs and threads per set based on a large hypothetical cluster with 100 nodes and CIB's CPUs and NICs. A Connect-IB NIC supports $\approx$ 400 QPs before READ throughput drops below RPC throughput (Figure 2). In this 100-node cluster, the 400 QPs are used to create 4 sets of 100 connections (QPs) to remote machines. CIB's CPUs have 14 cores, so sets of 3–4 threads share a QP set.

Figure 3 shows per-thread throughput in this experiment. For brevity, we only show results on CIB; the loss in CPU efficiency is comparable on CX3. The hypothetical configuration above requires sharing 100 QPs among at least 3 threads; we also show other configurations that may be relevant for other NICs and cluster sizes. With one thread, there is no sharing of QPs and throughput is high—up to 10.9 Mrps. Throughput with QP sharing between 3 threads, however, is 5.4x lower (2 Mrps).

This observation leads to an important question: If the increase in CPU utilization at the local CPU due to QP sharing is accounted for, do one-sided READs use fewer *cluster-wide* CPU cycles than FaSST's RPCs that do not require QP sharing? We show in Section 4 that the answer is no. FaSST's RPCs provide 3.4–4.3 Mrps per core on CIB—1.7–2.15x higher than READs with QP sharing between 3 threads. Note that in our symmetric setting, each core runs both client and server code. Therefore, READs use cluster CPU cycles at only the client, whereas RPCs use them at both the client and the server. However, RPCs consume fewer *overall* CPU cycles.

3.4 Reliability considerations

Unreliable transports do not provide reliable packet delivery, which can introduce programming complexity and/or have performance implications (e.g., increased CPU use), since reliability mechanisms such as timeouts and retransmissions must be implemented in the software RPC layer or application.

To understand FaSST's approach to handling potential packet loss, we make two observations. First, we note that transaction processing systems usually include a reconfiguration mechanism to handle node failures. Reconfiguration includes optionally pausing ongoing transactions, informing nodes of the new cluster membership, replaying transaction logs, and re-replicating lost data [12]. In FaSST, we assume a standard reconfiguration mechanism; we have not implemented such a mechanism because this paper's contribution is not in that space. We expect that, similar to DrTM+R [7], FaRM's recovery protocol [12] can be adapted to FaSST.

The second observation is that in normal operation, packet loss in modern RDMA-capable networks is extremely rare: in our experiments (discussed below), we observed zero packet loss in over 50 PB of data transferred. Packets can be lost during network hardware failures, and corner cases of the link/physical layer reliability protocols. FaSST's RPC layer detects these losses using coarse-grained timeouts maintained by the RPC requester (Section 4.3).

Based on these two observations, we believe that an acceptable first solution for handling packet loss in FaSST is to simply restart one of the two FaSST processes that is affected by the lost RPC packet, allowing the reconfiguration mechanism to make the commit decision for the affected transaction. We discuss this in more detail in Section 5.1.

3.4.1 Stress tests for packet loss

Restarting a process on packet loss requires packet losses to be extremely rare. To quantify packet loss on realistic RDMA networks, we set up an experiment on the CX3 cluster, which is similar to real-world clusters with multiple switches, oversubscription, and sharing. It is a shared cluster with 192 machines arranged in a tree topology with seven leaf and two spine switches, with an oversubscription ratio of 3.5. The network is shared by Emulab users. Our largest experiment used 69 machines connected to five leaf switches.

Threads on these machines use UD transport to exchange 256-byte RPCs. We used 256-byte messages to achieve both high network utilization and message rate. Threads send 16 requests to remote threads chosen uniformly at random, and wait for all responses to arrive before starting the next batch. A thread stops making progress if a request or reply packet is lost. Threads routinely output their progress messages to a log file; we manually inspect these files to ensure that all threads are making progress.

We ran the experiment without a packet loss for approximately 46 hours. (We stopped the experiment when a log file exhausted a node's disk capacity.) The experiment generated around 100 trillion RPC packets and 33.2 PB of network data. Including other smaller-scale experiments with 20–22 nodes, we have transferred over 50 PB of network data without a lost packet.

While we observed zero packet loss, we detected several reordered packets. Using sequence numbers embedded in the RPC packets, we observed around 1500 reordered packets in 100 trillion packets transferred. Reordering happens due to multi-path in CX3: although there is usually a single deterministic path between each source and destination node, the InfiniBand subnet manager sometimes reconfigures the switch routing tables to use different paths.

4 FaSST RPCs

FaSST's RPCs are designed for transaction workloads that use small (~ 100 byte) objects and a few tens of keys. This layer abstracts away details of RDMA, and is used by higher-layer systems such as our transaction processing system. Key features of FaSST's RPCs include integration with coroutines for efficient network latency hiding, and optimizations such as Doorbell batching and message coalescing.

4.1 Coroutines

RDMA network latency is on the order of 10 µs under load, which is much higher than the time spent by our applications in computation and local data store accesses. It is critical to not block a thread while waiting for an RPC reply. Similar to Grappa [22], FaSST uses coroutines (cooperative multitasking) to hide network latency: a coroutine yields after initiating network I/O, allowing other coroutines to do useful work while the RPC is in flight. Our experiments showed that a small number (~ 20) of coroutines per thread is sufficient for latency hiding, so FaSST uses standard coroutines from the Boost C++ library instead of Grappa's coroutines, which are optimized for use cases with thousands of coroutines. We measured the CPU overhead to switch between coroutines to be 13–20 ns.

In FaSST, each thread creates one RPC endpoint that is shared by the coroutines spawned by the thread. One coroutine serves as the master; the remaining are workers. Worker coroutines only run application logic and issue RPC requests to remote machines, where they are processed by the master coroutine of the thread handling the request. The master coroutine polls the network to identify any newly-arrived request or response packets. The master computes and sends responses for request packets. It buffers response packets received for each worker until all needed responses are available, at which time it invokes the worker.

