EECS 570
Lecture 8
Snooping Coherence
Winter 2022
Prof. Yatin Manerkar

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

• Discussion this Friday on PA2
Consider this program:

```
lock(l1);
flagA = true;
while (!flagB) {}
//update m
unlock(l1);
```

```
lock(l2);
while (!flagA) {}
flagB = true;
//update n
unlock(l2);
```

What if we replace lock and unlock with begin_transaction and end_transaction?
Transactions ≠ Critical Sections

What is wrong with this program?

```cpp
begin_transaction();
flagA = true;
while (!flagB) {}
//update m
end_transaction();

begin_transaction();
while (!flagA) {}
flagB = true;
//update n
end_transaction();
```

A less contrived example...

Queue* queueA = new Queue();
Queue* queueB = new Queue();

```cpp
begin_transaction();
...
queueA->enqueue(val1);
while (queueB->empty()){}
//access queueB
...
end_transaction();

begin_transaction();
...
queueB->enqueue(val2);
while (queueA->empty()){}
//access queueA
...
end_transaction();
```
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

- **Scenario I**: processors have no caches
  - No problem
  - Only one location where the value can reside!
Cache Incoherence

Scenario II: processors have write-back caches

- Potentially 3 copies of `accts[241].bal`: memory, p0$, p1$
- Can get incoherent (out of sync)
Coherence, more formally defined

- Coherence can be thought of as two invariants:
  - **SWMR** = Single-Writer Multiple Readers
    - There is either one writer or zero or more readers of a cache line at any (logical) time
  - **DVI** = Data Value Invariant
    - All cores see the values of the address/line update in the same order
    - e.g. if core 0 observes $x$ go from 0 -> 1 -> 3, then all other cores must observe this sequence of updates as well
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 later

• Hardware solutions are far more common
  - Today we will 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

```
A [V]: 0 1  
|           
| Write-through  
| No-write-allocate

A [V]: 0  

A [V]: 0 1  
|           
| Main Memory
```

```
P1

P2

Bus
```
Valid-Invalid Snooping Protocol

Actions:
Ld, St, BusRd, BusWr
Write-through, no-write-allocate cache
1 bit of storage overhead per cache frame

Valid
Load / --
Store / BusWr

Invalid
Load / BusRd
BusWr

Store / BusWr
Write Through Scheme 2: Write-Update Coherence

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

Load / BusRd

Invalid ➔ Shared

1: Load A

P1: A [↓ S]: 0

P2: A [I]

2: BusRd A

Bus

3: BusReply A

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

Load / BusRd

Invalid  \rightarrow  \text{Shared}

BusRd / [BusReply]  \rightarrow  \text{Load} / --

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

P1

A [S]: 0

P2

A [+ S]: 0

Bus

A: 0

P1

A [S]: 0

P2

A [+ S]: 0

Bus

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

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

P1

A [S]: 0

Bus

A: 0

P2

A [S I]
Modified-Shared -Invalid (MSI) Protocol

States:
- Invalid
- Shared
- Modified

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

Example sequence:
1: Store A
2: BusRdX A
3: BusReply A
Modified-Shared -Invalid (MSI) Protocol

Invalid

Load / BusRd

BusRdX / [BusReply]

Evict / --

Store / BusRdX

Shared

Load / --

BusRd / [BusReply]

Modified

Load, Store / --

Load A

Bus

A [↓ S]: 1

A [M S]: 1

BusReply A

Snarf A

P1

2: BusRd A

3: BusReply A

P2

1: Load A

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

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

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

- **Modified**
  - Store / BusInv
  - BusRd / BusReply

1: Store A aka “Upgrade”

- **P1**
  - A [§ M]: 2

- **P2**
  - A [§ I]

- **2: BusInv A**

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

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

- **Shared**
  - Load / --
  - BusRd / [BusReply]
  - BusRdX / BusReply
  - Store / BusInv

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

**Process P1**
- A [M I]: 2
- 3: BusReply A

**Process P2**
- A [↑ M]: 3
- 2: BusRdX A

**Bus**
- A: 1

**Notes:**
- Store A
- BusRdX A

**Diagram Details:**
- Modified state transitions to Invalid and Shared.
- Invalid state transitions to Shared.
- Shared state transitions to BusRd.
- BusRd state transitions to Load.
- Load state transitions to Share.
- Share state transitions to BusRdX, BusInv, and BusRd.
- BusRd state transitions to Store.
- Store state transitions to BusRdX and BusInv.
- BusRdX state transitions to BusReply.
- BusInv state transitions to BusReply.
- BusReply state transitions to Bus.

**Legend:**
- Modified
- Invalid
- Shared
- Modified
- Load
- Store
- BusRd
- BusRdX
- BusInv
- BusReply
- Bus
Modified-Shared -Invalid (MSI) Protocol

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

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

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

P1
- A [I]

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

Bus
- A: 1 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 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 cache)

BusRdX / [BusReply], BusInv / --

Evict / --

BusRd / BusReply

Shared

Load / --

Exclusive

BusRdX / BusReply

Evict / --

BusRd / BusReply

Modified

Store / BusInv

BusRdX / BusReply

Evict / BusWB

Store / --

Load, Store / --
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; writable; 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

- Only evictions in Dirty state trigger a Writeback

**States:**
- **Exclusive**
- **Shared**
- **Dirty**

**Transitions:**
- **Load Miss & !SL**
- **Store & !SL / --**
- **Load/Store Miss & SL**
- **BusRd / BusReply**
- **BusRd, BusWr / BusReply**
- **BusWr / snarf**
- **BusRd / BusReply (update mem)**
- **Load, Store / --**
- **Store Miss & !SL**
- **Store**

**Actions:**
- **Load, Store**
- **Store**
- **Load**
- **Store Miss & !SL**
- **BusRd**
- **BusWr**
- **BusRd, BusWr**
- **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

You’ll see a lot of this in PA2! 😊
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