Hardware-Assisted Transactions

Arvind Krishnamurthy

University of Washington

FaRM

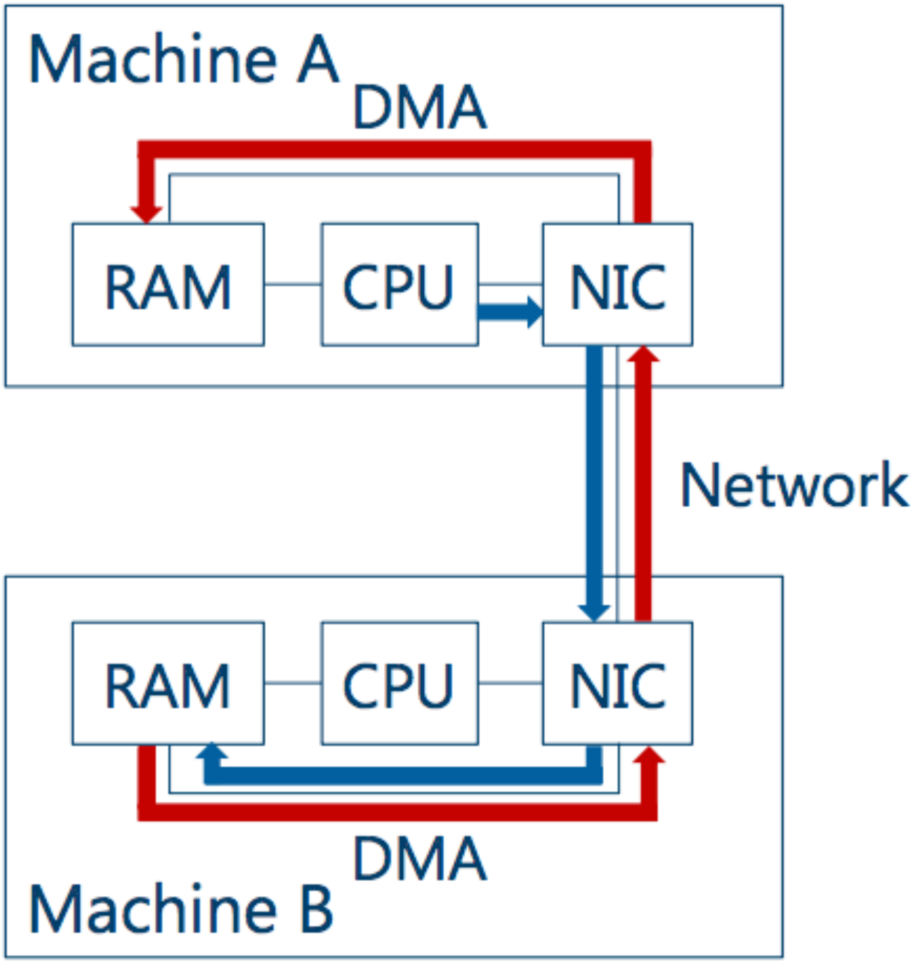
- Distributed transaction system that is in production at Microsoft
- Single datacenter solution
- Hardware assisted:
  - RDMA network
  - Form of NVRAM to provide durable transactions

- Other features:
  - How to perform transactions in a “shared memory” model?
  - What is the appropriate form of OCC?

Network often a performance bottleneck

- Usual setup:
    - Sockets -> Kernel TCP -> NIC driver -> NIC
- Expensive CPU operations
    - System calls
    - Message copies
    - Interrupts

RDMA provides kernel bypass



- App directly interacts with NIC
- Shared memory mapping between App and NIC
- Can perform remote reads/writes with no interrupts or kernel copies
- RPCs: sender writes to remote memory, receiver polls local queue & executes RPCs

FaRM's use of RDMA

- How to use RDMA for transactions and replication?
  - Protocols we have seen require receiver CPU to actively process messages

