No compromises: distributed transactions with consistency, availability, and performance

Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale,
Matthew Renzelmann, Alex Shamis, Anirudh Badam, Miguel Castro
Microsoft Research

Abstract

Transactions with strong consistency and high availability simplify building and reasoning about distributed systems. However, previous implementations performed poorly. This forced system designers to avoid transactions completely, to weaken consistency guarantees, or to provide single-machine transactions that require programmers to partition their data. In this paper, we show that there is no need to compromise in modern data centers. We show that a main memory distributed computing platform called FaRM can provide distributed transactions with strict serializability, high performance, durability, and high availability. FaRM achieves a peak throughput of 140 million TATP transactions per second on 90 machines with a 4.9 TB database, and it recovers from a failure in less than 50 ms. Key to achieving these results was the design of new transaction, replication, and recovery protocols from first principles to leverage commodity networks with RDMA and a new, inexpensive approach to providing non-volatile DRAM.

1. Introduction

Transactions with high availability and strict serializability [35] simplify programming and reasoning about distributed systems by providing a simple, powerful abstraction: a single machine that never fails and that executes one transaction at a time in an order consistent with real time. However, prior attempts to implement this abstraction in a distributed system resulted in poor performance. Therefore, systems such as Dynamo [13] or Memcached [1] improve performance by either not supporting transactions or by implementing weak consistency guarantees. Others (e.g., [3–6, 9, 28]), provide transactions only when all the data resides within a single machine, forcing programmers to partition their data and complicating reasoning about correctness.

This paper demonstrates that new software in modern data centers can eliminate the need to compromise. It describes the transaction, replication, and recovery protocols in FaRM [16], a main memory distributed computing platform. FaRM provides distributed ACID transactions with strict serializability, high availability, high throughput and low latency. These protocols were designed from first principles to leverage two hardware trends appearing in data centers: fast commodity networks with RDMA and an inexpensive approach to providing non-volatile DRAM. Non-volatility is achieved by attaching batteries to power supply units and writing the contents of DRAM to SSD when the power fails. These trends eliminate storage and network bottlenecks, but they also expose CPU bottlenecks that limit their performance benefit. FaRM's protocols follow three principles to address these CPU bottlenecks: *reducing message counts*, *using one-sided RDMA reads and writes instead of messages*, and *exploiting parallelism effectively*.

FaRM scales out by distributing objects across the machines in a data center while allowing transactions to span any number of machines. Rather than replicate coordinators and data partitions using Paxos (e.g., as in [11]), FaRM reduces message counts by using vertical Paxos [25] with primary-backup replication, and unreplicated coordinators that communicate directly with primaries and backups. FaRM uses optimistic concurrency control with a four phase commit protocol (lock, validation, commit backup, and commit primary) [16] but we improved the original protocol by eliminating the messages to backups in the lock phase.

FaRM further reduces CPU overhead by using one-sided RDMA operations. One-sided RDMA uses no remote CPU and it avoids most local CPU overhead. FaRM transactions use one-sided RDMA reads during transaction execution and validation. Therefore, they use no CPU at remote read-only participants. Additionally, coordinators use one-sided RDMA when logging records to non-volatile write-ahead logs at the replicas of objects modified in a transaction. For

SOSP'15, October 4–7, 2015, Monterey, CA.

ACM 978-1-4503-3834-9/15/10.
http://dx.doi.org/10.1145/2815400.2815425

example, the coordinator uses a single one-sided RDMA to write a commit record to a remote backup. Hence, transactions use no foreground CPU at backups. CPU is used later in the background when lazily truncating logs to update objects in-place.

Using one-sided RDMA requires new failure-recovery protocols. For example, FaRM cannot rely on servers to reject incoming requests when their leases [18] expire because requests are served by the NICs, which do not support leases. We solve this problem by using *precise membership* [10] to ensure that machines agree on the current configuration membership and send one-sided operations only to machines that are members. FaRM also cannot rely on traditional mechanisms that ensure participants have the resources necessary to commit a transaction during the prepare phase because transaction records are written to participant logs without involving the remote CPU. Instead, FaRM uses *reservations* to ensure there is space in the logs for all the records needed to commit and truncate a transaction before starting the commit.

The failure recovery protocol in FaRM is fast because it leverages parallelism effectively. It distributes recovery of every bit of state evenly across the cluster and it parallelizes recovery across cores in each machine. In addition, it uses two optimizations to allow transaction execution to proceed in parallel with recovery. First, transactions begin accessing data affected by a failure after a lock recovery phase that takes only tens of milliseconds to complete rather than wait several seconds for the rest of recovery. Second, transactions that are unaffected by a failure continue executing without blocking. FaRM also provides fast failure detection by leveraging the fast network to exchange frequent heart-beats, and it uses priorities and pre-allocation to avoid false positives.

Our experimental results show that you can have it all: consistency, high availability, and performance. FaRM recovers from single machine failures in less than 50 ms and it outperforms state-of-the-art single-machine in-memory transactional systems with just a few machines. For example, it achieves better throughput than Hekaton [14, 26] when running on just three machines and it has both better throughput and latency than Silo [39, 40].

2. Hardware trends

FaRM's design is motivated by the availability of plentiful, cheap DRAM in data center machines. A typical data center configuration has 128–512 GB of DRAM per 2-socket machine [29], and DRAM costs less than $12/GB[1]. This means that a petabyte of DRAM requires only 2000 machines, and this is sufficient to hold the data sets of many interesting applications. In addition, FaRM exploits two hardware trends to eliminate storage and network bottlenecks: non-volatile DRAM, and fast commodity networks with RDMA.

[1] 16 GB DDR4 DIMMs on newegg.com, 21 March 2015.

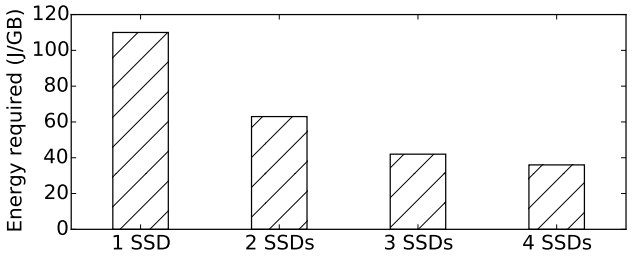


Figure 1. Energy to copy one GB from DRAM to SSD

2.1 Non-volatile DRAM

A "distributed uninterruptible power supply (UPS)" exploits the wide availability of Lithium-ion batteries to lower the cost of a data center UPS over a traditional, centralized approach that uses lead-acid batteries. For example, Microsoft's Open CloudServer (OCS) specification includes Local Energy Storage (LES) [30, 36], which integrates Li-ion batteries with the power supply units in each 24-machine chassis within a rack. The estimated LES UPS cost is less than $0.005 per Joule.[2] This approach is more reliable than a traditional UPS: Li-ion batteries are overprovisioned with multiple independent cells, and any battery failure impacts only a portion of a rack.

A distributed UPS effectively makes DRAM durable. When a power failure occurs, the distributed UPS saves the contents of memory to a commodity SSD using the energy from the battery. This not only improves common-case performance by avoiding synchronous writes to SSD, it also preserves the lifetime of the SSD by writing to it only when failures occur. An alternative approach is to use non-volatile DIMMs (NVDIMMs), which contain their own private flash, controller and supercapacitor (e.g., [2]). Unfortunately, these devices are specialized, expensive, and bulky. In contrast, a distributed UPS uses commodity DIMMs and leverages commodity SSDs. The only additional cost is the reserved capacity on the SSD and the UPS batteries themselves.

Battery provisioning costs depend on the energy required to save memory to SSDs. We measured an unoptimized prototype on a standard 2-socket machine. On failure, it turns off the HDDs and NIC and saves in-memory data to a single M.2 (PCIe) SSD, and it consumes 110 Joules per GB of data saved. Roughly 90 Joules is used to power the two CPU sockets on the machine during the save. Additional SSDs reduce the time to save data and therefore the energy consumed (Figure 1). Optimizations, like putting the CPUs into a low-power state, will further reduce energy consumption.

In the worst-case configuration, (single SSD, no optimization) at $0.005 per Joule, the energy cost of non-

[2] Li-ion is 5x cheaper than traditional lead-acid based UPS, which costs $31 million per 25 MW data center. A 25 MW data center can house 100,000 machines, and hence the Li-ion UPS cost per machine is $62. A 24-machine chassis has 6 PSUs, each with an LES that is provisioned for at least 1600 W for 5 seconds and 1425 W for a further 30 seconds, i.e. a total of 50 kJ per PSU or 12.5 kJ per machine, giving a cost per Joule of $0.0048.

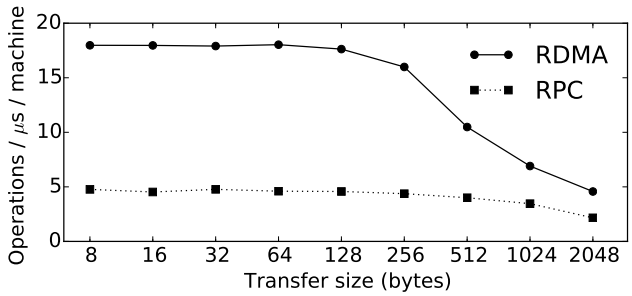


Figure 2. Per-machine RDMA and RPC read performance

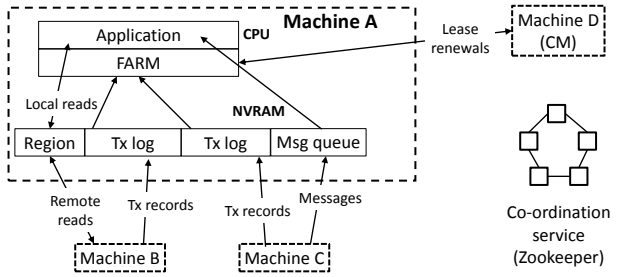


Figure 3. FaRM architecture

volatility is $0.55/GB and the storage cost of reserving SSD capacity is $0.90/GB[3]. The combined additional cost is less than 15% of the base DRAM cost, which is a significant improvement over NVDIMMs that cost 3–5x as much as DRAM. Therefore, it is feasible and cost-effective to treat all machine memory as non-volatile RAM (NVRAM). FaRM stores all data in memory, and considers it durable when it has been written to NVRAM on multiple replicas.

