Fast General Distributed Transactions with Opacity

Alex Shamis*
Microsoft Research

Matthew Renzelmann
Microsoft

Stanko Novakovic†
VMware

Georgios Chatzopoulos‡
EPFL

Aleksandar Dragojević
Microsoft Research

Dushyanth Narayanan
Microsoft Research

Miguel Castro
Microsoft Research

ABSTRACT

Transactions can simplify distributed applications by hiding data distribution, concurrency, and failures from the application developer. Ideally the developer would see the abstraction of a single large machine that runs transactions sequentially and never fails. This requires the transactional subsystem to provide *opacity* (strict serializability for both committed and aborted transactions), as well as transparent fault tolerance with high availability. As even the best abstractions are unlikely to be used if they perform poorly, the system must also provide high performance.

Existing distributed transactional designs either weaken this abstraction or are not designed for the best performance within a data center. This paper extends the design of FaRM — which provides strict serializability only for committed transactions — to provide opacity while maintaining FaRM's high throughput, low latency, and high availability within a modern data center. It uses timestamp ordering based on real time with clocks synchronized to within tens of microseconds across a cluster, and a failover protocol to ensure correctness across clock master failures. FaRM with opacity can commit 5.4 million neworder transactions per second when running the TPC-C transaction mix on 90 machines with 3-way replication.

*Also with Imperial College London.
†Work done while at Microsoft Research.
‡Work done while at Microsoft Research.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD '19, June 30-July 5, 2019, Amsterdam, Netherlands
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-5643-5/19/06…$15.00
https://doi.org/10.1145/3299869.3300069

KEYWORDS

distributed transactions, RDMA, opacity, global time, clock synchronization, multi-version concurrency control

ACM Reference Format:
Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojević, Dushyanth Narayanan, and Miguel Castro. 2019. Fast General Distributed Transactions with Opacity. In *2019 International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3299869.3300069

1 INTRODUCTION

Cloud data centers provide many relatively small, individually unreliable servers. Cloud services need to run on clusters of such servers to maintain availability despite individual server failures. They also need to scale out to increase throughput beyond that of a single server. For latency-sensitive applications that need to keep data in main memory, scale-out is also required to go beyond the memory limits of a single server.

The challenge is that distributed applications, especially stateful ones, are much harder to program than single-threaded or even multi-threaded applications. Our goal is to make them easier to program by providing the abstraction of a single large machine that runs transactions one at a time and never fails. This requires a distributed transactional system with the following properties:

- *Serializability*: All executions are equivalent to some serial ordering of committed transactions.
- *Strictness*: This ordering is consistent with real time.
- *Snapshot reads*: All transactions see a consistent snapshot of the database until they commit or abort.
- *High availability*: The system recovers transparently from server failures and downtimes are short enough to appear as transient dips in performance.

The combination of the first three properties is also referred to as *opacity* [13]. Intuitively, opacity extends the properties of strict serializability to aborted transactions, i.e. these

transactions also see a consistent snapshot at a point in time consistent with real-time ordering, until they abort.

As even the best abstractions are unlikely to be used if they perform poorly, the system must also provide scalability and high performance. Existing designs either weaken this abstraction or are not designed for the best performance within a data center. Spanner [6] is a geo-distributed database that provides opacity with availability but does not provide low latency and high throughput in the data center. Several transactional systems [4, 10, 20, 35] have leveraged large amounts of cheap DRAM per server, fast commodity networking hardware, and RDMA to achieve good performance in the data center. RDMA can improve networking throughput and latency by orders of magnitude compared to TCP [9]. One-sided RDMA can reduce CPU costs and latency further compared to two-way messaging using RDMA, as it bypasses the remote CPU. Several distributed transactional protocols [4, 10, 35] use one-sided RDMA to send fewer messages and achieve higher performance than two-phase commit (2PC) within the data center.

Current designs that use one-sided RDMA do not provide opacity. FaRMv1 [9, 10] and DrTM [4, 33] provide scalability, availability, and strict serializability for committed but not for aborted transactions. Optimistically executing transactions in these systems might read inconsistent state with the guarantee that such transactions would eventually abort. NAM-DB [35] provides read snapshots but not strictness, serializability, or high availability.

In this paper, we describe FaRMv2, which extends the original design and implementation of FaRMv1 to provide read snapshots to all transactions. FaRMv2 uses a novel timestamp-ordering protocol that leverages the low latency of RDMA to synchronize clocks. Timestamps are based on real time, which scales well as it allows machines to use their local clocks to generate timestamps. However, since clocks are not perfectly synchronized, the transaction protocol must "wait out the uncertainty" when generating read and write timestamps, which introduces latency. FaRMv2 leverages low-latency, CPU-efficient RDMA-based communication to synchronize clocks frequently over the network to achieve uncertainties in the tens of microseconds, two orders of magnitude lower than in Spanner [6]. Unlike Spanner, FaRMv2 does not require atomic clocks or GPS. Instead, servers use the CPU cycle counter and synchronize with a clock master elected from all the servers in the system. Timestamp ordering is maintained across clock master failures using a clock master failover protocol. Our design and implementation also supports multi-versioning which improves the performance of read-only transactions. Old versions are kept in memory with efficient allocation and garbage collection.

The paper makes the following novel contributions:

- A mechanism to synchronize clocks to within tens of microseconds by leveraging RDMA.
- A transaction protocol with opacity that uses global time and one-sided RDMA.
- A clock failover protocol that keeps timestamps monotonic across clock master failures without requiring special hardware such as atomic clocks or GPS.
- An efficient thread-local, block-based allocator and garbage collector for multi-versioning.

FaRMv2 can commit 5.4 million neworder transactions per second when running the TPC-C transaction mix on a cluster of 90 machines with 3-way replication for fault tolerance. It retains the high availability of FaRMv1 and can recover to full throughput within tens of milliseconds of a server failure. We believe FaRMv2 has the highest known throughput for this transaction mix of any system providing opacity and high availability.

2 MOTIVATION

Serializability is an easy isolation level for developers to understand because it avoids many anomalies. We found that *strictness* and *opacity* are also important for developers using a transactional platform.

Strict serializability [28] means that the serialization order of transactions corresponds to real time. If A completes before B starts, then any correct execution must be equivalent to a serial one where A appears before B. This is important when clients communicate using some channel outside the system as is often the case, for example, when other systems are layered on top of the database.

Opacity [13] is the property that transaction executions are strictly serializable for aborted transactions as well as committed transactions. This simplifies programming by ensuring that invariants hold during transaction execution. Many existing systems provide opacity either by using pessimistic concurrency control with read locks (e.g., Spanner [6]), or by using timestamp ordering to provide read snapshots during execution (e.g., Hekaton [7, 22]). But many systems that use optimistic concurrency control (OCC) [21] do not provide read snapshots for aborted transactions, e.g., FaRMv1 [9, 10] and DrTM [4, 33]. This design decision can improve performance but it imposes a large burden on developers. Since developers cannot assume invariants hold, they must program defensively by checking for invariants explicitly in transaction code. Relational databases reduce this burden by providing mechanisms to check constraints automatically after each SQL statement, but this can add a non-trivial performance overhead and it still requires the developer to write all the relevant constraints.

FaRM and DrTM provide a low level transactional memory model rather than a relational model. This allows developers

to write highly performant C++ code to manipulate arbitrary pointer-linked data structures in transactions, but this flexibility comes at a price. It is not feasible to do efficient automatic constraint checking after every C++ statement. Additionally, lack of opacity can lead to violations of memory safety. For example, a transaction could read memory that has been freed and reused, which could cause a crash or an infinite loop. We illustrate the difficulty of programming without opacity by discussing the implementation of hash table and B-tree indices on top of FaRM.

The FaRM hash table [9] uses chained associative hopscotch hashing. Each key lookup reads two adjacent array buckets and zero or more overflow buckets. FaRMv1 ensures that each of these objects is read atomically, but they may not all be read from the same consistent snapshot because FaRMv1 does not ensure opacity. This can lead to several anomalies, for example, a concurrent transaction could move key *A* from an overflow bucket to an array bucket while deleting key *B*, causing the lookup transaction to incorrectly miss *A*. FaRMv1 solves this problem by adding 64-bit incarnations to all object headers, replicating them in all overflow bucket pointers, and adding additional version fields to each array bucket. This adds complexity and overhead, which could be avoided by providing opacity.

