Advanced Computer Architecture

Coherence

Fall 2016

Pejman Lotfi-Kamran

Adapted from slides originally developed by Profs. Hill, Hoe, Falsafi and Wenisch of CMU, EPFL, Michigan, Wisconsin

Where Are We?

Fr	Sa	Su	Mo	Tu
	27-Shahrivar		29-Shahrivar	
	3-Mehr		5-Mehr	
	10-Mehr		12-Mehr	
	17-Mehr		19-Mehr	
	24-Mehr		26-Mehr	
	1-Aban		3-Aban	
	8-Aban		10-Aban	
	15-Aban		17-Aban	
	22-Aban		24-Aban	
	29-Aban		1-Azar	
	6-Azar		8-Azar	
	13-Azar		15-Azar	
	20-Azar		22-Azar	
	27-Azar		29-Azar	
	4-Dey		6-Dey	

- This Lecture
  - Coherence
- Next Lecture:
  - Advanced Coherence

Roadmap

- Cache Coherence
  - ➔Basic Coherence
  - Bus-based
  - Directory-based

Why Shared Memory?

- Pluses
  - For applications looks like multitasking uniprocessor
  - For OS only evolutionary extensions required
  - Easy to do communication without OS
  - Software can worry about correctness first then performance
- Minuses
  - Proper synchronization is complex
  - Communication is implicit so harder to optimize
  - Hardware designers must implement
- Result
  - Traditionally bus-based Symmetric Multiprocessors (SMPs), and now the Chip Multiprocessors (CMPs) are the most common form of parallel general-purpose machines
  - Embedded systems (mobile phones) are often not shared memory

In More Detail

- Efficient Naming
  - virtual to physical using TLBs
  - ability to name relevent portions of objects
- Ease and efficiency of caching
  - caching is natural and well understood
  - can be done in HW automatically
- Communication Overhead
  - low since protection is built into memory system
  - easy for HW to packetize requests / replies
- Integration of latency tolerance
  - demand-driven: consistency models, prefetching, multithreaded
  - Can extend to push data to PEs and use bulk transfer

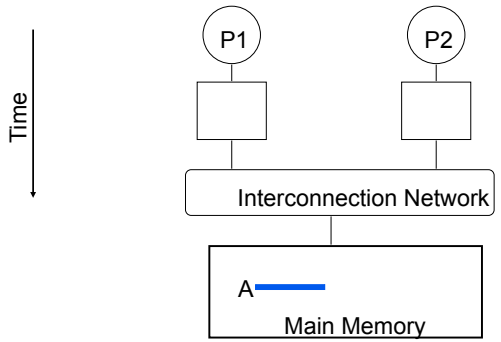
First Shared-Memory Machines: Symmetric Multiprocessors (SMP)

The basic form:

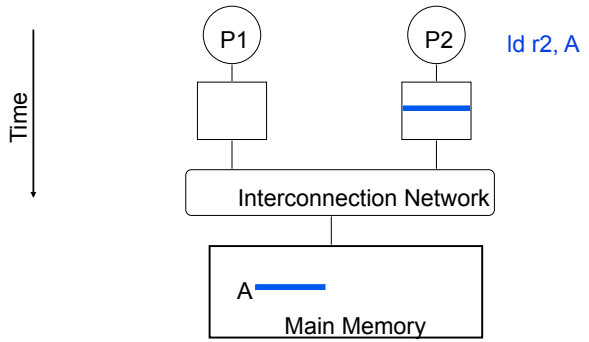
- Multiple (micro-)processors
- Each has cache(s)
- Connect with logical bus (ordered events)

- Implement Snooping Cache Coherence Protocol
  - Broadcast all cache "misses" on bus
  - All caches "snoop" bus and may act
  - Memory responds otherwise

