Memory Coherence in the Exascale Age

Srini Devadas

Massachusetts Institute of Technology
Department of EECS & CSAIL
Memory programming models

Shared Memory Model

- Sharing the same address space
- Reads/writes on shared variables automatically visible to all other threads

(+), Easy to program
(+), Communication between threads is implicitly done
(-), Synchronization issues, race conditions

Message Passing Model

- Each process with its own local space
- All shared data is exchanged by sending and receiving messages

(+), Fits well on supercomputers and clusters
(+), No race conditions (explicit control)
(-), Difficult to program (programmer’s responsibility to explicitly distribute and communicate data)

Processor A

......

Processor B

x = x + 1

Processor A

Send x
Receive into x

Processor B

Receive into temp
temp += 1
Send temp
Shared-memory architectures and shared-data models are preferred if performance and scalability were not an issue since they are easier to program.

### Why shared memory?

If there were no price differences and no scalability problems for shared-memory machines, on which machine would you prefer to run all of your code?

<table>
<thead>
<tr>
<th>shared-memory machine with shared data</th>
<th>distributed-memory machines with message passing</th>
</tr>
</thead>
<tbody>
<tr>
<td>39</td>
<td>23</td>
</tr>
<tr>
<td>5 D (20)</td>
<td>15 D (20)</td>
</tr>
<tr>
<td>20 I (36)</td>
<td>16 I (36)</td>
</tr>
<tr>
<td>15 E (25)</td>
<td>10 E (25)</td>
</tr>
</tbody>
</table>

Assume that there would be no performance penalties. Then, would you rather use one paradigm (be it message passing or multithreading) much more and like to do one the following?

<table>
<thead>
<tr>
<th>use software DSM (Virtual Shared Memory) on a distributed-memory machine (i.e. keep the shared-data model)</th>
<th>keep message passing on a shared-memory machine</th>
<th>neither of both or using HPF or other abstract model / library</th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>18</td>
<td>9</td>
</tr>
<tr>
<td>19 I (36)</td>
<td>13 I (36)</td>
<td></td>
</tr>
<tr>
<td>13 E (25)</td>
<td>12 E (25)</td>
<td></td>
</tr>
</tbody>
</table>

# Why shared memory?

**Message passing** establishes a higher initial barrier than shared memory

<table>
<thead>
<tr>
<th>Question</th>
<th>shared data</th>
<th>message passing</th>
<th>neither of both</th>
</tr>
</thead>
<tbody>
<tr>
<td>Which model do you find easier to use?</td>
<td>31</td>
<td>27</td>
<td>4</td>
</tr>
<tr>
<td>Which model did you find harder to learn?</td>
<td>18</td>
<td>33</td>
<td>11</td>
</tr>
<tr>
<td>Which model do you find easier to modify—or understand—if not written by yourself?</td>
<td>29</td>
<td>27</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>12 B (29)</td>
<td>15 B (29)</td>
<td>2 B (29)</td>
</tr>
<tr>
<td>Which model do you find easier to tune for performance?</td>
<td>28</td>
<td>26</td>
<td>8</td>
</tr>
<tr>
<td>Which model did you find harder to debug the first time you tried it?</td>
<td>19</td>
<td>33</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>8 B (29)</td>
<td>19 B (29)</td>
<td>2 B (29)</td>
</tr>
<tr>
<td>Which model do you find harder to debug now?</td>
<td>25</td>
<td>26</td>
<td>11</td>
</tr>
</tbody>
</table>

We need **distributed caches** because a centralized shared cache suffers from:

- high access latency and energy consumption
- high inter-thread conflicts because the number of ports cannot scale with number of cores

*How to support a Shared Memory Abstraction for a 1000-core machine?*
Snoopy Cache Coherence (Goodman 1983)

- Snoopy schemes broadcast requests over memory bus
- Every cache “snoops” the bus on every command
- Easy to implement
- Requires huge bus bandwidth
- Difficult to scale beyond 16 processors
Distributed “Private” Caches: Directory-Based Cache Coherence