2.2 RDMA networking

FaRM uses one-sided RDMA operations where possible because they do not use the remote CPU. We based this decision both on our prior work and on additional measurements. In [16], we showed that on a 20-machine RoCE [22] cluster, RDMA reads performed 2x better than a reliable RPC over RDMA when all machines read randomly chosen small objects from the other machines in the cluster. The bottleneck was the NIC message rate and our implementation of RPC requires twice as many messages as one-sided reads. We replicated this experiment on a 90-machine cluster where each machine has two Infiniband FDR (56 Gbps) NICs. This more than doubles the message rate per machine when compared with [16] and eliminates the NIC message rate bottleneck. Both RDMA and RPC are now CPU bound and the performance gap increases to 4x, as seen in Figure 2.This illustrates the importance of reducing CPU overhead to realize the potential of the new hardware.

3. Programming model and architecture

FaRM provides applications with the abstraction of a global address space that spans machines in a cluster. Each machine runs application threads and stores objects in the address space. The FaRM API [16] provides transparent access to local and remote objects within transactions. An application thread can start a transaction at any time and it becomes the transaction's coordinator. During a transaction's execution, the thread can execute arbitrary logic as well as read, write, allocate, and free objects. At the end of the execution, the thread invokes FaRM to commit the transaction.

FaRM transactions use optimistic concurrency control. Updates are buffered locally during execution and only made visible to other transactions on a successful commit. Commits can fail due to conflicts with concurrent transactions or failures. FaRM provides strict serializability [35] of all successfully committed transactions. During transaction execution, FaRM guarantees that individual object reads are atomic, that they read only committed data, that successive reads of the same object return the same data, and that reads of objects written by the transaction return the latest value written. It does not guarantee atomicity across reads of different objects but, in this case, it guarantees that the transaction does not commit ensuring committed transactions are strictly serializable. This allows us to defer consistency checks until commit time instead of re-checking consistency on each object read. However, it adds some programming complexity: FaRM applications must handle these temporary inconsistencies during execution [20]. It is possible to deal with these inconsistencies automatically [12].

The FaRM API also provides lock-free reads, which are optimized single-object read only transactions, and locality hints, which enable programmers to co-locate related objects on the same set of machines. These can be used by applications to improve performance as described in [16].

Figure 3 shows a FaRM instance with four machines. The figure also shows the internal components of machine A. Each machine runs FaRM in a user process with a kernel thread pinned to each hardware thread. Each kernel thread runs an event loop that executes application code and polls the RDMA completion queues.

A FaRM instance moves through a sequence of *configurations* over time as machines fail or new machines are added. A configuration is a tuple $\langle i, S, \mathcal{F}, \text{CM}\rangle$ where i is a unique, monotonically increasing 64-bit configuration identifier, S is the set of machines in the configuration, $\mathcal{F}$ is a mapping from machines to failure domains that are expected to fail independently (e.g., different racks), and CM $\in S$ is the *configuration manager*. FaRM uses a Zookeeper [21] coordination service to ensure machines agree on the current configuration and to store it, as in Vertical Paxos [25]. But it does not rely on Zookeeper to manage leases, detect failures, or coordinate recovery, as is usually done. The CM does these using an efficient implementation that leverages RDMA to recover fast. Zookeeper is invoked by the CM once per configuration change to update the configuration.

[3] Samsung M.2 256 GB MLC, newegg.com on 25 March 2015

The global address space in FaRM consists of 2 GB regions, each replicated on one primary and f backups, where f is the desired fault tolerance. Each machine stores several regions in non-volatile DRAM that can be read by other machines using RDMA. Objects are always read from the primary copy of the containing region, using local memory accesses if the region is on the local machine and using one-sided RDMA reads if remote. Each object has a 64-bit version that is used for concurrency control and replication. The mapping of a region identifier to its primary and backups is maintained by the CM and replicated with the region. These mappings are fetched on demand by other machines and cached by threads together with the RDMA references needed to issue one-sided RDMA reads to the primary.

Machines contact the CM to allocate a new region. The CM assigns a region identifier from a monotonically increasing counter and selects replicas for the region. Replica selection balances the number of regions stored on each machine subject to the constraints that there is enough capacity, each replica is in a different failure domain, and the region is co-located with a target region when the application specifies a locality constraint. It then sends a prepare message to the selected replicas with the region identifier. If all replicas report success in allocating the region, the CM sends a commit message to all of them. This two-phase protocol ensures a mapping is valid and replicated at all the region replicas before it is used.

This centralized approach provides more flexibility to satisfy failure independence and locality constraints than our previous approach based on consistent hashing [16]. It also makes it easier to balance load across machines and to operate close to capacity. With 2 GB regions, we expect up to 250 regions on a typical machine and hence that a single CM could handle region allocation for thousands of machines.

Each machine also stores ring buffers that implement FIFO queues [16]. They are used either as transaction logs or message queues. Each sender-receiver pair has its own log and message queue, which are physically located on the receiver. The sender appends records to the log using one-sided RDMA writes to its tail. These writes are acknowledged by the NIC without involving the receiver's CPU. The receiver periodically polls the head of the log to process records. It lazily updates the sender when it truncates the log, allowing the sender to reuse space in the ring buffer.

4. Distributed transactions and replication

FaRM integrates the transaction and replication protocols to improve performance. It uses fewer messages than traditional protocols, and exploits one-sided RDMA reads and writes for CPU efficiency and low latency. FaRM uses primary-backup replication in non-volatile DRAM for both data and transaction logs, and uses unreplicated transaction coordinators that communicate directly with primaries and backups. It uses optimistic concurrency control with read validation, as in some software transactional memory systems (e.g., TL2 [15]).

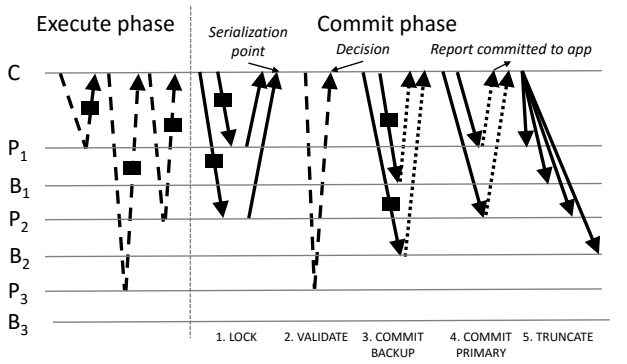


Figure 4. FaRM commit protocol with a coordinator C, primaries on P_1, P_2, P_3, and backups on B_1, B_2, B_3. P_1 and P_2 are read and written. P_3 is only read. We use dashed lines for RDMA reads, solid ones for RDMA writes, dotted ones for hardware acks, and rectangles for object data.

Figure 4 shows the timeline for a FaRM transaction and tables 1 and 2 list all log record and message types used in the transaction protocol. During the execution phase, transactions use one-sided RDMA to read objects and they buffer writes locally. The coordinator also records the addresses and versions of all objects accessed. For primaries and backups on the same machine as the coordinator, object reads and writes to the log use local memory accesses rather than RDMA. At the end of the execution, FaRM attempts to commit the transaction by executing the following steps:

1. Lock. The coordinator writes a LOCK record to the log on each machine that is a primary for any written object. This contains the versions and new values of all written objects on that primary, as well as the list of all regions with written objects. Primaries process these records by attempting to lock the objects at the specified versions using compare-and-swap, and send back a message reporting whether all locks were successfully taken. Locking can fail if any object version changed since it was read by the transaction, or if the object is currently locked by another transaction. In this case, the coordinator aborts the transaction. It writes an abort record to all primaries and returns an error to the application.

2. Validate. The coordinator performs read validation by reading, from their primaries, the versions of all objects that were read but not written by the transaction. If any object has changed, validation fails and the transaction is aborted. Validation uses one-sided RDMA reads by default. For primaries that hold more than t_r objects, validation is done over RPC. The threshold t_r (currently 4) reflects the CPU cost of an RPC relative to an RDMA read.

3. Commit backups. The coordinator writes a COMMIT-BACKUP record to the non-volatile logs at each backup and then waits for an ack from the NIC hardware without inter-

Log record type	Contents
LOCK	transaction ID, IDs of all regions with objects written by the transaction, and addresses, versions, and values of all objects written by the transaction that the destination is primary for
COMMIT-BACKUP	contents are the same as lock record
COMMIT-PRIMARY	transaction ID to commit
ABORT	transaction ID to abort
TRUNCATE	low bound transaction ID for non-truncated transactions and transaction IDs to truncate

Table 1. Log record types used in the transaction protocol. The low bound on transaction identifiers that have not been truncated and a transaction identifier for truncation are piggybacked on each record.

Message type	Contents
LOCK-REPLY	transaction ID, result indicating whether locking succeeded
VALIDATE	addresses and versions of objects read from destination (not sent when validation is done over RDMA reads)
NEED-RECOVERY	configuration ID, region ID, and transaction IDs to be recovered (sent by backup to primary)
FETCH-TX-STATE	configuration ID, region ID, and transaction IDs whose state is requested (sent by primary to backup)
SEND-TX-STATE	configuration ID, region ID, transaction ID, and contents of lock record for transaction requested by fetch
REPLICATE-TX-STATE	configuration ID, region ID, transaction ID, and contents of lock record (sent by primary to backup)
RECOVERY-VOTE	configuration ID, region ID, transaction ID, region IDs for regions modified by the transaction, and vote
REQUEST-VOTE	configuration ID, transaction ID, and region ID
COMMIT-RECOVERY	configuration ID, and transaction ID
ABORT-RECOVERY	configuration ID, and transaction ID
TRUNCATE-RECOVERY	configuration ID, and transaction ID

Table 2. Message types used in the transaction protocol. All but the first two are used only during recovery.

rupting the backup's CPU. The COMMIT-BACKUP log record has the same payload as a LOCK record.

