TSO-CC: Consistency-directed Coherence for TSO

Vijay Nagarajan
Multicores are here!

Power8: 12 cores

A8: 2 CPU + 4 GPU

Tile: 64 cores
Hardware Support for Shared Memory

• Cache coherence
  • ensures caches are transparent to programmer

• Memory consistency model
  • specifies what value a read can return

• Primitive synchronisation instructions
  • memory fence, atomic read-modify-write (RMW)
Cache Coherence

Initially data = 0, flag = 0

P1  P2

data = 1

flag = 1

while(!flag);

print data

The update to flag (data) should be visible to P2
Cache Coherence
Cache Coherence

P1
flag=0, shared
L1

P2
flag=0, shared
L1

... 

Pn
flag=0, shared, [P1=1, P2=1, P3=0,...Pn=0]

Last-Level Cache

Interconnect

Directory
Cache Coherence

P1
flag=1,-.
L1

P2
flag=0, shared
L1

Pn

... Interconnect

Last-Level Cache
flag=0, shared, [P1=1, P2=1, P3=0,…Pn=0]

Directory
Cache Coherence

P1
flag=1, mod.
L1

P2
flag=0, inv.
L1

... 

Pn
L1

Interconnect

Last-Level Cache
flag=0, mod., [P1=1, P2=0, P3=0, … Pn=0]

Directory
Memory Consistency

Initially data = 0, flag = 0

P1

data = 1
flag = 1

P2

while(!flag);

print data

If P2 sees update to flag, will it also see update to data?
Synchronisation Instructions

Initially data = 0, flag = 0

P1

- data = 1
- flag = 1

P2

while(!flag);

print data

If P2 sees update to flag, will it also see update to data?
Initially data = 0, flag = 0

P1

P2

data = 1

flag = 1

while(!flag);

print data

If P2 sees update to flag, will it also see update to data?
Performance Tension Programmability

- Simple, intuitive memory models like Sequential Consistency (SC) presumed too costly
  - None of the current processors enforce SC.
- Primitive synchronisation instructions expensive
  - For e.g. RMW in an Intel Sandybridge processor ~ 67 cycles
- Will cache coherence scale?
  - Coherence metadata per block scales linearly with processors
Performance co-exist Programmability

- Memory ordering via Conflict ordering
  - $SC = RC + 2\%$ [ASPLOS ’12];

- Efficient synchronisation instructions
  - Zero-overhead memory barriers [PACT ’10, ICS ’13, SC’14]
  - Fast, portable Intel x86 RMWs (latency halved) [PLDI ’13]

- Consistency-directed coherence
  - Coherence for x86 (TSO), without a sharer vector [HPCA ’14]
Performance co-exist Programmability

- Memory ordering via Conflict ordering
  - SC = RC + 2% [ASPLOS ‘12];

- Efficient synchronisation instructions
  - Zero-overhead memory barriers [PACT ’10, ICS ’13, SC’14]
  - Fast, portable Intel x86 RMWs (latency halved) [PLDI ’13]

- Consistency-directed coherence
  - Coherence for x86 (TSO), without a sharer vector [HPCA ’14]
Cache Coherence: Problem

Sharer vector increases linearly with number of processors
Cache Coherence

- Number of techniques attack directory and cache organisation
  - [Pugsley ’10] [Ferdman ’11] [Sanchez ’12]
Cache Coherence

- Number of techniques attack directory and cache organisation
  - [Pugsley ’10] [Ferdman ’11] [Sanchez ’12]

Can we do better if we consider memory consistency model?
Coherence and Consistency

- Cache coherence
  - ensures writes are visible to other processors

- Memory consistency
  - specifies when

- Traditional coherence protocols do this eagerly (target SC)
Eager Coherence for SC

- SC enforces w→r ordering
  - Write must be globally visible before a following read
- Writes are propagated eagerly to other processors
  - Via ensuring SWMR (Single Write Multiple Reader) invariant
- typically requires a sharer vector.
Lazy coherence for RC

