FaRM: Fast Remote Memory

Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro, *Microsoft Research*

https://www.usenix.org/conference/nsdi14/technical-sessions/dragojević

This paper is included in the Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14).

April 2–4, 2014 • Seattle, WA, USA

ISBN 978-1-931971-09-6

Open access to the Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14) is sponsored by USENIX

FaRM: Fast Remote Memory

Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, Miguel Castro

Microsoft Research

Abstract

We describe the design and implementation of FaRM, a new main memory distributed computing platform that exploits RDMA to improve both latency and throughput by an order of magnitude relative to state of the art main memory systems that use TCP/IP. FaRM exposes the memory of machines in the cluster as a shared address space. Applications can use transactions to allocate, read, write, and free objects in the address space with location transparency. We expect this simple programming model to be sufficient for most application code. FaRM provides two mechanisms to improve performance where required: lock-free reads over RDMA, and support for collocating objects and function shipping to enable the use of efficient single machine transactions. FaRM uses RDMA both to directly access data in the shared address space and for fast messaging and is carefully tuned for the best RDMA performance. We used FaRM to build a key-value store and a graph store similar to Facebook's. They both perform well, for example, a 20-machine cluster can perform 167 million key-value lookups per second with a latency of 31μs.

1 Introduction

Decreasing DRAM prices have made it cost effective to build commodity servers with hundreds of gigabytes of DRAM. A cluster with one hundred machines can hold tens of terabytes of main memory, which is sufficient to store all the data for many applications or at least to cache the applications' working sets [11, 38, 39]. This has the potential to enable applications that perform small random data accesses, because it removes the overhead of disk or flash, but network communication remains a bottleneck. Emerging fast networks are not going to solve this problem while systems continue to use traditional TCP/IP networking. For example, the results in [16] show a state-of-the-art key-value store performing 7x worse in a client-server setup using TCP/IP than in a single-machine setup despite extensive request batching.

RDMA provides reliable user-level reads and writes of remote memory. It achieves low latency and high throughput because it bypasses the kernel, avoids the overheads of complex protocol stacks, and performs remote memory accesses using only the remote NIC without involving the remote CPU. RDMA has long been supported by Infiniband but it has not seen widespread use in data centers because Infiniband has traditionally been expensive and it is not compatible with Ethernet. Today, RoCE [27] supports RDMA over Ethernet with data center bridging [25, 26] at competitive prices.

FaRM uses RDMA writes to implement a fast message passing primitive that achieves an order-of-magnitude improvement in message rate and latency relative to TCP/IP on the same Ethernet network (Figure 2). It also uses one-sided RDMA reads to achieve an additional two-fold improvement for read-only operations that dominate most workloads [9, 11]. We did not get this performance out of the box. We improved performance by up to a factor of eight with careful tuning and changes to the operating system and the NIC driver.

FaRM machines store data in main memory and also execute application threads. This enables locality-aware optimizations which are important because accessing local memory is still up to 23x faster than RDMA.

FaRM exposes the memory of all machines in the cluster as a shared address space. Threads can use ACID transactions with strict serializability to allocate, read, write, and free objects in the address space without worrying about the location of objects. FaRM provides an efficient implementation of this simple programming model that offers sufficient performance for most application code. Transactions use optimistic concurrency control with an optimized two-phase commit protocol that takes advantage of RDMA. FaRM achieves availability and durability using replicated logging [39] to SSDs, but it can also be deployed as a cache [11, 38].

FaRM offers two mechanisms to improve performance where required with only localized changes to the code: lock-free reads that can be performed with a single RDMA and are strictly serializable with transactions, and support for collocating objects and function shipping to allow applications to replace distributed transactions by optimized single machine transactions.

We designed and implemented a new hashtable algorithm on top of FaRM that combines hopscotch hashing [21] with chaining and associativity to achieve high space efficiency while requiring a small number of RDMA reads for lookups: small object reads are performed with only 1.04 RDMA reads at 90% occupancy. We optimize inserts, updates, and removes by taking advantage of FaRM's support for collocating related objects and shipping transactions.

We used YCSB [15] to evaluate the performance of FaRM's hashtable. We compared FaRM with a baseline system that uses TCP/IP for messaging and performs better than MemC3 [16] (which is the best main-memory key-value store in the literature). Our evaluation on a cluster of 20 servers connected by 40 Gbps Ethernet demonstrates good scalability and performance: FaRM provides an order-of-magnitude better throughput and latency than the baseline across a wide range of settings.

We also implemented a version of Facebook's Tao graph store [11] using FaRM. Once more FaRM achieves an order of magnitude better throughput and latency than the numbers reported in [11].

2 Background on RDMA

RDMA requests are sent over reliable connections (also called *queue pairs*) and network failures are exposed as a terminated connection. Requests are sent directly to the NIC without involving the kernel and are serviced by the remote NIC without interrupting the CPU. A *memory region* must be *registered* with the NIC before it can be made available for remote access. During registration the NIC driver pins the pages in physical memory, stores the virtual to physical page mappings in a page table in the NIC, and returns a *region capability* that the clients can use to access the region. When the NIC receives an RDMA request, it obtains the page table for the target region, maps the target offset and size into the corresponding physical pages, and uses DMA to access the memory. Many NICs (including the ones we are using) guarantee that RDMA writes (but not reads) are performed in increasing address order. DMA operations are cache coherent on our hardware platform.

NICs have limited memory for page tables and connection data. Therefore, many NICs (including ours) store this information in system memory and use NIC memory as a cache. Accessing information that is not cached requires issuing a DMA to fetch it from system memory across the PCI bus. This is a common limitation of offload technology and requires careful use of available memory to achieve good performance.

RDMA has long been supported by Infiniband networks which are widely used by the HPC community. There have been deployments with thousands of nodes and full bisection bandwidth (e.g., [45]). Today, Infiniband has become cost competitive with Ethernet [37], but Ethernet remains prevalent in data centers.

RoCE (RDMA over Converged Ethernet) hardware supports RDMA over Ethernet with data center bridging extensions, which are already available in many switches. These extensions add priority based flow control [26] and congestion notification [25]. They eliminate losses due to congestion and allow segregation of RDMA from other traffic. The hardware manages connection state and acknowledgments eliminating the need for a protocol stack like TCP to ensure reliable delivery.

RoCE is price competitive at the rack level: $19/Gbps for 40 Gbps RoCE compared to $60/Gbps for 10 Gbps Ethernet[1], but there are some concerns about the scalability of RoCE. We expect it to scale to hundreds of nodes and there is ongoing work to improve scalability to thousands of nodes. This paper presents results on a 20-machine cluster using 40 Gbps RoCE but we have also run FaRM on a 78-machine Infiniband cluster.

3 FaRM

This section describes the design and implementation of FaRM. It starts by discussing FaRM's communication primitives and how their implementation is optimized for RDMA. Then it describes how FaRM implements a shared address space and how it ensures consistent accesses to the address space with good performance.

3.1 Communication primitives

FaRM uses one-sided RDMA reads to access data directly and it uses RDMA writes to implement a fast message passing primitive. This primitive uses a circular buffer, as in Figure 1, to implement a unidirectional channel. The buffer is stored on the receiver, and there is one buffer for each sender/receiver pair. The unused portions of the buffer (marked as "Processed" and "Free") are kept zeroed to allow the receiver to detect new messages. The receiver periodically polls the word at the "Head" position to detect new messages. Any non-zero value L in the head indicates a new message, of length L. The receiver then polls the message trailer; when it

[1] Prices include NICs and switches as of October 2013. Ethernet prices are for Intel X520-T2 NICs and Juniper EX4550 switches. RoCE prices are for Mellanox ConnectX 3 NICs and SX1036 switches.