4. Commit primaries. After all COMMIT-BACKUP writes have been acked, the coordinator writes a COMMIT-PRIMARY record to the logs at each primary. It reports completion to the application on receiving at least one hardware ack for such a record, or if it wrote one locally. Primaries process these records by updating the objects in place, incrementing their versions, and unlocking them, which exposes the writes committed by the transaction.

5. Truncate. Backups and primaries keep the records in their logs until they are truncated. The coordinator truncates logs at primaries and backups lazily after receiving acks from all primaries. It does this by piggybacking identifiers of truncated transactions in other log records. Backups apply the updates to their copies of the objects at truncation time.

Correctness. Committed read-write transactions are serializable at the point where all the write locks were acquired, and committed read-only transactions at the point of their last read. This is because the versions of all read and written objects at the serialization point are the same as the versions seen during execution. Locking ensures this for objects that were written and validation ensures this for objects that were only read. In the absence of failures this is equivalent to executing and committing the entire transaction atomically at the serialization point. Serializability in FaRM is also *strict*: the serialization point is always between the start of execution and the completion being reported to the application.

To ensure serializability across failures, it is necessary to wait for hardware acks from all backups before writing COMMIT-PRIMARY. Assume that the coordinator does not receive an ack from some backup b for a region r. Then a primary could expose transaction modifications and later fail together with the coordinator and the other replicas of r without b ever receiving the COMMIT-BACKUP record. This would result in losing the updates to r.

Since the read set is stored only at the coordinator, a transaction is aborted if the coordinator fails and no commit record survives to attest to the success of validation. So it is necessary for the coordinator to wait for a successful commit at one of the primaries before reporting a successful commit to the application. This ensures that at least one commit record survives any f failures for transactions reported committed to the application. Otherwise, such a transaction could still abort if the coordinator and all the backups failed before any COMMIT-PRIMARY record was written, because only LOCK records would survive and there would be no record that validation had succeeded.

In traditional two-phase commit protocols, participants can reserve resources to commit the transaction when they process the prepare message, or refuse to prepare the transaction if they do not have enough resources. However, as our protocol avoids involving the backups' CPUs during the commit, the coordinator must reserve log space at all participants to guarantee progress. Coordinators reserve space for all commit protocol records including truncate records in primary and backup logs before starting the commit protocol. Log reservations are a local operation at the coordinator

because the coordinator writes records to the log it owns at each participant. The reservation is released when the corresponding record is written. Truncation record reservations are also released if the truncation is piggybacked on another message. If the log becomes full, the coordinator uses the reservations to write explicit truncate records to free up space in the log. This is rare but needed to ensure liveness.

Performance. For our target hardware, this protocol has several advantages over traditional distributed commit protocols. Consider a two-phase commit protocol with replication such as Spanner's [11]. Spanner uses Paxos [24] to replicate the transaction coordinator and its participants, which are the machines that store data read or written by the transaction. Each Paxos state machine takes the role of an individual machine in a traditional two-phase commit protocol [19]. This requires $2f + 1$ replicas to tolerate f failures and, since each state machine operation requires at least $2f + 1$ round trip messages, it requires $4P(2f + 1)$ messages (where P is the number of participants in the transaction).

FaRM uses primary-backup replication instead of Paxos state machine replication. This reduces the number of copies of data to $f + 1$, and also reduces the number of messages transmitted during a transaction. Coordinator state is not replicated and coordinators communicate directly with primaries and backups, further reducing latency and message counts. FaRM's overhead due to replication is minimal: a single RDMA write to each remote machine having a backup of any written object. Backups of read-only participants are not involved in the protocol at all. Additionally, read validation over RDMA ensures that primaries of read-only participants do no CPU work, and using one-way RDMA writes for COMMIT-PRIMARY and COMMIT-BACKUP records reduces waiting for remote CPUs and also allows the remote CPU work to be lazy and batched.

The FaRM commit phase uses $P_w(f + 3)$ one-sided RDMA writes where P_w is the number of machines that are primaries for objects written by the transaction, and P_r one-sided RDMA reads where P_r is the number of objects read from remote primaries but not written. Read validation adds two one-sided RDMA latencies to the critical path but this is a good trade-off: the added latency is only a few microseconds without load and the reduction in CPU overhead results in higher throughput and lower latency under load.

5. Failure recovery

FaRM provides durability and high availability using replication. We assume that machines can fail by crashing but can recover without losing the contents of non-volatile DRAM. We rely on bounded clock drift for safety and on eventually bounded message delays for liveness.

We provide durability for all committed transactions even if the entire cluster fails or loses power: all committed state can be recovered from regions and logs stored in non-volatile DRAM. We ensure durability even if at most f replicas per object lose the contents of non-volatile DRAM. FaRM can also maintain availability with failures and network partitions provided a partition exists that contains a majority of the machines which remain connected to each other and to a majority of replicas in the Zookeeper service, and the partition contains at least one replica of each object.

Failure recovery in FaRM has five phases described below: failure detection, reconfiguration, transaction state recovery, bulk data recovery, and allocator state recovery.

5.1 Failure detection

FaRM uses leases [18] to detect failures. Every machine (other than the CM) holds a lease at the CM and the CM holds a lease at every other machine. Expiry of any lease triggers failure recovery. Leases are granted using a 3-way handshake. Each machine sends a lease request to the CM and it responds with a message that acts as both a lease grant to the machine and a lease request from the CM. Then, the machine replies with a lease grant to the CM.

FaRM leases are extremely short, which is key to high availability. Under heavy load, FaRM can use 5 ms leases for a 90-machine cluster with no false positives. Significantly larger clusters may require a two-level hierarchy, which in the worst case would double failure detection time.

Achieving short leases under load required careful implementation. FaRM uses dedicated queue pairs for leases to avoid having lease messages delayed in a shared queue behind other message types. Using a reliable transport would require an additional queue pair at the CM for each machine. This would result in poor performance due to capacity misses in the NIC's queue pair cache [16]. Instead the lease manager uses Infiniband send and receive verbs with the connectionless unreliable datagram transport, which requires space for only one additional queue pair on the NIC. By default, lease renewal is attempted every $1/5$ of the lease expiry period to account for potential message loss.

Lease renewal must also be scheduled on the CPU in a timely way. FaRM uses a dedicated lease manager thread that runs at the highest user-space priority (31 on Windows). The lease manager thread is not pinned to any hardware thread and it uses interrupts instead of polling to avoid starving critical OS tasks that must run periodically on every hardware thread. This increases message latency by a few microseconds, which is not problematic for leases.

In addition, we do not assign FaRM threads to two hardware threads on each machine, leaving them for the lease manager. Our measurements show that the lease manager usually runs on these hardware threads without impacting other FaRM threads, but sometimes it is preempted by higher priority tasks that cause it to run on other hardware threads. So pinning the lease manager to a hardware thread would likely result in false positives when using short leases.

Finally, we preallocate all memory used by the lease manager during initialization and we page in and pin all the code it uses to avoid delays due to memory management.

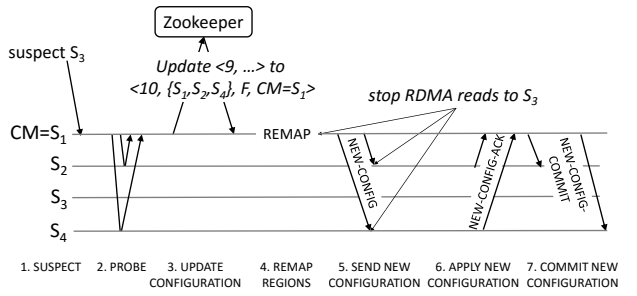


Figure 5. Reconfiguration

5.2 Reconfiguration

The reconfiguration protocol moves a FaRM instance from one configuration to the next. Using one-sided RDMA operations is important to achieve good performance but it imposes new requirements on the reconfiguration protocol. For example, a common technique to achieve consistency is to use leases [18]: servers check if they hold a lease for an object before replying to requests to access the object. If a server is evicted from the configuration, the system guarantees that the objects it stores cannot be mutated until after its lease expires (e.g., [7]). FaRM uses this technique when servicing requests from external clients that communicate with the system using messages. But since machines in the FaRM configuration read objects using RDMA reads without involving the remote CPU, the server's CPU cannot check if it holds the lease. Current NIC hardware does not support leases and it is unclear if it will in the future.

We solve this problem by implementing *precise membership* [10]. After a failure, all machines in a new configuration must agree on its membership before allowing object mutations. This allows FaRM to perform the check at the client rather than at the server. Machines in the configuration do not issue RDMA requests to machines that are not in it, and replies to RDMA reads and acks for RDMA writes from machines no longer in the configuration are ignored.

Figure 5 shows an example reconfiguration timeline that consists of the following steps:

1. Suspect. When a lease for a machine expires at the CM, it suspects that machine of failure and initiates reconfiguration. At this point it starts blocking all external client requests. If a non-CM machine suspects the CM of failure due to a lease expiry, it first asks one of a small number of "backup CMs" to initiate reconfiguration (the k successors of the CM using consistent hashing). If the configuration is unchanged after a timeout period then it attempts the reconfiguration itself. This design avoids a large number of simultaneous reconfiguration attempts if the CM fails. In all cases, the machine initiating the reconfiguration will try to become the new CM as part of the reconfiguration.

2. Probe. The new CM issues an RDMA read to all the machines in the configuration except the machine that is suspected. Any machine for which the read fails is also suspected. These *read probes* allow handling of correlated failures that affect several machines, e.g., power and switch failures, by a single reconfiguration. The new CM proceeds with the reconfiguration only if it obtains responses for a majority of the probes. This ensures that if the network is partitioned, the CM will not be in the smaller partition.

3. Update configuration. After receiving replies to the probes, the new CM attempts to update the configuration data stored in Zookeeper to $\langle c+1, S, \mathcal{F}, \text{CM}_{\text{id}} \rangle$, where c is the current configuration identifier, S is the set of machines that replied to the probes, $\mathcal{F}$ is the mapping of machines to failure domains, and CM_{id} is its own identifier. We use Zookeeper znode sequence numbers to implement an atomic compare-and-swap that succeeds only if the current configuration is still c. This ensures that only one machine can successfully move the system to the configuration with identifier $c+1$ (and become CM) even if multiple machines simultaneously attempt a configuration change from the configuration with identifier c.

