EECS 570  
Lecture 9  
Snooping Coherence  
Winter 2021  
Prof. Satish Narayanasamy  

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

Slides developed in part by Profs. Falsafi, Hardavellas, Nowatzyk, and Wenisch of EPFL, Northwestern, CMU, U-M.
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 Monday:
- 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`)
No-Cache, No-Problem

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
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

- r take action to ensure coherence
  - m invalidate
  - m update
  - m supply value
- r depends on state of the block and the protocol
Scalable Cache Coherence

• Scalable cache coherence: two part solution

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

• Part II: processor snooping bandwidth
  r Interesting: most snoops result in no action
  r Replace non-scalable broadcast protocol (spam everyone)...
  r ...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

- 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
  r Invalid – cache does not have a copy
  r Shared – cache has a read-only copy; clean
    m Clean == memory is up to date
  r Modified – cache has the only copy; writable; dirty
    m Dirty == memory is out of date

• Three processor actions
  r Load, Store, Evict

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

Inv alid  

Load / BusRd  

Sha red

1: Load A

P1

A [+ S]: 0

2: BusRd A

Bus

P2

A [I]

3: BusReply A

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

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

P1
A [S]: 0

P2
A [+S]: 0

Bus
A: 0

Load / BusRd

BusRd / [BusReply]

Load / --
Modified-Shared -Invalid (MSI) Protocol

Invalid

Load / BusRd

Evict / --

Shared

BusRd / [BusReply]

Load / --

Evict A

P1

A [S]: 0

Bus

A: 0

P2

A [S I]

Modified-Shared -Invalid (MSI) Protocol

- **Invalid**: Load / BusRd
  - BusRdX / [BusReply]
- **Shared**: BusRd / [BusReply]
  - Load / --
- **Modified**: Evict / --
- **Load, Store**: Store / BusRdX
- **Modified**: P1
  - A [S I]: θ
  - 1: Store A
  - 2: BusRdX A
  - 3: BusReply A
- **Modified**: P2
  - A [↑ M]: θ
  - 1
- **Bus**: A: 0
Modified-Shared -Invalid (MSI) Protocol

States:
- Invalid
- Shared
- Modified

Transitions:
- Load / BusRd
- BusRdX / [BusReply]
- Evict / --
- Store / BusRdX
- BusRd / BusReply

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

Memory:
- A [+ S]: 1
- A [+ M S]: 1
- A: θ 1
Modified-Shared -Invalid (MSI) Protocol

- Invalid
  - Store / BusRdX
  - Inv

- Shared
  - Load / BusRd
  - BusRdX, BusInv / [BusReply]

- Modified
  - Evict / --
  - Store / BusInv
  - Mod

- Load, Store / --

1: Store A aka “Upgrade”

2: BusInv A

Bus

P1

A [SM]: 2

P2

A [S I]

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

invalid

Modified

Load / BusRd

BusRdX, BusInv / [BusReply]

Load / --

BusRd / [BusReply]

Evict / --

Store / BusInv

Store / BusRdX

Load, Store / --

Modified

Shared

P1

A [M] = 2

P2

A [M] = 3

Bus

A: 1

1: Store A

2: BusRdX A

3: BusReply A
Modified-Shared -Invalid (MSI) Protocol

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

**Scenario:**

1. **P1**: A [I]: 3
   - Evict A
2. **P2**: A [M: I]: 3
   - BusWB A

**Bus State:** A: ± 3
MSI Protocol Summary

Cache Actions:
- Load, Store, Evict

Bus Actions:
- BusRd, BusRdX
- BusInv, BusWB, BusReply
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 1\textsuperscript{st} 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

Inv (Invalid)
- Load / BusRd (reply from mem)
- BusRdX, BusInv / [BusReply]
- Evict / --

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

Exclusive
- Evict / --
- BusRdX / BusReply
- Store / BusRdX
- Evict / BusWB

Modified
- Store / --
- Load / --
- Load, Store / --
MOESI Protocol

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

• Solution: add an “Owned” state
  r Owned – shared, but dirty; only one owner (others enter S)
    m Entered on M→S transition, aka “downgrade”
  r 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
DEC Firefly

• An update protocol for write-back caches

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

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

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

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

Load Miss & !SL

Excl usive

BusRd, BusWr / BusReply

Store & !SL / --

Shared

Load Miss & SL

BusRd / BusReply

BusWr / snarf

Store & SL / BusWr

Store

BusWr / snarf

BusRd / BusReply (update mem)

Dirty

Load, Store / --
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
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