4.2 RPC interface and optimizations

A worker coroutine operates on batches of $b \geq 1$ requests, based on what the application logic allows. The worker begins by first creating new requests without performing network I/O. For each request, it specifies the request type (e.g., access a particular database table, transaction logging, etc.), and the ID of destination machine. After creating a batch of requests, the worker invokes an RPC function to send the request messages. Note that an RPC request specifies the destination machine, not the destination thread; FaSST chooses the destination thread as the local thread's ID–based peer on the destination machine. Restricting RPC communication to between thread peers improves FaSST's scalability by reducing the number of coroutines that can send requests to a thread (Section 4.4).

Request batching. Operating on batches of requests has several advantages. First, it reduces the number of NIC Doorbells the CPU must ring from b to 1, saving CPU cycles. Second, it allows the RPC layer to coalesce messages sent to the same destination machine. This is particularly useful for multi-key transactions that access multiple tables with same primary key, e.g., in the SmallBank benchmark (Section 6). Since our transaction layer partitions tables by a hash of the primary key, the table access requests are sent in the same packet. Third, batching reduces coroutine switching overhead: the master yields to a worker only after receiving responses for all b requests, reducing switching overhead by a factor of b.

Response batching: Similar to request batching, FaSST also uses batching for responses. When the master coroutine polls the NIC for new packets, it typically receives more than one packet. On receiving a batch of B request packets, it invokes the request handler for each request, and assembles a batch of B response packets. These responses are sent using one Doorbell. Note that the master does not wait for a batch of packets to accumulate before sending responses to avoid adding latency.

Cheap RECV posting: FaSST's RPCs use two-sided verbs, requiring RECVs to be posted on the RECV queue before an incoming SEND arrives. On our InfiniBand hardware, posting RECVs requires creating descriptors in the host-memory RECV queue, and updating the queue's host-memory head pointer. No CPU-initiated PCIe transactions are required as the NIC fetches the descriptors using DMA reads. In FaSST, we populate the RECV queue with descriptors once during initialization, after which the descriptors are not accessed by the CPU; new RECVs re-use descriptors in a circular fashion, and can be posted with a single write to the cached head pointer. Doing so required modifying the NIC's device driver, but it saves CPU cycles.

It is interesting to note that in FaSST, the NIC's RECV descriptor DMA reads are redundant, since the descriptors never change after initialization. Avoiding these DMA reads may be possible with device firmware changes or with future programmable NICs; doing so will likely give FaSST's RPCs a large throughput boost [16].

4.3 Detecting packet loss

The master coroutine at each thread detects packet loss for RPCs issued by its worker coroutines. The master tracks the progress of each worker by counting the number of responses received for the worker. A worker's progress counter stagnates if and only if one of the worker's RPC packets (either the request or the response) is lost: If a packet is lost, the master never receives all responses for the worker; it never invokes the worker again, preventing it from issuing new requests and receiving more responses. If no packet is lost, the master eventually receives all responses for the worker. The worker gets invoked and issues new requests—we do not allow workers to yield to the master without issuing RPC requests.

If the counter for a worker does not change for `timeout` seconds, the master assumes that the worker suffered a packet loss. On suspecting a loss, the master kills the FaSST process on its machine (Section 5.1). Note that, before it is detected, a packet loss affects only the progress of one worker, i.e., other workers can successfully commit transactions until the loss is detected. This allows us to use a large value for `timeout` without affecting FaSST's availability. We currently set `timeout` to one second. In our experiments with 50+ nodes, we did not observe a false positive with this timeout value. We observed false positives with significantly smaller timeout values such as 100 ms. This can happen, for instance, if the thread handling the RPC response gets preempted [12].

4.4 RPC limitations

Although FaSST aims to provide general-purpose RPCs, we do currently not support workloads that require messages larger than the network's MTU (4 KB on our InfiniBand network). These workloads are likely to be bottlenecked by network bandwidth with both RPC- and one-sided RDMA-based designs, achieving similar performance. This limitation can be addressed in several performance-neutral ways if needed.[2]

FaSST also restricts each coroutine to one message per destination machine per batch; the message, however, can contain multiple coalesced requests. This restriction is required to keep the RECV queues small so that they can be cached by the NIC. Consider a cluster with N nodes, T threads per node, and c coroutines per thread. For a given thread, there are N peer threads, and $N * c$ coroutines that can send requests to it. At any time, each thread must provision as many RECVs in its RECV queue as the number of requests that can be sent to it. Allowing each coroutine m messages per destination machine requires maintaining $(N * c * m)$ RECVs per RECV queue. A fairly large cluster with $N = 100$, $c = 20$, and $T = 14$ requires 14 RECV queues of size $2000 * m$ at each machine. $m = 1$ was sufficient for our workloads and worked well in our experiments, but significantly larger values of m reduce RPC performance by causing NIC cache thrashing.

Supporting a larger cluster may require reducing RECV queue size. This can be achieved by reducing the number of requests allowed from a local thread to a particular remote machine from c to some smaller number; a coroutine yields if its thread's budget for a remote machine is temporarily exhausted. This will work well with large clusters and workloads without high skew, where the probability of multiple coroutines sending requests to the same remote machine is small.

4.5 Single-core RPC performance

We showed in Section 3.3 that FaSST RPCs provide good per-NIC throughput. We now show that they also provide good single-core throughput. To measure per-core throughput, we run one thread per machine, 20 coroutines per thread, and use 32-byte RPCs. We use all 11 available machines on CIB; we use 11 machines on CX3 for comparison. We evaluate RPC performance with multiple request batch sizes. To prevent request coalescing by our RPC layer, we choose a different machine for each request in the batch.

For our RPC baseline, we use a request batch size of one, and disable response batching. We then enable the request batching, cheap RECV posting, and response batching optimizations in succession. Figure 4 shows the results from this experiment. Note that the batching optimizations do not apply to READs, because Doorbells cannot be batched across the connected QPs. For brevity, we discuss only CIB here.