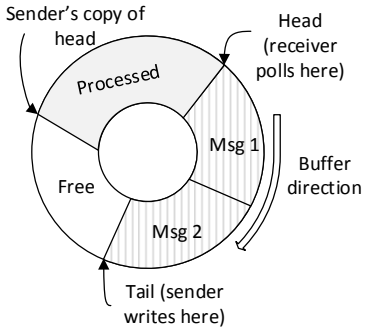


Figure 1: Circular buffer for RDMA messaging

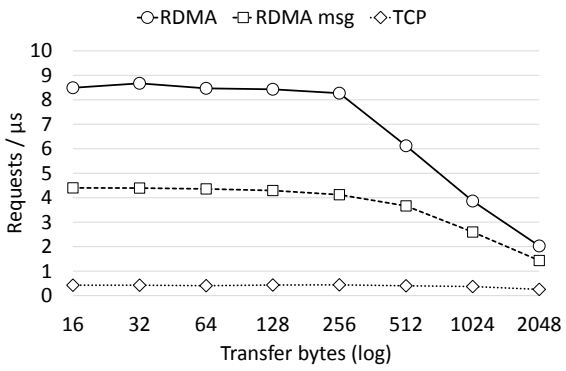


Figure 2: Random reads: request rate per machine

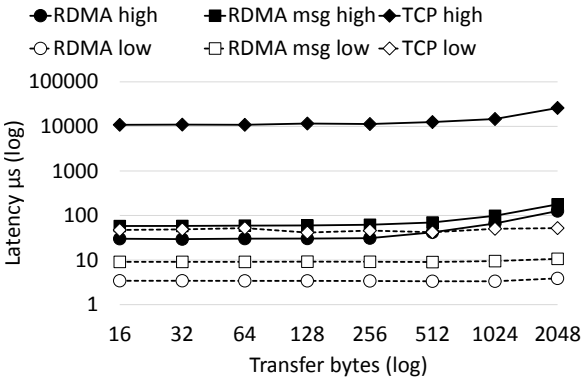


Figure 3: Random reads: latency with high and low load

becomes non-zero, the entire message has been received because RDMA writes are performed in increasing address order. The message is delivered to the application layer, and once it has been processed the receiver zeroes the message buffer and advances the head pointer.

The sender uses RDMA to write messages to the buffer tail and it advances the tail pointer on every send. It maintains a local copy of the receiver's head pointer and never writes messages beyond that point. The receiver makes processed space available to the sender lazily by writing the current value of the head to the sender's copy using RDMA. To reduce overhead, the receiver only updates the sender's copy after processing at least half of the buffer. The sender's copy of the head always lags the receiver's head pointer and thus the sender is guaranteed never to overwrite unprocessed messages.

Polling overhead increases linearly with the number of channels, so we establish a single channel from a thread to a remote machine. We observed negligible polling overhead with 78 machines. We also found that, at this scale, RDMA writes and polling significantly outperform the more complex Infiniband send and receive verbs. In large clusters, it may be better to use RDMA write with immediate and a shared receive queue [35], which would make polling overhead constant.

FaRM messaging is similar to the one described in [35] but our implementation uses a contiguous ring buffer as opposed to a ring of buffers to provide better memory utilization with variable-sized messages. Additionally, the receiver in [35] piggybacks updates to the sender's head pointer in messages.

We ran a micro-benchmark to compare the performance of FaRM's communication primitives with TCP/IP on a cluster with 20 machines connected by a 40 Gbps RoCE network (more details in Section 4). Each machine ran a number of threads that issued requests to read a random block of memory from a random remote machine in an all-to-all communication pattern. Figure 2 shows the average request rate per machine in a configuration optimized for peak throughput. Towards the left of the graph, FaRM's communication primitives are bottlenecked on packet rate and, towards the right, on bit rate. One-sided RDMA reads achieve a bit rate of nearly 33 Gbps with 2 KB request sizes and the bit rate saturates around 35 Gbps for request sizes greater than 8 KB.

FaRM's RDMA-based messaging achieves a request rate between 11x and 9x higher than TCP/IP for request sizes between 16 and 512 bytes, which are typical of data center applications (e.g., [9]). One-sided RDMA reads achieve an additional 2x improvement for sizes up to 256 bytes because they require half the network packets. We expect this performance gap to increase with the next generation of NICs that support 4x the message rate [36]; one-sided RDMA reads do not involve the remote CPU and RDMA-based messaging will be CPU bound.

We also measured UDP throughput and observed it is less than half the throughput of TCP configured for maximum throughput (with Nagle on). So we decided to compare against TCP in the rest of the paper.

Figure 3 shows average request latency both at peak request rate and using only 2 machines configured for minimum latency. The latency of TCP/IP at peak request rate is at least 145x higher than that of RDMA-based messaging across all request sizes. Using one-sided RDMA reads reduces latency by an extra factor of two for sizes up to 256 bytes. In an unloaded system, the latency of RDMA reads is at least 12x lower than TCP/IP

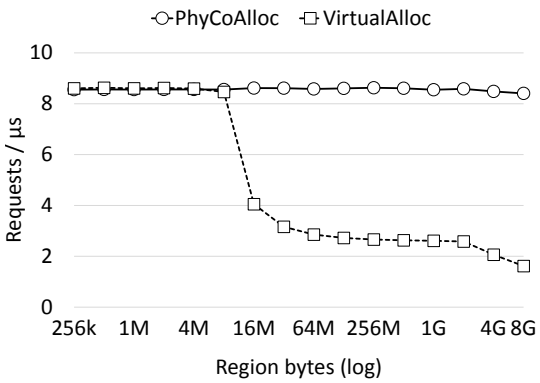


Figure 4: Impact of physically contiguous regions

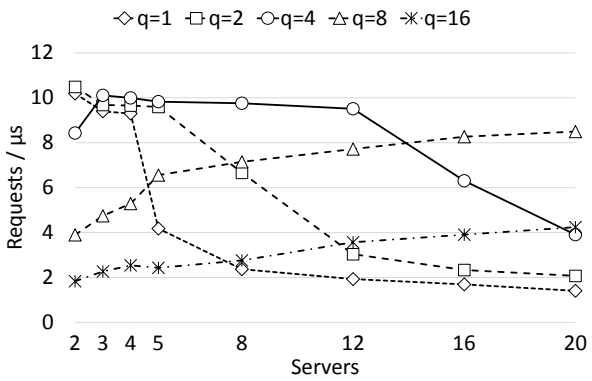


Figure 5: Impact of connection multiplexing

and 3x lower than RDMA-based messaging across all request sizes. The micro-benchmark shows that FaRM's communication primitives can achieve both low latency and high message rates at the same time.

Achieving this level of performance for the two communication primitives was non-trivial and it required solving several problems. The first problem we observed was that the performance of RDMA operations decreased significantly as we increased the amount of memory registered for remote access. The reason was that the NIC was running out of space to cache all the page tables. So it kept fetching page table entries from system memory across the PCI bus.

We fixed this problem by using larger pages to reduce the number of entries in NIC page tables. Unfortunately, existing large page support in Windows and Linux was not sufficient to eliminate all the fetches because of the large amount of memory registered by FaRM. So we implemented PhyCo, a kernel driver that allocates a large number of physically-contiguous and naturally-aligned 2 GB memory regions at boot time (2 GB is the maximum page size supported by our NICs). PhyCo maps the regions into the virtual address space of the FaRM process aligned on a 2 GB boundary. This allowed us to modify the NIC driver to use 2 GB pages, which reduced the number of page table entries per region from more than half a million to one.

We ran the random read benchmark to compare the request rate of 64-byte RDMA reads when regions are allocated with VirtualAlloc and with PhyCo. Figure 4 shows that with VirtualAlloc the request rate drops by a factor of 4 when more than 16 MBs of memory is registered with the NIC. With PhyCo, the request rate remains constant even when registering 100 GB of memory.

We also observed a significant decrease in request rate when the cluster size increased because the NIC ran out of space to cache queue pair data. Using a queue pair between every pair of threads requires $2 \times m \times t^2$ queue pairs per machine (where m is the number of machines and t is the number of threads per machine). We reduced this to $2 \times m \times t$ using a single connection between a thread and each remote machine. In addition, we introduced queue pair sharing among q threads in a NUMA-aware way, resulting in a total of $2 \times m \times t/q$ queue pairs per machine. This trades off parallelism for a reduction in the amount of queue pair data on the NIC.

We ran the random read benchmark with 64-byte transfers while varying the cluster size and the value of q. Figure 5 shows that the optimal value of q depends on the size of the cluster. Small values provide more parallelism and lower sharing overhead, which results in better performance in small clusters, but they also require more queue pair data, which results in degraded performance with larger clusters. In the remainder of the paper, we use these results to select the best value of q for each cluster size. We expect to solve this problem in the future by using Dynamically Connected Transport [36], which improves scalability by setting up connections on demand.

