EECS 570

Lecture 8

Bus-based SMPs

Winter 2025

Prof. Satish Narayanasamy

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.

Reading this week

Daniel J. Sorin, Mark D. Hill, and David A. Wood, A Primer on Memory Consistency and Cache Coherence (Ch. 6 & 7)

Announcements

- No discussion this week.
- Next week discussion: PA2

Midterm Exam

26th Wed 3-4:30pm, Location: TBD

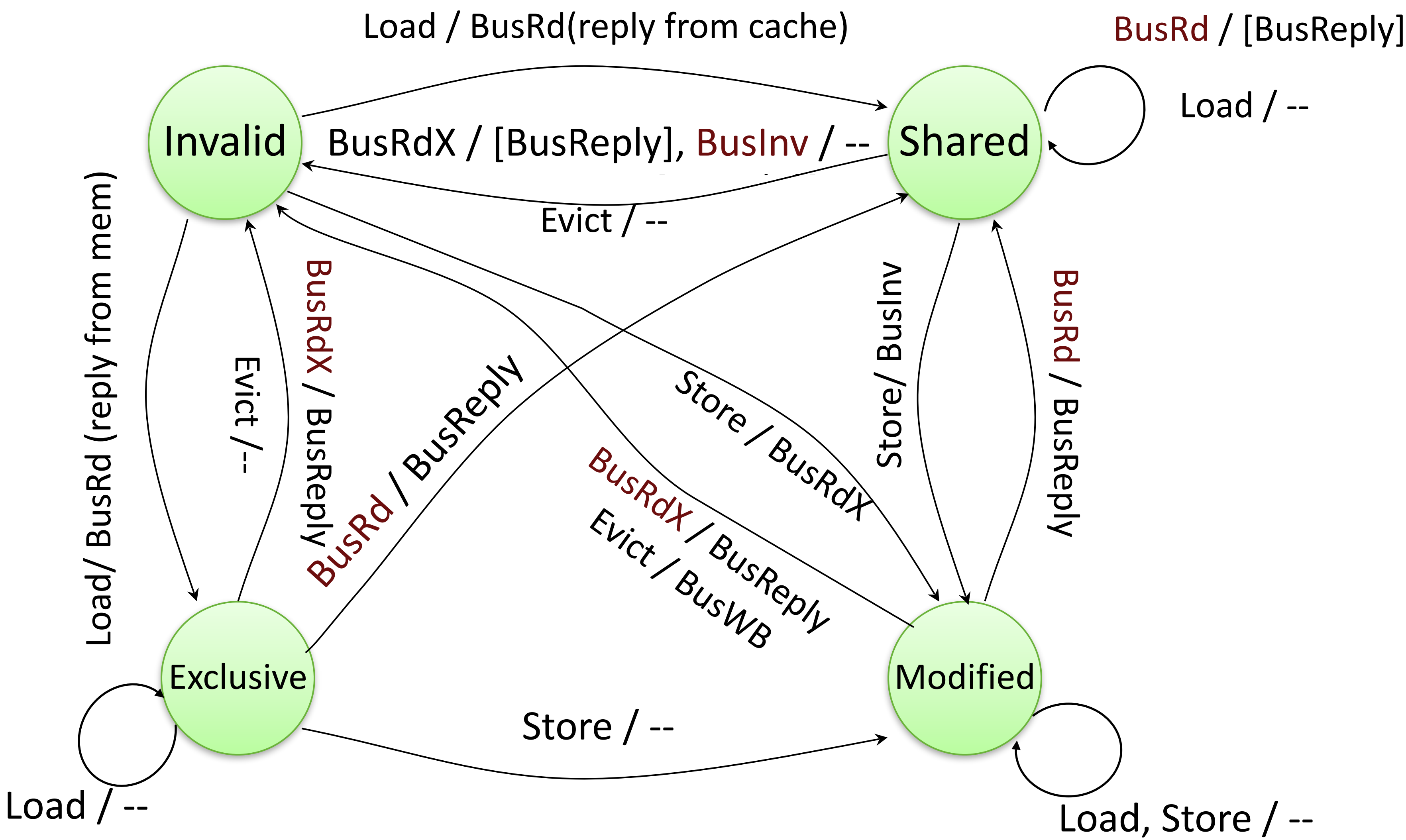
Alternate Exam Request from Online

- Please request before 2/12.
- *Fill this out even if we have discussed on Piazza/Email/SSD already to ensure they are all tracked on one sheet.*