[2]Infrequent small but > 4KB messages could be segmented in the RPC layer. Alternately, a three-message exchange wherein the sender requests that the recipient use a one-sided READ to obtain a large message payload could be used.

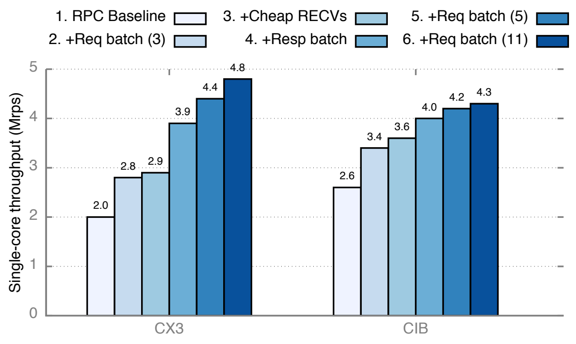


Figure 4: Per-core RPC throughput as optimizations 2–6 are added

Even without any optimizations, FaSST RPCs are more CPU-efficient than READs with QP sharing: our baseline achieves 2.6 Mrps, whereas READs achieve up to 2 Mrps with QP sharing between 3 or more threads (Figure 3). With a request batch size of 3 and all optimizations enabled, FaSST RPCs achieve 4 Mrps—2x higher than READs. Peak RPC throughput with one request per batch is 3.4 Mrps (not shown).

With 11 requests per batch, FaSST RPCs achieve 4.3 Mrps. At this request rate, each CPU core issues 17.2 million verbs per second on average: 4.3 million SENDs each for requests and responses, and 8.6 million for their RECVs. This large advantage over one-sided READs (which achieve 2 million verbs per second) arises from FaSST's use of datagram transport, which allows exclusive access to QPs and Doorbell batching.

Comparison with FaRM RPCs: FaRM's RPCs achieve up to 5 Mrps with one ConnectX-3 NIC and 16 CPU cores [11]. Their throughput does not increase noticeably when another ConnectX-3 NIC is added [12], so we expect them to provide ≈ 5 Mrps with a Connect-IB NIC. FaSST RPCs can achieve 34.9–40.9 Mrps (Figure 1), i.e., up to 8x higher throughput per NIC. FaRM's RPCs achieve $5/16 = 0.31$ Mrps per core; FaSST can achieve 3.4–4.3 Mrps per core depending on the request batch size (up to 13.9x higher).

5 Transactions

FaSST provides transactions with serializability and durability on partitioned distributed data stores. FaSST's data stores map 8-byte keys to opaque application-level objects. Each key is associated with an 8-byte header, consisting of a lock bit, and a 63-bit version number. The header is used for concurrency control and for ordering commit

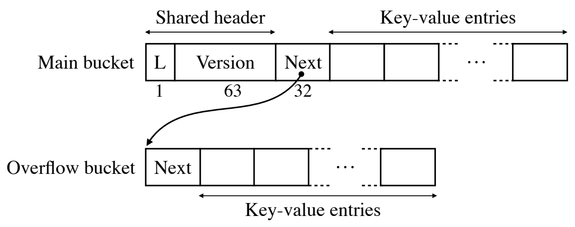


Figure 5: Layout of main and overflow buckets in our MICA-based hash table

log records during recovery. Several keys can map to the same header.

We have implemented transactions for an unordered key-value store based on MICA [18]. The key-value store uses a hash table composed of associative buckets (Figure 5) with multiple (7–15) slots to store key-value items. Each key maps to a *main* bucket. If the number of keys mapping to a main bucket exceeds the bucket capacity, the main bucket is dynamically linked to a chain of *overflow* buckets. The header for all keys stored in a main bucket and its linked overflow buckets is maintained in the main bucket.

In FaSST, worker coroutines run the transaction logic and act as transaction coordinators. FaSST's transaction protocol is inspired by FaRM's, with some modifications for simplicity. FaSST uses optimistic concurrency control and two-phase commit for distributed atomic commit, and primary-backup replication to support high availability. We use the Coordinator Log [24] variant of two-phase commit for its simplicity. Figure 6 summarizes FaSST's transaction protocol. We discuss the protocol's phases in detail below. All messages are sent using FaSST RPCs. We denote the set of keys read and written by the transaction by R (read set) and W (write set) respectively. We assume that the transaction first reads the keys it writes, i.e., $W \subseteq R$.

1. Read and lock: The transaction coordinator begins execution by reading the header and value of keys from their primaries. For a key in W, the coordinator also requests the primary to lock the key's header. The flexibility of RPCs allows us to read and lock keys in a single round trip. Achieving this with one-sided RDMA requires 2 round trips: one to lock the key using an ATOMIC operation, and one to read its value [7]. If any key in R or W is already locked, the coordinator aborts the transaction by sending unlock RPCs for successfully locked keys.

2. Validate: After locking the write set, the coordinator checks the versions of its read set by requesting the versions of R again. If any key is locked or its version has changed since the first phase, the coordinator aborts the transaction.

3. Log: If validation succeeds, the transaction can commit. To commit a transaction, the coordinator replicates

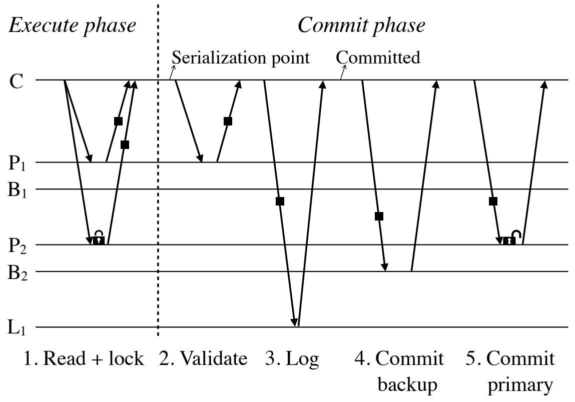


Figure 6: FaSST's transaction protocol with tolerance for one node failure. P_1 and P_2 are primaries and B_1 and B_2 are their backups. C is the transaction coordinator, whose log replica is L_1. The solid boxes denote messages containing application-level objects. The transaction reads one key from P_1 and P_2, and updates the key on P_2.

its commit log record at $f + 1$ log replicas so that the transaction's commit decision survives f failures. The coordinator's host machine is always a log replica, so we send f RPCs to remote log replicas. The commit log record contains W's key-value items and their fetched versions.