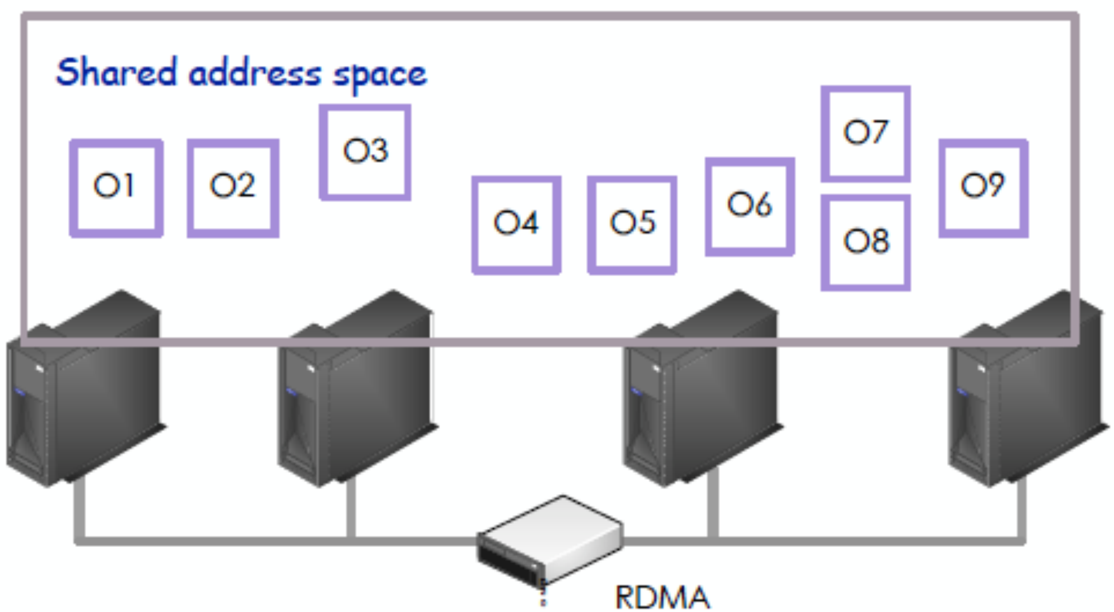
- RDMA used in three ways:
  - One-sided read of objects during transaction execution
  - RPC composed of one-sided writes
  - One-sided writes to backups from Primary

FaRM's "NVRAM"

- FaRM writes go to RAM not disk
- But RAM loses content in power failure
  - Could write to RAM of f+1 machines. But, cannot handle correlated failures
- FaRM uses batteries in every rack to run for a few minutes
  - Power hardware notifies software when power fails
  - Software halts all transaction processing
  - Writes FaRM's RAM to SSD in a few minutes
  - On restart, reads saved memory image from SSD

FaRM Programming Model

- Distributed shared memory abstraction
  - Fixed size objects, flat address space
- Transparent access to local and remote objects



RDMA choices

- RDMA supports two reliable modes:
  - one-sided RDMA uses reliable transfers
  - two-sided RDMA supports datagram transfers
- FaRM uses one-sided RDMA (“reliable transfer”)
  - This results in an all-to-all connectivity pattern.
  - Each side “authorizes” the other side of a connection to do read or write operations in a designed region of memory.

Use of one-sided RDMA

- But with very large numbers of long-lived RDMA pairs of this kind, the RDMA hardware can run into problems:
  - NIC caches data associated with the mapped memory regions. Cache can become over-full and performance then degrades.
  - NIC also caches the page mapping data. With large amounts of FaRM memory, the NIC memory for caching page table entry records will be exhausted.
  - Each active transfer has some state while the transfer is underway. With many concurrent transfers, NIC memory for active operations can overflow.