Early experiments showed that using interrupts and blocking could increase RDMA latency by a factor of four. Therefore, we use an event-based programming model. Each FaRM machine runs a user-level process and pins threads to hardware threads. Threads run an event loop which executes application work items and polls for the arrival of RDMA-based messages and the completion of RDMA requests. This polling is done at the user level without involving the OS.

3.2 Architecture and programming model

FaRM's architecture is motivated by the performance results presented in the previous section. FaRM's communication primitives are fast but accesses to main memory still achieve up to 23x higher request rate. Therefore, we designed FaRM to enable applications to improve performance by collocating data and computation on the same machine. FaRM machines store data in main memory and they also execute application threads. The memory of all machines in the cluster is exposed as a shared address space that can be read using one-sided RDMA.

```
Tx*  txCreate();
void txAlloc(Tx *t, int size, Addr a, Cont *c);
void txFree(Tx *t, Addr a, Cont *c);
void txRead(Tx *t, Addr a, int size, Cont *c);
void txWrite(Tx *t, ObjBuf *old, ObjBuf *new);
void txCommit(Tx *t, Cont *c);

Lf* lockFreeStart();
void lockFreeRead(Lf* op,Addr a,int size,Cont *c);
void lockFreeEnd(Lf *op);
Incarnation objGetIncarnation(ObjBuf *o);
void objIncrementIncarnation(ObjBuf *o);

void msgRegisterHandler(MsgId i, Cont *c);
void msgSend(Addr a, MsgId i, Msg *m, Cont *c);
```

Figure 6: FaRM's API

Currently, we support a single FaRM protection domain across the cluster.

Figure 6 shows the main operations in FaRM's interface. FaRM provides an event-based programming model. Operations that require polling to complete take a *continuation* argument, which consists of a *continuation function* and a *context* pointer. The continuation function is invoked when the operation completes and it is passed the result of the operation and the context pointer. The continuation is always invoked on the thread that initiated the operation.

FaRM provides strictly serializable ACID transactions as a general mechanism to ensure consistency. Applications start a transaction by creating a transaction context. They can allocate and free objects using `txAlloc` and `txFree` inside transactions. Allocations return opaque 64-bit pointers that can be used to access objects or stored in object fields to build pointer linked data structures. Applications can request that the new object is allocated close to an existing object by supplying the existing object's address to `txAlloc`. FaRM attempts to store the two objects in the same machine and keep them on the same machine even after recovering from failures or adding new machines. This allows applications to collocate data that is commonly accessed together.

The `txRead` operation can be used to read an object given its address and size. It allocates an object buffer and uses RDMA to read the object's data and meta-data into the buffer. When it completes, it passes the object buffer to the continuation. To update an object, a transaction must first read the object and then call `txWrite` to create a writable copy of the object buffer. Applications commit transactions by calling `txCommit`, which returns the outcome and frees any allocated buffers. Transactions can abort due to conflicts or failures; otherwise, the writes are committed.

General distributed transactions provide a simple programming model but can be too expensive to implement performance critical operations. FaRM's API allows applications to implement efficient lock-free read-only operations that are serializable with transactions. `lockFreeStart` and `lockFreeEnd` are used to bracket lock-free operations. `lockFreeRead` is similar to `txRead` but any object buffers it allocates are freed by `lockFreeEnd`. FaRM also exposes object incarnations, which can be used to combine several lock-free reads into more complex operations. Transactions and lock-free operations are described in Sections 3.4 and 3.5.

The last two API operations are used to send RDMA-based messages to a thread in the machine that stores an object, which allows shipping transactions to the server that stores the object. Together with the ability to collocate related data on the same machine, this enables replacing distributed transactions by single machine transactions, which are significantly less expensive.

FaRM also offers functions to allocate, read, and free arrays of objects. This allows efficient reads of consecutive elements in an array with a single RDMA.

FaRM uses replicated logging to provide ACID transactions with strict serializability and high availability under the following assumptions: crash failures, a bound on the maximum number of failures per replica group, a bound on clock drift in an otherwise asynchronous system for safety, and eventual synchrony for liveness. We do not describe or evaluate recovery from failures in this paper but the common-case (non failure) performance reported in this paper includes all the overheads of replication and logging.

We used this interface to implement a distributed hashtable (Section 3.6) and a graph store similar to Facebook's Tao [11] (Section 4.4).

3.3 Distributed memory management

FaRM's shared address space consists of many 2 GB *shared memory regions* that are the unit of address mapping, the unit of recovery, and the unit of registration for RDMA with the NIC. The address of an object in the shared address space consists of the 32-bit region identifier and the 32-bit offset relative to the start of the region. To access an object, FaRM uses a form of consistent hashing [31] to map the region identifier to the machine that stores the object. If the region is stored locally, FaRM obtains the base address for the region and uses local memory accesses. Otherwise, FaRM contacts the remote machine to obtain a capability for the region, and then uses the capability, the offset in the address and the object size to build an RDMA request. Capabilities for remote regions are cached to improve performance.

Consistent hashing is implemented using a one-hop distributed hashtable [6]. Each machine is mapped into k virtual rings by hashing its IP address with k hash functions. FaRM uses multiple rings to allow multiple regions to be recovered in parallel as in RAMCloud [39] and also to improve load balancing [44]. We currently

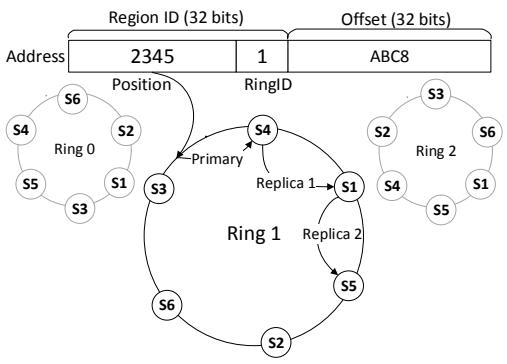


Figure 7: Resolving an address

use $k = 100$. The 32-bit shared region identifier identifies both a ring and a position in the ring. The primary copy and replicas of the region are then stored at the r machines immediately following the region's position in the ring. Figure 7 shows a simple example with $k = 3$ and $r = 3$. The mapping of the region to the machine can be performed locally given the set of machines in the cluster. Cluster membership can be maintained reliably using Zookeeper [24].

Memory allocators are organized into a three-level hierarchy — slabs, blocks, and regions — to reduce synchronization overheads (as in parallel allocators [10]). At the lowest level, threads have private *slab allocators* that allocate small objects from large blocks. Each block is used to allocate objects of the same size. FaRM supports 256 distinct sizes from 64 bytes to 1 MB. The sizes are selected so the average fragmentation is 1.8% and the maximum is 3.6%. An object is allocated in the smallest size class that can fit it. Slab allocators use a single bit in the header of each object to mark it allocated. This state is replicated when a transaction that allocates or frees objects commits and it is scanned during recovery to reconstruct allocator data structures.

The blocks are obtained from a machine-wide *block allocator* that allocates blocks from shared memory regions. It splits the regions into blocks whose size is a multiple of 1 MB. Each region has a table with an 8-byte allocation state per block. The regions are obtained from a cluster-wide *region allocator*. The region allocator uses PhyCo to allocate memory for the region and then it registers the region with the NIC to allow remote access (as described in Section 2). It picks an identifier for the region by selecting a ring at random and a position in the ring that ensures the local node stores the primary copy. Information about region and block allocations is replicated at allocation time.

FaRM allows applications to supply a location hint, which is the address of an existing object, when allocating an object. FaRM attempts to allocate the object in the following order: in the same block as the hint, in the same region, or in a region with a nearby position in the same virtual ring. This ensures that the allocated object and the hint remain collocated both on the primary and on the replicas with high probability even after failures and reconfigurations. If the hint is an address stored at another server, the allocation is performed using an RPC to the remote server.