4. Commit backup: If logging succeeds, the coordinator sends update RPCs to backups of W. It waits for an ACK from each backup before sending updates to the primaries. This wait ensures that backups process updates for a bucket in the same order as the primary. This ordering is not required in FaRM, which can drop out-of-order bucket updates as each update contains the contents of the entire bucket. FaSST's updates contain only one key-value item and are therefore smaller, but cannot be dropped.

5. Commit primary: After receiving all backup ACKs, the coordinator sends update RPCs to the primaries of W. On receiving an update, a primary updates the key's value, increments its version, and unlocks the key.

Similar to existing systems [12, 28], FaSST omits validation and subsequent phases for single-key read-only transactions.

5.1 Handling failures and packet loss

Currently, the FaSST implementation provides serializability and durability, but not high availability. Similar to prior single-node transaction systems [27], we have implemented the normal case datapath (logging and replication) to the extent that fast recovery is possible, but we have not implemented the actual logic to recover from a machine failure. We assume that FaRM's mechanisms to detect and recover from machine failures, such as leases, cluster membership reconfiguration, log replay, and re-replication of lost data can be adapted to FaSST; we discuss how packet losses can be handled below. Note that our implementation is insensitive to packet reordering since each RPC message is smaller than the network's MTU.

We convert a packet loss to a machine failure by killing the FaSST process on the machine that detects a lost RPC packet (Section 4.3). The transaction affected by the lost packet will not make progress until the killed FaSST process is detected (e.g., via leases); then the transaction's commit/abort decision will be handled by the recovery mechanism. This basic scheme can be improved (e.g., the victim node can be re-used to avoid data re-replication since it need not reboot), but that is not the focus of our work.

In Section 3.4, we measured the packet loss rate of our network at less than one in 50 PB of data. Since we did not actually lose a packet, the real loss rate may be much lower, but we use this upper-bound rate for a ballpark availability calculation. In a 100-node cluster where each node is equipped with 2x56 Gbps InfiniBand and transfers data at full-duplex bandwidth, 50 PB will be transferred in approximately 5 hours. Therefore, packet losses will translate to less than 5 machine failures per day. Assuming that each failure causes 50 ms of downtime as in FaRM [12], FaSST will achieve five-nines of availability.

5.2 Implementation

We now discuss details of FaSST's transaction implementation. Currently, FaSST provides transactions on 8-byte keys and opaque objects up to 4060 bytes in size. The value size is limited by our network's MTU (4096 bytes) and the commit record header overhead (36 bytes). To extend a single-node data store for distributed transactions, a FaSST user writes RPC request handlers for pre-defined key-value requests (e.g., get, lock, put, and delete). This may require changes to the single-node data store, such as supporting version numbers. The user registers database tables and their respective handlers with the RPC layer by assigning each table a unique RPC request type; the RPC subsystem invokes a table's handler on receiving a request with its table type.

The data store must support concurrent local read and write access from all threads in a node. An alternate design is to create exclusive data store partitions per thread, instead of per-machine partitions as in FaSST. As shown in prior work [18], this alternate design is faster for local data store access since threads need not use local concurrency control (e.g., local locks) to access their exclusive partition. However, when used for distributed transactions, it requires the RPC subsystem to support all-to-all communication between threads, which reduces scalability by amplifying the required RECV queue size (Section 4.4). We chose to sacrifice higher CPU efficiency on small clusters for a more pressing need: cluster-level scalability.

	Nodes	NICs	CPUs (cores used, GHz)
FaSST (CX3)	50	1	1x E5-2450 (8, 2.1 GHz)
FaRM [12]	90	2	2x E5-2650 (16, 2.0 GHz)
DrTM+R [7]	6	1	1x E5-2450-v3 (8, 2.3 GHz)

Table 3: Comparison of clusters used to compare published numbers. The NIC count is the number of ConnectX-3 NICs. All CPUs are Intel® Xeon ® CPUs. DrTM+R's CPU has 10 cores but their experiments use only 8 cores.

5.2.1 Transaction API

The user writes application-level transaction logic in a worker coroutine using the following API.

`AddToReadSet(K, *V)` and **`AddToWriteSet(K, *V, mode)`** enqueue key `K` to be fetched for reading or writing, respectively. For write set keys, the write mode is either insert, update, or delete. After the coroutine returns from `Execute` (see below) the value for key `K` is available in the buffer `V`. At this point, the application's transaction logic can modify `V` for write set keys to the value it wishes to commit.

`Execute()` sends the execute phase-RPCs of the transaction protocol. Calling `Execute` suspends the worker coroutine until all responses are available. Note that the `AddToReadSet` and `AddToWriteSet` functions above do not generate network messages immediately: requests are buffered until `Execute` is called. This allows the RPC layer to send all requests in with one Doorbell, and coalesce requests sent to the same remote machine. Applications can call `Execute` multiple times in one transaction after adding more keys. This allows transactions to choose new keys based on previously fetched keys.

`Execute` fails if a read or write set key is locked. In this case, the transaction layer returns failure to the application, which then must call `Abort`.

`Commit()` runs the commit protocol, including validation, logging and replication, and returns the commit status. **`Abort()`** sends unlock messages for write set keys.