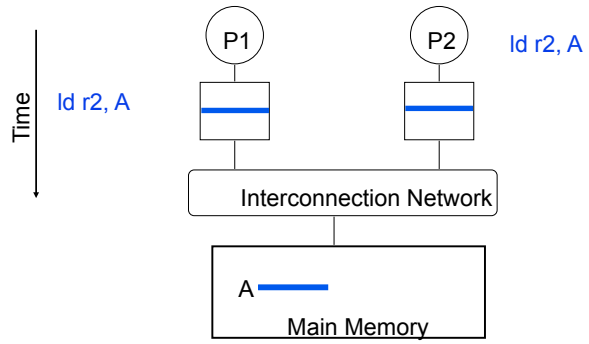
Cache Coherent Shared Memory



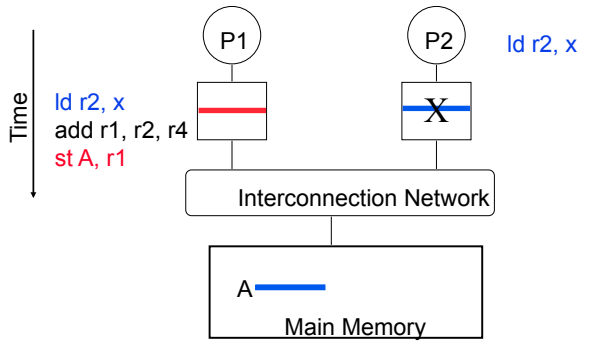
Cache Coherent Shared Memory



Cache Coherent Shared Memory



Cache Coherent Shared Memory

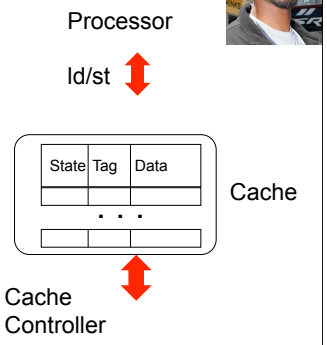


Snooping Cache-Coherence Protocols

- Bus provides serialization point
- Each cache controller "snoops" all bus transactions
  - relevant transactions if for a block it contains
  - take action to ensure coherence
    - invalidate
    - update
    - supply value
  - depends on state of the block and the protocol
- Simultaneous Operation of Independent Controllers

Snooping Design Choices

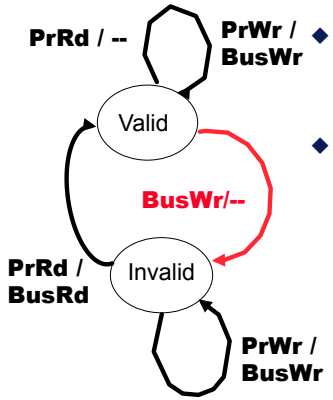
- Controller updates state of blocks in response to processor and snoop events and generates bus transactions
- Often have duplicate cache tags
- Snoopy protocol
  - set of states
  - state-transition diagram
  - actions
- Basic Choices
  - write-through vs. write-back
  - invalidate vs. update



A 2-State Write-Through Invalidation Protocol

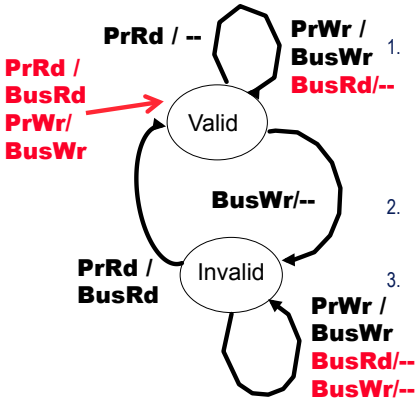
- 2-State Protocol
  - Use the valid bit to indicate presence
  - Write through on all writes/stores
  - Invalidate copies when "snooping" a bus write
- Processor
  - On a read
    - If Valid, read
    - If Invalid, fetch block from memory
  - On a write, write through (no allocate)
- Cache controller (bus side)
  - On a bus write (from another processor)
    - If Valid, invalidate (mark as Invalid)

The Simple Invalidate Snooping Protocol



- Write-through, no-write-allocate cache
- Actions/Reactions:
  - X/Y: Action X, Reaction Y
  - PrRd: Processor Read
  - PrWr: Processor Write
  - BusRd: Fetch a cache block
  - BusWr: Write through one word
  - --: No action