- If consistency model is relaxed, why should coherence propagate writes eagerly?
- Why not propagate writes lazily, as per consistency model?
- This has been explored for release consistency (RC)
  - Earlier works (Lazy RC) [Kehler et al. ‘94][Kontothanasis et al. ‘95]
  - Recent Works [Choi et al. ‘11] [Ros and Kaxiras ‘12]
Lazy coherence for RC

- Synchronization variables not cached locally
- **release**: shared blocks written back to shared cache (w/r → release)
- **acquire**: shared blocks in local cache self invalidated (acquire → r/w)
- No sharer vector!
Lazy coherence for RC

Initially data = 0

P1

Data written to shared cache before release

data = 1

release(flag)

P2

acquire(flag)

r1 = data

self-invalidate
Research Question

- Lazy coherence for RC exist, but none for other relaxed models

Can we implement any memory consistency model with lazy coherence (with similar benefits)?
Lazy coherence for TSO

- Prevalent in x86 and SPARC architectures
- TSO relaxes w→r ordering
- RC based approached won’t work for TSO
- Absence of explicit synchronisation
Lazy coherence for TSO

Initially data = 0, flag = 0

P1

data = 1
flag = 1

P2

while(flag==0);
r1 = data
Lazy coherence for TSO

Initially data = 0, flag = 0

P1

data = 1

flag = 1

✘

✘

✘

P2

while(flag==0);

r1 = data

Requirements

✤ write-propagation

✤ TSO ordering
TSO-CC: Basic protocol

- Coherence state
  - Shared L2 directory maintains pointer to last-writer/owner
  - Local L1 states: Invalid, Exclusive, Modified
  - Shared L2 states: Shared, Uncached
  - No sharer vector!
TSO-CC: Basic protocol

- Writes write-through (state) to the shared cache in program order
  - Enforces $w \rightarrow w$

- Shared reads hit in L1s, but miss after threshold accesses
  - Ensures write propagation

- Upon an L1 miss, and last writer not the current processor, then self invalidate shared lines
  - Ensures $r \rightarrow r$
TSO-CC: Basic protocol

Initially data = 0, flag = 0

P1

P2

data = 1

flag = 1

while(flag==0);

r1 = data

Data available from shared cache before flag

Flag eventually misses

self invalidate

data misses, gets correct value

Initially data = 0, flag = 0

P1

P2

data = 1

flag = 1

while(flag==0);

r1 = data

Data available from shared cache before flag

Flag eventually misses

self invalidate

data misses, gets correct value
Guaranteed write/release propagation?

- Does correctness depend on the threshold used?
- No!
  - No guaranteed write propagation delay
  - No memory model guarantees this (including SC)
  - Especially TSO where write propagation is relaxed!
How to reduce self-invalidations?

P1

\[ \text{data}_1 = 1 \]
\[ \text{data}_2 = 1 \]

flag = 1

P2

\[ \text{while}(\text{flag} == 0); \]

\[ \text{r1} = \text{data}_2 \]

\[ \text{r2} = \text{data}_1 \]

Flag eventually misses

self invalidate

data\_2 misses
should it self invalidate?
Transitive reduction using timestamps

- Each processor maintains monotonically increasing timestamp
- Upon write, store current timestamp in local cache line
- Each processor also maintains a table of last seen timestamps from other processors
- Upon a miss, only self-invalidate if
  - If time stamp of the block > last seen timestamp from that processor
Transitive reduction using timestamps

<table>
<thead>
<tr>
<th>Time</th>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>data₁ = 2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>data₂ = 1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>flag = 1</td>
<td>while(flag==0); print data₂; print data₁</td>
</tr>
</tbody>
</table>

Last-seen timestamp

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P3</th>
<th>P4</th>
<th>P4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Transitive reduction using timestamps

<table>
<thead>
<tr>
<th>Time</th>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>data₁ = 2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>data₂ = 1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>flag = 1</td>
<td>while(flag == 0);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>print data₂</td>
</tr>
<tr>
<td></td>
<td></td>
<td>print data₁</td>
</tr>
</tbody>
</table>