The FaRM B-Tree implementation keeps cached copies of internal nodes at each server to improve performance. Fence keys [12, 24] are used to check the consistency of parent-to-child traversals. Strict serializability is maintained by always reading leaf nodes uncached and adding them to the read set of the transaction. The cached internal nodes are shared read-only across all threads without making additional thread-local or transaction-local copies. This is extremely efficient as most lookups require a single uncached read, and do not make copies of internal nodes.

Lack of opacity can lead to several anomalies when using this B-tree, for example, one developer reported that they had found a bug because "They inserted a key into a B-Tree, looked up the same key in the same transaction, and it was not there." On investigation, we found that this was possible when a concurrent transaction created a split in the B-Tree that migrated the key in question (*A*) to a new leaf object, deleted *A*, and the server running the original transaction evicted some internal nodes on the path from the root to the new leaf from the cache. Even though the original transaction would have aborted in this case, the programmer still needs to reason about execution before a transaction aborts. Reasoning about complex corner cases like this one is hard. Opacity simplifies programming by providing strong isolation guarantees even for transactions that abort.

In this paper, our contribution is adding opacity to FaRM to improve programmability while retaining good performance. It is hard to quantify the benefit of providing a better developer experience. Based on our deployment experience — more than two years of FaRMv1 followed by more than two years of FaRMv2 — we can say that our developers praised the addition of opacity and we no longer see bug reports due to opacity violations. We were also able to remove the additional version fields per array bucket in the hash table and convert the "fat pointers" for overflow chains to normal pointers, which simplified the code and reduced space usage.

3 BACKGROUND

3.1 FaRM

FaRM [9, 10] provides a transactional API to access objects in a global flat address space that pools together the memory of a cluster. The API is exposed as library calls, and both application code and FaRM run within the same process on each machine. Within a transaction, the application can allocate, free, read and write objects regardless of their location in the cluster, as well as execute application logic. The thread executing the code for a transaction also acts as the coordinator for the distributed commit of that transaction. The execution model is symmetric: all threads in a cluster can be coordinators and all servers can hold in-memory objects.

FaRM objects are replicated using primary-backup replication. The unit of replication is a *region* (e.g., 2 GB). All objects in a region have the same primary and backup servers.

FaRM implements optimistic concurrency control to enable using one-sided RDMA to read objects from remote primaries during transaction execution. Locking remote objects would require using the remote CPU or using additional atomic RDMA operations. So no locks are taken during transaction execution. Writes are buffered within the transaction context. At the end of the execution phase, the application calls COMMIT invoking the commit protocol. The commit protocol integrates concurrency control and replication for fault-tolerance to achieve lower message counts and fewer round trips than approaches which build distributed commit on top of replication [6]. The commit protocol first locks write sets at their primary replicas and then *validates* read sets to ensure serializability.

FaRM has transparent fault tolerance with high availability through fast failure detection, a reconfiguration protocol for adding/removing machines, parallel transaction recovery after failure, and background data re-replication to restore replication levels. Unlike traditional 2PC, FaRM does not block transactions when a coordinator fails: coordinator state is recovered in parallel from logs on participants.

3.2 One-sided RDMA

CPU is the bottleneck when accessing in-memory data using the fast networking hardware deployed in data centers today. So FaRM uses one-sided RDMA operations, which are

handled entirely by the remote NIC, to improve performance. Remote objects are read using RDMA reads from their primary replica during transaction execution and read set validation uses RDMA reads of object versions from the primary. Unlike traditional 2PC protocols, primaries of read-only participants do no CPU work in FaRM as RDMA requests are served by their NICs. Backups of read-only participants do no work on the CPU or on the NIC. Backups of write-set objects do not do any CPU work on the critical path of the transaction; coordinators do a single one-sided RDMA write to each backup to commit a transaction, and only wait for the hardware acknowledgement from the NIC. This commit message is processed asynchronously by the backup's CPU.

There has been a lively debate on the merits of one-sided RDMA [17–20, 32]. The key issue is that one-sided operations and deployed congestion control mechanisms [26, 37] require the reliable connected (RC) mode of RDMA. This requires per-connection (queue pair) state which grows with the size of the cluster, and degrades performance when the state cannot be cached on the NIC and must be fetched from host memory instead. Sharing connection state across cores can improve scalability but adds CPU synchronization costs [9].

An alternative approach, eRPC [17], uses connectionless unreliable datagrams (UD). These scale better at the NIC level as a single endpoint (queue pair) can send and receive to all servers in the cluster [17, 18, 20]. This approach requires two-sided messaging, as one-sided RDMAs are not supported over UD. It uses an RTT-based congestion control mechanism [26] implemented in software.

The scalability of RC has been improving with newer NICs. The RDMA performance of the Mellanox CX3 starts dropping at 256 connections per machine. More recent NICs (CX4, CX5, and CX6) have better scalability. We measured the scalability of RDMA reads on CX4 RoCE NICs between two machines connected by a 100 Gbps switch. We emulated larger clusters by increasing the number of queue pairs per machine. We compared this with the performance of 64-byte reads over eRPC using one queue pair per thread, which is 15 million reads/s. The RDMA throughput is 35 million reads/s with 28 queue pairs per machine, and RDMA reads perform better than eRPC with up to 3200 queue pairs per machine, where they equal eRPC performance. We do not have CX5 hardware and could not measure the performance of 64-byte reads on CX5. With a small number of queue pairs, RDMA writes [17] and reads [16] on CX5 have up to 2x higher throughput than eRPC for reads between 512 bytes and 32 KB. For larger transfers both approaches are limited by the line rate.

These results and other recent work [32] show that one-sided RDMA can provide a significant performance advantage for moderate size clusters. So we added opacity to FaRM while retaining all the one-sided RDMA optimizations.

4 DESIGN

4.1 Global time

Using one-sided RDMA reads during execution makes it challenging to provide opacity with scalability. Pessimistic concurrency control schemes such as Spanner [6] provide opacity by using read locks but this requires two-way messaging and remote CPU usage on read-only participants.

Timestamp ordering enables opacity by allowing transactions to read a consistent snapshot defined by a read timestamp. The challenge is to generate timestamps scalably and with global monotonicity, i.e. the timestamp order must match the real time order in which the timestamps were generated across all servers.

Centralized sequencers do not scale to our target transaction rates. A state of the art centralized sequencer without fault tolerance [19] can generate 122 million timestamps per second. FaRMv1 can execute 140 million TATP transactions per second on 90 machines [10].

NAM-DB [35] uses a centralized timestamp server and caches timestamps at servers to avoid generating new timestamps per transaction. This improves scalability but it means that timestamps are not globally monotonic: timestamps generated on different servers will not be in real time order. Using non-monotonic timestamps as transaction read and write timestamps violates strictness.

Clock-SI [11] has no centralized time server but uses loosely synchronized physical clocks on each server. Remote reads in Clock-SI block until the remote server's clock moves past the transaction read timestamp. This operation is not supported on RDMA NICs and requires two-sided messaging. Clock-SI is also not globally monotonic.

Spanner [6] uses Marzullo's algorithm [25] to maintain globally synchronized real time. Servers synchronize their local clocks with a time master periodically. The algorithm accounts for synchronization uncertainty explicitly by representing time as an interval, which is computed using the round trip time of synchronization requests, the master time returned by requests, and an assumed bound on the rate drift of the local clock. Time masters use atomic clocks and/or GPS that are synchronized with global real time, and there are multiple clock masters for fault tolerance. Servers synchronize with clock masters every 30 seconds. Spanner's uncertainties are 1–7 ms, which is acceptable in a geo-distributed database where latencies are dominated by WAN round trips.

These uncertainties are too high for FaRM, which is designed for sub-millisecond transaction latencies using low latency RDMA networks to scale out within a data center. We also did not want to depend on atomic clocks and GPS as they are not widely available in all data centers.