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



MOESI Protocol

- MESI must write-back to memory on M→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→S transition, aka "downgrade"
  - Owner is responsible for writeback upon eviction

MOESI Framework

[Sweazey & Smith ISCA86]

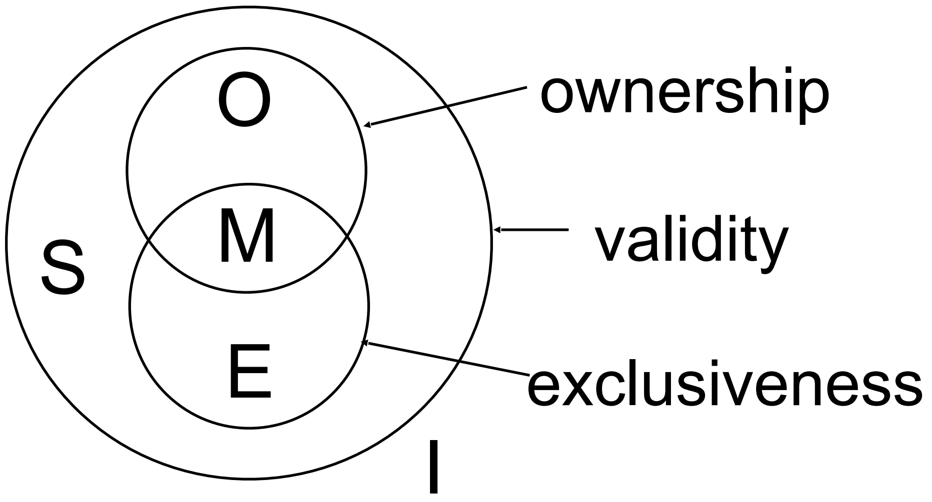
M - Modified (dirty)

O - Owned (dirty but shared) WHY?