6 Evaluation

We evaluate FaSST using 3 benchmarks: an object store, a read-mostly OLTP benchmark, and a write-intensive OLTP benchmark. We use the simple object store benchmark to measure the effect of two factors that affect FaSST's performance: multi-key transactions and the write-intensiveness of the workload. All benchmarks include 3-way logging and replication, and use 14 threads per machine.

We use the other two benchmarks to compare against two recent RDMA-based transaction systems, FaRM [12] and DrTM+R [7]. Unfortunately, we are unable do a direct comparison by running these systems on our clusters. FaRM is not open-source, and DrTM+R depends on Intel's Restricted Transactional Memory (RTM). Intel's RTM is disabled by default on Haswell processors due to a hardware bug that can cause unpredictable behavior. It can be re-enabled by setting model-specific registers [28], but we were not permitted to do so on the CIB cluster.

For a comparison against published numbers, we use the CX3 cluster which has less powerful hardware (NIC and/or CPU) than used in FaRM and DrTM+R; Table 3 shows the differences in hardware. We believe that the large performance difference between FaSST and other systems (e.g., 1.87x higher than FaRM on TATP with half the hardware resources; 1.68x higher than DrTM+R on SmallBank without locality assumptions) offsets performance variations due to system and implementation details. We also use the CIB cluster in our evaluation to show that FaSST can scale up to more powerful hardware.

TATP is an OLTP benchmark that simulates a telecommunication provider's database. It consists of 4 database tables with small key-value pairs up to 48 bytes in size. TATP is read-intensive: 70% of TATP transactions read a single key, 10% of transactions read 1–4 keys, and the remaining 20% of transactions modify keys. TATP's read-intensiveness and small key-value size makes it well-suited to FaRM's design goal of exploiting remote CPU bypass: 80% of TATP transactions are read-only and do not involve the remote CPU. Although TATP tables can be partitioned intelligently to improve locality, we do not do so (similar to FaRM).

SmallBank is a simple OLTP benchmark that simulates bank account transactions. SmallBank is write-intensive: 85% of transactions update a key. Our implementation of SmallBank does not assume data locality. In DrTM+R, however, single-account transactions (comprising 60% of the workload) are initiated on the server hosting the key. Similarly, only a small fraction (< 10%) of transactions that access two accounts access accounts on different machines. These assumptions make the workload well-suited to DrTM+R's design goal of optimizing local transactions by using hardware transactional memory, but they save messages during transaction execution and commit. We do not make either of these assumptions and use randomly chosen accounts in all transactions.

Although the TPC-C benchmark [26] is a popular choice for evaluating transaction systems, we chose not to include it in our benchmarks for two reasons. First, TPC-C has a high degree of locality: only around 10% of transactions (1% of keys) access remote partitions. The speed of local transactions and data access, which our work does not focus on, has a large impact on TPC-C performance. Second, comparing performance across TPC-C implementations is difficult. This is due to differences in data structures (e.g., using hash tables instead of B-Trees

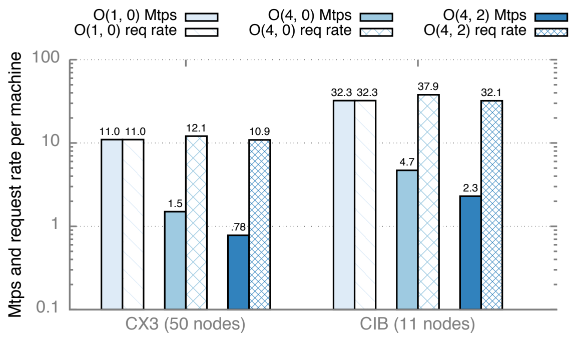


Figure 7: Object store performance. The solid and patterned bars show transaction throughput and RPC request rate, respectively. The Y axis is in log scale.

for some tables), interaction of the benchmark with system optimizations (e.g., FaRM and DrTM+R use caching to reduce READs, but do not specify cache hit rates), and contention level (DrTM+R uses 1 TPC-C "warehouse" per thread whereas FaRM uses ≈ 7, which may reduce contention).

6.1 Object store

We create an object store with small objects with 8-byte keys and 40-byte values. We scale the database by using 1 million keys per thread in the cluster. We use workloads with different read and write set sizes to evaluate different aspects of FaSST. An object store workload in which transactions read r keys, and update w of these keys (on average) is denoted by $O(r,w)$; we use $O(1,0)$, $O(4,0)$, and $O(4,2)$ to evaluate single-key read-only transactions, multi-key read-only transactions, and multi-key read-write transactions. All workloads choose keys uniformly at random; to avoid RPC-level coalescing, keys are chosen such that their primaries are on different machines. Figure 7 shows FaSST's performance on the object store workloads on the two clusters.

6.1.1 Single-key read-only transactions

With $O(1,0)$ FaSST achieves 11.0 million transactions per second (Mtps) per machine on CX3. FaSST is bottlenecked by the ConnectX-3 NIC: this throughput corresponds to 11.0 million RPC requests per second (Mrps), which is 96.5% of the NIC's maximum RPC throughput in this scenario.

On CIB, FaSST achieves 32.3 Mtps/machine and is CPU-limited. This is because $O(1,0)$ does not allow Doorbell batching for requests, leading to low per-core throughput. Although CIB's CPUs can saturate the NIC without request Doorbell batching for an RPC microbenchmark that requires little computation (Section 3.3.1), they cannot do so for $O(1,0)$ which requires key-value store accesses.

Comparison: FaRM [12] reports performance for the $O(1,0)$ workload. FaRM uses larger, 16-byte keys and

32-byte values. Our FaSST implementation currently supports only 8-byte keys, but we use larger, 40-byte values to keep the key-value item size identical. Using 16-byte keys is unlikely to change our results.[3]

FaRM achieves 8.77 Mtps/machine on a 90-node cluster with $O(1,0)$. It does not saturate its 2 ConnectX-3 NICs and is instead bottlenecked by its 16 CPU cores. FaSST achieves 1.25x higher per-machine throughput with 50 nodes on CX3, which has close to half of FaRM's hardware resources per node (Table 3). *Although $O(1,0)$ is well-suited to FaRM's design goal of remote CPU bypass (i.e., no transaction involves the remote CPU), FaRM performs worse than FaSST.* Note that with FaRM's hardware—2 ConnectX-3 NICs and 16 cores—FaSST will deliver higher performance; based on our CIB results, we expect FaSST to saturate the two ConnectX-3 NICs and outperform FaRM by 2.5x.