4. Remap regions. The new CM then reassigns regions previously mapped to failed machines to restore the number of replicas to $f+1$. It tries to balance load and satisfy application-specified locality hints subject to capacity and failure independence constraints. For failed primaries, it always promotes a surviving backup to be the new primary to reduce the time to recover. If it detects regions that lost all their replicas or there is no space to re-replicate regions, it signals an error.

5. Send new configuration. After remapping regions, the CM sends a NEW-CONFIG message to all the machines in the configuration with the configuration identifier, its own identifier, the identifiers of the other machines in the configuration, and all the new mappings of regions to machines. NEW-CONFIG also resets the lease protocol if the CM has changed: it acts as a lease request from the new CM to each machine. If the CM is unchanged, lease exchange continues during reconfiguration to detect additional failures quickly.

6. Apply new configuration. When a machine receives a NEW-CONFIG with a configuration identifier that is greater than its own, it updates its current configuration identifier and its cached copy of the region mappings, and allocates space to hold any new region replicas assigned to it. From this point, it does not issue new requests to machines that are not in the configuration and it rejects read responses and write acks from those machines. It also starts blocking requests from external clients. Machines reply to the CM with a NEW-CONFIG-ACK message. If the CM has changed, this both grants a lease to the CM and requests a lease.

7. Commit new configuration. Once the CM receives NEW-CONFIG-ACK messages from all machines in the configuration, it waits to ensure that any leases granted in previous configurations to machines no longer in the configuration have expired. The CM then sends a NEW-CONFIG-COMMIT to all the configuration members that also acts as

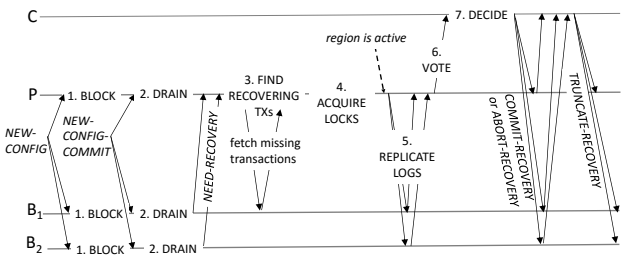


Figure 6. Transaction state recovery showing a coordinator C, primary P, and two backups B_1 and B_2

a lease grant. All members now unblock previously blocked external client requests and initiate transaction recovery.

5.3 Transaction state recovery

FaRM recovers transaction state after a configuration change using the logs distributed across the replicas of objects modified by a transaction. This involves recovering the state both at the replicas of objects modified by the transaction and at the coordinator to decide on the outcome of the transaction. Figure 6 shows an example transaction recovery timeline. FaRM achieves fast recovery by distributing work across threads and machines in the cluster. Draining (step 2) is done for all message logs in parallel. Step 1 and steps 3–5 are done for all regions in parallel. Steps 6–7 are done for all recovering transactions in parallel.

1. Block access to recovering regions. When the primary of a region fails, one of the backups is promoted to be the new primary during reconfiguration. We cannot allow access to the region until all transactions that updated it have been reflected at the new primary. We do this by blocking requests for local pointers and RDMA references to the region until step 4 when all write locks have been acquired for all recovering transactions that updated the region.

2. Drain logs. One-sided RDMA writes also impact transaction recovery. A general approach to consistency across configurations is to reject messages from old configurations. FaRM cannot use this approach because NICs acknowledge COMMIT-BACKUP and COMMIT-PRIMARY records written to transaction logs regardless of the configuration in which they were issued. Since coordinators only wait for these acks before exposing the updates and reporting success to the application, machines cannot always reject records from previous configurations when they process them. We solve this problem by *draining logs* to ensure that all relevant records are processed during recovery: all machines process all the records in their logs when they receive a NEW-CONFIG-COMMIT message. They record the configuration identifier in a variable *LastDrained* when they are done.

FaRM transactions have unique identifiers $\langle c, m, t, l\rangle$ assigned at the start of commit that encode the configuration c in which the commit started, the machine identifier m of the coordinator, the thread identifier t of the coordinator, and a thread-local unique identifier l. Log records for transactions with configuration identifiers less than or equal to *LastDrained* are rejected.

3. Find recovering transactions. A recovering transaction is one whose commit phase spans configuration changes, and for which some replica of a written object, some primary of a read object, or the coordinator has changed due to reconfiguration. During log draining, the transaction identifier and list of updated region identifiers in each log record in each log is examined to determine the set of recovering transactions. Only recovering transactions go through transaction recovery at primaries and backups, and coordinators reject hardware acks only for recovering transactions.

All machines must agree on whether a given transaction is a recovering transaction or not. We achieve this by piggybacking some extra metadata on the communication during the reconfiguration phase. The CM reads the *LastDrained* variable at each machine as part of the probe read. For each region r whose mapping has changed since *LastDrained*, the CM sends two configuration identifiers in the NEW-CONFIG message to that machine. These are *LastPrimaryChange*$[r]$, the last configuration identifier when the primary of r changed, and *LastReplicaChange*$[r]$, the last configuration identifier when any replica of r changed. A transaction that started committing in configuration $c-1$ is recovering in configuration c *unless*: for all regions r containing objects modified by the transaction *LastReplicaChange*$[r] < c$, for all regions r' containing objects read by the transaction *LastPrimaryChange*$[r'] < c$, and the coordinator has not been removed from configuration c.

Records for a recovering transaction may be distributed over the logs of different primaries and backups updated by the transaction. Each backup of a region sends a NEED-RECOVERY message to the primary with the configuration identifier, the region identifier, and the identifiers of recovering transactions that updated the region.

4. Lock recovery. The primary of each region waits until the local machine logs have been drained and NEED-RECOVERY messages have been received from each backup, to build the complete set of recovering transactions that affect the region. It then shards the transactions by identifier across its threads such that each thread t recovers the state of transactions with coordinator thread identifier t. In parallel, the threads in the primary fetch any transaction log records from backups that are not already stored locally and then lock any objects modified by recovering transactions.

When lock recovery is complete for a region, the region is *active* and local and remote coordinators can obtain local pointers and RDMA references, which allows them to read objects and commit updates to this region in parallel with subsequent recovery steps.

5. Replicate log records. The threads in the primary replicate log records by sending backups the REPLICATE-TX-STATE message for any transactions that they are missing.

The message contains the region identifier, the current configuration identifier, and the same data as the LOCK record.

6. Vote. The coordinator for a recovering transaction decides whether to commit or abort the transaction based on votes from each region updated by the transaction. These votes are sent by the primaries of each region. FaRM uses consistent hashing to determine the coordinator for a transaction, ensuring that all the primaries independently agree on the identity of the coordinator for a recovering transaction. The coordinator does not change if the machine it is running on is still in the configuration, but when a coordinator fails the responsibility for coordinating its recovering transactions is spread across the machines in the cluster.

The threads in the primary send RECOVERY-VOTE messages to their peer threads in the coordinator for each recovering transaction that modified the region. The vote is *commit-primary* if any replica saw COMMIT-PRIMARY or COMMIT-RECOVERY. Otherwise, it votes *commit-backup* if any replica saw COMMIT-BACKUP and did not see ABORT-RECOVERY. Otherwise, it votes *lock* if any replica saw a LOCK record and no ABORT-RECOVERY. Otherwise, it votes *abort*. Vote messages include the configuration identifier, the region identifier, the transaction identifier, and the list of region identifiers modified by the transaction.

Some primaries may not initiate voting for a transaction because either they never received a log record for the transaction or they already truncated the log records for the transaction. The coordinator sends explicit vote requests to primaries that have not already voted within a timeout period (set to 250 μs). The REQUEST-VOTE message includes the configuration identifier, the region identifier, and the transaction identifier. Primaries that do have log records for the transaction vote as before after first waiting for log replication for that transaction to complete.

Primaries that do not have any log records for the transaction vote *truncated* if the transaction has already been truncated and *unknown* if it has not. To determine if a transaction has already been truncated, each thread maintains the set of identifiers of transactions whose records have been truncated from its logs. This set is kept compact by using a lower bound on non-truncated transaction identifiers. The lower bound is updated based on the lower bounds at each coordinator, which are piggybacked on coordinator messages and during reconfiguration.

7. Decide. The coordinator decides to commit a transaction if it receives a *commit-primary* vote from any region. Otherwise, it waits for all regions to vote and commits if at least one region voted *commit-backup* and all other regions modified by the transaction voted *lock*, *commit-backup*, or *truncated*. Otherwise it decides to abort. It then sends COMMIT-RECOVERY or ABORT-RECOVERY to all participant replicas. Both messages include the configuration identifier and the transaction identifier. COMMIT-RECOVERY is processed similarly to COMMIT-PRIMARY if received at a primary and to COMMIT-BACKUP if received at a backup. ABORT-RECOVERY is processed similarly to ABORT. After the coordinator receives back acks from all primaries and backups, it sends a TRUNCATE-RECOVERY message.

Correctness. Next we provide some intuition on how the different steps of transaction recovery ensure strict serializability. The key idea is that recovery preserves the outcome for transactions that were previously committed or aborted. We say that a transaction is *committed* when either a primary exposes transaction modifications, or the coordinator notifies the application that the transaction committed. A transaction is *aborted* when the coordinator sends an abort message or notifies the application that the transaction has aborted. For transactions whose outcome has not yet been decided, recovery may commit or abort the transaction but it ensures that any recovery from additional failures preserves the outcome.

The outcome of transactions that are not recovering (step 3) is decided using the normal case protocol (Section 4). So we will not discuss them further.