Complete Simple Invalidate



1. When the block does not exist in the cache (tag mismatch)
   - All arches the same as if the protocol is in "Invalid"
2. BusRd in "Valid" results in a null action "—"
3. BusRd and BusWr in "Invalid" result in a null "action"

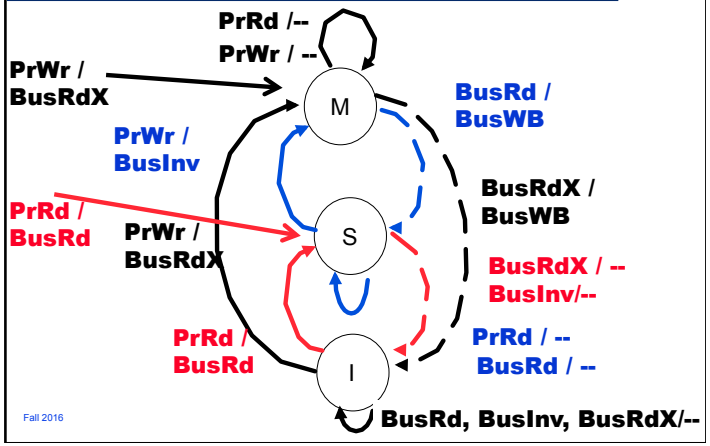
A 3-State Write-Back Invalidation Protocol

- 2-State Protocol
  - \+ Simple hardware and protocol
  - Bandwidth (every write goes on bus!)
- 3-State Protocol (MSI)
  - Modified
    - one cache has valid/latest copy
    - memory is stale
  - Shared
    - one or more caches have valid copy
  - Invalid
- Must invalidate other copies before entering modified
- Requires bus transaction (order and invalidate)

MSI Processor and Bus Actions

- Processor:
  - PrRd
  - PrWr
  - Writeback on replacement of modified block
- Bus
  - Bus Read (BusRd) Read without intent to modify, data could come from memory or another cache
  - Bus Read-Exclusive (BusRdX) Read with intent to modify, must invalidate all other caches copies
  - Bus Invalidate (BusInv or BusUpgr) Bus invalidate or upgrade, intent to "upgrade" an existing copy, must invalidate other copies
  - Writeback (BusWB) cache controller puts contents on bus and memory is updated
  - Definition: cache-to-cache transfer occurs when another cache satisfies BusRd or BusRdX request
- Let's draw it!

MSI State Diagram



An example

Proc Action	P1 State	P2 state	P3 state	Bus Act	Data from
1. P1 read A	S	--	--	BusRd	Memory
2. P3 read A	S	--	S	BusRd	Memory
3. P3 write A	I	--	M	BusInv	None
4. P1 read A	S	--	S	(BusWB) BusRd	P3's cache
5. P2 read A	S	S	S	BusRd	Memory

- Single writer, multiple reader protocol
- Why Modified to Shared?
- What if not in any cache?
  - Read, Write produces 2 bus transactions!

Summary

- Coherence
  - About whether value at an address is the most up-to-date
  - Do not confuse with consistency: order of loads/stores to multiple addresses
- 2-State protocol
  - Write-through invalidate
  - Uses existing uniprocessor valid bit for cache blocks
  - Changes cache controller to snoop on Bus Writes and invalidate
- 3-State protocol
  - Allows for write-back caches
  - More complex cache controller/bus design
- 3Cs for misses are now 4Cs (coherence)

Advanced Coherence

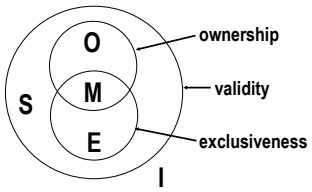
- 4-state machines
  - What do we use the fourth state for?
- Invalidate vs. Update
- Coherence implmentation
  - Actual machines are bigger than 3-4 states
  - Intermediate states
- Multi-level caches
  - Coherence in hierarchies
  - Inclusion
- Please note!
  - In the following diagrams, we will not show all transitions
    - ▲E.g., Transitions on a miss (not tag match) are identical to those from "I"
  - On the exam/homework, if asked, must draw all transitions!

