EECS 570
Lecture 16
Advanced Topics in
Coherence and Consistency
Winter 2022
Prof. Yatin Manerkar
http://www.eecs.umich.edu/courses/eecs570/
Outline

• The Peekaboo scenario
  ☐ Causes and solutions
  ☐ The Coherence-Consistency Interface (CCI)

• Lazy coherence
  ☐ DeNovo

• Dependencies

• Out-of-thin-air values

• If we have time:
  ☐ Cumulativity
Coherence and Consistency

At a high level:

- **Coherence Protocols**: Propagation of writes to other cores
- **Consistency Models**: Ordering rules for visibility of reads and writes

- Reality: Coherence and consistency are often intertwined!
Static Coherence/Consistency Verification

Coherence Verifiers
Ignore consistency even when protocol affects consistency!

Consistency Verifiers
Assume abstract coherence instead of protocol in use!

Coherence and consistency often interwoven
Motivating Example - “Peekaboo”

1. Invalidation before use
   - Repeated inv before use → livelock [Kubiatowicz et al. ASPLOS 1992]

2. Livelock avoidance: allow destination core to perform
   - Individual Opt. → No violation
   - Combination ofOpts. → Violation!
Motivating Example - “Peekaboo”

- Consider mp with the livelock-avoidance mechanism:

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i1) St $x$ ← 1</td>
<td>(i3) Ld r1 ← [y]</td>
</tr>
<tr>
<td>(i2) St $y$ ← 1</td>
<td>(i4) Ld r2 ← [x]</td>
</tr>
<tr>
<td>Under TSO: Forbid r1=1, r2=0</td>
<td></td>
</tr>
</tbody>
</table>

Core 0

- Prefetch x
- Data (x = 0)
- Inv
- Inv-Ack
- Request y
- Data (y = 1)

Core 1

- x: Invalid
- y: Shared
- r1 ← [y]
- r2 ← 0
The Coherence-Consistency Interface (CCI)

CCI = guarantees that coherence protocol provides to rest of microarchitecture + memory ordering guarantees that rest of microarch. expects from coherence protocol

- SWMR, DVI, No Stale Data
- Expected Coherence

C6hMismatch

Consistency Violation!
SWMR and DVI

- Coherence = SWMR + DVI [Sorin et al. 2011]
- **SWMR:** Single Writer, Multiple Readers

Consistency model can dictate coherence protocol correctness!

**Logical time, not physical time**
Coherence Verification

- State space exploration
  - Murphi: symmetry reduces state space
    [Dill et al. ICCD 1992]

- Designing protocols for easier verification
  - Fractal Coherence [Zhang et al. MICRO 2010]
  - PVCoherence [Zhang et al. HPCA 2014]

- Coherence verifiers typically **ignore** consistency and focus on accesses to a single address
Coherence in Consistency Verification

Solution:
Allow instr. to use inv. data if it was oldest load or store in program order when request for data was issued.

Prior techniques cannot model CCI events!

Litmus Test mp

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i1) St [x] ← 1</td>
<td>(i3) Ld r1 ← [y]</td>
</tr>
<tr>
<td>(i2) St [y] ← 1</td>
<td>(i4) Ld r2 ← [x]</td>
</tr>
</tbody>
</table>

Under TSO: Forbid r1=1, r2=0
CCICheck [Manerkar et al. MICRO 2015]: Static CCI-aware consistency verification

Coherence Orderings (SWMR, DVI, etc.)

**Microarch spec**

**Litmus Test**

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i1) St: x ← 1</td>
<td>(i3) Ld r1 ← y</td>
</tr>
<tr>
<td>(i2) St: y ← 1</td>
<td>(i4) Ld r2 ← x</td>
</tr>
</tbody>
</table>

Under TSO: Forbid r1=1, r2=0

Microarchitectural happens-before (μhb) graph
Modelling CCI Events

- Need to model **per-cache occupancy**
  - Lazy coherence and partial incoherence (e.g. GPUs)
- Need to model **coherence transitions** that relate to consistency (e.g. Peekaboo)
- Solution: ViCL (Value in Cache Lifetime) Abstraction
ViCL: Value in Cache Lifetime Abstraction

- 4-tuple: $(\text{cache}_\text{id}, \text{address}, \text{data}_\text{value}, \text{generation}_\text{id})$
- `\text{cache}_\text{id}` and `\text{generation}_\text{id}` uniquely identify each cache line
- A ViCL 4-tuple maps on to the period of time over which the cache line serves the data value for the address
- ViCLs start at a \text{ViCL Create} event and end at a \text{ViCL Expire} event
ViCL: Value in Cache Lifetime

Conventional co-mp timeline (M = Modified, S = Shared)

Litmus Test co-mp

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i1) St</td>
<td>(i3) Ld r1 ← x</td>
</tr>
<tr>
<td>(i2) St</td>
<td>(i4) Ld r2 ← x</td>
</tr>
<tr>
<td>In TSO: r1=2, r2=2 Allowed</td>
<td></td>
</tr>
</tbody>
</table>

Now model requests, downgrades, etc.