3.4 Transactions

FaRM supports distributed transactions as a general mechanism to ensure consistency. Our implementation uses optimistic concurrency control [32] and two-phase commit [18] to ensure strict serializability [41]. A transaction context records the version numbers of objects read by the transaction (the *read set*), the version numbers of objects written by the transaction (the *write set*), and it buffers writes. At commit time, the machine running the transaction acts as the *coordinator*. It starts by sending *prepare* messages to all *participants*, which are the primaries and replicas of objects in the write set. The primaries lock the modified objects and both primaries and replicas log the message before sending replies back. After receiving replies from all participants, the coordinator sends *validate* messages to the primaries of objects in the read set to check if the versions read by the transaction are up to date. If read set validation succeeds, the coordinator sends commit messages first to the participant replicas and then to the participant primaries. The primaries update the modified objects and unlock them, and both primaries and replicas log the commit message. The transaction aborts if any modified object is locked, if read set validation fails, or if the coordinator fails to receive replies for all the prepare and validate messages.

FaRM replicas keep the log on SSDs. To improve logging performance, they use a few megabytes of non-volatile RAM [2] to hold the circular message buffers and to buffer log entries [39]. The entries are flushed when the buffers fill up and log cleaning is invoked when the log is half full. These logs are used to implement a parallel recovery mechanism similar to RAMCloud [39].

The two-phase commit protocol is implemented using RDMA-based messaging, which was shown to have very low latency. This reduces conflicts and improves performance by reducing the amount of time locks are held. Despite these optimizations, two-phase commit may be too expensive to implement common case operations.

FaRM provides two mechanisms to achieve good performance in the common case: single machine transactions and lock-free read-only operations. Applications can use single machine transactions by collocating the objects accessed by a transaction on the same primary and on the same replicas, and by shipping the transaction

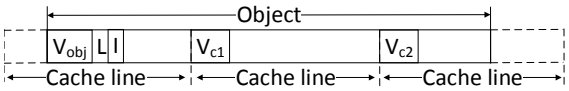


Figure 8: Versioning for lock-free reads. A read is consistent if the lock field L is zero and V_{c1} and V_{c2} match the low-order bits of V_{obj}. The incarnation I is used to detect if the object was freed concurrently with being read.

to the primary. In this case, write set locking and read set validation are local. Therefore, the prepare and validate messages are not needed and the primary only needs to send a commit message with the buffered writes to the replicas before unlocking the modified objects. Additionally, we use two locking modes: objects are first locked in a mode that allows lock-free reads and the primary locks objects in exclusive mode just before updating them (after the commit messages are delivered to the replicas). Single machine transactions improve performance by reducing the number of messages and by further reducing delays due to locks.

3.5 Lock-free operations

FaRM provides lock-free reads that are serializable with transactions and are performed using a single RDMA read without involving the remote CPU. The application is guaranteed to observe a consistent object state even if it is concurrent with writes to the same object. FaRM relies on cache coherent DMA: it stores an object's version number both in the first word of the object header and at the start of each cache line (except the first). These versions are not visible to the application; FaRM automatically converts the object layout on reads and writes.

A `lockFreeRead` reads the object with RDMA and checks if the header version is unlocked and matches all the cache line versions. If the check succeeds, the read is strictly serializable with transactions. Otherwise, the RDMA is retried after a randomized backoff. Figure 8 shows the version fields for an object that spans three cache lines.

An object is written during transaction commit using local memory accesses. The header version is locked with a compare-and-swap during the prepare phase. We use the two least significant bits in the header version to encode the lock mode. During the commit phase, an object is updated by first writing a special lock value to the cache line versions, then updating the data in each cache line, and finally updating the cache line versions and the header version. These steps are separated by memory barriers. On x86 processors, compiler barriers are sufficient to ensure the required ordering. Since DMA is cache-coherent on x86, any RDMA read observes memory writes within each cache line in the order enforced by the memory barriers. Therefore, matching versions across all cache lines spanned by an object ensures strict serializability for lock-free reads.

We use 64-bit header versions to prevent wrapping around but cache line versions only keep the least significant l bits of the version to save space. We can do this because there is a lower bound on the time it takes to perform a write and we abort RDMA reads that take longer than an upper bound to ensure that a read can never overlap two successive writes that produce versions with the same least significant l bits. This relies on a weak bound on clock drift that we already required to maintain leases with ZooKeeper. The results in this paper were obtained with $l = 16$, but our measurements show that $l = 8$ is sufficient in configurations with replication.

To provide consistency, FaRM must ensure that lock-free reads do not access objects that have been freed by concurrent transactions. FaRM uses type stability [19] to ensure that object meta-data remains valid and incarnation checks [46] to detect when objects are freed. Object headers have a 64-bit incarnation that is initially zero and is incremented when the object is freed. FaRM provides 128-bit *fat pointers* that include the object address, size, and expected incarnation. Applications check that the incarnation in the object buffer returned by a lock-free read matches the incarnation in the pointer, which guarantees that the object has not been freed.

FaRM can reuse freed memory to allocate another object of the same size because the incarnation in the object header remains valid. Reusing memory for different object sizes requires more work because the object header may be overwritten with arbitrary data. FaRM implements a distributed version of an epoch-based allocator [17] to do this. It sends an *end of epoch* request to the threads on all machines (we aggregate messages to/from the same machine). When a thread receives this request, it clears any cached pointers, starts a new epoch, and continues processing operations in the new epoch. Once all transactions and read-only operations that started in previous epochs complete, the thread sends a reply to the request. FaRM's API provides primitives that bracket operations to enable detecting when ongoing operations complete. Memory can be reused after receiving responses from all machines in the current configuration. This mechanism does not impact performance significantly because it runs in the background and only when available memory drops below a threshold.

3.6 Hashtable

FaRM also provides a general key-value store interface that is implemented as a hashtable on top of the shared address space. One important use of this interface is as a root to obtain pointers to shared objects given keys.

Designing a hashtable that performs well using RDMA is similar to other forms of memory hierarchy aware data structure design: it is important to balance achieving good space efficiency with minimizing the number and size of RDMAs required to perform common operations. Ideally, we would like to perform lookups, which are the most common operation, using a single RDMA read. We identified hopscotch hashing [21] as a promising approach to achieve this goal because it guarantees that a key-value pair is located in a small contiguous region of memory that may be read with a single RDMA. This contrasts with popular approaches based on cuckoo hashing [40] where a key-value pair is in one of several disjoint regions.

Each bucket in a hopscotch hashtable has a *neighbourhood* that includes the bucket and the $H-1$ buckets that follow. Hopscotch hashing maintains the invariant that a key-value pair is stored in the neighbourhood of the key's bucket (i.e., the bucket the key hashes to). To insert a key-value pair, the algorithm looks for an empty bucket close to the key's bucket by doing a linear probe forward. If the empty bucket is in the neighbourhood of the key's bucket, the key-value pair is stored there. Otherwise, the algorithm attempts to move the empty bucket towards the neighbourhood by repeatedly displacing key-value pairs while preserving the invariant. If the algorithm does not find an empty bucket or is unable to preserve the invariant, the hashtable is resized.

The original algorithm outperforms chaining and cuckoo hashtables at high occupancy using $H = 32$ [21] (where occupancy is the ratio between the number of key-value pairs inserted and the number of slots in the table). Unfortunately, large neighbourhoods perform poorly with RDMA because they result in large reads. For example, using $H = 32$ with 64-byte key-value pairs requires RDMA reads of at least 2 KB, which perform significantly worse than smaller RDMAs (Figure 2). Simply using small neighbourhoods does not work well as it requires frequent resizes and results in poor space efficiency. For example, the original algorithm achieves an average occupancy of only 37% with $H = 8$.

We designed a new algorithm, *chained associative hopscotch hashing*, that achieves a good balance between space efficiency and the size and number of RDMAs used to perform lookups by combining hopscotch hashing with chaining and associativity. For example, on average it requires only 1.04 RDMA reads per lookup with $H = 8$ at 90% occupancy. This is better than techniques based on cuckoo hashing [40] that require 3.2 RDMA reads at 75% occupancy (or 1.6 if key-value pairs were inlined in the table) [37].