We also use timestamp ordering based on real time, with clock synchronization using Marzullo's algorithm [25] but

$SyncState \leftarrow \emptyset$
function SYNC
  $S_{new}.T_{send} \leftarrow$ LOCALTIME()
  $S_{new}.T_{CM} \leftarrow$ MASTERTIME() ▷ this is an RPC
  $S_{new}.T_{recv} \leftarrow$ LOCALTIME()
  $Sync \leftarrow Sync \cup \{S_{new}\}$
function TIME
  $T \leftarrow$ LOCALTIME()
  $L \leftarrow max_{S \in Sync}(S.T_{CM} + (T - S.T_{recv})(1 - \epsilon))$
  $U \leftarrow min_{S \in Sync}(S.T_{CM} + (T - S.T_{send})(1 + \epsilon))$
  return $[L, U]$

Figure 1: Synchronization with Marzullo's algorithm

with a design and implementation that provide an average uncertainty below 20 microseconds, two orders of magnitude lower than in Spanner. We use only the cycle counters present on all CPUs, allowing any server in a cluster to function as a clock master (CM) without additional hardware.

Non-clock masters periodically synchronize their clocks with the current clock master using low-latency, CPU-efficient RPCs based on RDMA writes [9]. Round-trip times are in the tens of microseconds even under load and a single clock master can handle hundreds of thousands of synchronization requests per second while also running the application without noticeable impact on application throughput.

As shown in Figure 1, each successful synchronization records the local cycle counter value before sending the request and after receiving the response, and the time at the clock master. This synchronization state and the current local cycle counter value are used to compute a time interval $[L, U]$ that contains the current time at the clock master. The width of the interval — the uncertainty — is the RTT of the synchronization plus a drift factor. The drift factor is the clock drift bound ϵ multiplied by twice the time elapsed since the synchronization request was sent.

State from any valid past synchronization can be used to compute L and U. The smallest correct time interval $[L, U]$ is the intersection of all the intervals computed from all valid past synchronizations. Naively this would require keeping an unbounded amount of state. We optimize this by only remembering two past synchronizations: one that gives the best lower bound and one that gives the best upper bound. This returns the same bounds but keeps bounded state.

Synchronization state is kept in a shared memory data structure on each machine. Any thread can read this state to compute a time interval. We also allow any thread to synchronize with the clock master and update the state. For threads that also run application code, these requests are interleaved non-preemptively with execution of application code. This leads to scheduling delays in processing synchronization requests and responses causing high uncertainty. Hence in practice, we only send synchronization requests from a single, high-priority thread that does not run application code. FaRMv1 already uses such a high-priority thread to manage leases for failure detection, and we use the same thread to send synchronization requests in FaRMv2.

We assume that cycle counters are synchronized across threads on a server to some known precision. On our platform, the OS (Windows) synchronizes the cycle counters across threads to within 1024 ticks (about 400 ns). This uncertainty is included in the interval returned to the caller.

Time intervals in FaRMv2 are globally monotonic: if an event at time $[L_1, U_1]$ anywhere in the cluster happens-before an event at time $[L_2, U_2]$ anywhere in the cluster, then $U_2 > L_1$. We also guarantee that on any thread, the left bound L is non-decreasing.

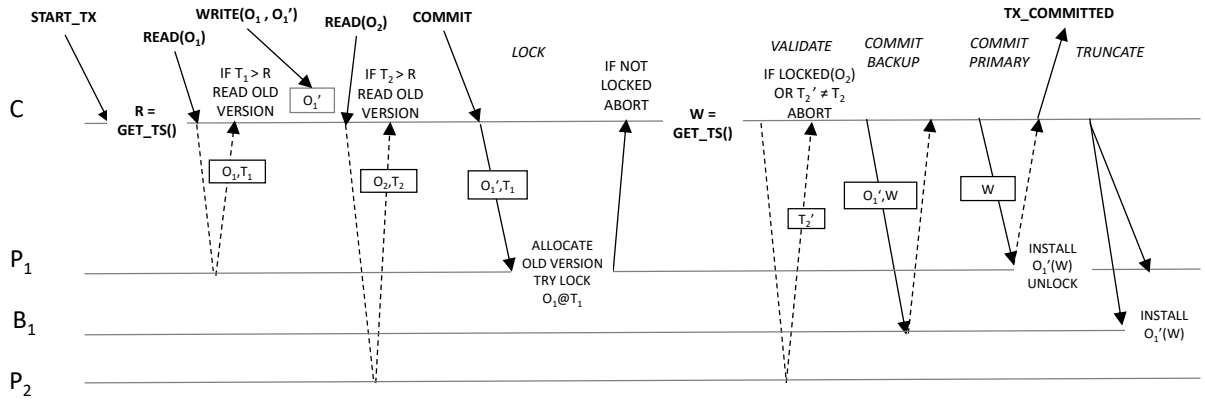
Frequent synchronization allows us to use conservative clock drift bounds. Currently we use a bound of 1000 parts per million (ppm), i.e. 0.1%. This is at least 10x higher than the maximum allowed by the hardware specification on our servers and at least 10x higher than the maximum observed rate drift across 6.5 million server hours on our production clusters with more than 700 machines.

Correctness in FaRMv2 requires clock frequencies to stay within these bounds. We use the local CPU clock on each machine, which on modern hardware is based on extremely accurate and reliable crystal oscillators. In rare cases, these can be faulty at manufacture: we detect these cases using an initial probation period when a server is added to the cluster, during which we monitor clock rates but do not use the server. Clock rates can also change slowly over time due to aging effects and temperature variation (e.g., [15]). FaRMv2 continuously monitors the clock rate of each non-CM relative to the CM. If this exceeds 200 ppm (5x more conservative than the bound we require for correctness), it is reported to a centralized service that removes either the non-CM or the CM, if the CM is reported by multiple servers.

4.2 FaRMv2 commit protocol

Figure 2 shows FaRMv2's transaction protocol as a time diagram for one example transaction. The line marked C shows the coordinator thread for this transaction. The other lines show other servers with primaries and backups of objects accessed by the transaction. FaRMv2 uses primary-backup replication. In this example the transaction reads two objects O_1 and O_2 and writes a new value O_1' to O_1. Each object is replicated on one primary and one backup. The backup for O_2 is not shown as backups of objects that are read but not written do not participate in the protocol.

Transactions obtain a read timestamp when they start executing and transactions that modify objects obtain a write



Execution and commit timeline for a transaction that reads two objects and writes one of them. Solid arrows show RDMA writes. Dashed arrows show RDMA reads and hardware acks for RDMA writes.

Figure 2: FaRMv2 commit protocol

function GET_TS
  $[L, U] \leftarrow$ TIME() ▷ U is in the future here.
  SLEEP($(U - L)(1 + \epsilon)$) ▷ Wait out uncertainty.
  return U ▷ U is now in the past.

ϵ is the clock drift bound.

Figure 3: Timestamp generation (strict serializability)

timestamp when they commit. FaRMv2 serializes transactions in timestamp order: read-only transactions are serialized by their read timestamp and read-write transactions are serialized by their write timestamp.

The application starts a transaction by calling START_TX, which acquires the read timestamp R. The reads of the transaction are then performed at time R, i.e., successful reads see the version with the highest write timestamp that is less than or equal to R. The time at the CM must exceed R before the first read can be issued, to ensure that no future writes can have a timestamp less than or equal to R. This is necessary for opacity. The time at the CM must also be less than or equal to R at the time START_TX is invoked. This is necessary for strictness, to avoid reading a stale snapshot. FaRMv2 satisfies both conditions by setting R to the upper bound of the time interval when START_TX is called and waiting out the uncertainty in this interval before the first read is issued (Figure 3).

The application can then issue reads and writes for objects in the global address space. Reads are always from the primary replica. If the read object has a timestamp greater than R, FaRMv2 will find and return the correct snapshot version of the object (not shown in the figure).

Writes to objects are buffered locally. When the application calls COMMIT, the coordinator sends LOCK messages to all primaries that have objects in the write set. If the versions of all these objects equal the versions read and they could be successfully locked, the coordinator acquires a write timestamp W. W must be greater than the time at the CM when all write locks are taken, to ensure that any conflicting reader with read timestamp $R' \geq W$ will see either the lock or the eventual commit. W must also be less than the time at the CM when read validation begins, to ensure that the transaction will see any conflicting writer with write timestamp $W' \leq W$. Both conditions are satisfied by computing W using the procedure shown in Figure 3: acquiring a timestamp in the future and then waiting until the timestamp has gone into the past.

Note that we do not send LOCK messages to backups. This makes writes more efficient but it means that reads of the latest version of an object must always be from the primary, preventing reads from backups for load balancing. In FaRM, we rely on sharding for load-balancing: each physical machine is the primary for some shards and the backup for other shards and thus reads to different shards can be balanced across different machines.

