Readings

For today

- Sorin et al - A Primer on Memory Consistency and Cache Coherence, Ch. 8

No reading for Monday 2/12. Yay!

For Wed 2/14

Announcements

Discussion this Friday to kick off Programming Assignment 2

Project Milestone 1 Report due 2/14

Midterm Exam on 2/21
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    ○ Dirty == memory is out of date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 \rightarrow \text{Load / BusRd} \rightarrow \text{Shared}

1: Load A

P1
A [\lφ]: 0

P2
A [l]

2: BusRd A

Bus

3: BusReply A

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

Invalid

Load / BusRd

Shared

BusRd / [BusReply]

Load / --

1: Load A

P1

A [S]: 0

3: BusReply A

A: 0

P2

A [↑ S]: 0

2: BusRd A
Modified-Shared -Invalid (MSI) Protocol

Invalid -> Load / BusRd -> Shared
Invalid -> Evict / --

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

P1
A [S]: 0

P2
A [S I]

Bus
A: 0

Evict A
Modified-Shared -Invalid (MSI) Protocol

Invalid

Load / BusRd

BusRdX / [BusReply]

Shared

Load / --

Evict / --

Store / BusRdX

Modified

Load, Store / --

P1

A [S I]: 0

P2

A [∩ M]: 0 1

1: Store A

2: BusRdX A

3: BusReply A

Bus

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

Invalid ➔ Load / BusRd

BusRdX / [BusReply] ➔ Invalid

Load / -- ➔ Shared

BusRd / BusReply ➔ Shared

Evict / -- ➔ Invalid

Store / BusRdX ➔ Invalid

Modified ➔ BusRd / BusReply

Load, Store / -- ➔ Modified

1: Load A ➔ P1

A [↓ S]: 1 ➔ P1

2: BusRd A ➔ Bus

3: BusReply A ➔ Bus

P2

A [M S]: 1 ➔ P2

4: Snarf A ➔ P2
Modified-Shared -Invalid (MSI) Protocol

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

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

Modified
- Store / BusRdX

Process P1:
- A [S M]: 2

Process P2:
- A [S I]

Bus
- A: 1

1: Store A aka “Upgrade”
2: BusInv A
Modified-Shared -Invalid (MSI) Protocol

Invalid

Modified

Load / BusRd

BusRdX / [BusReply]

BusInv, Evict / --

Load / --

P1

A [M I]: 2

3: BusReply A

P2

A [† M]: 3

2: BusRdX A

1: Store A

Bus

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

Invalid

Load / BusRd

BusRdX / [BusReply]

BusInv, Evict / --

Shared

Load / --

BusRd / [BusReply]

Store / BusRdX

Evict / BusWB

BusRdX / BusWB

Modified

Load, Store / --

Bus

A: 1 3

A [M 1]: 3

1: Evict A

2: BusWB A

P1

P2
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
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

Invalid

Load / BusRd (reply from cache)

BusRdX / [BusReply]

BusInv, Evict / --

Shared

Exclusive

Load, Store / --

Modified

Store / --

Evict / BusWB

Store / BusInv

BusRd / BusReply

BusRdX / BusReply

Load / --

Load / BusRd (reply from mem)
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 / --
- Load, Store / --
- BusRd, BusWr / BusReply

**Shared**
- Load Miss & SL
- BusRd / BusReply
- BusWr / snarf
- Store & SL / BusWr
- BusRd / BusReply (update mem)

**Dirty**
- BusWr / snarf

Flow chart: Exclusive -> Shared -> Exclusive

Legend:
- 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 $S \rightarrow M$

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
Implementing Snoopy Coherent SMPs
Outline

- Coherence Control Implementation
- Writebacks, non-atomicity, serialization/order
- Hierarchical caches
- Split Busses
- Deadlock, livelock & starvation
- TLB Coherence
Base Coherence SMP design

- Single-level write-back cache
- MSI coherence protocol
- One outstanding memory request per CPU
- Atomic memory bus transactions
  - No interleaving of transactions
- Atomic operations within process
  - One operation at a time in program order

- We will incrementally add more concurrency/complexity
Cache Controller & Tags

• On a miss in a uniprocessor
  ❑ Assert request for bus
  ❑ Wait for bus grant
  ❑ Drive address & command lines
  ❑ Wait for command to be accepted by target device
  ❑ Transfer data

• In a Snoop-based SMP, cache controller must:
  ❑ Monitor bus and CPU
    ❑ Can view as two controllers, bus-side and CPU-side
    ❑ With a single cache level, tags often duplicated or dual-ported
  ❑ Respond to bus transactions as needed
Reporting Snoop results: How?

• Collective response from caches must appear on bus

• Wired-OR signals
  - Shared: assert if any cache has a copy (recall: Firefly protocol)
  - Dirty/Inhibit: asserted if some cache has a dirty copy
    - Needn’t indicate which; it knows what it needs to do
    - Also indicates that memory controller should ignore request
  - Snoop-valid: asserted when OK to check other two signals

• Need arbitration/priority scheme for cache-to-cache xfers
  - Which cache should supply data in shared state?
Reporting Snoop results: When?