The new algorithm uses an overflow chain per bucket. If an insert fails to move an empty bucket into the right neighbourhood, it adds the key-value pair to the overflow chain of the key's bucket instead of resizing the table. This also lets us limit the length of linear probing during inserts. The algorithm uses associativity to amortize the space overhead of chaining and of FaRM's object meta-data across several key-value pairs. Each bucket is a FaRM object with $H/2$ slots to store key-value pairs. The algorithm guarantees that a key-value pair is stored in the key's bucket or the next one. Overflow blocks also store several key-value pairs (currently two) to improve performance and space efficiency.

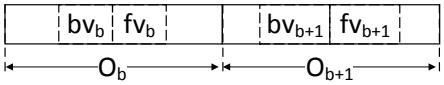


Figure 9: Joint versions for lock-free reads of adjacent buckets. The two objects are consistent with each other if they are individually consistent and $fv_b = bv_{b+1}$.

We implemented the new algorithm using FaRM's API. The hashtable is sharded across the machines in the cluster. Each machine allocates shards, which are FaRM arrays of buckets, and exchanges pointers to the shards with the other machines. We use consistent hashing to partition hash values across shards to enable elasticity.

Lookups are performed using lock-free read-only operations. A lookup for a key k starts by issuing a single RDMA to read both k's bucket b and the next bucket $b+1$. The lookup completes if it finds k in b or $b+1$. Otherwise, it uses lock-free reads to search for k in b's chain of overflow blocks. The chain uses fat pointers to link blocks and lookups check if the incarnation numbers in a fat pointer and the next block match. If they do not, the lookup is restarted. It is inefficient to store large key-value pairs inline in buckets because it results in large RDMAs. FaRM stores large or variable-sized key-value pairs as separate objects and it stores a key (or a hash for large keys) and a fat pointer to the object in the bucket. If the incarnation numbers in the fat pointer and the object do not match, the lookup is restarted.

The version checks in Section 3.5 guarantee that each bucket is individually consistent when read. For hash table lookups, however, we must also ensure that the two buckets in the neighbourhood are consistent with each other. To do this we add *joint versions* for adjacent pair of buckets, meaning that each object bucket stores a forward and a backward joint version (Figure 9). If the corresponding joint versions do not have the same value, the read is restarted. Transactions that update adjacent buckets increment the corresponding joint versions. We reduce the space overhead of joint versions using the same technique we used to reduce the size of cache line versions for lock-free reads. The results in this paper use 16-bit joint versions.

We optimize inserts, updates, and removes by ship-

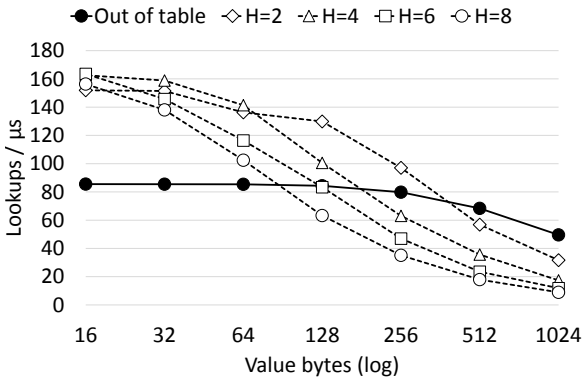


Figure 10: Hashtable: throughput with varying value size

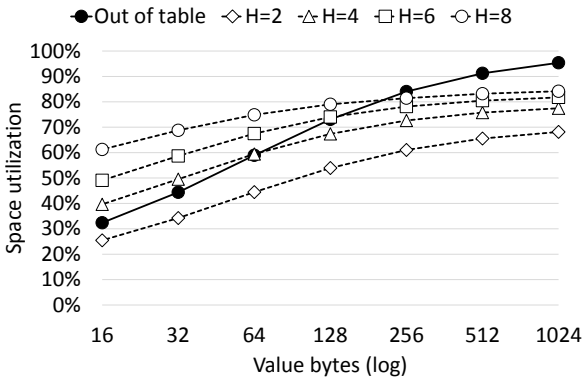


Figure 11: Hashtable: utilization with varying value size

ping transactions to the machine that store the relevant shard. Using transactions simplifies the implementation, which is significantly more complex than for lookups. We use FaRM's API to ensure that shards are collocated with their overflow blocks so we can use more efficient single machine transactions.

We implement inserts as described above. We use a technique inspired by flat combining [20] to combine concurrent inserts and updates to the same key into a single transaction. This improves throughput by more than 4x in our experiments with skewed YCSB workload by reducing the overheads of concurrency control and replication for hot keys. Removes attempt to collapse overflow chains to reduce the number of RDMAs for lookups. They always move the last key-value pair in the chain to the newly freed slot and free the last overflow block if it becomes empty. Otherwise, they increment its incarnation number to ensure lookups observe a consistent view.

FaRM's hashtable guarantees linearizability and it performs well. Figure 10 shows lookup throughput on a 20-machine cluster (see Section 4) at 90% occupancy with 8-byte keys and different value sizes when keys are chosen uniformly at random. It shows results when values are inlined with different neighbourhood sizes and when values are stored outside of the buckets (using $H = 8$). Figure 11 shows the space utilization in the same experiment: this is the ratio between the total number of bytes in key-value pairs and the total amount of memory used by the hashtable. The results show that inlining values with $H = 8$ or $H = 6$ provides a good balance between throughput and space utilization for objects up to 128 bytes. Applications that can tolerate low space utilization to achieve better throughput can inline objects up to 320 bytes with $H = 2$. Objects larger than 320 bytes should be stored outside the table.

4 Evaluation

We evaluate FaRM's performance and its ability to support applications with different data structures and access patterns. We first compare the performance of FaRM's key-value store with a state-of-the art implementation that uses TCP/IP. Then we evaluate FaRM's ability to serve a workload based on Facebook's Tao [8, 11].

4.1 Experimental setup

We ran the experiments on an isolated cluster with 20 machines. Each machine had a 40 Gbps Mellanox ConnectX-3 RoCE NIC connected to a single port of a Mellanox SX-1036 switch. The machines ran Windows Server 2012 R2 on two 2.4 GHz Intel Xeon E5-2665 CPUs with 8 cores and two hyper-threads per core (32 hardware threads per machine). Each machine had a 240 GB Intel 520 SSD for logging and 128 GBs of DRAM (2.5 TB of DRAM across the cluster). We configured machines with 28 GB of private memory and 100 GB of shared memory. We used the results in Figure 5 to select the best connection multiplexing factor q for the number of nodes in each experiment. We report the average of three runs for each experiment. The standard deviation was below 1% of the average except for a very small number of points where it was below 10%.

To evaluate the performance of FaRM's key-value store, we compare it to a distributed hashtable that uses the same chained associative hopscotch hashing algorithm and most of the optimizations in MemC3 [16]. This hashtable uses a neighbourhood size of 12 with six key-value pair slots per bucket. Each key-value pair slot has a 1-byte hash tag [16] and a pointer to the key and value (which are stored outside the table). Remote operations are implemented over TCP/IP running natively in the same RoCE network. We tuned TCP/IP carefully for maximum performance, e.g., we enabled header processing in the NIC and used 16 receiver side queues. We labeled this baseline TCP in the experiments.

The key-value store experiments ran with 16-byte keys and 32-byte values as in [16] to facilitate comparisons and because these are representative of real workloads [16, 38]. Our baseline achieves 40 million lookups

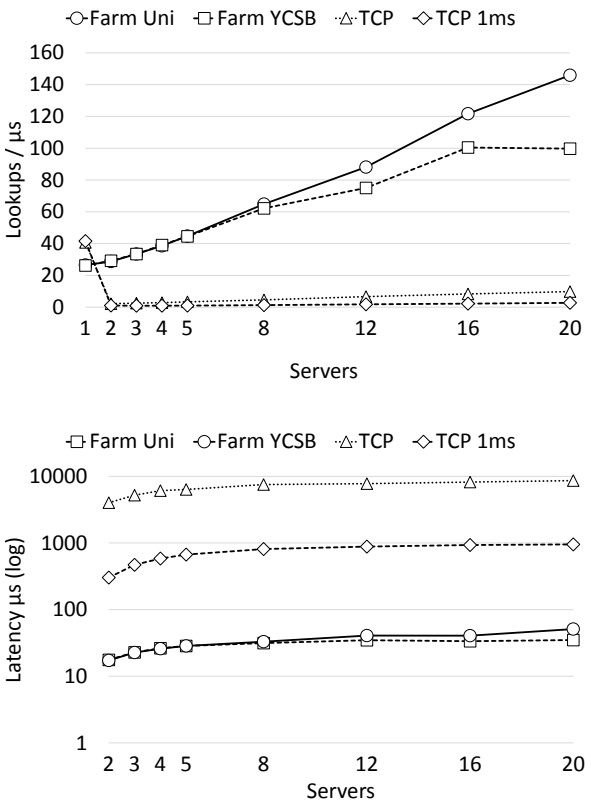


Figure 12: Key-value store: lookup scalability

per second on a single machine which is comparable with the 35 million reported by MemC3 [16] last year.