4-State (MESI) Invalidation Protocol

- Often called the Illinois protocol
- Modified (dirty)
- Exclusive (clean unshared) only copy, not dirty
- Shared
- Invalid
- Requires shared signal to detect if other caches have a copy of block
- Cache Flush for cache-to-cache transfers
  - Only one can do it though
- What does state diagram look like?

More Generally: MOESI

- [Sweazey & Smith ISCA86]
- M - Modified (dirty)
- O - Owned (dirty but shared) WHY?
- E - Exclusive (clean unshared) only copy, not dirty
- S - Shared
- I - Invalid

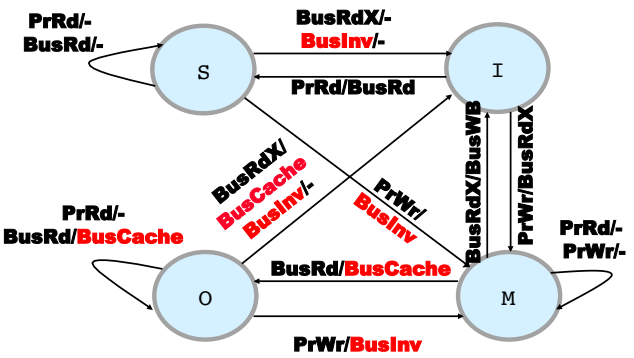


- Variants
  - MSI
  - MESI
  - MOSI
  - MOESI

Berkeley Protocol

- Multiprocessor Workstation (SPUR)
- Uses "O" state to optimize cache-to-cache transfers
- Read miss:
  - If block Modified (M) , transfer cache-to-cache
  - If block Shared (S), read from memory,
  - If block Owner (O) read from owner
- Write hit:
  - On Modified (M), proceed, on Owner (O) Invalidate
- Write miss:
  - Same as Read miss

Berkeley Diagram

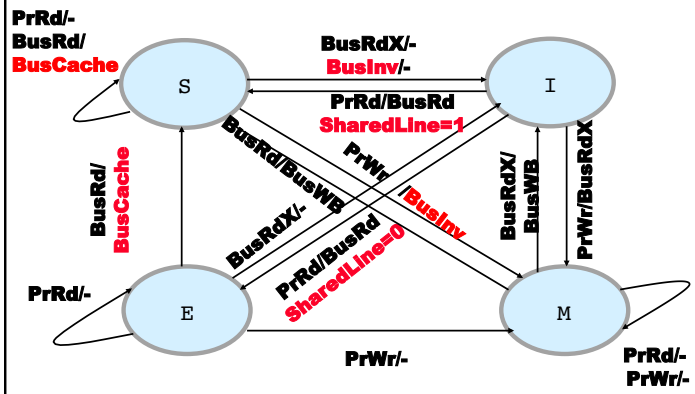


MP bus: support for cache-to-cache transfer + invalidation

Illinois Protocol

- Implemented in SGI multiprocessors in 1980's
- Missed data always comes from caches, bus SharedLine
- Read miss:
  - If block Modified (M), transfer cache-to-cache, and write back
  - If block Shared (S), Exclusive (E) transfer cache-to-cache
- Write hit:
  - On Modified (M) or Exclusive (E), proceed, on Shared (S), invalidate
- Write miss:
  - Same as Read miss

Illinois Diagram

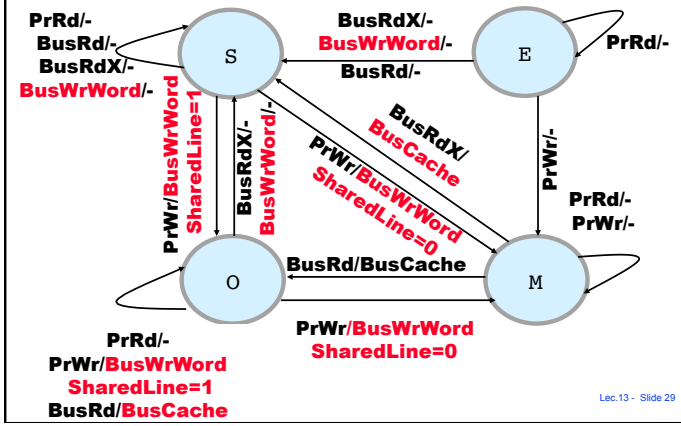


 Needs a shared line to vote for any "S" copies

Xerox Dragon Protocol

- Write-update protocol
  - Exclusive & Shared-Dirty states
- Read miss:
  - If block Modified (M), Owner (O), cache-to-cache transfer
  - If block Exclusive (E), Shared (S), data comes from memory
- Write hit:
  - On Modified (M), Exclusive (E), proceed, on Shared (S), Owner (O) update
- Write miss:
  - Same as Read miss
  - If SharedLine set, must update

