EECS 570
Lecture 9
Snooping Coherence
Winter 2019
Prof. Thomas Wenisch

http://www.eecs.umich.edu/courses/eecs570/

Slides developed in part by Profs. Falsafi, Hardavellas, Nowatzyk, and Wenisch of EPFL, Northwestern, CMU, U-M.
Announcements

Kick-off for PA #2 on Friday 2/15 in discussion

No lecture next Monday 2/18

Project Milestone 1 due 2/25
Readings

For Today:
- Daniel J. Sorin, Mark D. Hill, and David A. Wood, A Primer on Memory Consistency and Cache Coherence (Ch. 6 & 7)

For Wednesday:
- Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. Reactive NUCA: near-optimal block placement and replication in distributed caches. ISCA 2009
Unit 3 - Cache Coherence & Memory Consistency
Cache Coherence

- Two $100 withdrawals from account #241 at two ATMs
  - Each transaction maps to thread on different processor
  - Track `accts[241].bal` (address is in `r3`)

Processor 0
0: addi r1,accts,r3
1: ld 0(r3),r4
2: blt r4,r2,6
3: sub r4,r2,r4
4: st r4,0(r3)
5: call spew_cash

Processor 1
0: addi r1,accts,r3
1: ld 0(r3),r4
2: blt r4,r2,6
3: sub r4,r2,r4
4: st r4,0(r3)
5: call spew_cash
• Scenario I: processors have no caches
  - No problem

<table>
<thead>
<tr>
<th>Processor 0</th>
<th>Processor 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: addi r1,accts,r3</td>
<td>0: addi r1,accts,r3</td>
</tr>
<tr>
<td>1: ld 0(r3),r4</td>
<td>1: ld 0(r3),r4</td>
</tr>
<tr>
<td>2: blt r4,r2,6</td>
<td>2: blt r4,r2,6</td>
</tr>
<tr>
<td>3: sub r4,r2,r4</td>
<td>3: sub r4,r2,r4</td>
</tr>
<tr>
<td>4: st r4,0(r3)</td>
<td>4: st r4,0(r3)</td>
</tr>
<tr>
<td>5: call spew_cash</td>
<td>5: call spew_cash</td>
</tr>
</tbody>
</table>
Cache Incoherence

- Scenario II: processors have write-back caches
  - Potentially 3 copies of `accts[241].bal`: memory, p0$, p1$
  - Can get incoherent (inconsistent)
Snooping Cache-Coherence Protocols

Bus provides serialization point

Each cache controller “snoops” all bus transactions

- take action to ensure coherence
  - invalidate
  - update
  - supply value

- depends on state of the block and the protocol
Scalable Cache Coherence

- **Scalable cache coherence**: two part solution

- **Part I: bus bandwidth**
  - Replace non-scalable bandwidth substrate (bus)...
  - ...with scalable bandwidth one (point-to-point network, e.g., mesh)

- **Part II: processor snooping bandwidth**
  - Interesting: most snoops result in no action
  - Replace non-scalable broadcast protocol (spam everyone)...
  - ...with scalable directory protocol (only spam processors that care)
Approaches to Cache Coherence

• Software-based solutions
  □ Mechanisms:
    ○ Mark cache blocks/memory pages as cacheable/non-cacheable
    ○ Add “Flush” and “Invalidate” instructions
    ○ *When are each of these needed?*
  □ Could be done by compiler or run-time system
  □ Difficult to get perfect (e.g., what about memory aliasing?)
  □ Will revisit this briefly in Unit 3...

• Hardware solutions are far more common
  □ In Unit 2, we study schemes that rely on broadcast over a bus
Write-Through Scheme 1: Valid-Invalid Coherence

- **t1**: Store A=1

- **t2**: BusWr A=1

- **t3**: Invalidate A

### Valid-Invalid Coherence

- Allows multiple readers, but must write through to bus
  - Write-through, no-write-allocate cache
- All caches must monitor (aka “snoop”) all bus traffic
  - Simple state machine for each cache frame
Valid-Invalid Snooping Protocol

Actions:
Ld, St, BusRd, BusWr

Write-through, no-write-allocate cache

1 bit of storage overhead per cache frame
Write Through Scheme 2: Write-Update Coherence

- **t1:** Store A=1
- **t2:** BusWr A=1
- **t3:** Snarf A

**Write-Update Coherence**

- Instead of invalidation, “Snarf” new value of A off the Bus
- But, 15% of cache accesses are stores
  - Tremendous bus and cache tag BW requirement
Supporting Write-Back Caches