- Data replicated and cached locally for access
- Uncached data copied to local cache, writes invalidate
- Round-trip network cost for misses & invalidations
- **Complex protocol/directories** required to maintain coherence (hard to scale beyond 100 cores)
Distributed “Shared” Caches: Non-Uniform Cache Access (NUCA-RA)

- Each address has a “home” core, data NOT replicated
- Non-local addresses accessed via remote cache
- Round-trip network cost for every non-local WORD access
- Coherence and sequential consistency trivial
The Achilles’ heel of CC

most capacity misses go to off-chip memory

SPLASH2 avg, Graphite sim, 256 cores & threads, 16K L1$ + L2$ shown, full-map MSI
Off-chip access is a growing problem

more cores but less off-chip bandwidth per core

(figures: Karlsson & Hagersten, Uppsala Univ Tech Report 2005-040)
The Achilles’ heel of NUCA-RA

OCEAN, Graphite sim, 256 cores & threads, 16K L1$ + 64K L2$, non-local refs only
Shared memory in massive-scale multicore systems is not that simple

Two Traditional Approaches

- **Directory Cache Coherence (DirCC)**
  - Data replication increases off-chip accesses
  - Complex protocols and directories

- **NUCA w/ Remote Cache Access (RA)**
  - Round-trip network cost for every non-local WORD access

We propose two novel designs: Execution Migration and Library Cache Coherence
Execution Migration Machine (EM²)

• Idea: send thread on 1\textsuperscript{st} memory access
  – move context (RF etc.) to core where data lives
  – possibly evict a context currently at destination
  – continue thread execution at destination
  – migration entirely at hardware level for speed

• Keeps low cache miss rates of NUCA-RA
  – But takes advantage of spatio-temporal locality

• One-way trip for data access (not 2-way)

• Old idea, new application
  – used by OSs for load balancing
  – and to take advantage of heterogeneous cores
  – we use it to provide a shared memory abstraction
Context load/unload hardware

- At migration source: serialize RF as a packet on the on-chip network
- At migration destination: deserialize packet into the receiving RF
Baseline autonomous migration protocols

- **SWAP**: A deadlock-prone autonomous migration protocol

- An eviction *swaps* the locations of two threads.

  + The evicted thread is likely to find an available resource.

  - Deadlock may occur if the eviction is blocked.
Interleaved multithreading core for EM²

- Core multiplexes *native* and *guest* contexts (required for deadlock freedom when evicting)

- Thread scheduler decides which context to run

(extra blocks needed for execution migration are highlighted)
The life of a memory access in EM²

1. Memory access
2. Address cacheable in core A?
   - Yes: Access memory & continue execution
   - No: 
     - Network
     - Core where address can be cached (home)
3. Core originating memory access

4. # threads exceeded?
   - Yes: Migrate another thread back to its native core
   - No: Access memory & continue execution
Exclusive Native Context protocol (ENC)

- ENC: A thread goes to its native core *when evicted* by another thread.
**EM² Pluses and Minuses**

+ One-way data access through thread migration
  - Determine core miss and remote core destination in parallel with L1 lookup
+ No replication across on-chip caches lowers off-chip memory access rates
+ Coherence, consistency and atomic locks are trivial

- Context size of 1-2Kb significantly greater than 1 word (NUCA), and greater than 512-bit cache block size (CC)
  - Higher on-chip bandwidth helps
- Contention especially for reads of shared data
  - Writes have to be serialized in any case
- Does not even replicate read-only data
  - Compiler or application can do this
Library Cache Coherence (LCC)

Check out

Copies with Timestamp

EXPIRED
Library Cache Coherence (LCC)

Check out

Copies with Timestamp

EXPIRED

EXPIRED

EXPIRED

EXPIRED
We assume a global timer that is available to all cores and caches. Each line in the L1/L2 cache has a timestamp.