Always Enumerated
- ViCL Create (C)
- ViCL Expire (E)

Enumerated as Needed
- $ Line Request (R)
- $ Line Downgrade (D)

Not Enumerated
- ViCL Create (C)
- ViCL Expire (E)
- $ Line Request (R)
ViCLs in \( \mu \)hb Graphs

- Use pipeline model from PipeCheck, but add ViCL nodes and edges

Litmus Test \texttt{co-mp}

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i1) St ([x])  ( \leftarrow ) 1</td>
<td>(i3) Ld ( r1 )  ( \leftarrow ) ([x])</td>
</tr>
<tr>
<td>(i2) St ([x])  ( \leftarrow ) 2</td>
<td>(i4) Ld ( r2 )  ( \leftarrow ) ([x])</td>
</tr>
</tbody>
</table>

In TSO: \( r1=2, r2=2 \) Allowed
• Livelock prevention mechanism allows use of stale data

• Can now reason about the presence/absence of values in specific caches

• Can now reason about the orderings of relevant coherence protocol events with other events in the execution

• No longer just an abstract notion of coherence!
Solutions to the Peekaboo problem

1. Remove one of the three contributing factors (with some caveats!)
   1. Remove prefetch/spec req., issue memory requests in order
   2. Remove “hold for one operation” => Livelock again!
   3. Remove invalidation-before-use => Deadlock!

2. Ensure all prior stores have reached the memory hierarchy and become visible to all cores, and all prior loads have been performed before the coherence request for the Peekaboo instruction is issued
Peekaboo Solution

- Livelock prevention mechanism allows use of stale data
- “Peekaboo” edge completes cycle => outcome forbidden
- Consistency maintained
Outline

• The Peekaboo scenario
  □ Causes and solutions
  □ The Coherence-Consistency Interface (CCI)

• Lazy coherence
  □ DeNovo

• Dependencies

• Out-of-thin-air values

• If we have time:
  □ Cumulativity
Lazy Coherence

• In a typical coherence protocol, writes cause other cores’ lines to be invalidated

• **Key Idea:** What if we only enforced coherence at synchronization points?

• This is what lazy coherence does
  ☐ Typical organization: private L1s, shared L2
  ☐ On a release, ensure all prior writes have been flushed to L2
  ☐ On an acquire, self-invalidate all shared lines
  ☐ Why does this work?

• Originally proposed for software distributed shared mem (software coherence) [Keleher et al. ISCA 1992]
  ☐ Subsequently proposed for hardware coherence [Kontothanassis et al. SC 1995]

• A bit similar to coherence decoupling [Huh et al. ASPLOS 2004], but older
Lazy Coherence

• Example: mp with and without lazy coherence
  □ Lazy coherence allows forbidden outcome even if operations executed in order

• Benefit of lazy coherence: reduced invalidations and traffic in the case of false sharing

• Drawback: synchronization can be expensive

• However, for properly-synchronized programs, lazy coherence will not be programmer-visible!
  □ Why?

• Lazy coherence will typically not work well with strong MCMs
  □ Coherence-consistency interface again!
  □ TSO-CC: Lazy coherence for TSO [Elver and Nagarajan HPCA 2014]
    ◇ Does not self-invalidate on every load, but must self-invalidate after a bounded number of loads to the same line
Related to Lazy Coherence

- DeNovo
  - Similar idea to lazy coherence, but assumes DRF software to further simplify the coherence protocol

- GPU Coherence in some GPUs
  - [Hower et al. ASPLOS 2014] [Wickerson et al. OOPSLA 2015]
Outline

• The Peekaboo scenario
  □ Causes and solutions
  □ The Coherence-Consistency Interface (CCI)

• Lazy coherence
  □ DeNovo

• Dependencies

• Out-of-thin-air values

• If we have time:
  □ Cumulativity
Dependencies

• Can be used to enforce local ordering in some MCM scenarios
  - e.g., can use dependency to order mp’s loads

• Address dependency:
  - Result of a load is address for subsequent load or store
    ```
lwz r1,0(r2);
xor r3,r1,r1;
lwzx r4,r3,r5;
```  

• Data dependency:
  - Result of a load is data for a subsequent store
    ```
lwz r1,0(r2);
xor r3,r1,r1;
addi r3,r3,1;
stw r3,0(r4);
```
Dependencies

- Control dependency:
  - Result of load controls branch condition, and load or store is after branch in program order [Sarkar et al. PLDI 2011]
    
    ```
    lwz r1,0(r2)
    cmpw r1,r1
    beq LC00
    LC00:
    lwz r3,0(r4)
    ```

  - ARM, Power, RISC-V enforce load-store control dependencies but not load-load control dependencies
    - Reasoning: want to be able to predict a branch and execute loads after the branch
    - Speculative stores that can be seen by other cores before the speculation commits are a really bad idea
    - Can put isync (Power) or isb (ARM) after the branch to enforce load-load control dependency
Outline

• The Peekaboo scenario
  □ Causes and solutions
  □ The Coherence-Consistency Interface (CCI)

• Lazy coherence
  □ DeNovo

• Dependencies

• Out-of-thin-air values

• If we have time:
  □ Cumulativity
Out of thin air (OOTA) values

- This section draws heavily on [Boehm and Demsky MSPC 2014]

Consider the following program:

- Should r1 = 42, r2 = 42 be allowed?