The coordinator then validates objects that were read but not written, using RDMA reads to re-read object versions from the primaries. Validation succeeds only if all such objects are unlocked and at the version read by the transaction. The coordinator then sends COMMIT-BACKUP messages to the backups of the write-set objects using RDMA writes. When all COMMIT-BACKUP messages have been acknowledged by the remote NICs, the coordinator sends COMMIT-PRIMARY messages to the primaries of the write-set objects using RDMA writes. Finally, TRUNCATE messages are sent to write-set primaries and backups to clean up per-transaction state maintained on those servers. These are almost always piggybacked on other messages. On receiving COMMIT-PRIMARY, primaries install the new versions of write-set objects and unlock them. Backups install the new values when they receive TRUNCATE.

This protocol retains all the one-sided RDMA optimizations in FaRMv1's commit protocol with no added communication. It adds two uncertainty waits which are executed locally on the coordinator. As in FaRMv1, the protocol can also exploit locality, i.e. co-locating the coordinator with the primaries. If all primaries are co-located with the coordinator then only COMMIT-BACKUP and piggybacked TRUNCATE messages are sent and there is no remote CPU involvement at all on the critical path.

In addition, FaRMv2 skips validation for all read-only transactions, which was not possible in FaRMv1. Committing a read-only transaction in FaRMv2 is a no-op. This can reduce the number of RDMA reads issued by read-only transactions by up to 2x compared to FaRMv1.

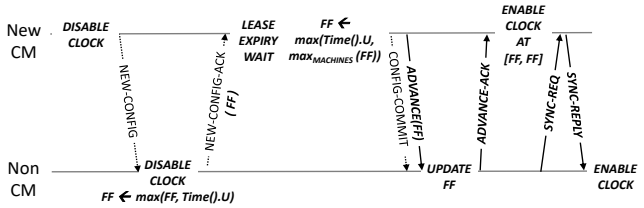
By default FaRMv2 transactions are strictly serializable. FaRMv2 also supports snapshot isolation (SI) and non-strict transactions. It does not support weaker isolation levels than snapshot isolation. Non-strictness and SI can be set per transaction: developers need only use this for transactions where they will improve performance significantly without affecting application correctness. SI and non-strictness are implemented with small changes to the protocol shown in Figure 2.

SI transactions in FaRMv2 skip the validation phase for read-only objects (written objects are implicitly validated during the lock phase, which fails if the object is not at the version read). The uncertainty wait of write timestamp acquisition is deferred until just before returning TX_COMMITTED to the application. Thus write locks are not held until the uncertainty wait completes, reducing contention. The uncertainty wait can also be overlapped with COMMIT-BACKUP and COMMIT-PRIMARY, improving latency.

Strictness can be relaxed in FaRMv2 both for serializable and SI transactions. For SI transactions, strictness means that if transaction A starts after transaction B, then the read timestamp of A is greater than or equal to the write timestamp of B. Non-strict transactions choose the lower bound L on the time interval $[L, U]$ as the read timestamp when START_TX is called, without any uncertainty wait. Non-strict SI transactions compute their write timestamp as their upper bound U of the timer interval at the point of write timestamp acquisition, again without any uncertainty wait. The uncertainty wait for the write timestamp is required for serializable read-write transactions, whether strict or non-strict.

4.3 Fast-forward for failover

FaRMv2 maintains global monotonicity across clock master failures. When the clock master fails or is removed, a new clock master is chosen. As we do not rely on externally synchronized time hardware such as atomic clocks or GPS, the new clock master must continue based on a time interval obtained by synchronizing with a previous clock master.



$Time()$ returns the current time interval $[L, U]$ on the coordinator.

Figure 4: Clock recovery after clock master failure

Adding the uncertainty in this interval to all time intervals generated in the future would maintain monotonicity, but uncertainty could grow without bound.

FaRMv2 shrinks the uncertainty on a new clock master during clock master failover. It uses a fast-forward protocol that we integrated with FaRM's reconfiguration protocol [10] which is used to add or remove servers from the cluster.

In FaRM, a configuration is identified by a unique sequence number and specifies the membership of the cluster, with one member distinguished as the "configuration manager" (CM) for the configuration. Leases are used for failure detection. Each non-CM maintains a lease at the CM and the CM maintains a lease at each non-CM. Lease renewals are periodically initiated by each non-CM via a 3-way handshake which renews both the non-CM's and the CM's leases.

Configurations are stored in Zookeeper and are changed by atomically incrementing the sequence number and installing the new configuration. A configuration change is initiated by the current CM if it suspects a non-CM has failed or a new server joins the cluster, and by a non-CM if it suspects the CM has failed. The server initiating the configuration change, if successful, becomes the new CM. After committing the configuration change to Zookeeper, the new CM uses a 2-phase protocol to commit the new configuration to all the servers in the new configuration. This mechanism handles single and multiple node failures as well as network partitions. If there is a set of connected nodes with the majority of the nodes from the previous configuration, with at least one replica of each object, and with at least one node that can update the configuration in Zookeeper, then a node from this partition will become the CM. Otherwise, the system will block until this condition becomes true.

In our design, the CM also functions as the clock master. This lets us reuse messages in the reconfiguration protocol for clock master failover, and lease messages for clock synchronization. Figure 4 shows the reconfiguration/fast-forward protocol. Dotted lines show existing messages in the reconfiguration protocol; bold lines show messages that were added for clock failover. For simplicity, the figure omits the interaction with Zookeeper (which remains unchanged) and shows only one non-CM. To implement fast-forward,

each server maintains a local variable *FF* marking the last time clocks were fast-forwarded. The new CM first disables its own clock: the clock continues to advance, but timestamps are not given out and synchronization requests from other servers are rejected. It then sends a NEW-CONFIG message to all non-CMs. On receiving NEW-CONFIG, each non-CM disables its clock and sets *FF* to the maximum of its current value and the upper bound on the current time interval. This updated value is piggybacked on the NEW-CONFIG-ACK sent back to the CM.

After receiving all acks from all non-CMs, the CM waits for one lease expiry period which ensures that servers removed from the configuration have stopped giving out timestamps. The CM then advances *FF* to the maximum of the upper bound of its current time interval and the maximum *FF* on all servers in the new configuration including itself. It then commits the new configuration by sending CONFIG-COMMIT, sends *FF* to all non-CMs (ADVANCE), and waits for acknowledgements (ADVANCE-ACK). After receiving acks from all non-CMs, the CM enables its clock with the time interval set to $[FF, FF]$. The additional round trip to propagate *FF* ensures that time moves forward even if the new CM fails immediately after enabling its own clock. After receiving CONFIG-COMMIT, non-CMs send periodic synchronization requests to the new CM. On the first successful synchronization, a non-CM clears all previous synchronization state, updates the synchronization state and enables its clock.

This protocol disables clocks for up to three round trips plus one lease expiry period (e.g., 10 ms) in the worst case. While clocks are disabled, application threads requesting timestamps are blocked. If the old CM was not removed from the configuration, then it remains the CM and we do not disable clocks or fast-forward. If only the old CM was removed (no other servers fail), the "lease expiry wait" is skipped as we know that the old CM's lease has already expired. Adding new servers to the configuration does not disable clocks. The worst-case clock disable time is incurred when the CM and at least one non-CM fail at the same time.

Fast-forward can cause the FaRMv2 clock to diverge from external time. If synchronization with external time is important, this can be done by "smearing", i.e., gradually adjusting clock rates without at any point violating the drift bound assumptions. Currently FaRMv2 does not do smearing as there is no need for timestamps to match external time.

4.4 Multi-versioning

FaRMv2 supports multi-version concurrency control [2]. This can help reduce aborts in read-only transactions caused by conflicting writes. Multi-versioning is implemented using a per-object, in-memory linked list of old versions in decreasing timestamp order. This is optimized for the case where most objects have no old versions, and those that do have only a few. These are reasonable assumptions for our current production workloads which have interactive read-only queries and transactional updates but not batch analytics that take hours or days. For example, we have built a distributed graph database, A1 [3] on FaRMv2 which must serve graph queries within 50 ms as part of a larger service.

Reading an object always starts at the *head version* whose location does not change during the lifetime of the object. This ensures that we can always read the head version using a single one-sided RDMA without indirection.