Last-seen timestamp

<table>
<thead>
<tr>
<th>P1</th>
<th>P3</th>
<th>P4</th>
<th>P4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

time-stamp is 3, last-seen is 0, so self invalidate
Transitive reduction using timestamps

<table>
<thead>
<tr>
<th>Time</th>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>data(_1) = 2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>data(_2) = 1</td>
<td></td>
</tr>
</tbody>
</table>
| 3    | flag = 1 | while(flag==0); 
|      |     | print data\(_2\) 
|      |     | print data\(_1\) 

Last-seen timestamp

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P3</th>
<th>P4</th>
<th>P4</th>
</tr>
</thead>
<tbody>
<tr>
<td>time-stamp</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

time-stamp is 3, last-seen is 0, so self invalidate
Transitive reduction using timestamps

<table>
<thead>
<tr>
<th>Time</th>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>data_1 = 2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>data_2 = 1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>flag = 1</td>
<td>while(flag==0);</td>
</tr>
</tbody>
</table>

- print data_2
- print data_1

<table>
<thead>
<tr>
<th>Last-seen timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

- time-stamp is 3, last-seen is 0, so self invalidate
- time-stamp is 2, last-seen is 3, so no self invalidate
Implementation

- Gem5 full system cycle accurate simulator
- Ruby memory simulator with garnet interconnect
- 32 out-of-order cores
- Programs from Splash-2, Parsec and Stamp
- Unmodified code running on top of linux
- Verification
- Litmus tests using diy tool.
Storage Overheads

At 32 cores = 40% reduction; 128 cores = 80% reduction.

(4-bit accesses-counter, 12-bit timestamp; 1MB per L2 tile)

32 cores: 40% reduction
128 cores: 80% reduction
Execution times

TSO-CC-optimized 3% (7%) faster than Mesi (TSO-CC-basic)
Self Invalidations

TSO-CC-optimized reduces self-invalidations by 87%.
Verification: Cons.-directed Coherence

- Conventional coherence protocols verified against local invariants
  - E.g. SWMR: Single Writer Multiple reader invariant
- But TSO-CC relaxes SWMR by design!
- Need to verify coherence implementation against TSO now!
Verification: Cons.-directed Coherence

- Conventional coherence protocols verified against local invariants
  - E.g. SWMR: Single Writer Multiple reader invariant
- But TSO-CC relaxes SWMR by design!
- Need to verify coherence implementation against TSO now!

Is this Hard?
But Wait…

Would it suffice to verify conventional coherence protocols against local invariants (e.g. SWMR)?
But Wait…

Would it suffice to verify conventional coherence protocols against local invariants (e.g. SWMR)?

No! Because coherence protocol can interact with other components to result in elusive bugs!
Case-study: Bugs in Gem5

- TSO, MESI (ensures SWMR).
- x86-64 ISA, OOO processor
- Found 2 bugs due to incorrect interaction of LSQ and coherence protocol.
Bug 1

1. Ld2 issued before Ld1
2. Directory responds to Ld2 (in transmission)
3. St1 is issued; directory sends inv. to P2
4. Invalidate reaches P2 (before 2)
   /* Bug: Invalidate not forwarded to LSQ */
5. St2 is issued.
6. Ld1 is issued.

&A, and &B cached in P1 (not in P2)
Verification goal

Coherence protocol and its interaction with other components (pipeline, memory controllers etc.) should be verified against memory model.
Verification options

* Litmus testing
  * Pros: On any memory consistency model
  * Cons: requires construction of tests?, slow on simulators

* (Parameterised) Model checking
  * Pros: easy to use
  * Cons: impractical for non-SC non-RC model?

* Theorem proving
  * Pros: Has been successfully applied for real systems
  * Cons: Not fully automated?
Ongoing Work

• Iteratively generate interesting instructions for checking

• Choice of instructions guided by coverage

• Detected 3 real bugs in Gem5.
Summary

Coherence protocols must be designed and verified against MCMs!

Better designs?

Verification techniques?