We loaded FaRM's key-value store and the baseline with 120 million key-value pairs per machine before taking measurements. We configured both stores for 90% occupancy and used a neighbourhood of 6 for FaRM because it provides good throughput and 62% space utilization (see Section 3.6). We used only 120 million key-value pairs to keep experiment running times low but we ran a control experiment with 20 machines and 1.3 billion key-value pairs per machine (58 GB of user data, 94 GB total per machine and 1.8 TB overall) and obtained the same lookup performance.

Except where noted, we measured performance for one minute after a 20 second warm-up. Keys were chosen randomly either with a uniform distribution or with the Zipf distribution prescribed by YCSB [15], which has $\theta = 0.99$. When using the Zipf distribution, the most popular key is accessed by 4% of the operations.

4.2 Key-value store lookups

Figure 12 shows throughput and latency for key-value store lookups as the number of machines increases. We show lines for FaRM with the uniform and YCSB distributions but only show baseline (TCP) performance with

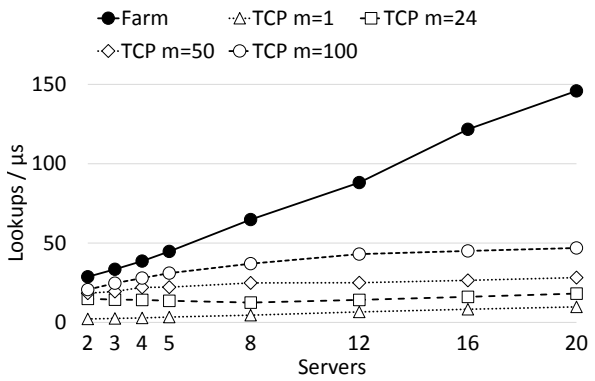


Figure 13: Key-value store: multi-get scalability

the uniform distribution because it is not impacted by the access distribution.

FaRM achieves 146 million lookups per second with a latency of only 35μs with 20 machines and the uniform distribution. This throughput is an order of magnitude higher than the baseline and the latency is two orders of magnitude lower. In fact, the latency of the baseline at peak throughput is more than 8ms, which is unacceptable for several applications (e.g., [38]). We decreased the number of concurrent requests to achieve a latency of 1ms for the baseline which degraded throughput by 3.6x as shown in Figure 12. Trading throughput for latency was not necessary for FaRM as it can achieve both high throughput and low latency at the same time.

The results also show that FaRM's performance scales well with the cluster size. The skew in the YCSB distribution impacts performance with more than 8 servers because it overloads the NICs on the machines that hold the hottest keys, but FaRM is able to achieve more than 100 million lookups per second with latency of 51μs.

Figure 12 also shows single-machine performance for both systems: the baseline can achieve 40 million lookups per second and FaRM achieves 26 million. This is because the baseline uses lock-free reads that are carefully tuned for this particular application whereas FaRM uses general support for lock-free reads, which involves an extra object copy even for local objects. But FaRM achieves 146 million lookups per second while providing 20 times more memory with 20 machines.

In the experiments so far, each lookup retrieves a single item. Main memory key-value stores like Memcached [1] provide a multi-get interface that allows applications to look up several keys. For example, Facebook reports that their applications issue multi-gets for 24 keys on average [38] to amortize communication overheads. We implemented a multi-get interface in our baseline key-value store but not in FaRM because current NICs do not support batching of RDMA. We could implement multi-get batching using FaRM's RDMA-based messaging but we have not yet done this.

Figure 13 compares FaRM's lookup throughput with the baseline key-value store (TCP) using different multi-get batch sizes. Multi-gets can improve the performance of the baseline significantly but FaRM still achieves 8x better throughput relative to multi-gets of 24 keys and 3x better relative to multi-gets of 100 (which are larger than the 95th percentile reported in [38]). However, multi-gets increase the latency of the baseline, which was already high. With multi-gets of 100 and 20 machines, lookup latency increases to 25ms. If we reduce the number of concurrent requests to achieve 10x lower latency, the throughput drops to 10x lower than FaRM's. Another limitation is that for a fixed multi-get size (which is determined by available client parallelism), the benefits of batching decrease with increasing cluster size because the batch must be broken into a message for each machine. For example, multi-gets of 24 improve throughput by 5.7x with 2 servers but only by 2x with 20.

The different mechanisms we discussed in the paper all contribute to the good performance we observed. The low level optimizations discussed in Section 2 improve lookup throughput by 8x. Using lock-free one-sided reads instead of RDMA-based messaging doubles throughput, and our hashtable design reduces the number of RDMAs per lookup by a factor of three when compared to the RDMA-aware design in Pilaf [37].

4.3 Key-value store updates

We also ran experiments with a mix of lookups and updates. We show results without replication (*NoRep*), with logging to SSDs in two replicas (*SSD*), and with logging to memory in two replicas (*Mem*). The first configuration corresponds to using FaRM as a cache and the last allows us to evaluate the overhead of the SSDs. We only show baseline (TCP) results without replication and with the uniform distribution. We set the log size to 32 GB and ran the experiment for 5 minutes with a longer warm-up period of 1 minute to allow the logger to reach a steady state mix of foreground logging and cleaning.

Figure 14 shows scalability with 5% updates (which corresponds to YCSB-B). This update rate is higher than those reported recently for main-memory systems [9, 11, 37, 38]. FaRM scales and performs well: it has an order of magnitude higher throughput than the non-replicated baseline key-value store even when logging to SSDs in two replicas. The throughput when logging to SSDs is 30% lower than without replication mostly due to the additional replication messages. We use a small amount of non-volatile RAM to remove the SSD latency from the critical path and the SSDs have enough bandwidth to cope with logging and cleaning with 5% updates.

The skewed access distribution in YCSB affects scalability as it did with read-only workloads. We omit the

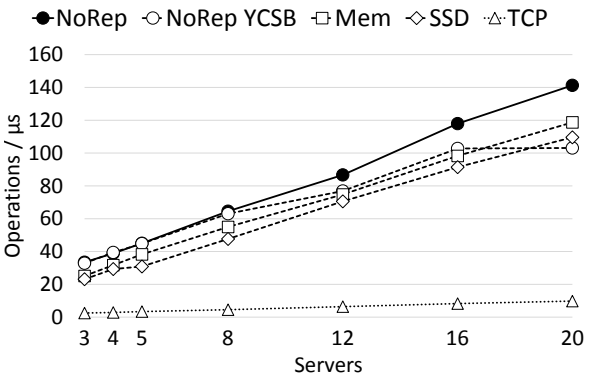
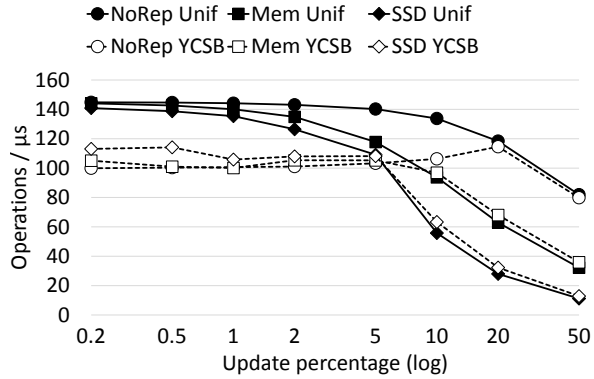
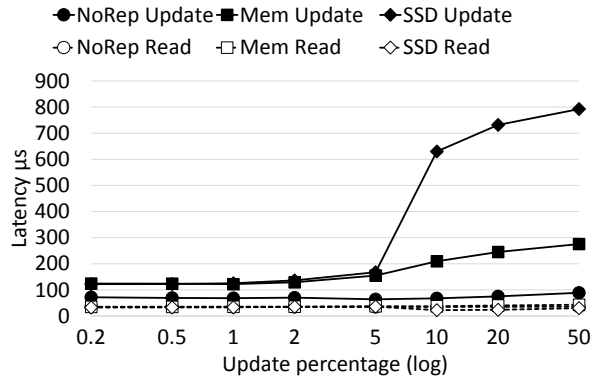


Figure 14: Key-value store: scalability with updates



(a) Throughput (YCSB and uniform)



(b) Latency for lookups and updates (uniform)

Figure 15: Key-value store: varying update rates

lines for the replicated configurations with YCSB because they are similar to the non-replicated configuration. Our measurements show that the update combining optimization in Section 3.6 improves throughput with skew by roughly 4x. Similarly, the dual-mode locking described in Section 3.4 improves the overall throughput by around 25% because it reduces the restart rate of lock-free reads. Both these optimizations have an even larger impact with higher update rates.