Each head version in FaRMv2 has a 128-bit header *H* which contains a lock bit *L*, a 53-bit timestamp *TS*, and (in multi-version mode) an old-version pointer *OVP*. *TS* contains the write timestamp of the last transaction to successfully update the object. *L* is used during the commit protocol to lock objects. *OVP* points to a singly linked list of older versions of the object, ordered by decreasing timestamp. Other header bits are inherited from FaRM [9, 10].

If the head version timestamp is beyond the transaction's read timestamp, the linked list is traversed, using RDMA reads if the version is remote, until a version with a timestamp less than or equal to the read timestamp is found. Old versions also have a 128-bit header but only use the *TS* and *OVP* fields. Old versions are allocated from globally-addressable, unreplicated, RDMA-readable regions with primaries co-located with the primaries of their head versions.

Old version memory is allocated in 1 MB blocks carved out of 2 GB regions. Blocks are owned and accessed by a single thread which avoids synchronization overheads. Each thread has a currently active block to which allocation request are sent until the block runs out of space. Within the block we use a bump allocator that allocates the bytes in the block sequentially. Allocating an old version thus requires one comparison and one addition, both thread-local, in the common case.

An old version is allocated when a thread on a primary receives a LOCK message. The thread allocates space for the old version, locks the head version, copies the contents as well as the timestamp and old-version pointer of the head version to the old version, and then acknowledges to the coordinator. When COMMIT-PRIMARY is received, a pointer to the old version is installed at the head version before unlocking. As allocation is fast, the dominant cost of creating old versions is the memory copy. This copy is required in order to keep the head version's location fixed, which lets us support one-sided RDMA reads.

FaRMv2 also supports stale snapshot reads, which are read-only transactions that can execute with a specified read timestamp *R* which can be in the past, i.e. less than the current time lower bound *L*. This is intended to support distributed read-only transactions that can speed up large queries by

parallelizing them across the cluster and also partitioning them to exploit locality. A master transaction that acquires read timestamp R can send messages to other servers to start slave transactions at time R, ensuring that all the transactions execute against the same snapshot. As R may be in the past when the message is received, this requires stale snapshot reads.

4.5 Garbage collection

FaRMv2 garbage-collects entire blocks of old-version memory without touching the object headers or data in the block. A block is freed by the thread that allocated it to a thread-local free block pool. No synchronization is required unless the free block pool becomes too large (at which point blocks are freed to a server-wide pool).

Each old version O has a GC time that is equal to the write timestamp of the transaction that allocated O. If the transaction that allocated O aborted, then the GC time of O is zero. The GC time of a block is the maximum GC time of all old versions in the block. It is kept in the block header and updated when transactions commit. A block can be freed and reused when its GC time is less than the GC safe point GC, which must be chosen such that no transaction will attempt to read old versions in the block after it is freed.

To compute GC, FaRMv2 uses the periodic lease renewal messages to propagate information about the oldest active transaction (OAT) in the system. Each thread maintains a local list of currently executing transactions with that thread as coordinator, and tracks the oldest read timestamp of these: this is the per-thread OAT. When sending a lease request, each non-CM includes the minimum of this value across all threads, and of the lower bound of the current time interval: this is the per-machine OAT. The CM tracks the per-machine OAT of all machines in the configuration including itself. The global OAT is the minimum of all of these values and is included in all lease responses to non-CMs.

Each non-CM thus has a slightly stale view of the global OAT, which is guaranteed to be less than or equal to the global OAT and will catch up to the current value of the global OAT on the next lease renewal. The global OAT is guaranteed to be less than or equal to the read timestamp of any currently running transaction in the system. It is also guaranteed to be less than or equal to the lower bound L of the time interval on any thread in the system. L is non-decreasing on every thread and transactions created in the future, both strict and non-strict, will have a read timestamp greater than L.

It is not safe to use global OAT as the GC safe point GC in the presence of stale snapshot reads and failures. The coordinator of a master transaction with read timestamp R can fail after sending a message to execute a slave transaction but before the slave transaction has been created. The global OAT could then advance beyond R, causing a block to be freed that is then read by a slave transaction, violating opacity. FaRMv2 uses a second round of propagation to solve this problem. The global OAT value received by each server is propagated to all threads, and then included in lease requests. The minimum global OAT value known to all servers is tracked at the CM and sent back on lease responses. This value is then used as the safe GC point GC.

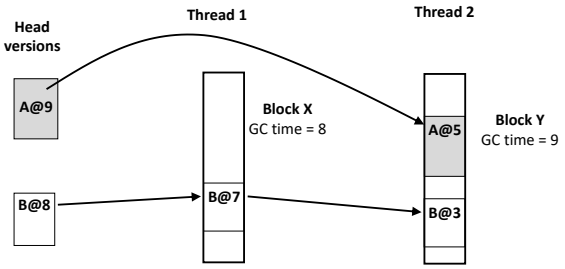


Figure 5: Old versions in FaRMv2

Stale snapshot reads can only be created on a thread if the timestamp R is greater than or equal to the thread's view of global OAT; otherwise START_TX returns an error. The global OAT on any thread is guaranteed to be greater than equal to the safe GC point on any other thread: this guarantees that stale snapshot reads will never read freed memory.

Figure 5 shows a simple example with two objects A and B with one and two old versions respectively, in two blocks owned by two different threads. The list for a given object is in decreasing timestamp order and the list pointers can cross block and thread boundaries arbitrarily. Old versions of the same object can be freed out of order and the list is not guaranteed to be null-terminated. We use OAT and GC to ensure that readers never follow a pointer to an old version that has already been freed.

4.6 Early aborts

FaRMv2 keeps primary but not backup copies of old versions. With n-way replication, this reduces the space and CPU overhead of multi-versioning by a factor of n. When a primary for a region fails and a backup is promoted to primary, the new primary will have no old versions for objects in that region. Readers attempting to read the old version will abort. This is a transient condition: if the transaction is retried it will be able to read the latest version of O, which exists at the new primary. As we expect failures to be relatively infrequent we allow such "early aborts" during failures in return for reduced common-case overheads.

Unlike the opacity violations in FaRMv1 described at the beginning of this section, these early aborts do not require significant additional developer effort. Developers need not

write code to check invariants after every read, detect infinite loops, or handle use-after-free scenarios: all of which were required when using FaRMv1. The code must already handle aborts during the commit phase due to optimistic concurrency control, e.g., by retrying the transaction. Early aborts can be handled in the same way.

Allowing early aborts also lets us do *eager validation* [27] as a performance optimization. If a serializable transaction with a non-empty write set attempts to read an old version, then FaRMv2 fails the read and aborts the transaction even if the old version is readable, as the transaction will eventually fail read validation. We also allow applications to hint that a serializable RW transaction is likely to generate writes; in this case we abort the transaction when it tries to read an old version even if the write set is currently empty.

For some workloads, old versions are accessed so infrequently that the cost of multi-versioning outweighs the benefit. FaRMv2 can operate in single-version mode. In this mode, it does not maintain old versions even at the primaries.

5 EVALUATION

In this section, we measure the throughput and latency of FaRMv2 and compare it to a baseline system without opacity. We then measure the costs and benefits of multi-versioning. Finally, we demonstrate FaRMv2's scalability and high availability.

5.1 Setup

Our experimental testbed consists of up to 90 machines. As we did not always have access to all 90 machines, all experiments used 57 machines unless stated otherwise. Each machine has 256 GB of DRAM and two 8-core Intel E5-2650 CPUs (with hyper-threading enabled) running Windows Server 2016 R2. FaRM runs in one process per machine with one OS thread per hardware thread (hyperthread). We use 15 cores for the foreground work and 1 core for lease management and clock synchronization. By default clocks are synchronized at an aggregate rate of 200,000 synchronizations per second, divided evenly across all non CMs. Each machine has one Mellanox ConnectX-3 56 Gbps Infiniband NIC, connected to a single Mellanox SX6512 switch with full bisection bandwidth. All graphs show average values across 5 runs with error bars showing the minimum and maximum across the 5 runs. Most experiments ran for 60 seconds after a warmup period, but the availability experiments ran for 10 minutes.