E - Exclusive (clean unshared) only copy, not dirty

S - Shared

I - Invalid



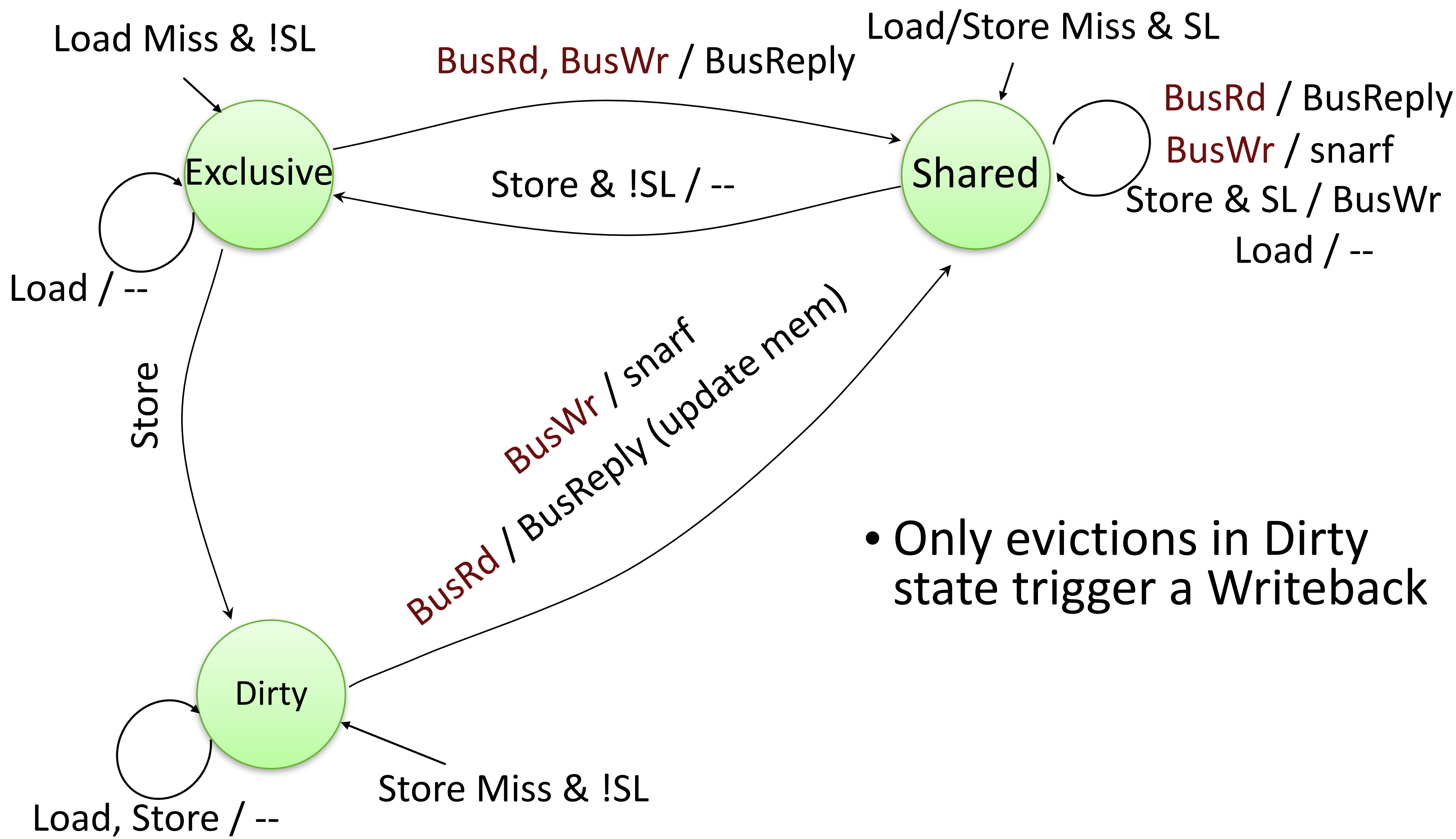
Variants

- 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

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

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 (used in Firefly protocol)
  - Shared: assert if any cache has a copy
  - 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

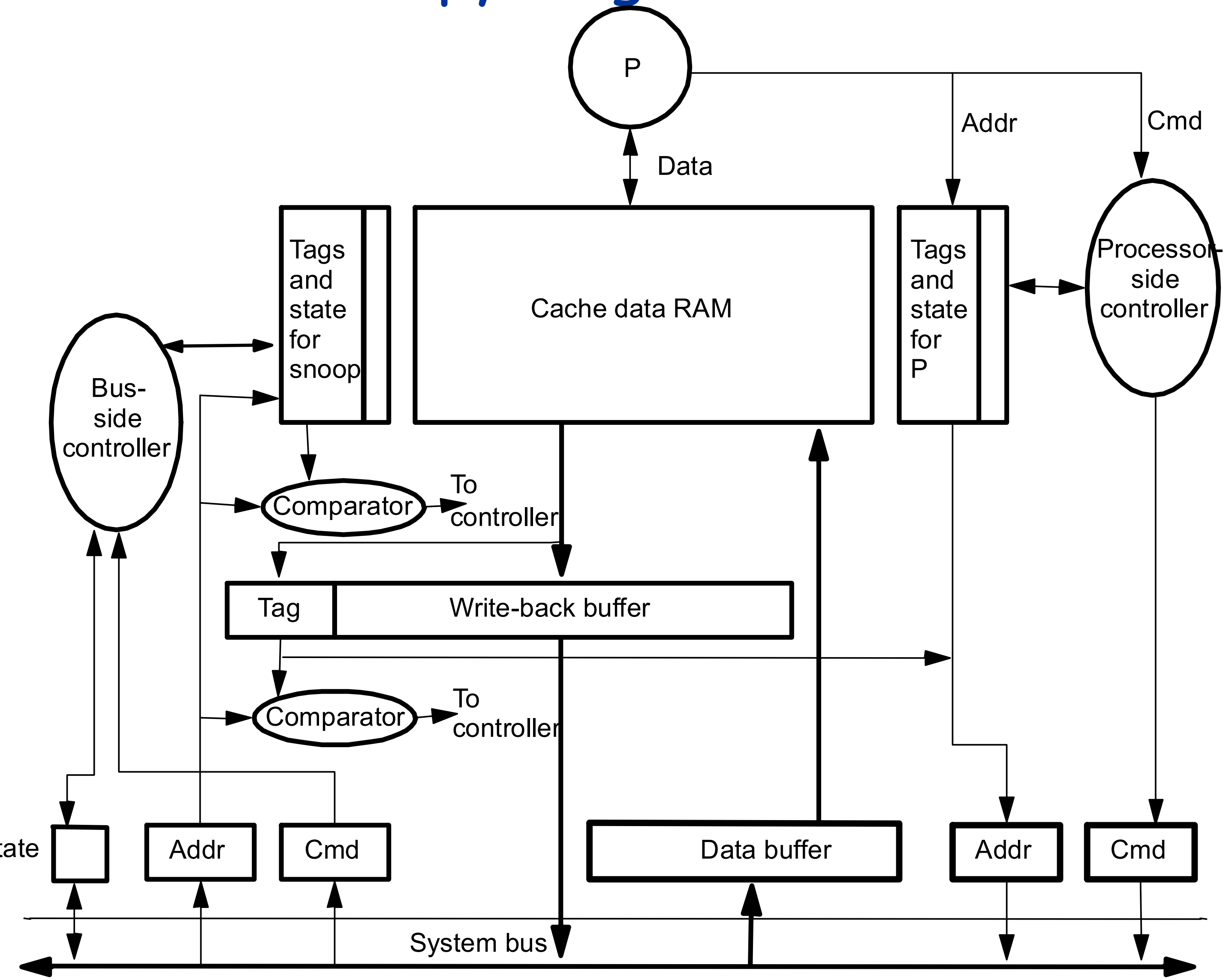
- Solution 2: Variable delay
  - Memory assumes cache will supply data until all say “sorry”
  - Less conservative, more flexible, more complex

