EECS 570
Lecture 11
Directory-based Coherence
Winter 2016
Prof. Thomas Wenisch
http://www.eecs.umich.edu/courses/eecs570/

Slides developed in part by Profs. Adve, Falsafi, Hill, Lebeck, Martin, Narayanasamy, Nowatzyk, Reinhardt, Roth, Smith, Singh, and Wenisch.
Announcements

- Midterm, Wednesday 2/24
Readings

For Wednesday 2/17:


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)

BusRdX, BusInv / [BusReply]

Shared

Load / BusRd (reply from cache)

Evict / --

Invalid

BusRdX / BusReply

Evict / --

Shared

Store / BusInv

BusRd / [BusReply]

Exclusive

Load / --

BusRdX / BusReply

Store / --

Modified

Load, Store / --

Store / BusWB

BusRdX / BusReply
**MOESI Protocol**

- MESI must write-back to memory on $M\to S$ 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 $M\to S$ 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

Load Miss & !SL

Exclusive

BusRd, BusWr / BusReply

Exclusive

Store & !SL / --

Shared

Load Miss & SL

BusRd / BusReply

BusWr / snarf

Store & SL / BusWr

Dirty

BusWr / snarf

BusRd / BusReply (update mem)

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

We will revisit this at length later
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
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 components of a snoopy cache organization, including:
- P (Processor)
- Bus-side controller
- Tags and state for snoop
- Comparator
- Write-back buffer
- Tag
- Data buffer
- System bus
- Processor-side controller
- Addr
- Cmd
- Tags and state for P

The diagram illustrates the flow of data and commands between these components.
Serialization and Ordering

• CPU-cache handshake must preserve serialization
  □ E.g., write in S state → first obtain permission

• Write completion → 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

**Diagram:**
- Direct mapped L2
- 2-way set associative L1

Symbols:
- 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 (Pipilined) Bus

- Supports multiple simultaneous transactions
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
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
Directory-Based Coherence
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)
Directory Coherence Protocols

- Observe: physical address space statically partitioned
  - Can easily determine which memory module holds a given line
    - That memory module sometimes called “home”
  - Can’t easily determine which processors have line in their caches
- Bus-based protocol: broadcast events to all processors/caches
  - Simple and fast, but non-scalable

- **Directories**: non-broadcast coherence protocol
  - Extend memory to track caching information
  - For each physical cache line whose home this is, track:
    - **Owner**: which processor has a dirty copy (i.e., M state)
    - **Sharers**: which processors have clean copies (i.e., S state)
  - Processor sends coherence event to home directory
    - Home directory only sends events to processors that care
Basic Operation: Read

Load A (miss)

Get-S A

A: Shared, #1

Data A
Basic Operation: Write

Node #1

Read A (miss)

Read A

Fill A

Invalidate A

Inv-Ack A

Directory

A: Shared, #1

Get-M A

A: Mod., #2

Node #2

Data A
Centralized Directory

- **Single directory** contains a copy of cache tags from all nodes

- **Advantages:**
  - Central serialization point: easy to get memory consistency (just like a bus...)

- **Problems:**
  - Not scalable (imagine traffic from 1000’s of nodes...)
  - Directory size/organization changes with number of nodes
Distributed Directory

- Distribute directory among memory modules
  - Memory block = coherence block (usually = cache line)
  - “Home node” → node with directory entry
    - Usually also dedicated main memory storage for cache line
  - Scalable – directory grows with memory capacity
    - Common trick: steal bits from ECC for directory state
  - Directory can no longer serialize accesses across all addresses
    - Memory consistency becomes responsibility of CPU interface
What is in the directory?

- Directory State
  - Invalid, Exclusive, Shared, ... ("stable" states)
  - # outstanding invalidation messages, ... (" transient" states)
- Pointer to exclusive owner
- Sharer list
  - List of caches that may have a copy
  - May include local node
  - Not necessarily precise, but always conservative
Directory State

- Few stable states – 2-3 bits usually enough

- Transient states
  - Often 10’s of states (+ need to remember node ids, ...)
  - Transient state changes frequently, need fast RMW access
  - Design options:
    - Keep in directory: scalable (high concurrency), but slow
    - Keep in separate memory
    - Keep in directory, use cache to accelerate access
    - Keep in protocol controller
      - Transaction State Register File – like MSHRs
Pointer to Exclusive Owner

• Simple node id – $\log_2$ nodes
• Can share storage with sharer list (don’t need both...)
• May point to a group of caches that internally maintain coherence (e.g., via snooping)
• May treat local node differently
**Sharer List Representation**

- Key to scalability – must efficiently represent node subsets
- Observation: most blocks cached by only 1 or 2 nodes
  - But, there are important exceptions (synchronization vars.)