Xerox Dragon Diagram



Tradeoffs in Protocol Design

- New State Transitions
- What Bus Transactions
- Cache block size
- Workload dependence
- Compute bandwidth, miss rates, from state transitions

Computing Bandwidth

- Why bandwidth?
- How do I compute it?
- Monitor State Transitions
  - tells me bus transactions
  - I know how many bytes each bus transaction requires

MESI State Transitions and Bandwidth

FROM/TO	NP	I	E	S	M
NP	--	--	BusRd 6+64	BusRd 6+64	BusRdX 6+64
I	--	--	BusRd 6+64	BusRd 6+64	BusRdX 6+64
E	--	--	--	--	--
S	--	--	NA	--	**BusInv** 6
M	BusWB 6 + 64	BusWB 6+64	NA	BusWB 6 + 64	--

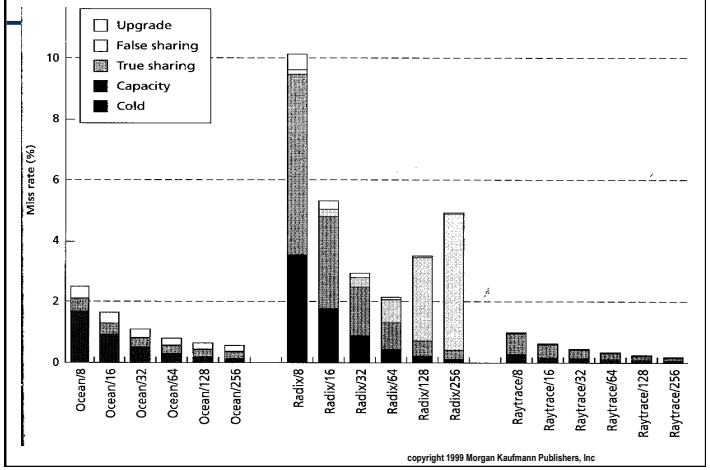
Bandwidth of MSI vs. MESI

- For an X MIPS/MFLOPS processor
  - use with measured state transition counts to obtain transitions/sec
- Compute state transitions/sec
- Compute bus transactions/sec
- Compute bytes/sec
- What is BW savings of MESI over MSI?
- Difference between protocols is Exclusive State
  - Add BusInv/BusUpgr for E->M transtion
- Result is very small benefit!
  - Small number of E->M transitions
  - Only 6 bytes on bus

Cache Block Size

- Block size is unit of transfer and of coherence
  - Doesn't have to be, could have coherence smaller [Goodman]
- Uniprocessor 3C's
  - (Compulsory, Capacity, Conflict)
- 4th C: Coherence Miss Type
  - True Sharing miss fetches data written by another processor
  - False Sharing miss results from independent data in same coherence block
- Increasing block size
  - Usually fewer 3C misses but more bandwidth
  - Usually more false sharing misses
- Increasing cache size
  - Usually fewer capacity/conflict misses (& compulsory don't matter)
  - No effect on true/false "coherence" misses (so may dominate)

Cache Block Size: Miss Rate



Invalidate vs. Update

- Pattern 1:

```
for i = 1 to k
    P1(write, x);          // one write before reads
    P2--PN-1(read, x);
end for i
```

- Pattern 2:

```
for i = 1 to k
    for j = 1 to m
      P1(write, x);        // many writes before reads
    end for j
    P2(read, x);
end for i
```

Invalidate vs. Update, cont.