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



Serialization and Ordering

- CPU-cache handshake must preserve serialization
  - E.g., write in S state → first obtain permission
- Write completion for SC → 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

The Inclusion Property

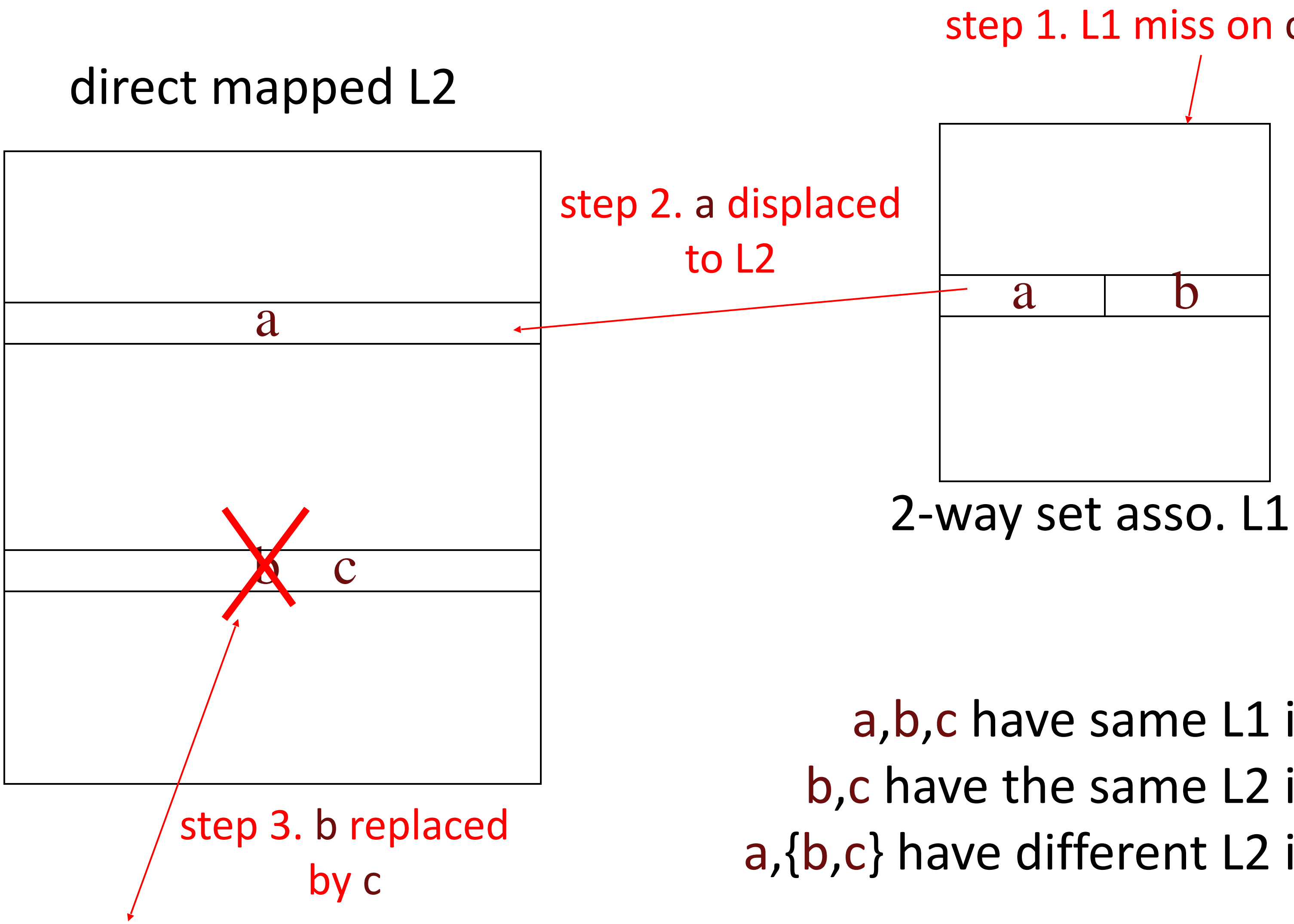
- Inclusion means L2 is a superset of L1 (ditto for L3…)
  - Also, must propagate "dirty" bit through cache hierarchy
- Now, only need to snoop last level cache
  - If L2 says not present, can't be in L1 either
- **Inclusion takes effort to maintain**
  - L2 must track what is cached in L1
  - On L2 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