A log record for a recovering transaction that committed is guaranteed to be processed and accepted before or during log draining (step 2). This is true because primaries expose modifications only after processing the COMMIT-PRIMARY record. If the coordinator notified the application, it must have received hardware acks for all COMMIT-BACKUP records and for at least one COMMIT-PRIMARY record before receiving NEW-CONFIG (because it ignores the acks after changing configuration). Therefore, since the new configuration includes at least one replica for each region, at least one replica for at least one region will process COMMIT-PRIMARY or COMMIT-BACKUP records, and at least one replica for each other region will process COMMIT-PRIMARY, COMMIT-BACKUP, or LOCK records.

Steps 3 and 4 ensure that the primaries for the regions modified by the transaction see these records (unless they have been truncated). They replicate these records to the backups (step 5) to guarantee that voting will produce the same results even if there are subsequent failures. Then the primaries send votes to the coordinator based on the records they have seen (step 6).

The decision step guarantees that the coordinator decides to commit any transaction that has previously committed. If any replica truncated the transaction records, all primaries will vote *commit-primary*, *commit-backup*, or *truncated*. At least one primary will send a vote other than *truncated* because otherwise the transaction would not be recovering. If no replicas truncated the transaction records, at least one primary will vote *commit-primary* or *commit-backup* and the others will vote *commit-primary*, *commit-backup* or *lock*. Similarly, the coordinator will decide to abort if the transaction was previously aborted because in this case there will either be no *commit-primary* or *commit-backup* records or all replicas will have received ABORT-RECOVERY.

Blocking access to recovering regions (step 1) and lock recovery (step 4) guarantee that until a recovering transaction has committed or aborted, no other operation can access objects it modified.

Performance. FaRM uses several optimizations to achieve fast failure recovery. Identifying recovering transactions limits its recovery work to only those transactions and regions that were affected by the reconfiguration, which could be a small subset of the total when a single machine in a large cluster fails. Our results indicate that this can reduce the number of transactions to recover by an order of magnitude. The recovery work itself is parallelized across regions, machines, and threads. Making regions available immediately after lock recovery improves foreground performance as new transactions that access these regions do not block for long. Specifically, they need not wait while new replicas of these regions are brought up to date which requires bulk movement of data over the network.

5.4 Recovering data

FaRM must recover (re-replicate) data at new backups for a region to ensure that it can tolerate f replica failures in the future. Data recovery is not necessary to resume normal case operation, so we delay it until all regions become active to minimize impact on latency-critical lock recovery. Each machine sends a REGIONS-ACTIVE message to the CM when all regions for which it is primary become active. After receiving all REGIONS-ACTIVE messages, the CM sends a message ALL-REGIONS-ACTIVE to all machines in the configuration. At this point, FaRM begins data recovery for new backups in parallel with foreground operations.

A new backup for a region initially has a freshly allocated and zeroed local region replica. It divides the region across worker threads that recover it in parallel. Each thread issues one-sided RDMA operations to read a block at a time from the primary. We currently use 8 KB blocks, which is large enough to use the network efficiently but small enough not to impact normal case operation. To reduce impact on foreground performance, recovery is paced by scheduling the next read to start at a random point within an interval after the start of the previous read (set to 4ms).

Each recovered object must be examined before being copied to the backup. If the object has a version greater than the local version, the backup locks the local version with a compare-and-swap, updates the object state, and unlocks it. Otherwise, the object has been or is being updated by a transaction that created a version greater than or equal to the one recovered, and the recovered state is not applied.

5.5 Recovering allocator state

The FaRM allocator splits regions into blocks (1 MB) that are used as slabs for allocating small objects. It keeps two pieces of meta-data: block headers, which contain the object size, and slab free lists. Block headers are replicated to backups when a new block is allocated. This ensures they are available on the new primary after a failure. Since block headers are used in data recovery, the new primary sends them to all backups immediately after receiving NEW-CONFIG-COMMIT. This avoids any inconsistencies when the old primary fails while replicating the block header.

The slab free lists are kept only at the primary to reduce the overheads of object allocation. Each object has a bit in its header that is set by an allocation and cleared by a free during transaction execution. This change to the object state is replicated during transaction commit as described in Section 4. After a failure, the free lists are recovered on the new primary by scanning the objects in the region, which is parallelized across all threads on the machine. To minimize the impact on transaction lock recovery, allocation recovery starts after ALL-REGIONS-ACTIVE is received and to minimize the impact on the foreground work it is paced by scanning 100 objects at a time every 100 µs. Object deallocations are queued until a slab's free list is recovered.

6. Evaluation

6.1 Setup

Our experimental testbed consists of 90 machines used for a FaRM cluster and 5 machines for a replicated Zookeeper instance. Each machine has 256 GB of DRAM and two 8-core Intel E5-2650 CPUs running Windows Server 2012 R2. We enabled hyper-threading and used the first 30 threads for the foreground work and the remaining 2 threads for the lease manager. Machines have two Mellanox ConnectX-3 56 Gbps Infiniband NICs, each used by threads on a different socket, and are connected by a single Mellanox SX6512 switch with full bisection bandwidth. FaRM was configured to use 3-way replication (one primary and two backups) with a lease time of 10 ms.

6.2 Benchmarks

We use two transactional benchmarks to measure FaRM's performance. We implemented both benchmarks in C++ against the FaRM API. Since FaRM uses a symmetric model to exploit locality, each machine both runs the benchmark code and stores data. Each machine runs the benchmark code linked with FaRM's code on the same process. In the future, we will compile the application from a safe language like SQL to prevent application bugs from corrupting data.

Telecommunication Application Transaction Processing (TATP) [32] is a benchmark for high-performance main-memory databases. Each database table is implemented as a FaRM hash table [16]. TATP is read dominated. 70% of the operations are single-row lookups which use FaRM's lock free reads [16]. They can usually be performed with a single RDMA read and do not require a commit phase. 10% of the operations read 2–4 rows and require validation during the commit phase. The remaining 20% of the operations are updates and require the full commit protocol. Since 70% of

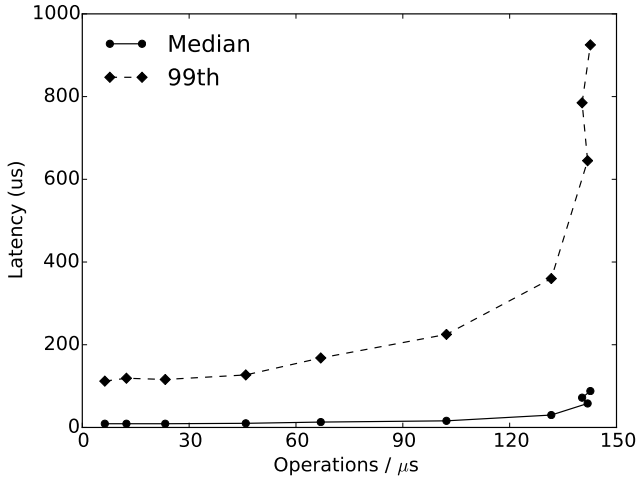


Figure 7. TATP performance

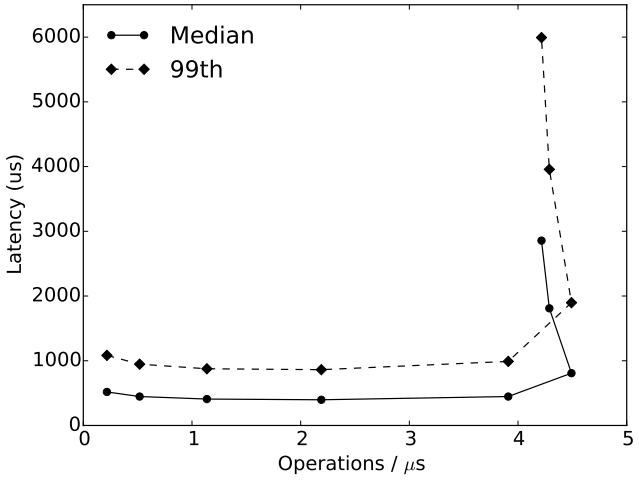


Figure 8. TPC-C performance

the updates only modify a single object field, we function ship these to the primary of the object as an optimization. We used a database with 9.2 billion subscribers (except where noted). TATP is partitionable but we have not partitioned it, so most operations access data on remote machines.

TPC-C [38] is a well-known database benchmark with complex transactions that access hundreds of rows. Our implementation uses a schema with 16 indexes. Twelve of these only require unordered (point) queries and updates and are implemented as FaRM hash tables. Four of the indexes also require range queries. These are implemented using the FaRM B-tree. The B-Tree caches internal nodes at each machine and hence lookups require a single FaRM RDMA read in the common case. We reserve 8 GB per machine for the cache. We use fence keys [17, 27] to ensure traversal consistency, similar to Minuet [37]. We omit a more detailed description of the B-tree for space reasons.

We use a database with 21,600 warehouses. We co-partition most of the hash table indexes as well as the clients by warehouse, which means that around 10% of all transactions access remote data. As specified by the benchmark, "new order" transactions are 45% of the transaction mix. We run the full mix but we report performance as the number of successfully committed "new orders".

6.3 Normal-case performance

We present the normal case (failure-free) performance of FaRM as throughput-latency curves. For each benchmark, we varied the load by first increasing the number of active threads per machine from 2 to 30 and then increasing the concurrency per thread, until the throughput saturated. Note that the left end of each graph still shows significant concurrency and hence throughput. It does not show the minimum latency that can be achieved by FaRM.

TATP. Figure 7 shows that FaRM performs 140 million TATP transactions per second with 58 µs median latency and 645 µs 99th percentile latency. On the left hand side of the graph, the median latency is only 9 µs, the 99th percentile latency drops to 112 µs, and FaRM performs 2 million operations per second. The multi-object distributed transactions used by TATP commit in tens of microseconds, with a mean commit latency of 19 µs at the lowest throughput and 138 µs at the highest.

FaRM outperforms published TATP results for Hekaton [14, 26], a single-machine in-memory transactional engine, by a factor of 33. The Hekaton results were obtained using different hardware but we expect a factor of 20 improvement when running Hekaton on one of our testbed machines. In a smaller-scale experiment, FaRM outperformed Hekaton with just three machines. In addition, FaRM supports much larger data sets because it scales out and it provides high availability unlike single machine systems.