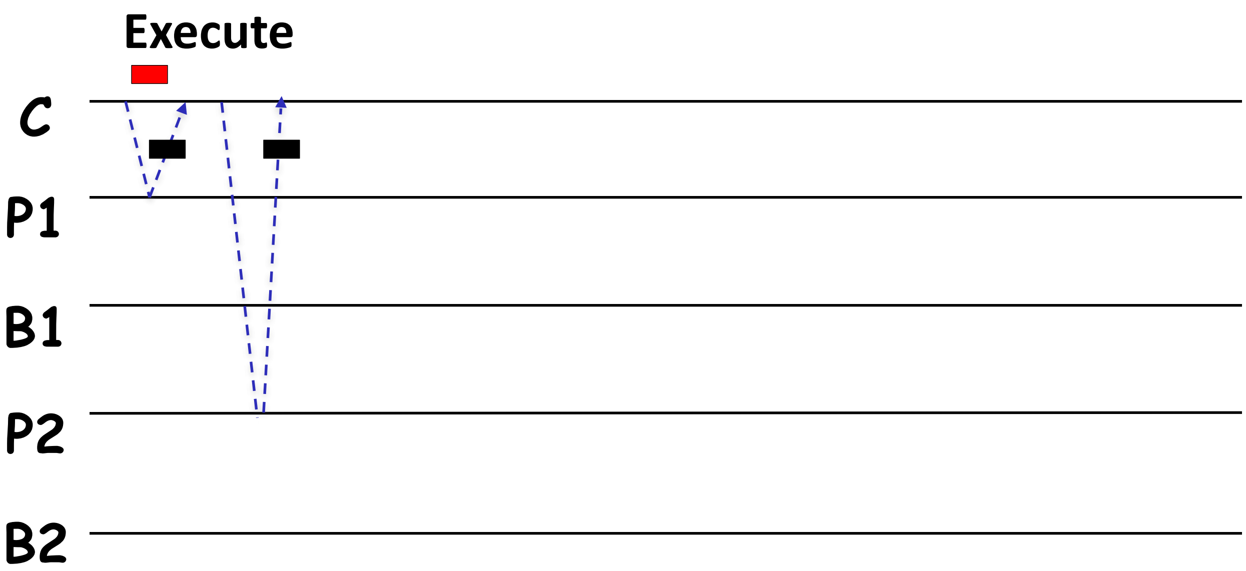
Solutions

- Number of FaRM servers is actually limited by the NIC capacity (128 with older NICs, 1024 with newer NICs)
- FaRM employs 1MB pages (“huge” kernel pages).
- Careful attention to load balancing reduces risk of hot-spots that might have large numbers of simultaneous transfers.

FaRM Setup

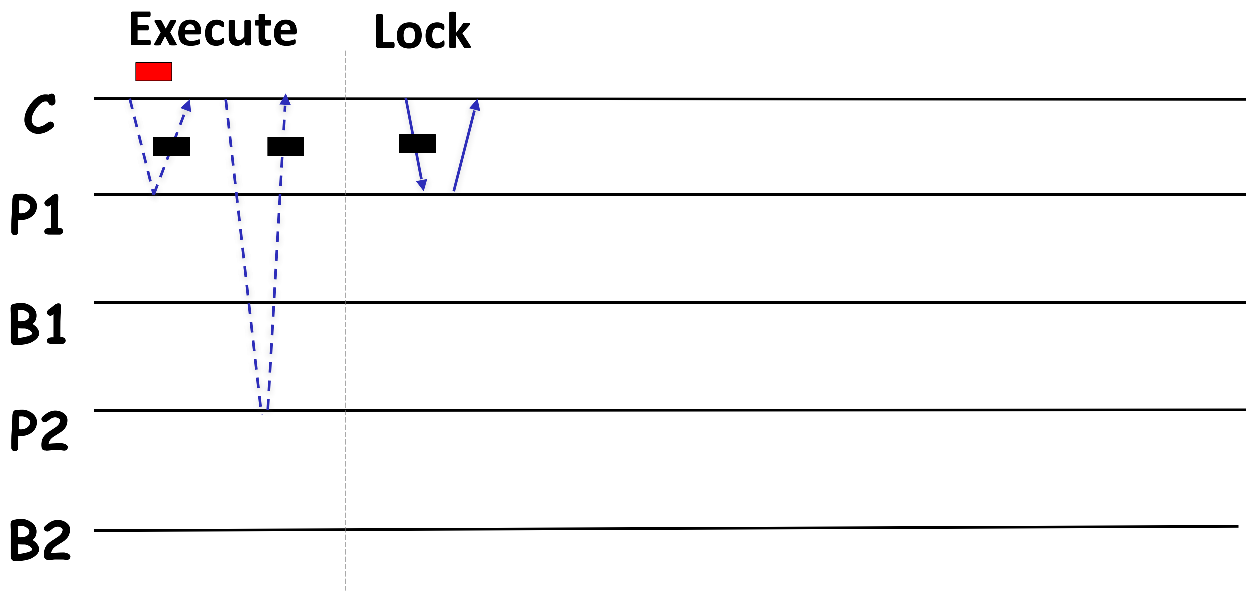
- Every region replicated on one primary and f backups
  - Only the primary serves reads, all f+1 see commits+writes
  - Replication yields availability even with one node (similar to chain replication)
- Regions: each an array of objects
  - Object layout: header with version # and lock
- For each other server:
  - Incoming log/message queue, written by RDMA, read by polling
  - All this in non-volatile RAM