We evaluate the performance of FaRM with higher up-

date rates in Figure 15. The overhead of replication is visible with more than 2% updates and SSDs become the bottleneck with more than 5% because we saturate the available I/O bandwidth. With 5% updates, there are 215 MB/s of log writes and 215 MB/s of log reads for cleaning. Roughly half of the writes log new updates and the other half are cleaning writes. The performance with YCSB is slightly better than with the uniform distribution with high update rates because the update combining optimization is able to combine more operations.

Figure 15(b) shows that FaRM's lookup latency is independent of the update rate. Since lookups are implemented using RDMA reads, they are not impacted (as updates are) by increased CPU utilization and queuing delays in FaRM's messaging layer. With low update rates, update operations have a latency of 60μs without replication and 120μs with replication. The latency increases with the update rate because of longer queuing delays.

4.4 Tao

Facebook's Tao [11] is an in-memory graph store. It stores both nodes (e.g., users, comments) and edges (e.g., friend, author of). Both nodes and edges have types and application-specific data. Tao's workload is read dominated (99.8%) with four main operation types. Clients can read a node and its data (`obj_get`), read the most recent outbound edges of a given type from a given node (`assoc_range`), count the number of outbound edges of a given type from a given node (`assoc_count`), or find all outbound edges of a given type from a given node to a set of other nodes (`assoc_get`).

We have implemented a version of Tao. Nodes are FaRM objects with application data inlined and they are uniquely named using fat pointers. Edges are stored as a linked list per edge type in reverse timestamp order and they are collocated with the source node. Each linked list node stores multiple edges. The head pointers and counts of the linked lists are stored in the source node.

`obj_get` and `assoc_count` are implemented with a lock-free read of the node object. `assoc_range` is implemented with a lock-free read of the linked list head using a fat pointer cached by the client. When a new head is inserted, the old head's incarnation is incremented to ensure that clients can detect this and re-fetch the head pointer. These three operations account for 85% of the Tao workload and they require a single RDMA read in the common case. `assoc_get` requires a scan of the edge list. So it is implemented by function-shipping the operation to the machine storing the source node (and the edge list). Update operations use distributed transactions but they account for only 0.2% of the workload.

We evaluated the graph store using Facebook's LinkBench [8] with its default parameters for degree and data size distributions. We used the recommended "full" scale for LinkBench: a graph with 1 billion nodes for which LinkBench generated 4.35 billion edges. The workload was parametrized using the operation mix in [11]. We measured a throughput of 126 million operations per second on our 20-machine cluster. FaRM's per-machine throughput of 6.3 million operations per second is 10x that reported for Tao. FaRM's average latency at peak throughput was 41μs which is 40–50x lower than reported Tao latencies. The three operation types that use lock-free reads required only 1.02 RDMA reads per operation on average. Note that our results were obtained on hardware different from Facebook's and using LinkBench rather than the real workload. Nevertheless, the order-of-magnitude improvements in throughput and latency show that FaRM can implement graph stores efficiently when the graph fits in the cluster's memory.

5 Related work

RDMA has been primarily used to improve message passing performance, e.g., several MPI implementations [34, 35, 42] use RDMA. FaRM's RDMA-based messaging improves on the implementation in [35].

Several libraries (e.g., [22, 49]) and programming languages (e.g., [4, 13, 14, 47, 50]) provide a Partitioned Global Address Space (PGAS) abstraction where processes have both private memory and memory that can be accessed remotely using one-sided operations. Some of them use RDMA to implement one-sided operations but unlike FaRM they do not support efficient lock-free RDMA reads. Instead, they ensure consistency using locks, barriers or messages. Additionally, they were designed for batch computation and are not well suited to building interactive online services, e.g., they lack support for persistence and either provide no fault tolerance or create periodic checkpoints. Distributed shared memory systems (e.g., [5, 12, 43]) are similar to PGAS but lack support for user controlled data placement. FaRM provides a PGAS with ACID transactions.

Several projects have used Infiniband messaging primitives and RDMA to improve the performance of distributed file systems [28, 33, 48], HBase [23], and Memcached [7, 29, 30, 37]. These projects use RDMA to improve performance of a specific service whereas FaRM provides a general distributed computing platform. Additionally, they use RDMA to optimize message passing and do not support one-sided RDMA reads with the exception of [7, 37]. The work in [7] supports one-sided RDMA reads but provides no consistency guarantees.

Pilaf [37] implements a key-value store that uses send/receive verbs to ship update operations to the server and one-sided RDMA reads to implement lookups. It provides linearizability using 64-bit CRCs to detect in-

consistent reads. FaRM's technique to detect inconsistent reads is more general. It provides serializability with respect to general transactions. Additionally, FaRM's RDMA-aware hashtable design performs better because it requires fewer RDMAs to perform lookups with higher space utilization. It is hard to perform a direct performance comparison because the evaluation in [37] uses a single core in a single server and does not address the scalability problems that we discuss in Section 2.

RAMCloud [39] describes techniques for logging and recovery in a main memory key-value store but provides little information about normal case operation. We use similar techniques for logging and recovery but extend them to deal with transactions on general data structures in a shared address space. Unlike [39], we focus on techniques to achieve good performance in the normal case.

Like FaRM, Sinfonia [3] offers a shared address space with transactions. It introduces "mini-transactions" that improve performance by piggybacking execution onto the 2-phase commit protocol. FaRM offers general distributed transactions optimized to take advantage of RDMA together with lock-free reads that require a single RDMA and locality optimizations that enable single machine transactions.

6 Conclusion

We described the design and implementation of FaRM, a new distributed computing platform that stores application data in main memory and exploits RDMA communication to achieve high throughput and low latency at the same time. FaRM provides a shared address space and general distributed transactions to simplify programming. Since distributed transactions can be too expensive for performance critical operations, FaRM also provides two mechanisms to improve performance where needed: lock-free read-only operations and locality optimizations that enable single machine transactions. We demonstrated the effectiveness of these techniques by building RDMA-aware key value and graph stores. Our results show that FaRM performs well: it consistently achieves an order of magnitude better throughput and latency than main memory systems that use TCP/IP on the same physical network.

References

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

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

[3] AGUILERA, M. K., MERCHANT, A., SHAH, M., VEITCH, A., AND KARAMANOLIS, C. Sinfonia: A New Paradigm for Building Scalable Distributed Systems. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles* (2007), SOSP '07.

[4] ALLEN, E., CHASE, D., HALLETT, J., LUCHANGCO, V., MAESSEN, J.-W., RYU, S., STEELE JR, G. L., TOBIN-HOCHSTADT, S., DIAS, J., EASTLUND, C., ET AL. The Fortress language specification. *Sun Microsystems 139* (2005), 140.

[5] AMZA, C., COX, A. L., DWARKADAS, S., KELEHER, P. J., LU, H., RAJAMONY, R., YU, W., AND ZWAENEPOEL, W. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer 29*, 2 (1996), 18–28.