Writes can be performed only at the home core location, and only when there are no unexpired copies.
High-level Example of LCC

Suppose Core A is the home core for memory address X, and we add a constant $T_{\text{delta}}$ of 150 to the system time to decide a new timestamp.
High-level Example of LCC

Suppose Core A is the home core for memory address X, and we add a constant $T_{\text{delta}}$ of 150 to the system time to decide a new timestamp.

**Remote read at $t = 1000$**

**New timestamp**

$= t + 150 = 1150$
High-level Example of LCC

Suppose Core A is the home core for memory address X, and we add a constant $T_{\text{delta}}$ of 150 to the system time to decide a new timestamp.

Remote read at $t = 1000$

Write at $t = 1100$ is delayed until 1150

New timestamp $= t + 150 = 1150$
Suppose Core A is the home core for memory address X, and we add a constant $T_{\text{delta}}$ of 150 to the system time to decide a new timestamp.

Remote read at $t = 1000$

Write at $t = 1100$ is delayed until 1150

New timestamp $= t + 150 = 1150$

Write performed at $t = 1150$
Baseline Architecture for LCC

Data can be replicated in L1:
The L1 cache can hold both the home or the remote data.

Timestamps have different meanings!

Home? The maximum timestamp among all the checked out copies of the block (the time until when writes need to be delayed)

Remote? The time until when the block is valid

Data is NOT replicated in L2:
Only the home data can be in the L2 cache.
Timestamps always indicate the maximum timestamp among all the checked out copies of the block.
Baseline LCC Protocol

[Read operations]

Requesting Core

Remote Core

Network

Core

Switch

Core

Switch

L1 $ T/S

T/S

L1 $ T/S

Shared L2 $ T/S

Shared L2 $ T/S

T/S
If it’s a **core hit**, the line is valid regardless of its timestamp.

If it’s a **core miss**, the line is valid only when the line has not yet expired (i.e., global system time $T < \text{the timestamp } t \text{ of the line in L1})

*If it misses in its local L1,*
If **Core hit**, it looks up its local L2.
If **Core miss**, it sends a remote request to the home core’s L1
(The line can be potentially brought from backing DRAM or the next-level cache)
Baseline LCC Protocol

[Read operations]

The cache block is returned to L1 with a timestamp, which is the timestamp stored in L2 (no change in its value).

The cache block is returned with a timestamp, which is decided by timestamp choosing logic. The timestamp at home is updated if necessary, to maintain the maximum timestamp among all timestamps issued.
A write request is directly forwarded to its home L1. (Local if core hit, remote if core miss) (The line can be potentially brought from backing DRAM or the next-level cache)
A write is delayed until the current system time reaches the timestamp in the library (home L1) ➡️ *Write Delay*

Once the write is performed, acknowledgement is sent back.
LCC Pluses and Minuses

+ Leverages temporary data replication to complete most loads locally
+ No broadcast/multicast of invalidation required
  • and thus, no collection of invalidation ack’s is required
+ Library size does not grow with the number of cores
  • unlike directory, library incurs a constant overhead regardless of the number of cores

- Writes performed similarly to NUCA-RA w/o benefit of private cache locality
- Performance depends on effective timestamp heuristics
Analytical Models on Average Memory Latency (AML)

The model focuses on the average cost of a single memory access for four different memory implementations (DirCC, NUCA-RA, EM² and LCC)

A uniform network model is assumed. On a 8 by 8 mesh,

- core-to-core message travels the same number of 12 hops (2 cycles/hop)
- experiences similar congestion (50%)
- packets are divided into equal-size (256-bit) flits

\[
\text{cost}_{\rightarrow, \text{data size}} = \text{cost}_{\text{avg net dist}} + \left[ \frac{\text{data size}}{\text{flit size}} \right]
\]