6.1.2 Multi-key transactions

With multi-key transactions, FaSST reduces per-message CPU use by using Doorbell batching for requests. With $O(4,0)$, FaSST achieves 1.5 and 4.7 Mtps/machine on CX3 and CIB, respectively. (The decrease in Mtps from $O(1,0)$ is because the transactions are larger.) Similar to $O(1,0)$, FaSST is NIC-limited on CX3. On CIB, however, although FaSST is CPU-limited with $O(1,0)$, it becomes NIC-limited with $O(4,0)$. With $O(4,0)$ on CIB, each machine generates 37.9 Mrps on average, which matches the peak RPC throughput achievable with a request batch size of 4.

With multi-key read-write transactions in $O(4,2)$, FaSST achieves 0.78 and 2.3 Mtps/machine on CX3 and CIB, respectively. FaSST is NIC-limited on CX3. On CIB, the bottleneck shifts to CPU again because key-value store inserts into the replicas' data stores are slower than lookups.

Comparison: FaRM does not report object store results for multi-key transactions. However, as FaRM's connected transport does not benefit from Doorbell batching, we expect the gap between FaSST's and FaRM's performance to increase. For example, while FaSST's RPC request rate increases from 32.3 Mrps with $O(1,0)$ to 37.9 Mrps with $O(4,0)$, the change in FaRM's READ rate is likely to be negligible.

6.2 TATP

We scale the TATP database size by using one million TATP "subscribers" per machine in the cluster. We use all CPU cores on each cluster and increase the number of machines to measure the effect of scaling. Figure 8 shows the throughput on our clusters. On CX3, FaSST achieves 3.6 Mtps/machine with 3 nodes (the minimum required for 3-way replication), and 3.55 Mtps/machine with 50 nodes. On CIB, FaSST's throughput increases to 8.7 Mtps/machine with 3–11 nodes. In both cases, FaSST's throughput scales linearly with cluster size.

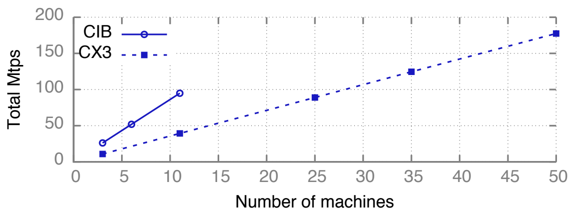


Figure 8: TATP throughput

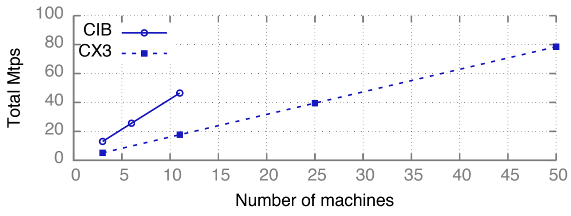


Figure 9: SmallBank throughput

Comparison: FaRM [12] reports 1.55 Mtps/machine for TATP on a 90-node cluster. With a smaller 50-node cluster, however, FaRM achieves higher throughput (≈ 1.9 Mtps/machine) [1]. On 50 nodes on CX3, FaSST's throughput is 87% higher. Compared to $O(1,0)$, the TATP performance difference between FaSST and FaRM is higher. TATP's write transactions require using FaRM's RPCs, which deliver 4x lower throughput than FaRM's one-sided READs, and up to 8x lower throughput than FaSST's RPCs (Section 4.5).

6.3 SmallBank

To scale the SmallBank database, we use 100,000 bank accounts per thread. 4% of the total accounts are accessed by 90% of transactions. (Despite the skew, the workload does not have significant contention due to the large number of threads, and therefore "bank accounts" in the workload/cluster.) This configuration is the same as in DrTM [28]. Figure 9 shows FaSST's performance on our clusters. FaSST achieves 1.57–1.71 Mtps/machine on CX3, and 4.2–4.3 Mtps/machine on CIB, and scales linearly with cluster size.

Comparison: DrTM+R [7] achieves 0.93 Mtps/machine on a cluster similar to CX3 (Table 3), but with more powerful CPUs. FaSST outperforms it by over 1.68x on CX3, and over 4.5x on CIB. DrTM+R's lower performance comes from three factors. First, ATOMICs lead to a fundamentally slower protocol. For example, excluding logging and replication, for a write-set key, DrTM+R uses four separate messages to read, lock, update, and unlock the key; FaSST uses only two messages. Second, as shown

[3]On a single node, FaSST's data store (MICA) delivers similar GET throughput (within 3%) for these two key-value size configurations. Throughput is *higher* with 16-byte keys, which could be because MICA's hash function uses fewer cycles.

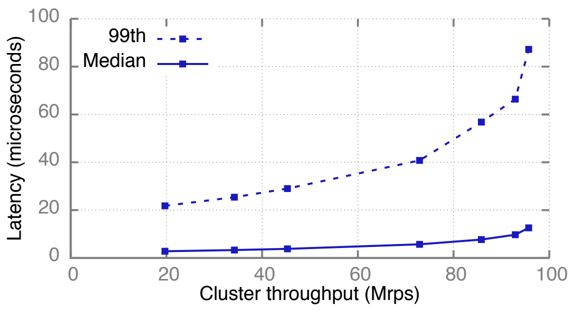


Figure 10: TATP latency on CIB

in our prior work, ATOMICs perform poorly (up to 10x worse than READs) on the ConnectX-3 NICs [16] used in DrTM+R; evaluation on Connect-IB NICs may yield better performance, but is unlikely to outperform FaSST because of the more expensive protocol. Third, DrTM+R does not use queue pair sharing, so their reported performance may be affected by NIC cache misses.