a,b,c have same L1 idx bits

b,c have the same L2 idx bits

a,{b,c} have different L2 idx bits

Multi-level Cache Hierarchies

- How to snoop with multi-level caches?
  - Independent bus snooping at each level?
  - Multiple duplicate tag arrays
  - Maintain cache **inclusion**

Is inclusion a good idea?

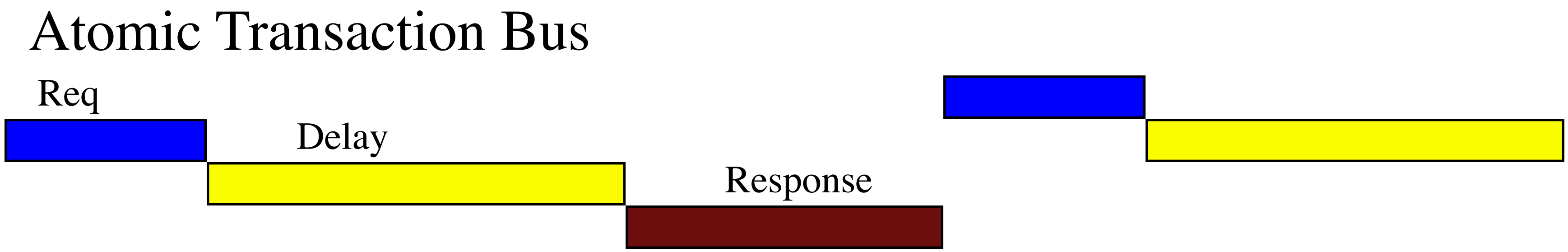
- Most common inclusion solution:
  - Ensure L2 holds a superset of L1I and L1D
  - On L2 replacement or coherence action that supplies data, forward actions to L1s
- But…
  - Restricted L2 associativity may limit blocks in split L1s
  - Not that hard to always snoop the L1s
- Many recent designs do not maintain inclusion
  - Leads to more complex coherence protocols

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



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

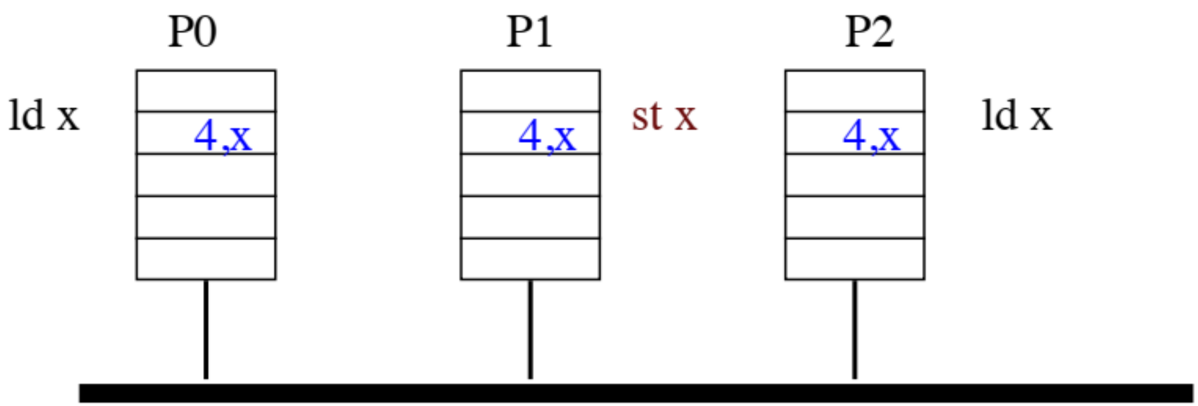
Possible Solutions

- 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 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