: Network Delivery Cost
+ Serialization Cost

\[
\text{cost}_{\rightarrow, \text{ack}} = \text{cost}_{\rightarrow, \text{address}} = \text{cost}_{\rightarrow, \text{value}} = (12 \times 2 + 50\%) + \left[ \frac{32}{256} \right] = 36 + 1 = 37 \text{ cycles},
\]

\[
\text{cost}_{\rightarrow, \text{addr \& value}} = (12 \times 2 + 50\%) + \left[ \frac{32 + 32}{256} \right] = 36 + 1 = 37 \text{ cycles},
\]

\[
\text{cost}_{\rightarrow, \text{ cacheline}} = (12 \times 2 + 50\%) + \left[ \frac{512}{256} \right] = 36 + 2 = 38 \text{ cycles}
\]
The model focuses on the average cost of a single memory access for four different memory implementations (DirCC, NUCA-RA, EM² and LCC)
**Directory-based Cache Coherence**

\[ AML_{CC} = \text{cost}_{L1\text{access}} + \text{rate}_{L1\text{miss},CC} \times \text{cost}_{L1\text{miss},CC} \]

- \( \text{cost}_{L1\text{miss},CC} = \text{rate}_{rdl,wr1,rdS} \times \text{cost}_{rdl,wr1,rdS} \)  
  + \( \text{rate}_{wrS} \times \text{cost}_{wrS} \)  
  + \( \text{rate}_{rdM,wrM} \times \text{cost}_{rdM,wrM} \)

- \( \text{cost}_{rdl,wr1,rdS} = \text{rate}_{\text{coremiss}} \times \text{cost}_{\rightarrow,addr} \)  
  + \( \max(\text{cost}_{\text{dir lookup}}, \text{cost}_{L2\text{request}}) \)  
  + \( \text{rate}_{\text{coremiss}} \times \text{cost}_{\rightarrow,\text{cacheline}} + \text{cost}_{L1\text{insert}} \)

- \( \text{cost}_{wrS} = \text{rate}_{\text{coremiss}} \times \text{cost}_{\rightarrow,addr} \)  
  + \( \max(\text{cost}_{\text{dir lookup}}, \text{cost}_{L2\text{request}}) \)  
  + \( \text{cost}_{\rightarrow,\text{addr}} + \text{cost}_{L1\text{inv}} + \text{cost}_{\rightarrow,\text{ack}} \)  
  + \( \text{rate}_{\text{coremiss}} \times \text{cost}_{\rightarrow,\text{cacheline}} + \text{cost}_{L1\text{insert}} \)

- \( \text{cost}_{rdM} = \text{rate}_{\text{coremiss}} \times \text{cost}_{\rightarrow,addr} + \text{cost}_{\text{dir lookup}} \)  
  + \( \text{cost}_{\rightarrow,\text{addr}} + \text{cost}_{L1\text{flush}} + \text{cost}_{\rightarrow,\text{cacheline}} + \text{cost}_{L2\text{write}} \)  
  + \( \text{rate}_{\text{coremiss}} \times \text{cost}_{\rightarrow,\text{cacheline}} + \text{cost}_{L1\text{insert}} \)

- \( \text{cost}_{wrM} = \text{rate}_{\text{coremiss}} \times \text{cost}_{\rightarrow,addr} + \text{cost}_{\text{dir lookup}} \)  
  + \( \text{cost}_{\rightarrow,\text{addr}} + \text{cost}_{L1\text{flush}} + \text{cost}_{\rightarrow,\text{cacheline}} \)  
  + \( \text{rate}_{\text{coremiss}} \times \text{cost}_{\rightarrow,\text{cacheline}} + \text{cost}_{L1\text{insert}} \)

L1 miss costs depend on the directory state & the request type.
NUCA-RA

\[ AML_{RA} = \text{cost}_{L1\text{access}} + \text{rate}_{L1\text{miss}} \times \text{cost}_{L1\text{miss,RA}} + \text{rate}_{core\text{miss}} \times \text{cost}_{core\text{miss,RA}} \]

\[ \text{cost}_{L1\text{miss,RA}} = \text{cost}_{L2\text{request}} + \text{cost}_{L1\text{insert}} \]

\[ \text{cost}_{L2\text{request}} = \text{cost}_{L2\text{access}} + \text{rate}_{L2\text{miss}} \times (\text{cost}_{DRAM} + \text{cost}_{L2\text{insert}}) \]

\[ \text{cost}_{core\text{miss,RA}} = \text{rate}_{read} \times (\text{cost}_{\rightarrow,addr} + \text{cost}_{\rightarrow,value}) + \text{rate}_{write} \times (\text{cost}_{\rightarrow,addr\&value} + \text{cost}_{\rightarrow,ack}) \]