6.4 Latency

For brevity, we discuss FaSST's latency only for TATP on CIB. Figure 10 shows FaSST's median and 99^{th} percentile latency for successfully committed TATP transactions. To plot a throughput-latency curve, we vary the request load by increasing the number of worker coroutines per thread from 1 to 19; each machine runs 14 threads throughout. We use at most 19 worker coroutines per thread to limit the RECV queue size required on a (hypothetical) 100-node cluster to 2048 RECVs (Section 4.4). Using the next available RECV queue size with 4096 RECVs can cause NIC cache misses for some workloads. With one worker coroutine per thread, the total transaction throughput is 19.7 Mtps with 2.8 µs median latency and 21.8 µs 99^{th} percentile latency. Since over 50% of committed transactions in TATP are single-key reads, FaSST's median latency at low load is close to the network's RTT. This shows that our batching optimizations do not add noticeable latency. With 19 worker coroutines per thread, cluster throughput increases to 95.7 Mtps, and median and 99^{th} percentile latency increase to 12.6 µs and 87.2 µs, respectively.

7 Future trends

7.1 Scalable one-sided RDMA

A key limitation of one-sided RDMA on current commodity hardware is its low scalability. This limitation itself comes from the fundamentally connection-oriented nature of the Virtual Interface Architecture. Two attempts at providing scalable one-sided RDMA are worth mentioning.

DCT: Mellanox's Dynamically Connected Transport [2] (DCT) preserves their core connection-oriented design, but dynamically creates and destroys one-to-one connections. This provides software the illusion of using one QP to communicate with multiple remote machines, but at a prohibitively large performance cost for our workloads: DCT requires three additional network messages when the target machine of a DCT queue pair changes: a disconnect packet to the current machine, and a two-way handshake with the next machine to establish a connection [8]. In a high fanout workload such as distributed OLTP, this increases the number of packets associated with each RDMA request by around 1.5x, reducing performance.

A detailed evaluation of DCT on CIB is available in FaSST's source code repository. Here, we discuss DCT's performance in the READ rate benchmark used in Section 3.3.1. We use 6 machines and 14 threads per machine, which issue 32-byte READs to machines chosen uniformly at random. We vary the number of outstanding READs per thread, and the number of DCT QPs used by each thread. (Using only one DCT QP per thread limits its throughput to approximately one operation per multiple RTTs, since a QP cannot be used to READ from multiple machines concurrently. Using too many DCT QPs causes cache NIC misses.) We achieve only up to 22.9 Mrps per machine—55.3% lower than the 51.2 Mrps achievable with standard READs over the RC transport (Figure 1).

Portals: Portals is a less-widespread RDMA specification that provides scalable one-sided RDMA using a connectionless design [4]. Recently, a hardware implementation of Portals—the Bull eXascale Interconnect [10]—has emerged, currently available only to HPC customers. The availability of scalable one-sided RDMA may reduce the performance gap between FaSST and FaRM/DrTM+R. However, even with scalable one-sided RDMA, transaction systems that use it will require large or multiple READs to access data stores, reducing performance. Further, it is likely that RPC implementations that use scalable one-sided WRITEs will provide even better performance than FaSST's two-sided RPCs by avoiding the NIC and PCIe overhead of RECVs. However, WRITE-based RPCs do not scale well on current commodity hardware.

In this paper, we took an extreme position where we used only RPCs, demonstrating that remote CPU bypass is not required for high performance transactions, and that a design using optimized RPCs can provide better performance. *If scalable one-sided RDMA becomes commonly available in the future, the best design will likely be hybrid of RPCs and remote bypass, with RPCs used for accessing data structures during transaction execution, and scalable one-sided WRITEs used for logging and replication during transaction commit.*

7.2 More queue pairs

Our experiments show that the newer Connect-IB NIC can cache a larger number of QPs than ConnectX-3 (Figure 2). Just as advances in technology yield NICs that are faster and have more/better cache, newer CPUs will also have more cores. We showed earlier that sharing QPs between only 2 threads causes CPU efficiency to drop by several factors (Figure 3). Avoiding QP sharing with next-generation CPUs (e.g., 28 cores in Intel's upcoming Skylake processors) on a 100-node cluster will require NICs that can cache 2800 QPs—7 times more than Connect-IB's 400 QPs. This trend lends additional support to datagram-based designs.

7.3 Advanced one-sided RDMA

Future NICs may provide advanced one-sided RDMA operations such as multi-address atomic operations, and B-Tree traversals [23]. Both of these operations require multiple PCIe round trips, and will face similar flexibility and performance problems as one-sided RDMA (but over the PCIe bus) if used for high-performance distributed transactions. On the other hand, we believe that "CPU onload" networks such as Intel's 100 Gbps OmniPath [6] are well-suited for transactions. These networks provide fast messaging over a reliable link layer, but not one-sided RDMA, and are therefore cheaper than "NIC offload" networks such as Mellanox's InfiniBand. FaSST requires only messaging, so we expect our design to work well over OmniPath.

8 Related work

High-performance RDMA systems: FaSST draws upon our prior work on understanding RDMA performance [16], where we demonstrated the effectiveness of Doorbell batching for RDMA verbs. A number of recent systems have used one-sided verbs to build key-value stores and distributed shared memory [20, 21, 11, 21]. These systems demonstrate that RDMA is now a practical primitive for building non-HPC systems, though their one-sided designs introduce additional complexities and performance bottlenecks. FaSST's two-sided RPC approach generalizes our approach in HERD [15]. HERD used a hybrid of unreliable one-sided and two-sided RDMA to implement fast RPCs in a client-server setting; FaSST extends this model to a symmetric setting and further describes a technique to implement reliability.