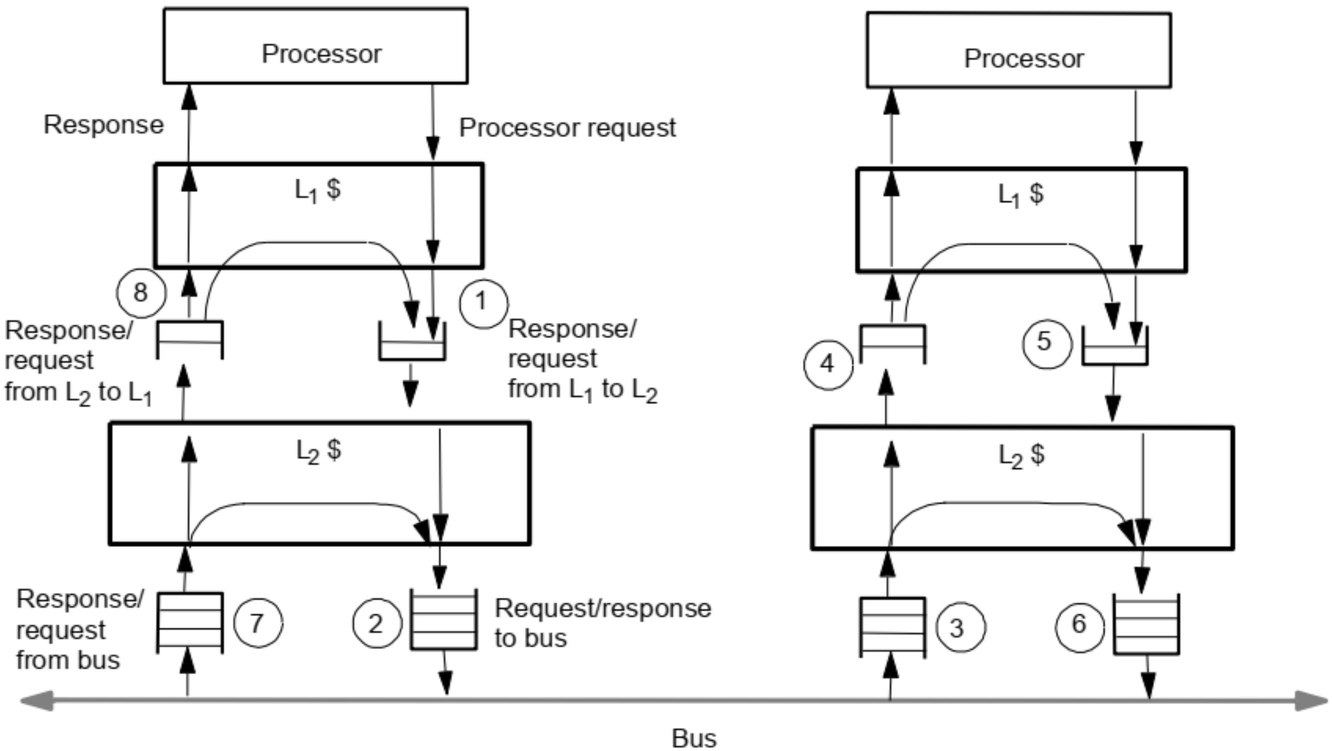
Split-Transaction Bus Example



Per-processor request table tracks all transactions

P2 Can snoop data from first ld
P1 Must hold st operation until entry is clear

Multi-Level Caches with Split Bus

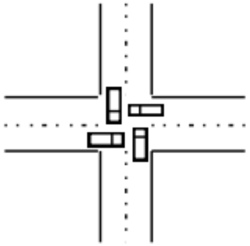


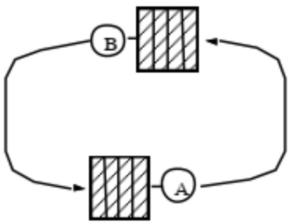
Multi-level Caches with Split-Transaction Bus

