FAQ FaRM

Q: What are some systems that currently uses FaRM?

A: Just FaRM. It's not in production use; it's a recent research prototype. I suspect it will influence future designs, and perhaps itself be developed into a production system.

Q: Why did Microsoft Research release this? Same goes for the research branches of Google, Facebook, Yahoo, etc. Why do they always reveal the design of these new systems? It's definitely better for the advancement of tech, but it seems like (at least in the short term) it is in their best interest to keep these designs secret.

A: These companies only publish papers about a tiny fraction of the software they write. One reason they publish is that these systems are partially developed by people with an academic background (i.e. who have PhDs), who feel that part of their mission in life is to help the world understand the new ideas they invent. They are proud of their work and want people to appreciate it. Another reason is that such papers may help the companies attract top talent, because the papers show that intellectually interesting work is going on there.

Q: Does FaRM really signal the end of necessary compromises in consistency/availability in distributed systems?

A: I suspect not. For example, if you are willing to do non-transactional one-sided RDMA reads and writes, you can do them about 5x as fast as FaRM can do full transactions. Perhaps few people want that 5x more performance today, but they may someday. Along another dimension, there's a good deal of interest in transactions for geographically distributed data, for which FaRM doesn't seem very relevant.

Q: This paper does not really focus on the negatives of FaRM. What are some of the biggest limitations FaRM?

A: Here are some guesses. The data has to fit in RAM. OCC will produce lots of aborts if transactions conflict a lot. The transaction API (described in their NSDI 2014 paper) looks awkward to use because replies return in callbacks. Application code has to tightly interleave executing application transactions and polling RDMA NIC queues and logs for messages from other computers. Application code can see inconsistencies while executing transactions that will eventually abort. For example, if the transaction reads a big object at the same time that a commiting transaction is overwriting the object. The risk is that the application may crash if it isn't defensively written. Applications may not be able to use their own threads very easily because FaRM pins threads to cores, and uses all cores. Of course, FaRM is a research prototype intended to explore new ideas. It is not a finished product intended for general use. If people continue this line of work, we might eventually see descendants of FaRM with fewer rough edges.

Q: What's a NIC?

A: A Network Interface Card -- the hardware that connects a computer to the network.

Q: What is RDMA? One-sided RDMA?

A: RDMA is a special feature implemented in some modern NICs (network interface cards). The NIC looks for special command packets that arrive over the network, and executes the commands itself (and does

not give the packets to the CPU). The commands specify memory operations such as write a value to an address or read from an address and send the value back over the network. In addition, RDMA NICs allow application code to directly talk to the NIC hardware to send the special RDMA command packets, and to be notified when the "hardware ACK" packet arrives indicating that the receiving NIC has executed the command.

"One-sided" refers to a situation where application code in one computer uses these RDMA NICs to directly read or write memory in another computer without involving the other computer's CPU. FaRM's "Validate" phase in Section 4 / Figure 4 uses only a one-sided read.

FaRM sometimes uses RDMA as a fast way to implement an RPC-like scheme to talk to software running on the receiving computer. The sender uses RDMA to write the request message to an area of memory that the receiver's FaRM software is polling (checking periodically); the receiver sends its reply in the same way. The FaRM "Lock" phase uses RDMA in this way.

The benefit of RDMA is speed. A one-sided RDMA read or write takes as little as 1/18 of a microsecond (Figure 2), while a traditional RPC might take 10 microseconds. Even FaRM's use of RDMA for messaging is a lot faster than traditional RPC: user-space code in the receiver frequently polls the incoming NIC queues in order to see new messages quickly, rather than involving interrupts and user/kernel transitions.

Q: Why is FaRM's RDMA-based RPC faster than traditional RPC?

A: Traditional RPC requires the application to make a system call to the local kernel, which asks the local NIC to send a packet. At the receiving computer, the NIC writes the packet to a queue in memory and interrrupts the receving computer's kernel. The kernel copies the packet to user space and causes the application to see it. The receving application does the reverse to send the reply (system call to kernel, kernel talks to NIC, NIC on the other side interrupts its kernel, &c). This point is that a huge amount of code is executed for each RPC, and it's not very fast.

Q: Much of FaRM's performance comes from the hardware. In what ways does the software design contribute to performance?

A: It's true that one reason FaRM is fast is that the hardware is fast. But the hardware has been around for many years now, yet no-one has figured out how to put all the parts together in a way that really exploits the hardware's potential. One reason FaRM does so well is that they simultaneously put a lot of effort into optimizing the network, the persistent storage, and the use of CPU; many previous systems have optimized one but not all. A specific design point is the way FaRM uses fast one-sided RDMA (rather than slower full RPC) for many of the interactions.

Q: Do other systems use UPS (uninterruptable power supplies, with batteries) to implement fast but persistent storage?

A: The idea is old; for example the Harp replicated file service used it in the early 1990s. Many storage systems use batteries in related ways (e.g. in RAID controllers) to avoid having to wait for disk writes. However, the kind of battery setup that FaRM uses isn't particularly common, so software that has to be general purpose can't rely on it. If you configure your own hardware to have batteries, then it would make sense to modify your Raft (or k/v server) to exploit your batteries.

Q: Would the FaRM design still make sense without the battery-backed RAM?