- Write-back caches drastically reduce bus write bandwidth

- Key idea: add notion of “ownership” to Valid-Invalid
  - Mutual exclusion – when “owner” has only replica of a cache block, it may update it freely
  - Sharing – multiple readers are ok, but they may not write without gaining ownership

- Need to find which cache (if any) is an owner on read misses
- Need to eventually update memory so writes are not lost
Modified-Shared-Invalid (MSI) Protocol

- Three states tracked per-block at each cache
  - Invalid – cache does not have a copy
  - Shared – cache has a read-only copy; clean
    - Clean == memory is up to date
  - Modified – cache has the only copy; writable; dirty
    - Dirty == memory is out of date

- Three processor actions
  - Load, Store, Evict

- Five bus messages
  - BusRd, BusRdX, BusInv, BusWB, BusReply
  - Could combine some of these
Modified-Shared -Invalid (MSI) Protocol

Invalid → Shared

Load / BusRd

1: Load A

P1

A [↓S]: 0

2: BusRd A

Bus

3: BusReply A

A: 0

P2

A [↓l]
Modified-Shared -Invalid (MSI) Protocol

Invalid \( \rightarrow \) Shared

Load / BusRd

Shared \( \leftrightarrow \) BusRd / [BusReply]

Load / --

1: Load A
2: BusRd A
3: BusReply A

A [S]: 0
A [I S]: 0
A: 0

P1
P2
Modified-Shared -Invalid (MSI) Protocol

Invalid → Load / BusRd → Shared → BusRd / [BusReply] → Load / -- → Evict / --

P1
A [S]: 0
A [S I]
Bus
A: 0

P2

Evict A
Modified-Shared -Invalid (MSI) Protocol

Invalid → Shared:
- Load / BusRd
- BusRdX / [BusReply]

Shared → Invalid:
- BusRd / [BusReply]
- Load / --

Invalid → Modified:
- Store / BusRdX

Modified → Invalid:
- Load, Store / --

Load, Store / --

1: Store A
2: BusRdX A
3: BusReply A

P1

A [S I]: 0

P2

A [† M]: 0 1

Bus

A: 0
Modified-Shared-Invalid (MSI) Protocol

1: Load A
2: BusRd A
3: BusReply A
4: Snarf A

A [I S]: 1
A [M S]: 1

P1

P2

Bus

A: 0 1
Modified-Shared -Invalid (MSI) Protocol

Invalid

BusRdX, BusInv / [BusReply]

Load / BusRd

BusRd / [BusReply]

Load / --

Evict / --

Modified

Store / BusRdX

Load, Store / --

Shared

Store / BusInv

P1

A [S M]: 2

P2

A [S I]

2: BusInv A

1: Store A aka “Upgrade”

Bus

A: 1
Modified-Shared -Invalid (MSI) Protocol

- **Invalid**
  - Load / BusRd
  - BusRdX, BusInv / [BusReply]
  - Store / BusRdX
  - BusRdX / BusReply

- **Modified**
  - Load, Store / --
  - BusRdX / BusReply
  - Store / BusInv

- **Shared**
  - Load / --
  - BusRd / [BusReply]
  - Evict / --
  - BusRd / BusReply

- **Bus**
  - A: 1
  - A: 2
  - A: 3

- **Processors**
  - P1
  - A [M]: 2
  - 1: Store A
  - 3: BusReply A
  - P2
  - A [M]: 3
  - 2: BusRdX A
Modified-Shared -Invalid (MSI) Protocol

- Load / BusRd
- BusRdX, BusInv / [BusReply]
- Shared
- BusRd / [BusReply]
- Evict / --
- BusRdX, BusInv / [BusReply]
- Invalid
- Store / BusRdX
- BusRdX / BusReply
- Modified
- Evict / BusWB
- Store / BusInv
- Load, Store / --

Bus

P1
A [I]

P2
A [M | I]: 3

1: Evict A
2: BusWB A

A: 1 3
**MSI Protocol Summary**

Cache Actions:
- Load, Store, Evict

Bus Actions:
- BusRd, BusRdX, BusInv, BusWB, BusReply

**Diagram**

- **Invalid**
  - Load / BusRd
  - BusRdX, BusInv / [BusReply]
  - Store / BusRdX
  - Evict / BusWB