[6] ANJALI, G., BARBARA, L., AND RODRIGO, R. Efficient routing for peer-to-peer overlays. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation* (2004), NSDI '04.

[7] APPAVOO, J., WATERLAND, A., DA SILVA, D., UHLIG, V., ROSENBURG, B., VAN HENSBERGEN, E., STOESS, J., WISNIEWSKI, R., AND STEINBERG, U. Providing a cloud network infrastructure on a supercomputer. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing* (2010), HPDC '10.

[8] ARMSTRONG, T. G., PONNEKANTI, V., BORTHAKUR, D., AND CALLAGHAN, M. LinkBench: a database benchmark based on the Facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (2013), SIGMOD '13.

[9] ATIKOGLU, B., XU, Y., FRACHTENBERG, E., JIANG, S., AND PALECZNY, M. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems* (2012), SIGMETRICS '12.

[10] BERGER, E. D., MCKINLEY, K. S., BLUMOFE, R. D., AND WILSON, P. R. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems* (2000), ASPLOS-IX.

[11] BRONSON, N., AMSDEN, Z., CABRERA, G., CHAKKA, P., DIMOV, P., DING, H., FERRIS, J., GIARDULLO, A., KULKARNI, S., LI, H., MARCHUKOV, M., PETROV, D., PUZAR, L., SONG, Y. J., AND VENKATARAMANI, V. TAO: Facebook's distributed data store for the social graph. In *Proceedings of the 2013 USENIX Annual Technical Conference* (2013), USENIX ATC'13.

[12] CARTER, J. B., BENNETT, J. K., AND ZWAENEPOEL, W. Implementation and performance of Munin. In *ACM SIGOPS Operating Systems Review* (1991), vol. 25, ACM.

[13] CHAMBERLAIN, B. L., CALLAHAN, D., AND ZIMA, H. P. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications 21*, 3 (2007), 291–312.

[14] CHARLES, P., GROTHOFF, C., SARASWAT, V., DONAWA, C., KIELSTRA, A., EBCIOGLU, K., VON PRAUN, C., AND SARKAR, V. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications* (2005), OOPSLA '05.

[15] 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.

[16] FAN, B., ANDERSEN, D. G., AND KAMINSKY, M. MemC3: Compact and concurrent MemCache with dumber caching and smarter hashing. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (2013), NSDI'13.

[17] FRASER, K. *Practical Lock Freedom*. PhD thesis, Cambridge University Computer Laboratory, 2003. Also available as Technical Report UCAM-CL-TR-579.

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

[19] GREENWALD, M., AND CHERITON, D. The synergy between non-blocking synchronization and operating system structure. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation* (1996), OSDI '96.

[20] HENDLER, D., INCZE, I., SHAVIT, N., AND TZAFRIR, M. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures* (2010), SPAA '10.

[21] HERLIHY, M., SHAVIT, N., AND TZAFRIR, M. Hopscotch hashing. In *Proceedings of the 22nd International Symposium on Distributed Computing* (2008), DISC '08.

[22] HOEFLER, T., DINAN, J., THAKUR, R., BARRETT, B., BALAJI, P., GROPP, W., AND UNDERWOOD, K. Remote memory access programming in MPI-3. *ACM Trans. Parallel Comput.* (Mar. 2013).

[23] HUANG, J., OUYANG, X., JOSE, J., W.-U. R. M., WANG, H., LUO, M., SUBRAMONI, H., MURTHY, C., AND PANDA, D. K. High-performance design of HBase with RDMA over Infiniband. In *Parallel and Distributed Processing Symposium* (2012).

[24] 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'10.

[25] IEEE. 802.1Qau - Congestion Notification, 2010.

[26] IEEE. 802.1Qbb - Priority-based Flow Control, 2011.

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

[28] ISLAM, N. S., RAHMAN, M., JOSE, J., RAJACHANDRASEKAR, R., WANG, H., SUBRAMONI, H., MURTHY, C., AND PANDA, D. K. High performance RDMA-based design of HDFS over InfiniBand. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (2012), IEEE Computer Society Press, p. 35.

[29] JOSE, J., SUBRAMONI, H., KANDALLA, K., WASI-UR RAHMAN, M., WANG, H., NARRAVULA, S., AND PANDA, D. K. Scalable Memcached design for Infiniband clusters using hybrid transports. In *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* (2012).

[30] JOSE, J., SUBRAMONI, H., LUO, M., ZHANG, M., HUANG, J., WASI-UR RAHMAN, M., ISLAM, N. S., OUYANG, X., WANG, H., SUR, S., AND PANDA, D. K. Memcached design on high performance RDMA capable interconnects. In *Proceedings of the 2011 International Conference on Parallel Processing* (2011).

[31] KARGER, D., LEHMAN, E., LEIGHTON, T., PANIGRAHY, R., LEVINE, M., AND LEWIN, D. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing* (1997), STOC '97.

[32] KUNG, H. T., AND ROBINSON, J. T. On optimistic methods for concurrency control. *ACM Transactions on Database Systems* (1981).

[33] LI, B., ZHANG, P., HUO, Z., AND MENG, D. Early experiences with write-write design of NFS over RDMA. In *Networking, Architecture, and Storage, 2009. NAS 2009. IEEE International Conference on* (2009), IEEE, pp. 303–308.

[34] LIU, J., JIANG, W., WYCOFF, P., PANDA, D., ASHTON, D., BUNTINAS, D., GROPP, W., AND TOONEN, B. Design and implementation of MPICH2 over Infiniband with RDMA support. In *Parallel and Distributed Processing Symposium* (2004).

[35] LIU, J., WU, J., AND PANDA, D. K. High performance RDMA-based MPI implementation over InfiniBand. *International Journal of Parallel Programming 32*, 3 (June 2004).

[36] MELLANOX TECHNOLOGIES. Connect-IB: Architecture for Scalable High Performance Computing, 2013.

[37] 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.

[38] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H. C., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., STAFFORD, D., TUNG, T., AND VENKATARAMANI, V. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (2013), NSDI'13.

[39] 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.

[40] PAGH, R., AND RODLER, F. F. Cuckoo hashing. *Journal of Algorithms* (2004).

[41] SETHI, R. Useless actions make a difference: Strict serializability of database updates. *Journal of the ACM* (1982).

[42] SHIPMAN, G., WOODALL, T., GRAHAM, R., MACCABE, A., AND BRIDGES, P. Infiniband scalability in Open MPI. In *Parallel and Distributed Processing Symposium* (2006).

[43] STETS, R., DWARKADAS, S., HARDAVELLAS, N., HUNT, G., KONTOTHANASSIS, L., PARTHASARATHY, S., AND SCOTT, M. Cashmere-2L: Software coherent shared memory on a clustered remote-write network. In *ACM SIGOPS Operating Systems Review* (1997), vol. 31, ACM, pp. 170–183.

[44] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (2001), SIGCOMM '01.

[45] SUBRAMONI, H., POTLURI, S., KANDALLA, K., BARTH, B., VIENNE, J., KEASLER, J., TOMKO, K., SCHULZ, K., MOODY, A., AND PANDA, D. K. Design of a scalable InfiniBand topology service to enable network-topology-aware placement of processes. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (2012).

[46] TREIBER, R. K. Systems programming: Coping with parallelism. Tech. Rep. RJ 5118 (53162), IBM, Thomas J. Watson Research Center, 1986.

[47] UPC CONSORTIUM. UPC Language Specifications, v1.2., Technical Report. Lawrence Berkeley National Laboratory LBNL-59208, 2005.

[48] WU, J., WYCKOFF, P., AND PANDA, D. PVFS over InfiniBand: Design and performance evaluation. In *Parallel Processing, 2003. Proceedings. 2003 International Conference on* (2003), IEEE, pp. 125–132.

[49] WWW.OPENSHMEM.ORG. OpenSHMEM Application Programming Interface, 2012.

[50] YELICK, K., SEMENZATO, L., PIKE, G., MIYAMOTO, C., LIBLIT, B., KRISHNAMURTHY, A., HILFINGER, P., GRAHAM, S., GAY, D., COLELLA, P., ET AL. Titanium: A high-performance Java dialect. *Concurrency Practice and Experience 10*, 11-13 (1998), 825–836.