FaRM Transaction Execution



- One-sided RDMA reads; remember simple objects
- Buffer writes on local node

FaRM Commit Protocol

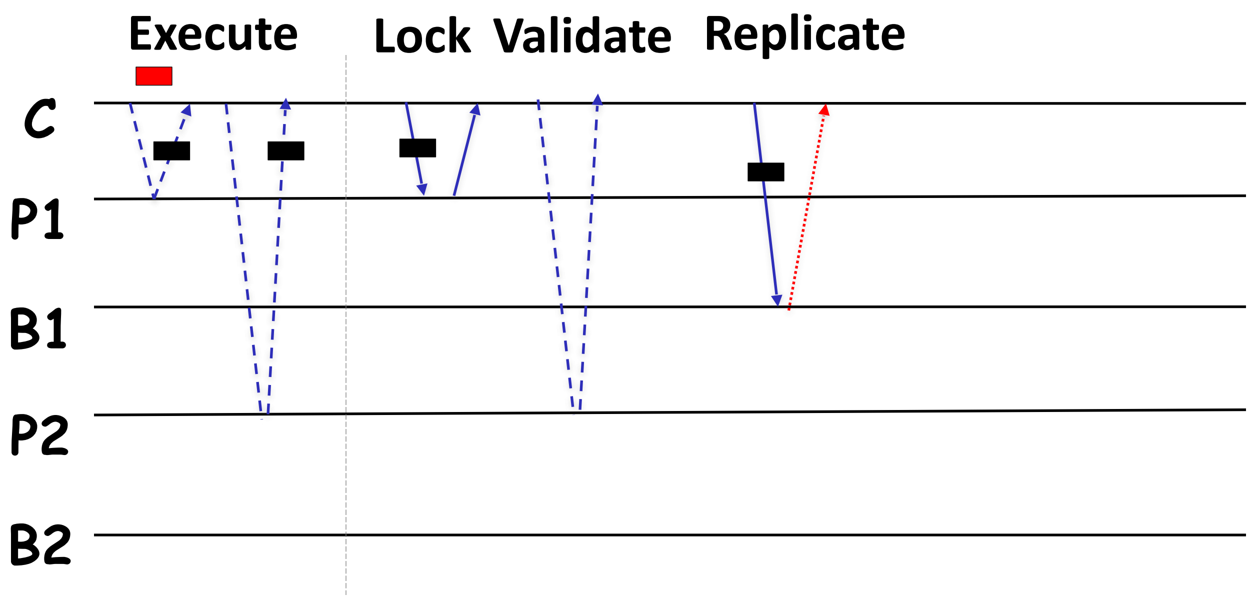


- Writes LOCK record to primary of written objects
- Primary attempts to lock and sends back message reporting succeed or not