Distributed transactions in the datacenter: Like FaSST, FaRM [12] uses primary-backup replication and optimistic concurrency control for transactions. FaRM's design (unlike FaSST) is specialized to work with their desire to use one-sided RDMA verbs. FaRM also provides fast failure detection and recovery, and a sophisticated programming model, which was not a goal of this work. Several projects use one-sided ATOMICs for transactions [28, 7, 5]. Though an attractive primitive, ATOMICs are slow on current NICs (e.g., ConnectX-3 serializes all ATOMIC operations [16]), use connected QPs, and fundamentally require more messages than an RPC-based approach (e.g., separate messages are needed to read and lock a key). Calvin [25] uses conventional networking without kernel bypass, and is designed around avoiding distributed commit. Designs that use fast networks, however, can use traditional distributed commit protocols to achieve high performance [12, 28].

9 Conclusion

FaSST is a high-performance, scalable, distributed in-memory transaction processing system that provides serializability and durability. FaSST achieves its performance using FaSST RPCs, a new RPC design tailored to the properties of modern RDMA hardware that uses two-sided verbs and datagram transport. It rejects one of the seemingly most attractive properties of RDMA—CPU bypass—to keep its communication overhead low and its system design simple and fast. The combination allows FaSST to outperform recent RDMA-based transactional systems by 1.68x–1.87x with fewer resources and making fewer workload assumptions. Finally, we provide the first large-scale study of InfiniBand network reliability, demonstrating the rarity of packet loss on such networks.

Acknowledgments We are tremendously grateful to Joseph Moore and NetApp for providing continued access to the CIB cluster. We thank Miguel Castro for initial discussions that shaped this project, and for his feedback on the paper. We also thank Milo Martin for his feedback that helped improve the paper, Chao Xin for conducting initial performance measurements, Hyeontaek Lim for his generous support with running MICA, and Y. Charlie Hu for shepherding. Emulab [29] and PRObE [14] resources were used in our experiments. PRObE is supported in part by NSF awards CNS-1042537 and CNS-1042543 (PRObE). This work was supported by funding from the National Science Foundation under awards 1345305, 1314721, and 1535821, and by Intel via the Intel Science and Technology Center for Cloud Computing (ISTC-CC).

References

[1] Private communication with FaRM's authors.

[2] Mellanox Connect-IB product brief. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_Connect-IB.pdf, 2015.

[3] Mellanox OFED for Linux user manual. http://www.mellanox.com/related-docs/prod_software/Mellanox_OFED_Linux_User_Manual_v2.2-1.0.1.pdf, 2015.

[4] B. W. Barrett, R. Brightwell, S. Hemmert, K. Pedretti, K. Wheeler, K. Underwood, R. Riesen, A. B. Maccabe, and T. Hudson. The Portals 4.0 network programming interface november 14, 2012 draft.

[5] 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*, New Delhi, India, Aug. 2016.

[6] M. S. Birrittella, M. Debbage, R. Huggahalli, J. Kunz, T. Lovett, T. Rimmer, K. D. Underwood, and R. C. Zak. Intel Omni-path architecture: Enabling scalable, high performance fabrics. In *Proceedings of the 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, 2015.

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

[8] D. Crupnicoff, M. Kagan, A. Shahar, N. Bloch, and H. Chapman. Dynamically-connected transport service, May 19 2011. URL https://www.google.com/patents/US20110116512. US Patent App. 12/621,523.

[9] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. 21st ACM Symposium on Operating Systems Principles (SOSP)*, Stevenson, WA, Oct. 2007.

[10] S. Derradji, T. Palfer-Sollier, J.-P. Panziera, A. Poudes, and F. W. Atos. The BXI interconnect architecture. In *Proceedings of the 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, 2015.

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

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

[13] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. M. Merritt, E. Gronke, and C. Dodd. The virtual interface architecture. *IEEE Micro*, pages 66–76, 1998.

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

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

[16] A. Kalia, M. Kaminsky, and D. G. Andersen. Design guidelines for high-performance RDMA systems. In *Proc. USENIX Annual Technical Conference*, Denver, CO, June 2016.

[17] L. Lamport, D. Malkhi, and L. Zhou. Vertical Paxos and primary-backup replication. Technical report, Microsoft Research, 2009.

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

[19] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *Proc. 7th ACM European Conference on Computer Systems (EuroSys)*, Bern, Switzerland, Apr. 2012.

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

[21] C. Mitchell, K. Montgomery, L. Nelson, S. Sen, and J. Li. Balancing CPU and network in the cell distributed B-Tree store. In *Proc. USENIX Annual Technical Conference*, Denver, CO, June 2016.

[22] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Latency-tolerant software distributed shared memory. In *Proc. USENIX Annual Technical Conference*, Santa Clara, CA, June 2015.

[23] S. Raikin, L. Liss, A. Shachar, N. Bloch, and M. Kagan. Remote transactional memory, 2015. US Patent App. 20150269116.

[24] J. W. Stamos and F. Cristian. Coordinator log transaction execution protocol. *Distrib. Parallel Databases*, 1(4):383–408, Oct. 1993. ISSN 0926-8782. doi: 10.1007/BF01264014. URL http://dx.doi.org/10.1007/BF01264014.

[25] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, May

2012.

[26] TPC-C. TPC benchmark C. `http://www.tpc.org/tpcc/`, 2010.

[27] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proc. 24th ACM Symposium on Operating Systems Principles (SOSP)*, Farmington, PA, Nov. 2013.

[28] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast in-memory transaction processing using RDMA and HTM. In *Proc. 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, Oct. 2015.

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

[30] H. Zhang, D. G. Andersen, A. Pavlo, M. Kaminsky, L. Ma, and R. Shen. Reducing the storage overhead of main-memory OLTP databases with hybrid indexes. In *Proc. ACM SIGMOD*, San Francisco, USA, June 2016.