Two message round trip for “core miss” memory accesses
Execution Migration (EM²)

\[
A_{ML}^{EM^2} = c_{L1\text{access}} + r_{L1\text{miss}} \times c_{L1\text{miss, } EM^2} + r_{\text{core miss}} \times c_{\rightarrow, \text{context}}
\]

\[
c_{L1\text{miss, } EM^2} = c_{L2\text{request}} + c_{L1\text{insert}}
\]

\[
c_{L2\text{request}} = c_{L2\text{access}} + r_{L2\text{miss}} \times (c_{\text{DRAM}} + c_{L2\text{insert}})
\]

From

\[
c_{\rightarrow, \text{data size}} = c_{\text{avg net dist}} + \left\lfloor \frac{\text{data size}}{\text{flit size}} \right\rfloor
\]

\[
c_{\rightarrow, \text{context}} = (12 \times 2 + 50\%) + \left\lfloor \frac{1088}{256} \right\rfloor = 36 + 5 = 41 \text{ cycles}
\]

Plus, we add an additional 3 cycles to restart a 5-stage pipeline with a new instruction at the migration destination core, resulting in \textit{44 cycles}.

\[1088\text{ bits} = 32\text{-bit regs} + 32\text{-bit instruc4on pointer reg} + 32\text{-bit status reg}\]
Library Cache Coherence (LCC)

\[ AML_{LCC} = rate_{read} \times cost_{read,LCC} + rate_{write} \times cost_{write,LCC} \]

\[ cost_{read,LCC} = cost_{L1access} + rate_{L1miss} \times cost_{L1readmiss,LCC} \]

\[ cost_{L1readmiss,LCC} = cost_{L2request} + rate_{coremiss} \times (cost_{\rightarrow,addr} + cost_{\rightarrow,cacheline}) + cost_{L1insert} \]

\[ cost_{write,LCC} = cost_{L1access} + rate_{L1miss} \times cost_{L1writemiss,LCC} + rate_{coremiss} \times (cost_{\rightarrow,addr&value} + cost_{\rightarrow,ack}) + cost_{expirationwait,LCC} \]

\[ cost_{L1writemiss,LCC} = cost_{L2request} + cost_{L1insert} \]