TPC-C. We ran TPC-C for 60 s and we report latency and average throughput over that period in Figure 8. FaRM performs up to 4.5 million TPC-C "new order" transactions per second with median latency of 808 µs and 99th percentile latency of 1.9 ms. The latency can be halved with a small 10% impact in throughput. The best published TPC-C performance we know of is from Silo [39, 40] which is a single-machine in-memory system with logging to FusionIO SSDs. FaRM's throughput is 17x higher than Silo without logging, and its latency at this throughput level is 128x better than Silo with logging.[4]

Read performance. Although the focus of this paper is on transactional performance and failure recovery, we were also able to improve read-only performance relative to [16]. We ran a key-value lookup-only workload with 16-byte keys and 32-byte values and a uniform access pattern. We achieved a throughput of 790 million lookups/s with median latency

[4] Silo reports total transaction counts which we multiplied by 45% to get the "new order" count.

of 23 µs and 99th percentile latency of 73 µs. This improves on previously reported per-machine throughput for the same benchmark by 20% [16]. We do not double performance despite doubling the number of NICs because the benchmark becomes CPU bound.

6.4 Failures

To evaluate performance with failures, we ran the same benchmarks and we killed the FaRM process on one of the machines 35 s into the experiment. We show timelines with the throughput of the 89 surviving machines aggregated at 1 ms intervals. The timelines are synchronized at experiment start using RDMA messaging.

Figures 9 and 10 show a typical run of each benchmark on different time scales. Both show throughput as a solid line. The "time to full throughput" is a zoomed-in view around the failure. It shows the time at which the failed machine's lease expired on the CM ("suspect"); the time at which all read probes completed ("probe"); the time at which the CM successfully updated Zookeeper ("zookeeper"); the time at which the new configuration was committed at all surviving machines ("config-commit"); the time at which all regions are active ("all-active"); and the time at which background data recovery begins ("data-rec-start"). The "time to full data recovery" shows a zoomed-out view that includes the time when all data is recovered at backups ("done"). A dashed line shows the cumulative number of backup regions recovered over time by data recovery.

TATP. The timelines for a typical TATP run are shown in Figure 9. We configured it for maximum throughput: each machine runs 30 threads with 8 concurrent transactions per thread. Figure 9(a) shows that throughput drops sharply at the failure but recovers rapidly. The system is back to peak throughput in less than 40 ms. All regions become active in 39 ms. Figure 9(b) shows that data recovery, which is paced, does not impact foreground throughput. The failed machine hosted 84 2 GB regions. Each thread fetches 8 KB blocks every 2 ms, which means that it takes around 17 s to recover a 2 GB region on a single machine. Machines recover one region at a time in parallel with each other and at roughly the same pace, hence the number of regions recovered moves in large steps. The recovery load (i.e., the number of regions per-machine that had a replica on the failed machine) is well balanced across the cluster: 64 machines recover one region and 10 machines recover two. This explains why re-replication of most regions completes in around 17 s and why all regions are fully re-replicated in less than 35 s. Some regions are not fully allocated, so their recovery takes less time. This is why re-replication of some regions completes in less than 17 s.

The figure also shows that TATP has some dips in throughput even when there are no failures. We believe that this is because of skewed access in the benchmark; the throughput drops when many transactions conflict and back off on hot keys at the same time.

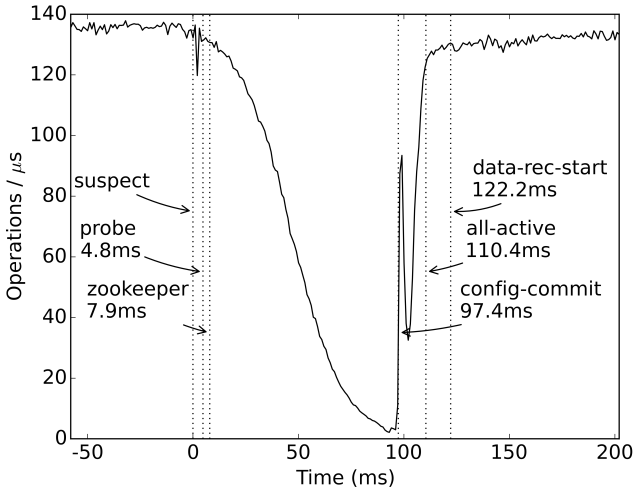
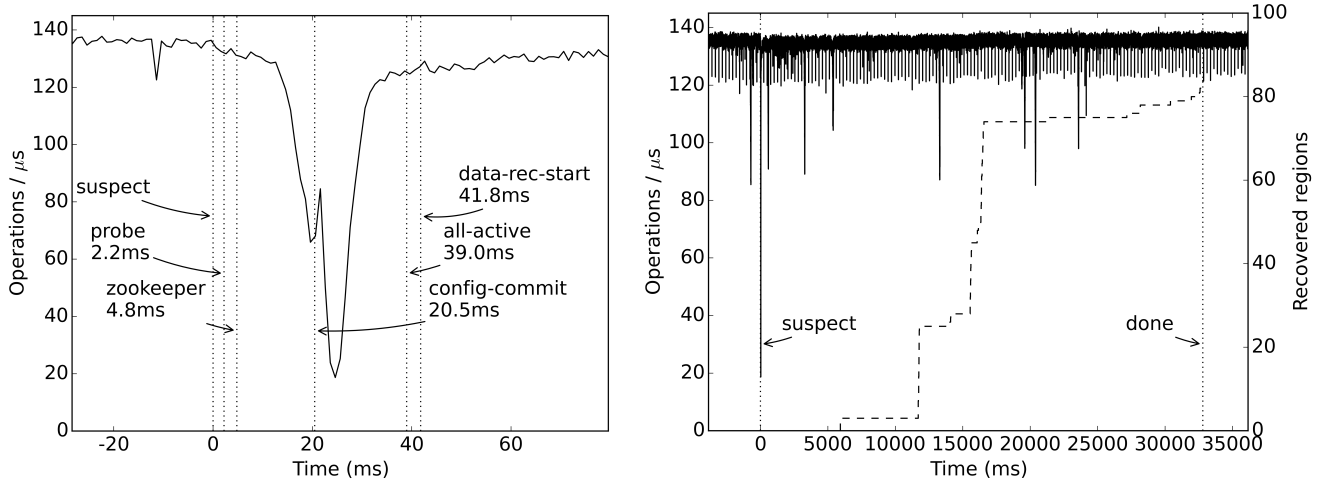


Figure 11. TATP performance timeline with CM failure

TPC-C. Figure 10 shows the timelines for TPC-C. Figure 10(a) shows that the system regains most of the throughput in less than 50 ms and that all regions become active shortly after that. It takes the system slightly more time to recover transaction locks than with TATP because TPC-C has more complex transactions. The main difference is that recovery of data takes longer (Figure 10(b)) even though TPC-C recovers only 63 regions in the experiment. This is because TPC-C co-partitions its hash tables to exploit locality and improve performance, which results in reduced recovery parallelism because multiple regions are replicated on the same set of machines to satisfy the locality constraints specified by the application. In the experiment, two machines recover 17 regions each, which leads to data recovery taking over 4 minutes. Note that TPC-C throughput degrades gradually over time in Figure 10(b) because the size of the database increases very quickly.

Failing the CM. Figure 11 shows TATP throughput over time when the CM process fails. Recovery is slower than when a non-CM process fails. It takes about 110 ms for throughput to get back to the same level as before the failure. The main reason for the increase in recovery time is an increase in the reconfiguration time: from 20 ms in Figure 9(a) to 97 ms. Most of this time is spent by the new CM building data structures that are only maintained at the CM. It should be possible to eliminate this delay by having all the machines maintain these data structures incrementally as they learn region mappings from the CM.

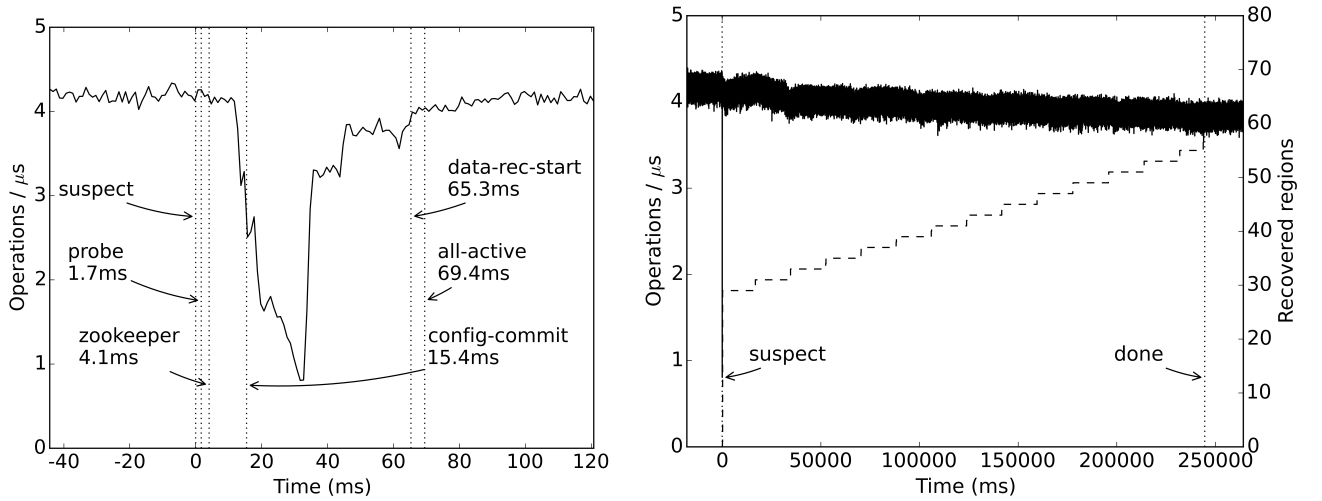
Distribution of recovery times. We repeated the TATP recovery experiment (without CM failures) 40 times to obtain a distribution of recovery times. The experiments were run with a smaller data set (3.5 billion subscribers) to shorten experiment times, but we confirmed that the time to regain



(a) Time to full throughput

(b) Time to full data recovery

Figure 9. TATP performance timeline with failure



(a) Time to full throughput

(b) Time to full data recovery

Figure 10. TPC-C performance timeline with failure

throughput after a failure was the same as for the larger data sets. This is because this time is dominated by recovering transaction state, and the number of concurrently executing transactions is the same for both data set sizes. Figure 12 shows the distribution of recovery times. We measured recovery time from the point where the failed machine is suspected by the CM until throughput recovers to 80% of the average throughput before the failure. The median recovery time is around 50 ms and in more than 70% of the executions the recovery time is less than 100 ms. In the remaining cases, the recovery took more than 100 ms, but always less than 200 ms.