We use two benchmarks: the first is TPC-C [30], a well-known database benchmark with transactions that access hundreds of rows. Our implementation uses a schema with 16 indexes. Twelve of these only require point queries and updates and are implemented as hash tables. Four of the

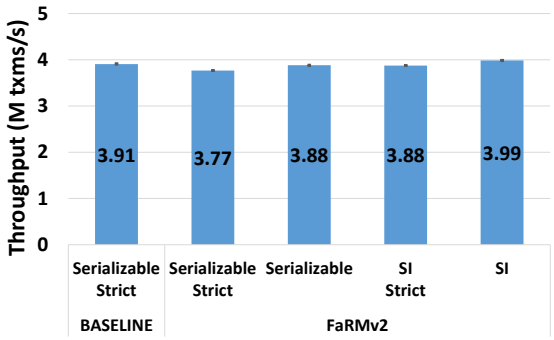


Figure 6: TPC-C throughput

indexes also require range queries and are implemented as B-Trees. We load the database with 240 warehouses per server, scaling the database size with the cluster size. We partition all tables by warehouse except for the small, read-only *ITEM* table which is replicated to all servers. We run the full TPC-C transaction mix and report throughput as the number of neworder transactions committed per second.

Our second benchmark is based on YCSB [5]. We used a database of 285 million keys, with 16-byte keys and 1 KB values, stored in a single B-Tree with leaves spread randomly across a 57-machine cluster, i.e., without any range partitioning. The B-Tree leaves were large enough to hold exactly one key-value pair, so a single key read or update caused one FaRMv2 object read or write.

We evaluate two systems. The first, BASELINE, is an optimized version of FaRMv1 [9, 10] that significantly outperforms our previously reported TPC-C performance for FaRMv1. The second system is FaRMv2, which adds opacity and multi-versioning to this optimized baseline. Both systems are run with strict serializability and 3-way replication unless stated otherwise. When running TPC-C we use the single-version mode of FaRMv2 by default as this gives the best performance for this workload.

5.2 Overhead of opacity

Figure 6 shows the saturation throughput of BASELINE, and of FaRMv2 in single-version mode with different combinations of serializability/SI and strictness/non-strictness.

FaRMv2 has a throughput of 3.77 million neworders per second on 57 machines with strict serializability, 3.6% lower than BASELINE. The overhead of opacity comes from uncertainty waits, clock synchronization RPCs, timestamp generation (querying the cycle counter and thread-safe access to shared synchronization state) and *OAT* propagation across threads and machines (which is enabled in both single-version and multi-version mode). Abort rates are extremely

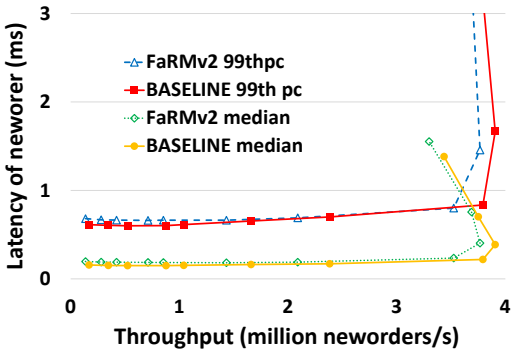


Figure 7: Throughput/latency

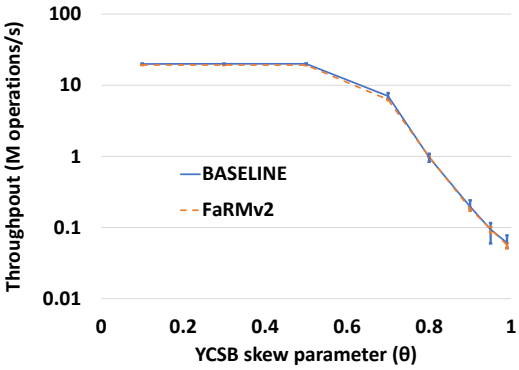


Figure 8: YCSB throughput with skew

low (0.002%) for both BASELINE and FaRMv2. Relaxing strictness improves performance by 3% with serializability, by removing the overhead of the uncertainty wait on the read timestamp. Using SI rather than serializability improves performance by a further 2.7% by removing the uncertainty wait on the write timestamp, and also the overhead of validation.

Figure 7 shows a throughput-latency curve generated by varying the level of concurrency in the workload. Both FaRMv2 and BASELINE are able to sustain close to peak throughput with low latency, e.g., FaRMv2 has a 99th percentile neworder latency of 835 μs at 94% of peak throughput. At high load, latency is dominated by queueing effects in both systems. At low load, the cost of opacity is an additional 69 μs (11%) of latency at the 99th percentile and 39 μs (25%) at the median. The throughput cost of opacity is small. The added latency can be significant for short transactions but we believe it is a price worth paying for opacity.

We also measured the effect of skew on performance, using the YCSB benchmark with a 50-50 ratio of key lookups and updates (Figure 8). Keys are selected from a Zipf distribution, with a higher skew parameter θ giving a more skewed distribution. At low skew, FaRMv2 has around 3% lower

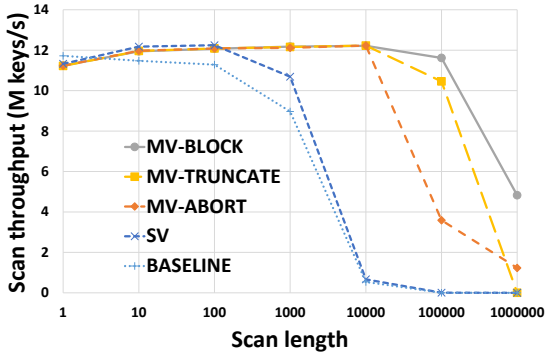


Figure 9: Throughput of mixed scan/update workload

performance than BASELINE. At high skew, both systems have two orders of magnitude lower throughput because of aborts due to high conflict rates and performance differences are within experimental error. The cost of opacity does not increase with conflict rates.

5.3 Multi-versioning

Multi-versioning can improve performance by avoiding aborts of read-only transactions, at the cost of allocating and copying old versions. For TPC-C, this cost is a 3.2% reduction in peak throughput (1.5% for allocation/GC and 1.7% for copying). There is no performance benefit for TPC-C from using MVCC as the abort rate is extremely low even in "single-version" mode.

Multi-versioning has benefits when single-version operation would frequently abort read-only transactions due to conflicts with updates. FaRMv2 bounds the memory usage of old versions to keep sufficient space for head versions and their replicas. When this limit is reached, writers will not be able to allocate old versions during the *LOCK* phase. We implemented three strategies for handling this case, which does not occur in the TPC-C experiments shown so far. We can *block* writers at the lock phase until old version memory becomes available; we can *abort* them when old version memory is not immediately available; or we can allow them to continue by allowing old version allocation to fail, and *truncating* the entire history of objects for which old versions could not be allocated. The last option improves write performance at the risk of aborting readers.

We measured the performance of these different approaches using YCSB. The workload contained scans (range queries) of varying length as well as single-key updates. The start of each scan and the key for each update is chosen from a uniform random distribution. We maintained a 50:50 ratio of keys successfully scanned (keys in scans that completed without aborting) to keys successfully updated. Old-version

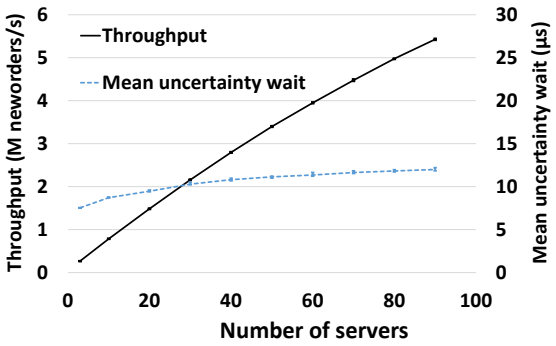


Figure 10: Scalability of FaRMv2

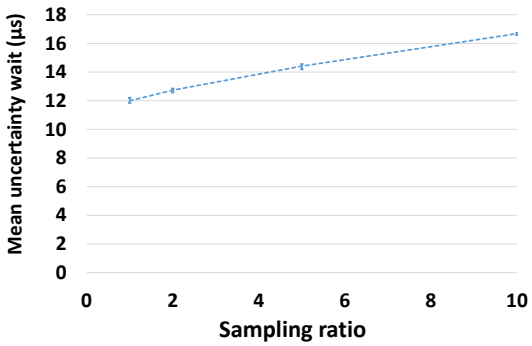


Figure 11: Scalability of clock synchronization

memory was limited to 2 GB per server. Transactions that aborted were retried until they committed. We report the average throughput from 10 min of steady-state measurement for each experimental configuration.