Lock Details

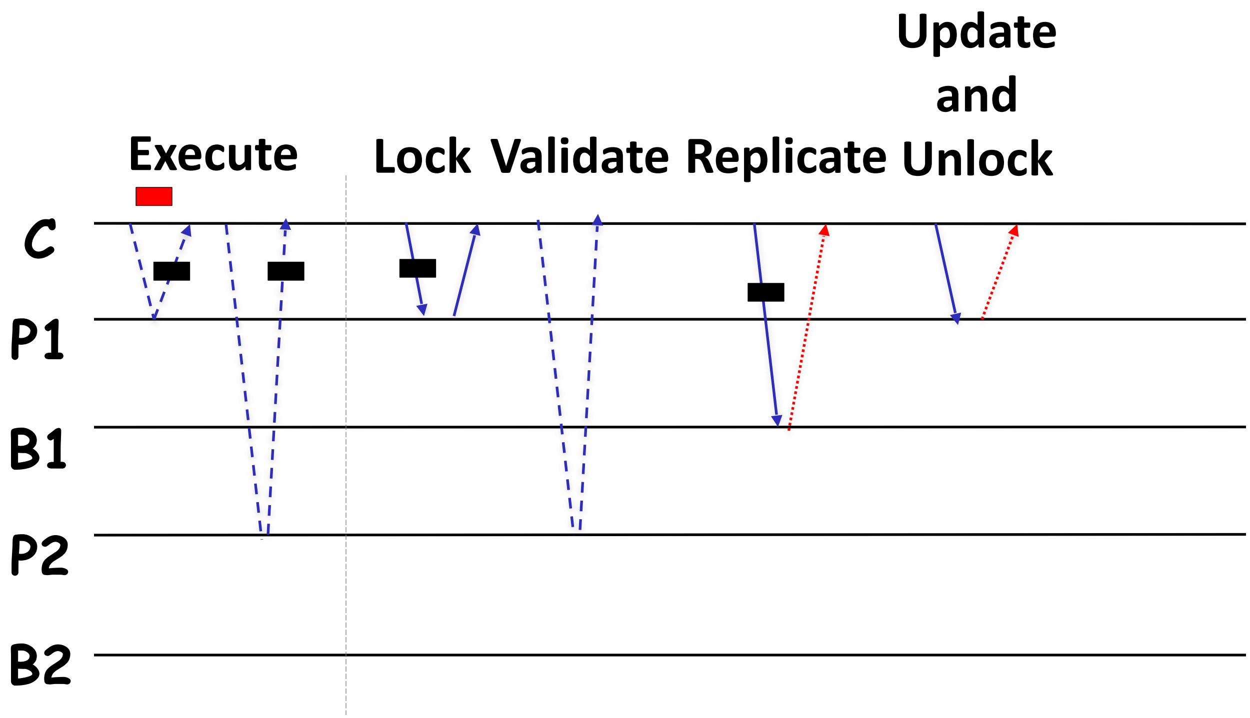
- Coordinator sends to each primary of written object
  - Object ID, Version # read initially, new value
- Primary polls log, sees record
  - Validates whether the version is the same
  - Locks object if possible
    - Atomic compare & swap, “locked” flag is high-order bit in version #
  - Sends yes/no

FaRM Commit Protocol



- Logs to Backups using COMMIT-BACKUP
  - Same as LOCK command: oid, v#, new-value
- Coordinator waits until it receives all hardware acks

FaRM Commit Protocol



- Writes COMMIT-PRIMARY
- Primary processes by updating and unlocking
- Responds to application

Fault Tolerance

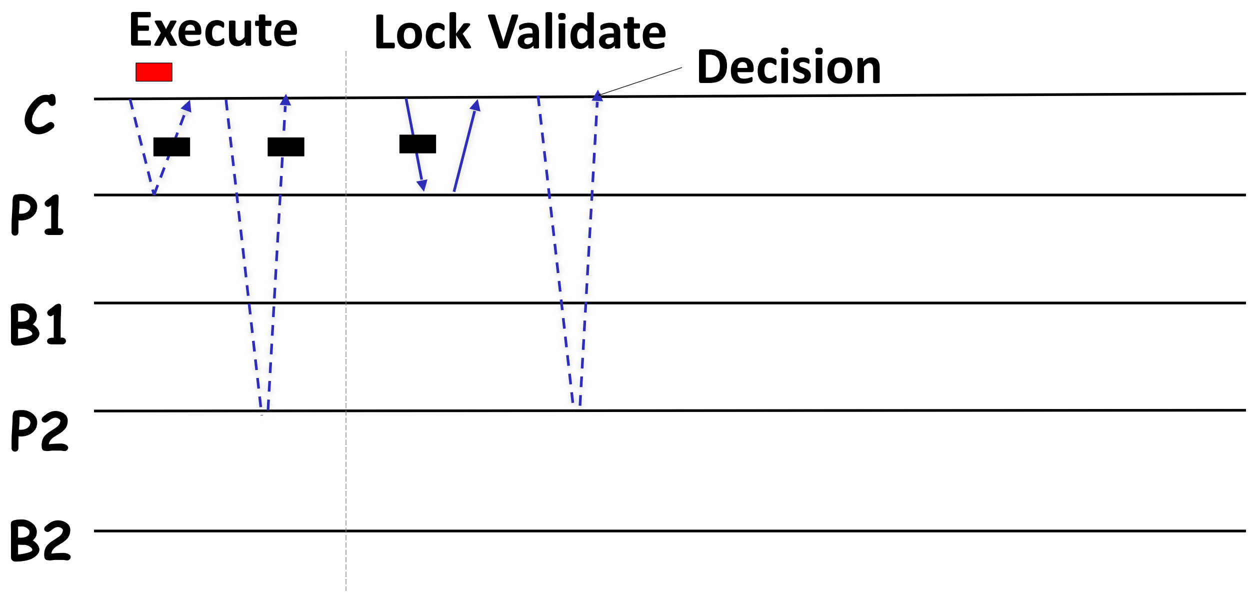
- Regions replicated on f+1 nodes
- Configuration manager monitors liveness
- Zookeeper (Paxos RSM) maintains configuration information
  - Referred to as “Vertical Paxos”
- Configuration manager detects failed nodes
  - Updates configuration in Zookeeper
  - Swaps in a new replica