- Pattern 1 (one write before reads)
  - N = 16, M = 10, K = 10
  - Update
    - Iteration 1: N regular cache misses (70 bytes)
    - Remaining iterations: update per iteration (14 bytes; 6 cntrl, 8 data)
  - Total Update Traffic = 16*70 + 9*14 = 1246 bytes
    - book assumes 10 updates instead of 9…
  - Invalidate
    - Iteration 1: N regular cache misses (70 bytes)
    - Remaining: P1 generates upgrade (6), 15 others Read miss (70)
  - Total Invalidate Traffic = 16*70 + 9*6 + 15*9*17 = 10,624 bytes
- Pattern 2 (many writes before reads)
  - Update = 1400 bytes
  - Invalidate = 824 bytes

Qualitative Sharing Patterns

- [Weber & Gupta, ASPLOS3]
- Read-Only
- Migratory Objects
  - Maniputalated by one processor at a time
  - Often protected by a lock
  - Usually a write causes only a single invalidation
- Synchronization Objects
  - Often more processors imply more invalidations
- Mostly Read
  - More processors imply more invalidations, but writes are rare
- Frequently Read/Written
  - More processors imply more invalidations

But in More Detail ...

- How does memory know another cache will respond so it need not?
- Is it okay a cache miss is not an atomic event (check tags, queue for bus, get bus, etc.)?
- What about L1/L2 caches?

Snooping SMP Design Goals

- Goals
  - Correctness
  - High Performance
  - Minimal Hardware => reduced complexity & cost
- Often at odds
  - High Performance

    => multiple outstanding low-level events
    => more complex interactions
    => more potential correctness bugs

Base Cache Coherence Design

- Single-level write-back cache
- Invalidation protocol
- One outstanding memory request per processor
- Atomic memory bus transactions
  - no interleaving of transactions
- Atomic operations within process
  - one finishes before next in program order
- Examine write serialization, completion, atomicity
- Then add more concurrency and re-examine

Cache Controller and Tags

- On a miss in uniprocessor:
  - Assert request for bus
  - Wait for bus grant
  - Drive address and command lines
  - Wait for command to be accepted by relevant device
  - Transfer data
- In snoop-based multiprocessor, cache controller must:
  - Monitor bus and processor
    - ▲Can view as two controllers: bus-side, and processor-side
    - ▲With single-level cache: dual tags (not data) or dual-ported tag RAM
    - ▲synchronize on updates
  - Respond to bus transactions when necessary

Reporting Snoop Results: How?

- Collective response from caches must appear on bus
- Wired-OR signals
  - Shared: asserted if any cache has a copy
  - Dirty/Inhibit: asserted if some cache has a dirty copy
    - ▲needn't know which, since it will do what's necessary
  - Snoop-valid: asserted when OK to check other two signals
- May require priority scheme for cache-to-cache transfers
  - Which cache should supply data when in shared state?
  - Commercial implementations allow memory to provide data

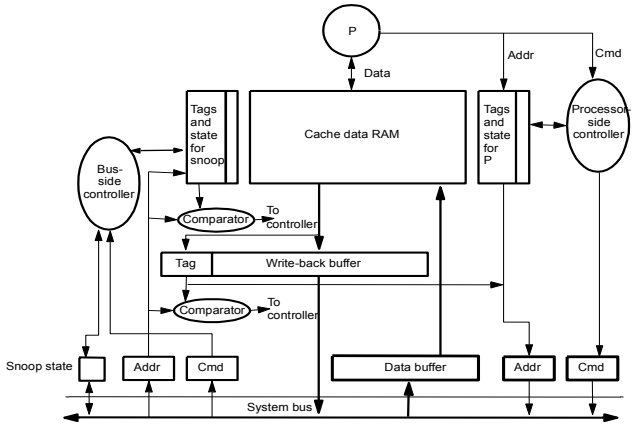
Reporting Snoop Results: When?