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i1) r1 = x;</td>
<td>(i2) r2 = y;</td>
</tr>
<tr>
<td>(i2) y = r1;</td>
<td>(i3) x = 42;</td>
</tr>
</tbody>
</table>

Outcome: r1 = 42, r2 = 42

How about the following program?

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i1) r1 = x;</td>
<td>(i2) r2 = y;</td>
</tr>
<tr>
<td>(i2) y = r1;</td>
<td>(i3) x = r2;</td>
</tr>
</tbody>
</table>

Outcome: r1 = 42, r2 = 42

- If r1 = r2 = 42, the value 42 has appeared "out of thin air"!
- Typical cases involve C11 relaxed atomics and Ld-St dependencies
OOTA in Hardware and Software

• Today, virtually all commercial ISAs enforce dependencies
  □ Result: no dependency cycles, and no OOTA
• However, the software situation is different
• Consider:

```c
if (x) {
    a[i++] = 1;
} else {
    a[i++] = 2;
}
```

• A compiler may want to hoist the increment of i out of the conditional
  □ But this would break the Ld-St dependency!
OOTA in Hardware and Software

• What about long, obscure dependence chains?

    y.store(f(x.load(memory_order_relaxed)),
    memory_order_relaxed);

    ✓ If we want to preserve dependencies, f would also need to
      maintain dependencies here (even if it had no atomic
      accesses and was compiled separately)

• Key takeaway: developing a programming language
  specification that forbids out-of-thin-air values but allows
  reasonable compiler optimizations has proven to be
  extremely difficult
Why does OOTA matter?

• If OOTA is permitted, how do we prove that e.g., malicious code cannot generate secret passwords out of thin air?

• Certain kinds of OOTA computation can break the SC-for-DRF guarantee
Outline

• The Peekaboo scenario
  □ Causes and solutions
  □ The Coherence-Consistency Interface (CCI)

• Lazy coherence
  □ DeNovo

• Dependencies

• Out-of-thin-air values

• If we have time:
  □ Cumulativity
Cumulativity

In an MCA or rMCA system, fences need only enforce ordering locally (among operations on the same core)

- Below, LOCFENCE denotes a fence that only orders instructions on the same core

But what about an nMCA system?
- Local ordering is not enough! (dependencies don’t work either!)
- Fences must be cumulative!
- ISAs with nMCA like Power have cumulative fences

Write-to-Read Causality (WRC) litmus test

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
<th>Core 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i1) x = 1;</td>
<td>(i2) r1 = x;</td>
<td>(i5) r2 = y;</td>
</tr>
<tr>
<td>(i3) LOCFENCE;</td>
<td>(i3) LOCFENCE;</td>
<td>(i6) LOCFENCE;</td>
</tr>
<tr>
<td>(i4) y = 1;</td>
<td>(i4) y = 1;</td>
<td>(i7) r3 = x;</td>
</tr>
</tbody>
</table>

MCA and rMCA forbid: r1 = 1, r2 = 1, r3 = 0
Cumulativity

- Lightweight cumulative fence (lwsync on Power)
  - Set A: any writes **observed** by the fencing core before the fence (even those from other cores)
  - Set B: writes after the fence (set B)
  - lwsync orders writes in set A before those in set B

**Write-to-Read Causality (WRC) litmus test**

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
<th>Core 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i1) x = 1;</td>
<td>(i2) r1 = x;</td>
<td>(i5) r2 = y;</td>
</tr>
<tr>
<td></td>
<td>(i3) lwsync;</td>
<td>(i6) lwsync;</td>
</tr>
<tr>
<td></td>
<td>(i4) y = 1;</td>
<td>(i7) r3 = x;</td>
</tr>
</tbody>
</table>

Power forbids: r1 = 1, r2 = 1, r3 = 0

- lwsyncs do not order writes with respect to subsequent reads
  - Do not enforce a total order!
  - Similar to release-acquire semantics

- Aside: the lwsync on core 2 could be replaced with a dependency
Cumulativity

• lwsyncs cannot forbid IRIW!

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
<th>Core 2</th>
<th>Core 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1;</td>
<td>y = 1;</td>
<td>r1 = x;</td>
<td>r3 = y;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>lwsync;</td>
<td>lwsync;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r2 = y;</td>
<td>r4 = x;</td>
</tr>
</tbody>
</table>

Power allows: r1 = 1, r2 = 0, r3 = 1, r4 = 0

• Must use heavyweight cumulative fence (sync on Power)
  All writes observed by the fencing core before the fence must be made visible to all other cores, and all reads before the fence must be performed before any memory operations after the fence are performed

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
<th>Core 2</th>
<th>Core 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1;</td>
<td>y = 1;</td>
<td>r1 = x;</td>
<td>r3 = y;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sync;</td>
<td>sync;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r2 = y;</td>
<td>r4 = x;</td>
</tr>
</tbody>
</table>

Power forbids: r1 = 1, r2 = 0, r3 = 1, r4 = 0