*OLTP workload [Data from Nowatzyk]*
Idea #1: Sharer Bit Vectors

- One bit per processor / node / cache
  - Storage requirement grows with system size

```
0 1 1 0 0 0 0 1
```
Idea #2: Limited Pointers

- Fixed number (e.g., 4) of pointers to node ids
- If more than $n$ sharers:
  - Recycle one pointer (force invalidation)
  - Revert to broadcast
  - Handle in software (maintain longer list elsewhere)
Idea #3: Linked Lists

- Each node has fixed storage for next (prev) sharer
- Doubly-linked (Scalable Coherent Interconnect)
- Singly-linked (S3.mp)
- Poor performance:
  - Long invalidation latency
  - Replacements – difficult to get out of sharer list
    - Especially with singly-linked list... – how to do it?
Directory representation optimizations

- Coarse Vectors (CV)
- Cruise Missile Invalidations (CMI)
- Tree Extensions (TE)
- List-based Overflow (LO)
Clean Eviction Notification

• Should directory learn when clean blocks are evicted?

• Advantages:
  □ Avoids broadcast, frees pointers in limited pointer schemes
  □ Avoids unnecessary invalidate messages

• Disadvantages:
  □ Read-only data never invalidated (extra evict messages)
  □ Notification traffic is unnecessary
  □ New protocol races
Sparse Directories

• Most of memory is invalid; why waste directory storage?

• Instead, use a directory cache
  - Any address w/o an entry is invalid
  - If full, need to evict & invalidate a victim entry
  - Generally needs to be highly associative
Cache Invalidation Patterns

• Hypothesis: On a write to a shared location, # of caches to be invalidated is typically small

• If this isn’t true, directory is no better than broadcast/snoop

• Experience tends to validate this hypothesis
Common Sharing Patterns

• Code and read-only objects
  - No problem since rarely written

• Migratory objects
  - Even as number of caches grows, only 1-2 invalidations

• Mostly-read objects
  - Invalidations are expensive but infrequent, so OK

• Frequently read/written objects (e.g., task queues)
  - Invalidations frequent, hence sharer list usually small

• Synchronization objects
  - Low-contention locks result in few invalidations
  - High contention locks may need special support (e.g. MCS)

• Badly-behaved objects
Designing a Directory Protocol: Nomenclature

- Local Node (L)
  - Node initiating the transaction we care about

- Home Node (H)
  - Node where directory/main memory for the block lives

- Remote Node (R)
  - Any other node that participates in the transaction
Read Transaction

- L has a cache miss on a load instruction

![Diagram]

1: Get-S

2: Data
4-hop Read Transaction

- L has a cache miss on a load instruction
  - Block was previously in modified state at R

![Diagram of 4-hop Read Transaction]

1: Get-S
2: Recall
3: Data
4: Data

State: M
Owner: R
3-hop Read Transaction

- L has a cache miss on a load instruction
  - Block was previously in modified state at R
An Example Race: Writeback & Read

- L has dirty copy, wants to write back to H
- R concurrently sends a read to H

To make your head really hurt:
Can optimize away SI^A & Put-Ack!
L and H each know the race happened, don’t need more msgs.
Store-Store Race

- Line is invalid, both L and R race to obtain write permission

1: Get-M

4: Data [ack=0]

3: Fwd-Get-M to L; New Owner: R

2: Fwd-Get-M

6: Fwd-Get-M

7: Data [ack=0]

Race! Stall for Data, do 1 store, then Fwd to R
Worst-case scenario?

- L evicts dirty copy, R concurrently seeks write permission

Race! Put-M floating around! Wait till its gone...

Put-M from NonOwner: Race!
L waiting to ensure Put-M gone...

IIA

1: Put-M
2: Get-M
3: Fwd-Get-M
4: Data [ack=0]
5: 
6: Put-Ack
Design Principles

• Think of sending and receiving messages as separate events
• At each “step”, consider what new requests can occur
  ❑ E.g., can a new writeback overtake an older one?
• Two messages traversing same direction implies a race
  ❑ Need to consider both delivery orders
    ❑ Usually results in a “branch” in coherence FSM to handle both orderings
  ❑ Need to make sure messages can’t stick around “lost”
    ❑ Every request needs an ack; extra states to clean up messages
  ❑ Often, only one node knows how a race resolves
    ❑ Might need to send messages to tell others what to do
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
Sources of correctness problems

- 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