Correlated failures. Some failures affect more than one machine at the same time, e.g., power or switch failures. To deal with such coordinated failures, FaRM allows specifying a failure domain for each machine and the CM places each replica of a region in a different failure domain. We group machines in our cluster into five failure domains with 18 machines each. This corresponds to the number of ports in each leaf module in our switch. We fail all the processes in one of these failure domains at the same time to simulate the failure of a top-of-rack switch.

Figure 13 shows TATP throughput over time for the 72 machines that do not fail. TATP was configured to use around 55 regions on each machine (6.9 billion subscribers

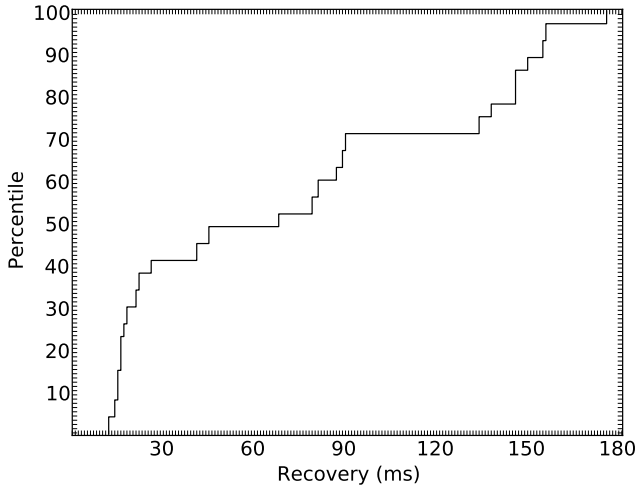


Figure 12. Distribution of recovery times for TATP

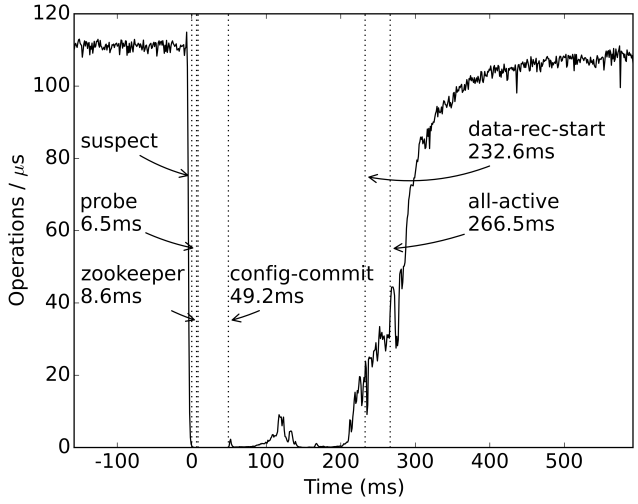


Figure 13. TATP throughput when failing 18 out of 90 machines at the same time

across the cluster) to allow enough space to re-replicate failed regions after the failure. FaRM regains peak throughput less than 400 ms after the failure. We repeated the experiment 20 times and this time was the median of all experiments. Most of this time is spent recovering transactions. We need to recover all in-flight transactions that modified any region with a replica in a failed machine, that read a region with the primary in a failed machine, or that had the coordinator on one of the failed machines. This results in roughly 130,000 transactions that need to be recovered, compared to 7500 with a single failure. Re-replication of data takes 4 minutes because there are 1025 regions to re-replicate. As in previous experiments, this does not impact throughput during recovery because of pacing. Note that during this time each region still has two available replicas, so there is no need to re-replicate more aggressively.

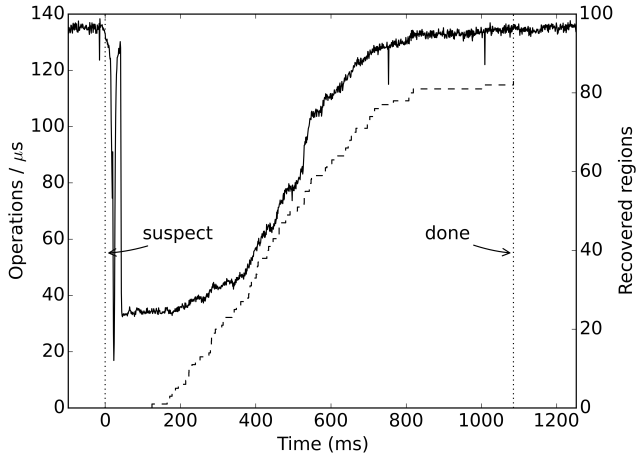


Figure 14. TATP throughput when optimizing for re-replication delay

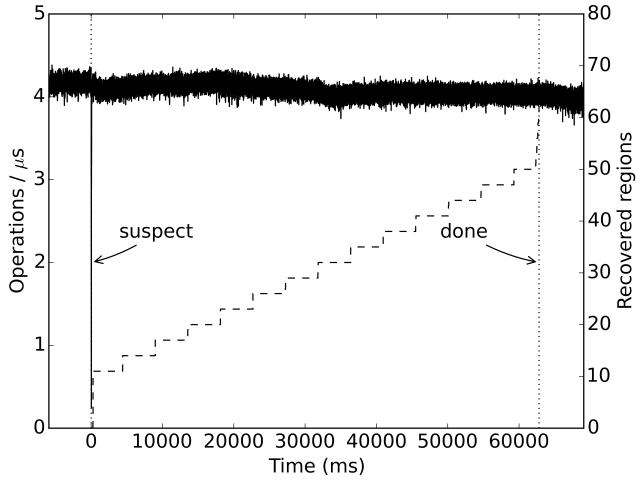


Figure 15. TPC-C throughput with more aggressive data recovery

Data recovery pacing. FaRM paces data recovery to reduce its impact on throughput. This increases the time to complete re-replication of regions at new backups. Figure 14 shows throughput over time for TATP with very aggressive data recovery: each thread fetches four 32 KB blocks concurrently. The system only recovers peak throughput after the majority of regions are re-replicated 800 ms after the failure. However, data recovery completes much faster: recovering 83 region replicas (166 GB) takes just 1.1 s. We use this aggressive recovery setting only when regions lose all but one replica. The aggressive recovery rate compares favorably with RAMCloud [33] which recovers 35 GB on 80 machines in 1.6 s.

TPC-C is less sensitive to interference from background recovery traffic than TATP because only a small fraction of accesses are to objects on remote machines. This means that, in settings in which application-specific tuning is possible, we could re-replicate data more aggressively without

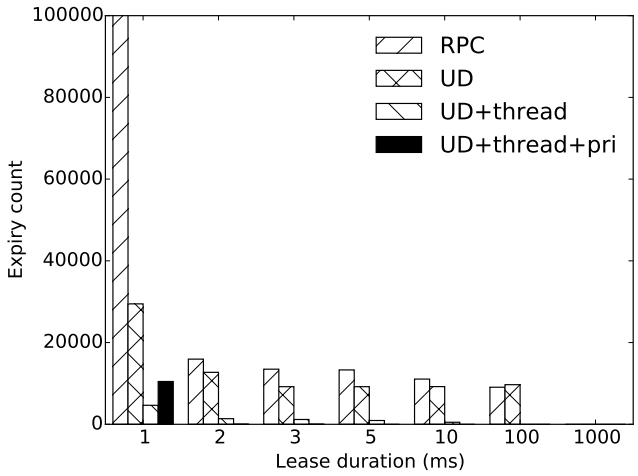


Figure 16. False positives with different lease managers

impacting performance. Figure 15 shows TPC-C throughput over time during recovery when threads fetch 32 KB blocks every 2 ms. Re-replication completes in 65 s, which is four times faster than with the default settings, without any impact on throughput.

6.5 Lease times

To evaluate our lease manager optimizations (Section 5.1), we ran an experiment where all threads in all machines repeatedly issue RDMA reads to the CM for 10 min. We disabled recovery and counted the number of (false positive) lease expiry events across the cluster for different lease manager implementations and different lease durations. This benchmark is a good stress test because it generates more traffic at the CM than any of the benchmarks we described.

Figure 16 compares four lease manager implementations. The first uses FaRM's RPC (RPC). The others use unreliable datagrams: on a shared thread (UD), on a dedicated thread at normal priority (UD+thread), and with high-priority, interrupts and no pinning (UD+thread+pri).

The results show that all the optimizations are necessary to enable using lease times of 10 ms or less without false positives. With shared queue pairs, even 100 ms leases expire very often. The number of false positives is reduced by using unreliable datagrams but it is not eliminated due to contention for the CPU. Using a dedicated thread allows us to use 100 ms leases with no false positives, but 10 ms leases still expire due to CPU contention from background processes running on the FaRM machines. With the interrupt-driven lease manager running at high priority, we can use 5 ms leases for 10 min with no false positives. With shorter leases, we still sometimes have false positives. We are limited by the network round trip time, which was up to 1 ms with load, and by the resolution of the system timer, which is 0.5 ms. The limited resolution of the system timer explains why the interrupt-driven lease manager has more false positives than the polling-based one with 1 ms leases.

We conservatively set the leases to 10 ms in all our experiments and have not observed any false positives during their execution.

