FaRM: Fast Remote Memory

Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, Miguel Castro
Hardware trends

• Main memory is inexpensive
  • 100 GB – 1 TB per server
  • 10 – 100 TBs in a small cluster

• New data centre networks
  • 40 Gbps throughput (100 this year)
  • 1-3 µs latency
  • RDMA primitives
Remote direct memory access

• Read and write remote memory
  • NIC performs DMA requests
  • Remote CPU not involved

• We use RDMA extensively
  • Reads for directly reading data
  • Writes into remote buffers for messaging

• Great performance
  • Bypasses kernel
  • Bypasses remote CPU
Applications

• Data centre applications
  • Irregular access patterns
  • Low latency

• Data serving
  • Graph store
  • Key value-store

• Enabling new applications
Outline

• FaRM programming model
• Design
  • Synchronization
  • Hashtable
• Experimental results
• Future work
How to program a modern cluster?

We have:
• TBs of DRAM
• 100s of CPU cores
• RDMA network

Desirable:
• Keep data in memory
• Access data using RDMA
• Collocate data and computation
Symmetric model

Access to local memory is much faster

Server CPUs are mostly idle with RDMA

Machines store data and execute application
Shared address space

Supports direct RDMA of objects

Programmability a welcome bonus
Transactions: simplify programming

General primitive

Strong consistency: serializability

Transparent:
- location
- concurrency
- failures

Shared address space

Atomic execution of multiple operations
FaRM API: transactions

Tx *TxStart();

Addr TxAlloc(Tx *tx, int size, Addr hint);
void TxFree(Tx *tx, Addr addr);

ObjBuf *TxRead(Tx *tx, Addr addr, int size);
ObjBuf *TxOpenForWrite(Tx *tx, ObjBuf *obj);

bool TxCommit(Tx *tx);
Optimizations: lock-free reads

Efficient: read is a single RDMA

Strong consistency: serializability

Harder to compose: custom validation

Shared address space

Atomic execution of a single read
Optimizations: locality awareness
Optimizations: locality awareness

Collocate data accessed together

Ship computation to target data

Optimized single-server transactions

Addr TxAlloc(Tx *tx, int size, Addr hint);
void SendMsg(Addr a, Msg *m);
Consistency model

• Strong consistency
  • Strict serializability for transactions
  • Linearizability for data structures

• Weak timing assumptions
  • Eventual synchrony
  • Bounded clock drift
Outline

• FaRM programming model
• Design
  • Synchronization
  • Hashtable
• Experimental results
• Future work
## FaRM runtime

<table>
<thead>
<tr>
<th>Applications</th>
<th>Key-value store</th>
<th>Graph store</th>
<th>FaRM Hasutable</th>
<th>Synchronization</th>
<th>Shared address space</th>
<th>Communication</th>
</tr>
</thead>
<tbody>
<tr>
<td>FaRM</td>
<td>3x</td>
<td>24x better than published RDMA key-value store</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2x</td>
<td>10x-40x better than TCP state-of-the-art key-value store</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Transactions

Buffer writes

S$_1$

S$_2$

S$_3$

Execution

Lock

Validate

Update and unlock

RDMA

RDMA

RDMA

RDMA
Lock-free reads

• Transactions can be expensive
  • Require many messages

• FaRM exposes lock-free reads
  • Consistent object state
  • One RDMA operation

• Strictly serializable with transactions
  • Equivalent to a one-read transaction
Lock-free reads

Header version

64-bit version to avoid overflow

W

Consistent if versions match and object is not locked

Read requires three network accesses
FaRM lock-free reads

Space efficiency:
- 16-bit cache-line versions
  - RDMA read, check versions match
  - and read does not take too long

Update:
- Unlock
- Lock update

Header version

\[
t_{\text{update_min}} = 40 \text{ ns}
\]
\[
t_{\text{read_max}} = 40 \text{ ns} \times 2^{16} \times (1 - \varepsilon) = 2 \text{ ms}
\]
Outline

• FaRM programming model
• Design
  • Synchronization
  • Hashtable
• Experimental results
• Future work
FaRM hashtable

• Optimize for lookups
  • Majority of accesses are lookups
  • Goal: lookup with a single RDMA read

• Update with transactions
  • Simplifies updates
  • Performance: ship updates to data owner

• Correctness
  • Goal: linearizability
Distributed hashtable
First attempt: chaining

Overflow chaining

Inlined data

Lookup(5)

One read in the common case. Not quite.
Hopscotch hashtable [Herlihy ‘08]

Invariant: element in neighbourhood

Lookup(5)

Hashtable lookup with a single RDMA
Maintaining invariant

Displace items to make room in neighbourhood

Resize when displacing does not help

Use large neighbourhoods: 32 elements
FaRM hashtable

Overflow chaining

Element in neighbourhood

Space efficiency: multiple items per FaRM object
Overlapping neighbourhoods
Consistent neighbourhoods

Neighbourhood versions

if \( \text{obj}_1 . \text{nv}_{\text{next}} \) != \( \text{obj}_2 . \text{nv}_{\text{prev}} \):
  retry

Lookup( )

Remove

\[ V_1 \quad V_2 \quad V_3 \quad V_4 \quad V_5 \]
Outline

• FaRM programming model
• Design
  • Synchronization
  • Hashtable
• Experimental results
• Future work
TAO [Bronson ‘13, Armstrong ‘13]

• Facebook’s in-memory graph store
• Workload
  • Read-dominated (99.8%)
  • 10 operation types
• FaRM implementation
  • Nodes and edges as FaRM objects
  • FaRM pointers between them
  • Lock-free reads for lookups
  • Transactions for updates

6 Mops/s/srv
(10x improvement)

42 µs average latency
(40 – 50x improvement)
A step towards future data centres

• Enabling new applications
  • Graph processing
  • Scale-out OLTP
  • Deep neural networks

• Future hardware
  • Software hardware co-design
  • Integrated network
  • Non-volatile memory

Fault tolerance

Graphs
Transactions
Data structures
Deep neural networks
FaRM [NSDI ‘14]

• Platform for distributed computing
  • RDMA
  • Data is in memory

• Shared memory abstraction
  • Transactions
  • Lock-free reads

• Order-of-magnitude performance improvements
  • Enables new applications