- Memory needs to know what, if anything, to do

- Solution 1: Fixed # of clocks after request message
  - Usually needs duplicate tags to avoid contention w/ CPU
  - Pentium Pro, HP Servers, Sun Enterprise

- Solution 2: Variable delay
  - Memory assumes cache will supply data until all say “sorry”
  - Less conservative, more flexible, more complex
Writebacks

• Allow CPU to proceed on a miss ASAP
  ❑ Fetch the requested block
  ❑ Do the writeback of the victim later

• Requires write buffer
  ❑ Must snoop/handle bus transactions in write buffer
  ❑ Must maintain order of writes/reads (maintain consistency)
Base Snoopy Organization

Diagram showing the Base Snoopy Organization:

- Bus-side controller
- Tags and state for snoop
- Cache data RAM
- Tags and state for P
- Processor-side controller
- Comparators
- Write-back buffer
- Snoop state
- Addr
- Cmd
- Data buffer
- System bus
Serialization and Ordering

- CPU-cache handshake must preserve serialization
  - E.g., write in S state \(\rightarrow\) first obtain permission

- Write completion for SC \(\rightarrow\) need to send invalidations
  - Wait to get bus, then can consider writes complete
    - Must serialize bus transactions in program order
      - Split transaction bus still must retire transactions in order
Multi-level Cache Hierarchies

• How to snoop with multi-level caches?
  ❑ Independent bus snooping at each level?
  ❑ Multiple duplicate tag arrays
  ❑ Maintain cache inclusion
The Inclusion Property

- **Inclusion** means L2 is a superset of L1 (ditto for L3...)
  - Also, must propagate “dirty” bit through cache hierarchy

- Now, only need to snoop last level cache
  - If L2 says not present, can’t be in L1 either

- **Inclusion takes effort to maintain**
  - L2 must track what is cached in L1
  - On L2 replacement, must flush corresponding blocks from L1

  _How can this happen?_

  **Consider:**
  1. L1 block size < L2 block size
  2. different associativity in L1
  3. L1 filters L2 access sequence; affects LRU ordering
Possible Inclusion Violation

- Step 1: L1 miss on c
- Step 2: a displaced to L2
- Step 3: b replaced by c

2-way set associative L1

- a, b, c have the same L1 idx bits
- b, c have the same L2 idx bits
- a, {b, c} have different L2 idx bits
Is inclusion a good idea?

• Most common inclusion solution:
  ❑ Ensure L2 holds a superset of L1I and L1D
  ❑ On L2 replacement or coherence action that supplies data, forward actions to L1s

• But...
  ❑ Restricted L2 associativity may limit blocks in split L1s
  ❑ Not that hard to always snoop the L1s

• Many recent designs do not maintain inclusion
Shared Caches

- Share low level caches among multiple processors
  - Sharing L1 adds to latency, *unless* multithreaded processor

- Advantages
  - Eliminates need for coherence protocol at shared level
  - Reduces latency within sharing group
  - Processors essentially prefetch for each other
  - Can exploit working set sharing
  - Increases utilization of cache hardware

- Disadvantages
  - Higher bandwidth requirements
  - Increased hit latency
  - May be more complex design
  - Lower effective capacity if working sets don’t overlap
Split-transaction (Pipelined) Bus

- Supports multiple simultaneous transactions

Atomic Transaction Bus

Split-transaction Bus
Potential Problems

• Two transactions to same block (conflicting)
  ☐ Mid-transaction snoop hits

• Buffer requests and responses
  ☐ Need flow control to prevent deadlock

• Ordering of Snoop responses
  ☐ when does snoop response appear wrt data response
Possible Solutions

• Disallow conflicting transactions
• NACK for flow control
• Out-of-order responses
  ☐ snoop results presented with data response
Deadlock, Livelock, Starvation

- **Deadlock:**
  - all system activity ceases
  - Cycle of resource dependences

- **Livelock:**
  - no processor makes forward progress
  - constant on-going transactions at hardware level
  - e.g. simultaneous writes in invalidation-based protocol

- **Starvation:**
  - some processors make no forward progress
  - e.g. interleaved memory system with NACK on bank busy
Correctness problems in SMP Bus

- Request-reply protocols can lead to *deadlock*
  - When issuing requests, must service incoming transactions
  - e.g. cache awaiting bus grant must snoop & flush blocks
  - else may not respond to request that will release bus: deadlock

- Livelock:
  - window of vulnerability problem [Kubi et al., MIT]
  - Handling invalidations between obtaining ownership & write
  - Solution: don’t let exclusive ownership be stolen before write

- Starvation:
  - solve by using fair arbitration on bus and FIFO buffers
Deadlock Avoidance

- Responses are never delayed by requests waiting for a response
- Responses are guaranteed to be sunk
- Requests will eventually be serviced since the number of responses is bounded by outstanding requests
- Must classify transactions according to deadlock and coherence semantics
Case Study: Sun Enterprise 10000

- How far can you go with snooping coherence?
- Quadruple request/snoop bandwidth using four address busses
  - each handles 1/4 of physical address space
  - impose *logical* ordering for consistency: for writes on same cycle, those
    on bus 0 occur “before” bus 1, etc.
- Get rid of data bandwidth problem: use a network
  - E10000 uses 16x16 crossbar betw. CPU boards & memory boards
  - Each CPU board has up to 4 CPUs: max 64 CPUs total
- 10.7 GB/s max BW, 468 ns unloaded miss latency
- See “Starfire: Extending the SMP Envelope”, IEEE Micro 1998