A: I'm not sure FaRM would work without non-volatile RAM, because then the one-sided log writes (e.g. COMMIT-BACKUP in Figure 4) would not persist across power failures. You could modify FaRM so that all log updates were written to SSD before returning, but then it would have much lower performance. An SSD write takes about 100 microseconds, while FaRM's one-sided RDMA writes to non-volatile RAM take only a few microseconds.

Q: Isn't DRAM inherently volatile?

A: The authors make RAM "non-volatile" by using UPS to write out the memory to an SSD on a power failure. But, this is indeed not completely non-volatile, because if the computer crashes for any other reason than a power failure, the content of the memory of the failed machine is lost. This is the reason why they replicate region across several machines and have a fast recovery protocol.

Q: A FaRM server copies RAM to SSD if the power is about to fail. Could they use mechanical hard drives instead of SSDs?

A: They use SSDs because they are fast. They could have used hard drives without changing the design. However, it would then have taken longer to write the data to disk during a power outage, and that would require bigger batteries. Maybe the money they'd save by using hard drives would be outweighed by the increased cost of batteries.

Q: What is the distinction between primaries, backups, and configuration managers in FaRM? Why are there three roles?

A: The data is sharded among many primary/backup sets. The point of the backups is to store a copy of the shard's data and logs in case the primary fails. The primary performs all reads and writes to data in the shard, while the backups perform only the writes (in order to keep their copy of the data identical to the primary's copy). There's just one configuration manager. It keeps track of which primaries and backups are alive, and keeps track of how the data is sharded among them. At a high level this arrangement is similar to GFS, which also sharded data among many primary/backup sets, and also had a master that kept track of where data is stored.

Q: Would FaRM make sense at small scale?

A: I think FaRM is only interesting if you need to support a huge number of transactions per second. If you only need a few thousand transactions per second, you can use off-the-shelf mature technology like MySQL. You could probably set up a considerably smaller FaRM system than the authors' 90-machine system. But FaRM doesn't make sense unless you are sharding and replicating data, which means you need at least four data servers (two shards, two servers per shard) plus a few machines for ZooKeeper (though probably you could run ZooKeeper on the four machines). Then maybe you have a system that costs on the order of $10,000 dollars and can execute a few million simple transactions per second, which is pretty good.

Q: Section 3 seems to say that a single transaction's reads may see inconsistent data. That doesn't seem like it would be serializable!

A: Farm only guarantees serializability for transactions that commit. If a transaction sees the kind of inconsistency Section 3 is talking about, FaRM will abort the transaction. Applications must handle inconsistency in the sense that they should not crash, so that they can get as far as asking to commit, so that FaRM can abort them.

Q: How does FaRM ensure that a transaction's reads are consistent?

What happens if a transaction reads an object that is being modified
by a different transaction?

A: There are two dangers here. First, for a big object, the reader may
read the first half of the object before a concurrent transaction has
written it, and the second half after the concurrent transaction has
written it, and this might cause the reading program to crash. Second,
the reading transaction can't be allowed to commit if it might not be
serializable with a concurrent writing transaction.

Based on my reading of the authors' previous NSDI 2014 paper, the
solution to the first problem is that every cache line of every object
has a version number, and single-cache-line RDMA reads and writes are
atomic. The reading transaction's FaRM library fetches all of the
object's cache lines, and then checks whether they all have the same
version number. If yes, the library gives the copy of the object to
the application; if no, the library reads it again over RDMA. The
second problem is solved by FaRM's validation scheme described in
Section 4. In the VALIDATE step, if another transaction has written an
object read by our transaction since our transaction started, our
transaction will be aborted.

Q: How does log truncation work? When can a log entry be removed?
If one entry is removed by a truncate call, are all previous entries
also removed?

A: The TC tells the primaries and backups to delete the log entries
for a transaction after the TC sees that all of them have a
COMMIT-PRIMARY or COMMIT-BACKUP in their log. In order that recovery
will know that a transaction is done despite truncation, page 62
mentions that primaries remember completed transaction IDs even after
truncation. Truncation implies that all log entries before the
truncation point are deleted; this works because each primary/backup
has a separate log per TC.

Q: Is it possible for an abort to occur during COMMIT-BACKUP, perhaps
due to hardware failure?

A: I believe so. If one of the backups doesn't respond, and the TC
crashes, then there's a possibility that the transaction might be
aborted during recovery.

Q: Since this is an optimistic protocol, does it suffer when many
transactions need to modify the same object?

A: When multiple transactions modify the same object at the same time,
some of them will see during Figure 4's LOCK phase that the lock is
already held. Each such transaction will abort (they do not wait until
the lock is released), and will restart from the beginning. If that
happens a lot, performance will indeed suffer. But, for the
applications the authors measure, FaRM gets fantastic performance.
Very likely one reason is that their applications have relatively few
conflicting transactions, and thus not many aborts.

Q: Figure 7 shows significant increase in latency when the number of
operations exceeds 120 per microsecond. Why is that?

A: I suspect the limit is that the servers can only process about 140
million operations per second in total. If clients send operations
faster than that, some of them will have to wait; this waiting causes
increased latency.

Q: What is vertical Paxos?

A: It is a style of Paxos protocols where an external master performs

reconfiguration while the Paxos group can continue performing
operations while reconfiguration is in progress (see
https://lamport.azurewebsites.net/pubs/vertical-paxos.pdf for the
details). In the FaRM paper, the authors use the term "vertical
Paxos" loosely to mean that the configuration management is done by an
external service (Zookeeper and CM) and processing writes of a
transaction is done with standard primary/backup protocol.