Fault Tolerance Analysis

- Why does transaction coordinator send COMMIT-PRIMARY only after getting acks for COMMIT-BACKUPs?

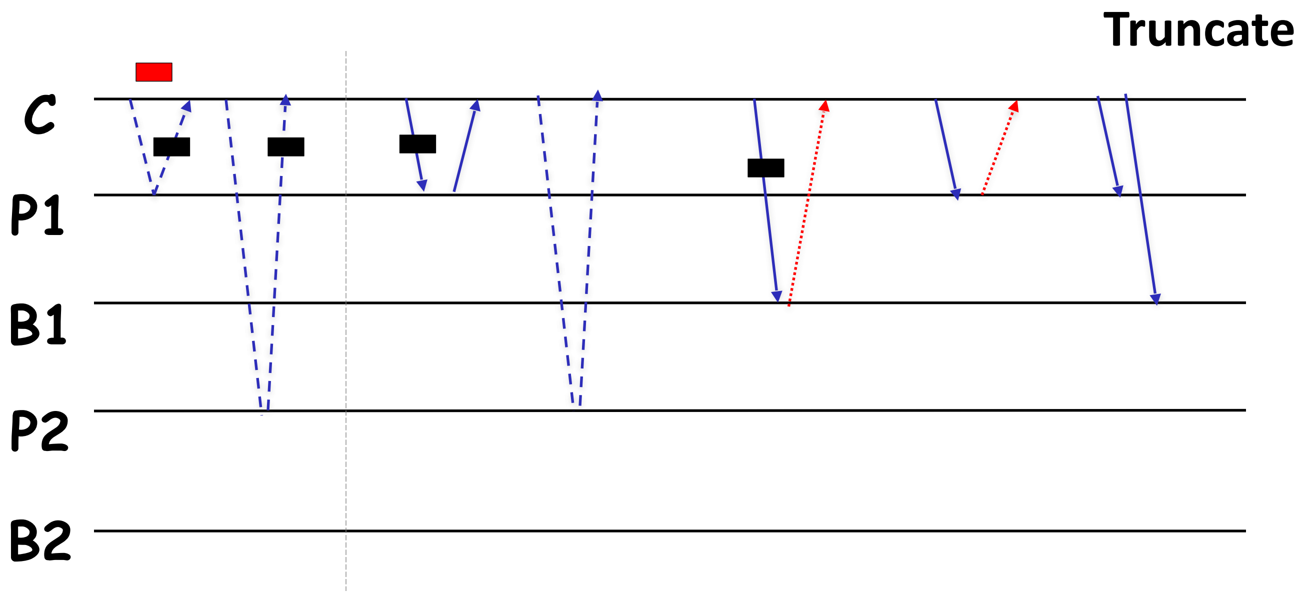
- Transaction coordinator can respond back to application after receiving just one COMMIT-PRIMARY ack. Why?

FaRM Commit Protocol



- Validates read set using one-sided RDMA
- Check whether version # hasn't changed
- Why is this correct? Why is this desirable?

FaRM Commit Protocol



- Coordinator truncates after receiving all ack from Primaries
- Piggybacking in other log records
- Backups apply updates at truncation time

Performance

- f+1 replicas instead of 2f+1 replicas
- Reads satisfied only at the primary
- Coordinator is not replicated - just the App Server as in TAPIR
- Read validation ensures that primaries do not obtain locks
  - No CPU involvement
  - But adds additional latency due to a separate phase