Figure 9 shows the throughput in millions of keys scanned per second on a linear scale against the scan length on a log scale. For scans that read a single key, BASELINE performs slightly better than FaRMv2 as it does not have the overhead of maintaining opacity and transactions that read a single object do not perform validation. For scans of multiple objects, BASELINE must validate every object read and performs worse than FaRMv2. Beyond scans of length 100, both single-version FaRMv2 (sv) and BASELINE lose throughput rapidly because of aborted scans. The three multi-versioning variants of FaRMv2 maintain high throughput up to scans of length 10,000: at this point, MV-ABORT begins to lose performance. MV-BLOCK and MV-TRUNCATE maintain throughput up to scans of length 100,000 and then lose performance, with MV-BLOCK performing slightly better than MV-TRUNCATE. For longer scans, all strategies perform poorly.

The average scan latency for 10,000 keys was 750–850 ms and all the MV variants perform well at this point. Our target applications have lower update rates and shorter queries. All the MV variants perform well for these relatively short queries, without any drop in throughput. In production we use MV-TRUNCATE by default.

5.4 Scalability

Figure 10 shows the throughput of FaRMv2 with strict serializability as we vary the cluster size from 3, the minimum possible with 3-way replication, to 90. Throughput scales well, achieving 5.4 million neworders/s at 90 machines with 21,600 warehouses. This is the highest TPC-C transactional throughput we know of for a system with strict serializability and high availability.[1]

We use an aggregate synchronization rate of 200,000 synchronizations per second at the CM. As the cluster size increases, this means individual non CMs synchronize less frequently, causing the uncertainty to increase. Figure 10 also shows the average uncertainty wait as a function of cluster size, which increases only moderately from 7.5 μs to 12 μs: a 60% increase in uncertainty for a 30x increase in cluster size.

We emulated the effect of clusters larger than 90 machines on uncertainty by down-sampling the synchronization responses from the CM, on each non CM. E.g., at a sampling ratio of 10 with 90 machines, non-CMs discard 9 out of 10 synchronization responses, which emulates the rate at which non-CMs would synchronize in a cluster of 900 machines. Figure 11 shows the effect of down-sampling on a 90-machine cluster with a sampling ratio varying from 1 to 10. Across this factor of 10, the mean uncertainty wait increases by 39%, from 12 μs to 16.7 μs.

Thus our centralized time synchronization mechanism scales well with only small increases in uncertainty when the cluster size is increased by orders of magnitude. While other factors such as network bandwidth and latency might limit application performance, the synchronization mechanism scales well to moderate sized clusters of up to 1000 machines.

5.5 Availability

We measured availability in FaRMv2 by killing one or more FaRMv2 processes in a 57-machine cluster running the TPC-C workload and using 10 ms leases for failure detection. For these experiments we measure throughput over time in 1 ms intervals. Table 1 summarizes results for 3 failure cases: killing a non-CM, killing the CM, and killing both the

[1] Like those of other in-memory systems [4, 10, 36], our results do not satisfy the TPC-C scaling rules for number of warehouses.

Machines failed	Clock disable time		Recovery time		Re-replication time
1 non-CM	0		49 ms	(44–56 ms)	340 s (336–344 s)
CM	4 ms	(3–4 ms)	58 ms	(27–110 ms)	271 s (221–344 s)
CM and 1 non-CM	16 ms	(11–20 ms)	71 ms	(61–85 ms)	292 s (263–336 s)

Table 1: Recovery statistics

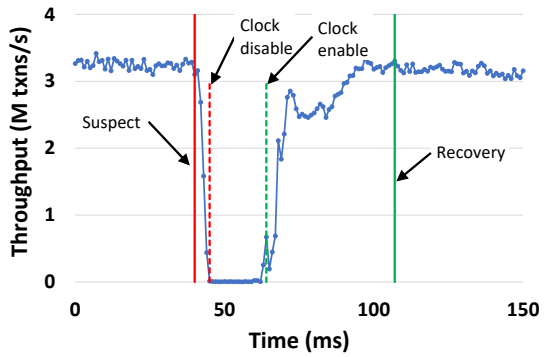


Figure 12: Failure and recovery timeline

CM and a non-CM. It shows mean values across 5 runs with the min/max range in parentheses.

The clock disable time is the elapsed time between the new CM disabling and re-enabling the clock. The recovery time is the time from when a failure was first suspected, to the time when the throughput after failure reaches the mean throughput before the failure. The re-replication time is the time taken for FaRMv2 to bring all regions affected by the failure back to full (3-way) redundancy.

Figure 12 shows throughput over time for one of the runs where we killed the CM as well as a non CM. The solid vertical lines mark the point at which FaRMv2 first suspects a failure, and the point at which we recover full throughput. The dashed vertical lines show the points at which the CM's clock was disabled and then enabled. The timeline does not show the re-replication time, which is much longer than the timescale of the graph.

FaRMv2 provides high availability with full throughput regained within tens of milliseconds in most cases. Even before this point, transactions can continue to execute if they only read objects whose primaries did not change, they only write objects none of whose replicas changed, and clocks are enabled. When clocks are disabled, transactions block when trying to acquire read or write timestamps. Clocks are not disabled if only non-CMs fail. If only the CM fails they are disabled for 4 ms, and if both the CM and a non-CM fail they are disabled for 16 ms on average.

Data re-replication takes around 5 minutes because it is paced to minimize the performance impact on the foreground workload. Re-replication happens in the background and concurrently with the foreground workload. It does not affect availability. The pacing is a configurable parameter: more aggressive re-replication will shorten the window of vulnerability to additional failures but will contend for network and CPU resources with the foreground workload.

5.6 Operation logging

NAM-DB [35] uses replicated in-memory operation logging with no checkpointing or logging to disk. Data is not replicated; instead each committed RW transaction writes the transaction description, inputs, and write timestamp to in-memory logs on three machines. The logging is load-balanced across all machines. This configuration has high TPC-C throughput (6.5 million neworders/s on 57 machines) but it is not realistic: once the in-memory logs fill up, the system cannot accept any more writes. A realistic configuration would need frequent checkpointing to truncate the logs, with substantial overhead.

Operation logging also hurts availability: on any failure, all the logs must be collected in a single location, sorted, and the operations re-executed sequentially, which can be orders of magnitude slower than the original concurrent execution.

FaRMv2 provides high availability through replication. It uses in-memory logs at backups which are continuously truncated by applying them to in-memory object replicas. During recovery, after clocks have been enabled, transactions can read any region whose primary did not change and write to a region if none if its replicas changed. Other regions can be accessed after a short lock recovery phase during which the untruncated portion of the in-memory logs are scanned in parallel by all threads and write-set objects are locked.

For a fair comparison we implemented operation logging in FaRMv2 and measured the same configuration as NAM-DB: operation logging, multi-versioning, non-strict snapshot isolation. FaRMv2 achieves 9.9 million neworders/s with 90 machines. With 57 machines, the throughput is 6.7 million neworders/s, 4% higher than NAM-DB's throughput with the same number of machines, comparable CPUs, and using older, slower CX3 NICs.

6 RELATED WORK

In this section we compare FaRMv2's design with some other systems that support distributed transactions. We do not aim to cover the entire space of distributed transaction protocols or systems [2, 14]. Instead we focus on a few systems that highlight differences in the use of one-sided RDMA, strictness, serializability, opacity, availability, and timestamp generation.

Most systems with distributed transactions and data partitioning use RPCs to read remote objects during execution which requires CPU processing at the remote participants. Traditional 2-phase commit also requires processing of *PREPARE* and *COMMIT* messages at all participants, including read-only ones. Calvin [29] uses deterministic locking of predeclared read and write sets to avoid 2PC but read locks must still be taken using messages. Sinfonia [1] can piggyback reads on the 2PC messages in specialized cases to reduce the number of messages at read-only participants. Sundial [34] uses logical leases to dynamically reorder transactions to minimize conflicts, and caching to reduce remote accesses. This provides serializability but not strictness, and lease renewals still require RPCs to read-only participants.