- General structure uses queues between
  - Bus and L2 cache
  - L2 cache and L1 cache
- Deadlock!
- Classify all transactions
  - Request, only generates responses
  - Response, doesn't generate any other transactions
- Requestor guarantees space for all responses
- Use Separate Request and Response queues
  - This ideal will evolve into "virtual channels" in Unit 3

More on Correctness

- Partial correctness (never wrong): Maintain coherence and consistency (safety)
- Full correctness (always right): Prevent: (liveness)
- 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

Deadlock, Livelock, Starvation

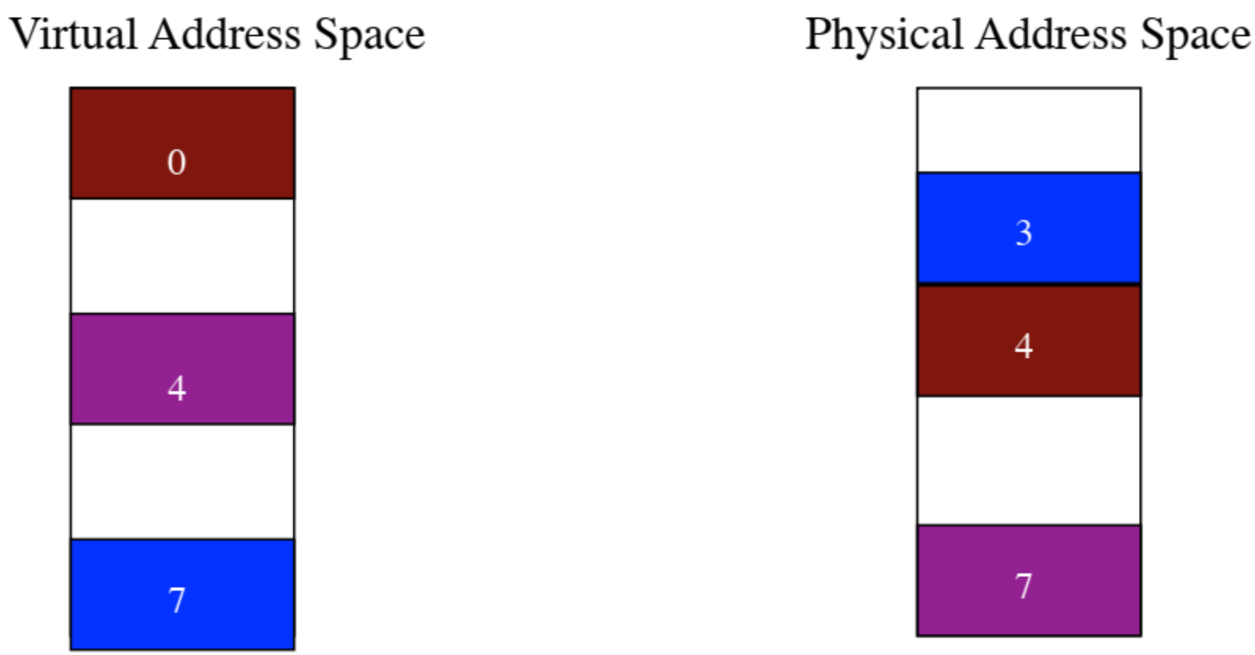
- Request-reply protocols can lead to *deadlock*
  - When issuing requests, must service incoming transactions
  - e.g. cache awaiting bus grant must snoop & flush blocks
  - else may not respond to request that will release bus: deadlock
- Livelock:
  - window of vulnerability problem [Kubi et al.,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

Translation Lookaside Buffer

- Cache of Page Table Entries
- Page Table Maps Virtual Page to Physical Frame



The TLB Coherence Problem

- Since TLB is a cache, must be kept coherent
- Change of PTE on one processor must be seen by all processors
- Process migration
- Changes are infrequent
  - get OS to do it
  - Always flush TLB is often adequate

TLB Shootdown

- To modify TLB entry, modifying processor must
  - LOCK page table,
  - flush TLB entries,
  - queue TLB operations,
  - send interprocessor interrupt,
  - spin until other processors are done
  - UNLOCK page table

- SLOW...
  - But most common solution today
- Some ISAs have “flush TLB entry” instructions