- Memory needs to know what, if anything, to do
- Fixed number of clocks from address appearing on bus
  - Dual tags required to reduce contention with processor
  - Still must be conservative (update both on write: E -> M)
  - Pentium Pro, HP servers, Sun Enterprise
- Variable delay
  - Memory assumes cache will supply data till all say "sorry"
  - Less conservative, more flexible, more complex
  - Memory can fetch data early and hold (SGI Challenge)

Writebacks

- Must allow processor to proceed on a miss
  - fetch the block
  - perform writeback later
- Need writeback buffer
  - Must handle bus transactions in writeback buffer
  - Snoop writeback buffer
  - Must care about the order of reads and writes

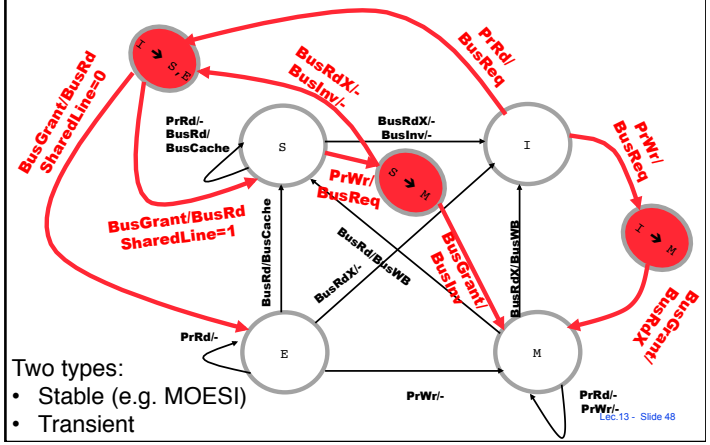
Base Organization



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

Non-Atomicity ➔Transient States (from 4 to 7 states)



Two types:
- Stable (e.g. MOESI)
- Transient

Serialization and Ordering

Let A and flag be 0

P1	P2
A += 5	while (flag == 0)
flag = 1	print A

- Assume A and flag are in different cache blocks
- What happens?
- How do you implement it correctly?

Serialization and Ordering

- Processor-cache handshake must preserve serialization
- e.g. write to S state=> first obtain ownership
- why?
- Write completion for SC => need bus invalidation:
  - Wait to get bus, can proceed afterwards
- Must serialize bus operations in program order

Multi-level Cache Hierarchies

- How to snoop with multi-level caches?
  - independent bus snooping at every level?
  - maintain cache inclusion
- Requirements for Inclusion
  - data in higher-level is subset of data in lower-level
  - modified in higher-level => marked modified in lower-level
- Now only need to snoop lowest-level cache
  - If L2 says not present (modified), then not so in L1
- Is inclusion automatically preserved
  - Replacements: all higher-level misses go to lower level
  - Modifications

Violations of Inclusion

- The two caches (L1, L2) may choose to replace different block

  Example: Local LRU not sufficient

  Assume that L1 and L2 hold two and three blocks and both use local LRU

  Processor references: 1, 2, 1, 3, 1, 4

  Final contents of L1: 1, 4

  L1 misses: 1, 2, 3, 4

  Final contents of L2: 2, 3, 4, but not 1

Violations of Inclusion

- Split higher-level caches
  - instruction, data blocks go in different caches at L1, but collide in L2
- Differences in Associativity
  - What if L1 is set-associative and L2 is direct-mapped?
- Differences in block size
  - Blocks in two L1 sets may both map to same L2 set
- *But* a common case works automatically
  - L1 direct-mapped, fewer sets than in L2, and block size same

Inclusion: to have or not to have

- Most common inclusion solution
  - Ensure L2 holds superset of L1I and L1D
  - On L2 replacement or coherence request that must source data or invalidate, forward actions to L1 caches
  - Can maintain bits in L2 cache to filter some actions from forwarding
  - virtual L1 / physical [Wang, et al., ASPLOS87]
- But
  - Restricted associativity in unified L2 can limit blocks in split L1's
  - Not that hard to always snoop L1's *e.g., on-chip)
- Thus, many new designs don't 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
- Bottom Line
  - Packaging has a lot to do with it
  - As levels of integrations increase, there will be more sharing

Summary

- Lots of possibilities with FSMs
  - Invalidate/upgrade protocols
  - What do we do with the 4th state (Owner-based, Exclusive-based protocols)
- Think about implementation too
  - 4 states become many more states. Why?
  - Multiple cache levels
  - Inclusion
  - Serialization and ordering