Systems that use one-sided RDMA reads can avoid CPU processing at read-only participants. NAM-DB [35] uses RDMA reads during execution and only takes write locks during commit, but it only provides SI and not serializability. DrTM [4, 33] provides serializability by using hardware transactional memory (HTM) to detect conflicting writes. FaRM uses an additional validation phase with RDMA reads to detect conflicting writes.

With traditional 2PC, if a coordinator fails, the system becomes unavailable until it recovers. Spanner [6] replicates both participants and coordinators to provide availability. FaRM and RAMCloud [23] replicate data but not coordinators: they recover coordinator state for untruncated transactions from participants. In FaRM, transaction recovery is parallelized across all machines and cores and runs concurrently with new application transactions. Calvin replicates transactional inputs and sequence numbers to all nodes in the system and re-executes transactions issued since the last checkpoint. NAM-DB replicates inputs of committed transactions but does not checkpoint or replicate data and must re-execute all past transactions sequentially on failure before the system becomes available.

Opacity requires consistent read snapshots during execution which can be provided with pessimistic concurrency control, or with timestamp ordering. FaRMv1, DrTM, RAMCloud, Sundial, and FaSST [20] use OCC with per-object versions rather than global timestamps, and hence do not provide opacity. NAM-DB uses timestamp vectors with one element per server, read from a timestamp server and cached and reused locally. NAM-DB does not provide strictness. Clock-SI [11] uses physical clocks at each server: remote reads must use RPCs that block at the remote server until the local clock has passed the transaction read timestamp. Clock-SI does not rely on a clock drift bound for correctness but needs physical clocks to be loosely synchronized for performance. It also does not provide strictness.

Spanner [6] and FaRMv2 use timestamp ordering based on real time with explicit uncertainty computed according to Marzullo's algorithm [25], and both provide opacity. Unlike Spanner, FaRMv2 does not rely on globally synchronized hardware such as atomic clocks or GPS, and it achieves two orders of magnitude lower uncertainty than Spanner within the data center by exploiting fast RDMA networks. Spanner uses pessimistic concurrency control for serializability whereas FaRMv2 uses OCC and supports one-sided RDMA reads.

Timestamp ordering and OCC have been used in many scale-up in-memory systems, both software transactional memories (STMs) and in-memory databases. They typically use shared-memory primitives such as CAS to generate timestamps. TL2 [8] and LSA [27] are some of the first STMs to use timestamp ordering and OCC to provide strict serializability and opacity. TL2 is single-versioned and LSA is multi-versioned with eager validation. Silo [31, 36] uses OCC with read-set validation and without opacity for strictly serializable transactions, and timestamp ordering based on epochs for stale snapshot reads with opacity. Hekaton [22] uses OCC and timestamp ordering with both single- and multi-versioning. It also supports pessimistic concurrency control.

7 CONCLUSION

FaRMv2 is a distributed transactional system that provides opacity, high availability, high throughput, and low latency within a data center. It uses timestamp ordering based on real time with clocks synchronized to within tens of microseconds; a protocol to ensure correctness across clock master failures; and a transactional protocol that uses one-sided RDMA reads and writes instead of two-sided messages. FaRMv2 achieves 5.4 million neworders/s on a TPC-C transaction mix and can recover to full throughput within tens of milliseconds of a failure. To our knowledge this is the highest throughput reported for a system with opacity and high availability, which are important to simplify distributed application development.

ACKOWLEDGEMENTS

We would like to thank Dan Alistarh, Dmitry Bimatov, Paul Brett, Chiranjeeb Buragohain, Wonhee Cho, Ming-Chuan Wu, Joshua Cowhig, Dana Cozmei, Anders Gjerdrum, Orion Hodson, Karthik Kalyanaraman, Richie Khanna, Alan Lawrence, Ed Nightingale, Greg O'Shea, John Pao, Knut Magne Risvik, Tim Tan, and Shuheng Zheng for their contributions to A1 and FaRM. We would also like to thank the anonymous reviewers for their feedback.

REFERENCES

[1] 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.

[2] Bernstein, P. A., Hadzilacos, V., and Goodman, N. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1986.

[3] Castro, M., Dragojević, A., Narayanan, D., Nightingale, E., Shamis, A., Khanna, R., Tan, T., Renzelmann, M., Buragohain, C., and Risvik, K. M. A1 and FaRM: scalable graph database on top of a transactional memory layer. In *HPTS* (2015).

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

[5] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud computing* (2010), SoCC'10.

[6] 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.

[7] 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.

[8] Dice, D., Shalev, O., and Shavit, N. Transactional locking ii. In *Proceedings of the 20th International Symposium on Distributed Computing* (2006), DISC 2006.

[9] 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.

[10] Dragojević, A., Narayanan, D., Nightingale, E. B., Renzelmann, M., Shamis, A., Badam, A., and Castro, M. No compromises: distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), SOSP 2015.

[11] Du, J., Elnikety, S., and Zwaenepoel, W. Clock-SI: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *IEEE 32nd Symposium on Reliable Distributed Systems* (2013), SRDS 2013.

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

[13] 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.

[14] Harding, R., Van Aken, D., Pavlo, A., and Stonebraker, M. An evaluation of distributed concurrency control. *Proc. VLDB Endow. 10*, 5 (Jan. 2017), 553–564.

[15] Intel Corporation. Intel 8 Series/C220 Series Chipset Family Platform Controller Hub (PCH), 2014.

[16] Kalia, A. personal communication.

[17] Kalia, A., Kaminsky, M., and Andersen, D. Datacenter RPCs can be general and fast. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation* (2019), NSDI'19 (to appear).

[18] 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.

[19] Kalia, A., Kaminsky, M., and Andersen, D. G. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference* (2016), USENIX ATC 16, USENIX Association.

[20] Kalia, A., Kaminsky, M., and Andersen, D. G. FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation* (2016), OSDI 16, USENIX Association.

[21] Kung, H.-T., and Robinson, J. T. On optimistic methods for concurrency control. *ACM Transactions on Database Systems 6*, 2 (1981).

[22] 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).

[23] Lee, C., Park, S. J., Kejriwal, A., Matsushita, S., and Ousterhout, J. Implementing linearizability at large scale and low latency. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), SOSP '15.

[24] Lehman, P. L., and Yao, S. B. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems 6*, 4 (1981).

[25] Marzullo, K., and Owicki, S. Maintaining the time in a distributed system. *SIGOPS Oper. Syst. Rev. 19*, 3 (July 1985), 44–54.

[26] Mittal, R., Lam, T., Dukkipati, N., Blem, E., Wassel, H., Ghobadi, M., Vahdat, A., Wang, Y., Wetherall, D., and Zats, D. TIMELY: RTT-based congestion control for the datacenter. In *Proceedings of the 2015 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (2015), SIGCOMM'15.

[27] Riegel, T., Felber, P., and Fetzer, C. A lazy snapshot algorithm with eager validation. In *Proceedings of the 20th International Symposium on Distributed Computing* (2006), vol. 4167 of *DISC 2006*.

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

[29] Thomson, A., Diamond, T., Weng, S.-C., Ren, K., Shao, P., and Abadi, D. J. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (2012), SIGMOD '12.

[30] Transaction Processing Performance Council (TPC). TPC benchmark C: Standard specification. http://www.tpc.org.

[31] 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.

[32] Wei, X., Dong, Z., Chen, R., Chen, H., and Jiao, S. Deconstructing RDMA-enabled distributed transactions: Hybrid is better! In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation* (2018), OSDI'18.

[33] Wei, X., Shi, J., Chen, Y., Chen, R., and Chen, H. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), SOSP '15.

[34] Yu, X., Xia, Y., Pavlo, A., Sanchez, D., Rudolph, L., and Devadas, S. Sundial: Harmonizing concurrency control and caching in a distributed OLTP database management system. *PVLDB 11*, 10 (2018).

[35] Zamanian, E., Binnig, C., Harris, T., and Kraska, T. The end of a

myth: Distributed transactions can scale. *PVLDB 10*, 6 (2017).

[36] 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.

[37] Zhu, Y., Eran, H., Firestone, D., Guo, C., Lipshteyn, M., Liron, Y., Padhye, J., Raindel, S., Yahia, M. H., and Zhang, M. Congestion control for large-scale RDMA deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (2015), SIGCOMM 2015.