“Core miss” writes always turn into round-trips

Possible write delays due to unexpired timestamps
Results

We vary some of the model parameters from their defaults (shown below), and examine how each affects the AML for each implementation:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( cost_{L1} ) access</td>
<td>2 cycles</td>
<td>( rate_{read}, rate_{write} )</td>
<td>70%, 30%</td>
</tr>
<tr>
<td>( cost_{L1} ) insert/inv/flush</td>
<td>3 cycles</td>
<td>( rate_{rdI,wrI,rdS} )</td>
<td>40% + 40% + 5%</td>
</tr>
<tr>
<td>( cost_{L2} ) access</td>
<td>7 cycles</td>
<td>( rate_{wrS} )</td>
<td>5%</td>
</tr>
<tr>
<td>( cost_{L2} ) insert</td>
<td>9 cycles</td>
<td>( rate_{rdM} )</td>
<td>10%</td>
</tr>
<tr>
<td>( cost_{dir} ) lookup</td>
<td>2 cycles</td>
<td>( rate_{wrM} )</td>
<td>negligible</td>
</tr>
<tr>
<td>( size_{ack/address/value} )</td>
<td>32 bits</td>
<td>( rate_{L1} ) miss</td>
<td>6%</td>
</tr>
<tr>
<td>( size_{cache/line} )</td>
<td>512 bits</td>
<td>( rate_{L2} ) miss</td>
<td>1%</td>
</tr>
<tr>
<td>( size_{context} )</td>
<td>1088 bits</td>
<td>( rate_{core} ) miss</td>
<td>2%</td>
</tr>
<tr>
<td>( cost_{DRAM} )</td>
<td>250 cycles</td>
<td>( cost_{expiration wait,LCC} )</td>
<td>3 cycles</td>
</tr>
<tr>
<td>flit size</td>
<td>256 bits</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( cost_{avg net dist} )</td>
<td>36 cycles</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
DirCC is most affected by L1 cache miss rate due to its expensive coherence protocol costs (of two or more interconnect round-trips plus directory and often invalidation overheads) for every L1 miss.
A core miss may occur even if the data resides on-chip in some L1 cache for $\text{EM}^2$, RA and stores under LCC. $\text{EM}^2$ and RA perform well only when the core miss rate is low.
When the access pattern has high spatiotemporal locality, $EM^2$ core miss rates will be lower than the RA equivalent. By varying core miss rates only for $EM^2$, we can see that lower core miss rates favor $EM^2$. 
The execution contexts migrated in EM² are larger than both coherence messages and cache-line sizes; EM² is sensitive to the context size and performance degrades with larger context size.
To simulate different network bandwidths, we vary the size of a flit that can be transferred in one cycle. EM$^2$ is sensitive to network bandwidth and performance improves with higher on-chip network bandwidths.
Under LCC, “checked out” cache lines can only be read; writes can only be performed at the home, possibly incurring round-trips and write delays. Thus, LCC works very well for workloads highly skewed to load instruction.
There is no clear WINNER…

**RA and LCC**
Simple implementation, *but:*
- performance is subpar except for niche applications

**DirCC and EM²**
Often perform better, *but:*
- EM² is sensitive to context sizes and the amount of locality
- DirCC protocols are complex to implement and verify

**Hybrid Approaches**
1) EM² + RA  
2) EM² + LCC  
3) LCC + DirCC
Hybrid Approaches (1) EM² + RA

many accesses only retrieve one data element and return; turn them into RA

OCEAN, Graphite sim, 256 cores & threads, 16K L1$ + 64K L2$, non-local refs only
How should we decide between remote access and migration?
DP Oracle trades off migrations and Remote Accesses

- **DISTANCE**
  - If home core is far away, round-trip RA time > one-way migration time
  - So migrate if time(migration) < time(remote-access)

- **HISTORY**
  - Migrate on *second* access to the same remote core
  - On access to *native* core, *always* migrate

Both migrations and RAs useful

Possible online heuristic decision schemes
Hybrid Approaches (2) \( EM^2 + LCC \)

Recall:
\( EM^2 \) neatly avoids the problem of invalidating copies for writes since it completely eschews replication

May suffer from **high core misses** (e.g., frequently accessed read-data, mapped to the other core)

**Allow automatic replication by combining with LCC!**

- **Per-access decision** on whether to migrate or check out a read-only limited-time copy.
- A successful hybrid would reduce core miss rates and network traffic by obviating migrations needed to read shared data.

Possible online decision schemes

- Always migrate for writes, but check out a copy for reads
Hybrid Approaches (3) LCC + DirCC

Recall:
LCC performs well for workloads with high load-to-store ratios, but is limited in performance by its centralized writes

Allow cache lines to be “checked out” not only for reads but also for writes (i.e., locality for writes)

Per-access decisions

➤ **Read-only copy**
  can be checked out by multiple caches before any of the timestamps expire (only writes can be delayed)

➤ **Read-write copy**
  read-write copies can only be checked out to one client at a time. Both reads and writes can be delayed, and thus, it would need more carefully tuned timestamps.
Summary

Traditional approach to cache coherence is using directories

Execution Migration

- Distribute address space over different cores’ local caches
  - when address not cached at local core, migrate execution
- Deadlock-free
  - provably deadlock-free given a dual-issue multithreaded core

Library Cache Coherence

- Use timestamps for cache lines to expire locally, enabling data replication without the complex invalidation protocol
- No directories required
  - Library size does not grow with the number of cores

Searching for best-performing hybrid protocol