- **Shared**
  - Load / --
  - BusRd / [BusReply]

- **Modified**
  - Evict / --
  - BusRd / BusReply
  - Store / BusInv
  - BusRdX / BusReply
  - Load, Store / --
Update vs. Invalidate

• Invalidation is bad when:
  - Single producer and many consumers of data

• Update is bad when:
  - Multiple writes by one CPU before read by another
  - Junk data accumulates in large caches (e.g., process migration)
Coherence Decoupling
[Huh, Chang, Burger, Sohi ASPLOS04]

- After invalidate, keep stale data around
  - On subsequent read, speculatively supply stale value
  - Confirm speculation with a normal read operations
  - Need a branch-prediction-like rewind mechanism
  - Completely solves false sharing problem
  - Also addresses “silent”, “temporally-silent” stores

- Can use update-like mechanisms to improve prediction
  - Paper explores a variety of update heuristics
  - E.g., piggy-back value of 1st write on invalidation message
MESI Protocol (aka Illinois)

• MSI suffers from frequent read-upgrade sequences
  - Leads to two bus transactions, even for private blocks
  - Uniprocessors don’t have this problem

• Solution: add an “Exclusive” state
  - Exclusive – only one copy; writable; clean
    - Can detect exclusivity when memory provides reply to a read
  - Stores transition to Modified to indicate data is dirty
    - No need for a BusWB from Exclusive
**MESI Protocol Summary**

- **Invalid**
  - Load / BusRd (reply from mem)
  - Evict / --
  - BusRdX / BusReply
  - Load / BusRd (reply from cache)
- **Shared**
  - Evict / --
  - Store / BusInv
- **Exclusive**
  - BusRdX / BusReply
  - Store / BusInv
- **Modified**
  - Store / --
  - Load, Store / --
  - BusRd / BusReply

- **BusRd** / [BusReply]
MOESI Protocol

- MESI must write-back to memory on $M \rightarrow S$ transitions
  - Because protocol allows “silent” evicts from shared state, a dirty block might otherwise be lost
  - But, the writebacks might be a waste of bandwidth
    - E.g., if there is a subsequent.store
    - Common case in producer-consumer scenarios

- Solution: add an “ Owned” state
  - Owned – shared, but dirty; only one owner (others enter S)
    - Entered on $M \rightarrow S$ transition, aka “downgrade”
  - Owner is responsible for writeback upon eviction
MOESI Framework

[Sweazey & Smith ISCA86]

M - Modified (dirty)
O - Owned (dirty but shared) WHY?
E - Exclusive (clean unshared) only copy, not dirty
S - Shared
I - Invalid

Variants
- MSI
- MESI
- MOSI
- MOESI
DEC Firefly

• An update protocol for write-back caches

• States
  - Exclusive – only one copy; writeable; clean
  - Shared – multiple copies; write hits write-through to all sharers and memory
  - Dirty – only one copy; writeable; dirty

• Exclusive/dirty provide write-back semantics for private data

• Shared state provides update semantics for shared data
  - Uses “shared line” bus wire to detect sharing status

• Well suited to producer-consumer; process migration hurts
DEC Firefly Protocol Summary

- **Exclusive**
  - Load Miss & !SL
  - Store & !SL / --
  - Store

- **Shared**
  - Load Miss & SL
  - BusRd, BusWr / BusReply
  - Store & SL / BusWr

- **Dirty**
  - Load, Store / --
  - BusRd / BusReply (update mem)
  - BusWr / snarf
  - BusRd / BusReply
Non-Atomic State Transitions

Operations involve multiple actions
- Look up cache tags
- Bus arbitration
- Check for writeback
- Even if bus is atomic, overall set of actions is not
- Race conditions among multiple operations

Suppose P1 and P2 attempt to write cached block A
- Each decides to issue BusUpgr to allow S → M

Issues
- Handle requests for other blocks while waiting to acquire bus
- Must handle requests for this block A

We will revisit this at length in Unit 3
Scalability problems of Snoopy Coherence

• **Prohibitive bus bandwidth**
  - Required bandwidth grows with # CPUs...
  - ... but available BW per bus is fixed
  - Adding busses makes serialization/ordering hard

• **Prohibitive processor snooping bandwidth**
  - All caches do tag lookup when ANY processor accesses memory
  - Inclusion limits this to L2, but still lots of lookups

• **Upshot**: bus-based coherence doesn’t scale beyond 8–16 CPUs