7. Related work

To our knowledge, FaRM is the first system to simultaneously provide high availability, high throughput, low latency, and strict serializability. In prior work [16], we provided an overview of an early version of FaRM that logged to SSDs for durability and availability but we did not describe recovery from failures. This paper describes a new fast recovery protocol and an optimized transaction and replication protocol that sends significantly fewer messages and leverages NVRAM to avoid logging to SSDs. The optimized protocol sends up to 44% fewer messages than the transaction protocol described in [16] and also replaces messages by one-sided RDMA reads during the validation phase. The work in [16] only evaluated the performance of single-key transactions in the absence of failures using the YCSB benchmark. Here we evaluate the performance of transactions with and without failures using the TATP and TPC-C benchmarks.

RAMCloud [33, 34] is a key-value store that stores a single copy of data in memory and uses a distributed log for durability. It does not support multi-object transactions. On a failure, it recovers in parallel on multiple machines, and during this period, which can take seconds, the data on failed machines is unavailable. FaRM supports transactions, makes data available within tens of milliseconds of a failure, and has an order of magnitude higher throughput per machine.

Spanner [11] was discussed in Section 4. It provides strict serializability but is not optimized for performance over RDMA. It uses $2f + 1$ replicas compared to FaRM's $f + 1$, and sends more messages to commit than FaRM. Sinfonia [8] offers a shared address space with serializable transactions implemented using 2-phase commit and piggybacking reads into the 2-phase commit in specialized cases. FaRM offers general distributed transactions optimized to take advantage of RDMA.

HERD [23] is an in-memory RDMA-based key-value store that delivers high performance per server in an asymmetric setting where clients run on different machines from servers. It uses RDMA writes and send/receive verbs for messaging but does not use RDMA reads. The authors of [23] show that one-sided RDMA reads perform worse than a specialized RPC implementation without reliability in an asymmetric setting. Our results use reliable communication in a symmetric setting where every machine is both a client and a server. This allows us to exploit locality, which is important because accessing local DRAM is significantly faster than using RDMA to access remote DRAM [16]. Pilaf [31] is a key-value store that uses RDMA reads. Neither Pilaf nor HERD support transactions. HERD is not fault tolerant whereas Pilaf gets durability but not availability by logging to a local disk.

Silo [39, 40] is a single-machine main-memory database that achieves durability by logging to persistent storage. It writes committed transactions to storage in batches to achieve high throughput. Failure recovery involves reading checkpoints and log records from storage. The storage in Silo is local and thus availability is lost when the machine fails. In contrast, FaRM is distributed and uses replication in NVRAM for durability and high availability. FaRM can regain peak throughput after a failure more than two orders of magnitude faster than Silo for a much larger database. By scaling out and using replication in NVRAM, FaRM also achieves higher throughput and lower latency than Silo. Hekaton [14, 26] is also a single-machine main-memory database without support for scale-out or distributed transactions. FaRM with 3 machines matches Hekaton's performance and with 90 machines has 33x the throughput.

8. Conclusion

Transactions make it easier to program distributed systems but many systems avoid them or weaken their consistency to improve availability and performance. FaRM is a distributed main memory computing platform for modern data centers that provides strictly serializable transactions with high throughput, low latency, and high availability. Key to achieving this are new transaction, replication, and recovery protocols designed from first principles to leverage commodity networks with RDMA and a new, inexpensive approach to providing non-volatile DRAM. The experimental results show that FaRM provides significantly higher throughput and lower latency than state of the art in-memory databases. FaRM can also recover from a machine failure back to providing peak throughput in less than 50 ms, making failures transparent to applications.

Acknowledgments

We would like to thank Jason Nieh, our shepherd, and the anonymous reviewers for their comments. We would also like to thank Richard Black for his help in performance debugging, Andy Slowey and Oleg Losinets for keeping the test cluster running, and Chiranjeeb Buragohain, Sam Chandrashekar, Arlie Davis, Orion Hodson, Flavio Junqueira, Richie Khanna, James Lingard, Samantha Lüber, Knut Magne Risvik, Tim Tan, Ming Wu, Ming-Chuan Wu, Fan Yang, and Lidong Zhou for innumerous discussions and for letting us use the whole cluster for extended periods of time to run the final experiments.

References

[1] Memcached. http://memcached.org.

[2] Viking Technology. http://www.vikingtechnology.com/.

[3] Apache Cassandra. http://cassandra.apache.org/, 2015.

[4] MySQL. http://www.mysql.com/, 2015.

[5] neo4j. http://neo4j.com/, 2015.

[6] redis. http://redis.io/, 2015.

[7] ADYA, A., DUNAGAN, J., AND WOLMAN, A. Centrifuge: Integrated lease management and partitioning for cloud services. In *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation* (2010), NSDI'10.

[8] AGUILERA, M. K., MERCHANT, A., SHAH, M., VEITCH, A., AND KARAMANOLIS, C. Sinfonia: A new paradigm for building scalable distributed systems. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles* (2007), SOSP'07.

[9] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation* (2006), OSDI'06.

[10] CHOCKLER, G. V., KEIDAR, I., AND VITENBERG, R. Group communication specifications: a comprehensive study. *ACM Computing Surveys (CSUR) 33*, 4 (2001).

[11] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J. J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., HSIEH, W. C., KANTHAK, S., KOGAN, E., LI, H., LLOYD, A., MELNIK, S., MWAURA, D., NAGLE, D., QUINLAN, S., RAO, R., ROLIG, L., SAITO, Y., SZYMANIAK, M., TAYLOR, C., WANG, R., AND WOODFORD, D. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation* (2012), OSDI'12.

[12] DALESSANDRO, L., AND SCOTT, M. L. Sandboxing transactional memory. In *Proceedings of the 21st ACM International Conference on Parallel Architectures and Compilation Techniques* (2012), PACT'12.

[13] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon's highly available key-value store. In *Proceedings of the the 21st ACM Symposium on Operating Systems Principles* (2007), SOSP'07.

[14] DIACONU, C., FREEDMAN, C., ISMERT, E., LARSON, P.-Å., MITTAL, P., STONECIPHER, R., VERMA, N., AND ZWILLING, M. Hekaton: SQL Server's memory-optimized OLTP engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2013), SIGMOD'13.

[15] DICE, D., SHALEV, O., AND SHAVIT, N. Transactional locking II. In *Proceedings of the 20th International Symposium on Distributed Computing* (2006), DISC'06.

[16] DRAGOJEVIĆ, A., NARAYANAN, D., HODSON, O., AND CASTRO, M. FaRM: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (2014), NSDI'14.

[17] GRAEFE, G. Write-optimized B-trees. In *Proceedings of the 30th International Conference on Very Large Data Bases* (2004), VLDB'04.

[18] GRAY, C., AND CHERITON, D. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. *SIGOPS Operating Systems Review (OSR) 23*, 5 (1989).

[19] GRAY, J., AND REUTER, A. *Transaction Processing: Concepts and Techniques*. 1992.

[20] GUERRAOUI, R., AND KAPALKA, M. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2008), PPoPP'08.

[21] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Annual Technical Conference* (2010), USENIX ATC'10.

[22] INFINIBAND TRADE ASSOCIATION. Supplement to InfiniBand Architecture Specification Volume 1 Release 1.2.2 Annex A16: RDMA over Converged Ethernet (RoCE), 2010.

[23] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Using RDMA efficiently for key-value services. In *Proceedings of the 2014 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (2014), SIGCOMM'14.

[24] LAMPORT, L. The part-time parliament. *ACM Transactions on Computer Systems 16*, 2.

[25] LAMPORT, L., MALKHI, D., AND ZHOU, L. Vertical Paxos and primary-backup replication. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing* (2009), PODC'09.

[26] LARSON, P.-Å., BLANAS, S., DIACONU, C., FREEDMAN, C., PATEL, J. M., AND ZWILLING, M. High-performance concurrency control mechanisms for main-memory databases. *PVLDB 5*, 4 (2011).

[27] LEHMAN, P. L., AND YAO, S. B. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems 6*, 4 (Dec. 1981).

[28] MICROSOFT. Scaling out SQL Server. http://www.microsoft.com/en-us/server-cloud/solutions/high-availability.aspx.

[29] MICROSOFT. Open CloudServer OCS V2 specification: Blade, 2014.

[30] MICROSOFT. OCS Open CloudServer power supply v2.0. http://www.opencompute.org/wiki/Server/SpecsAndDesigns, 2015.

[31] MITCHELL, C., YIFENG, G., AND JINYANG, L. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *Proceedings of the 2013 USENIX Annual Technical Conference* (2013), USENIX ATC'13.

[32] NEUVONEN, S., WOLSKI, A., MANNER, M., AND RAATIKKA, V. Telecom Application Transaction Processing benchmark. http://tatpbenchmark.sourceforge.net/.

[33] ONGARO, D., RUMBLE, S. M., STUTSMAN, R., OUSTERHOUT, J., AND ROSENBLUM, M. Fast crash recovery in RAMCloud. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (2011), SOSP'11.

[34] RUMBLE, S. M., KEJRIWAL, A., AND OUSTERHOUT, J. Log-structured Memory for DRAM-based Storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies* (2014), FAST'14.

[35] SETHI, R. Useless actions make a difference: Strict serializability of database updates. *JACM 29*, 2 (1982).

[36] SHAUN HARRIS. Microsoft reinvents datacenter power backup with new Open Compute project specification. http://blogs.msdn.com/b/windowsazure/archive/2012/11/13/windows-azure-benchmarks-show-top-performance-for-big-compute.aspx, 2015.

[37] SOWELL, B., GOLAB, W. M., AND SHAH, M. A. Minuet: A scalable distributed multiversion B-tree. *PVLDB 5*, 9 (2012).

[38] TRANSACTION PROCESSING PERFORMANCE COUNCIL (TPC). TPC benchmark C: Standard specification. http://www.tpc.org.

[39] TU, S., ZHENG, W., KOHLER, E., LISKOV, B., AND MADDEN, S. Speedy transactions in multicore in-memory databases. In *Proceedings of the 24th Symposium on Operating Systems Principles* (2013), SOSP'13.

[40] ZHENG, W., TU, S., KOHLER, E., AND LISKOV, B. Fast databases with fast durability and recovery through multicore parallelism. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation* (2014), OSDI'14.