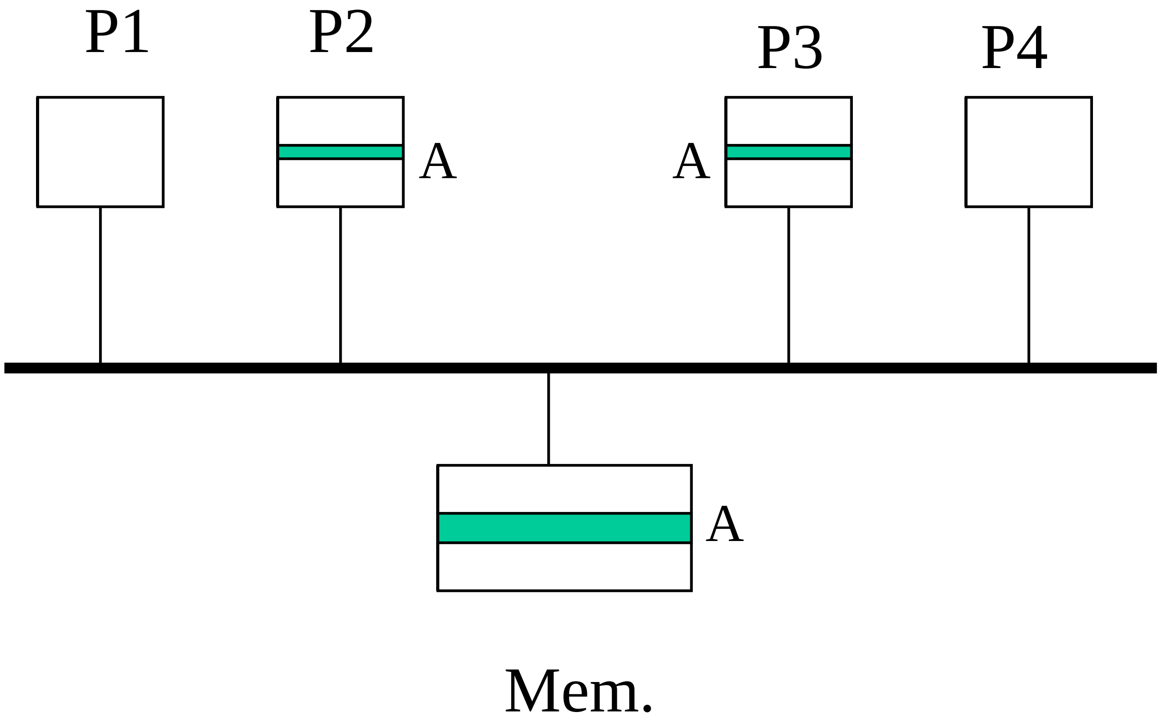
Cache Coherence

- Recall the memory wall
  - In multiprocessors the wall might even be higher!
  - Contention on shared-bus
  - Time to travel through an interconnection network
- In addition to the 3 C’s of the cache hierarchy
  - Cache coherence misses
- Cache coherence protocols
  - Shared-bus: Snoopy protocols
  - Other interconnection schemes: Directory protocols

Cache Coherence: The problem

Initial state: P2 reads A; P3 reads A (note already a decision to make: who sends the value of A?)

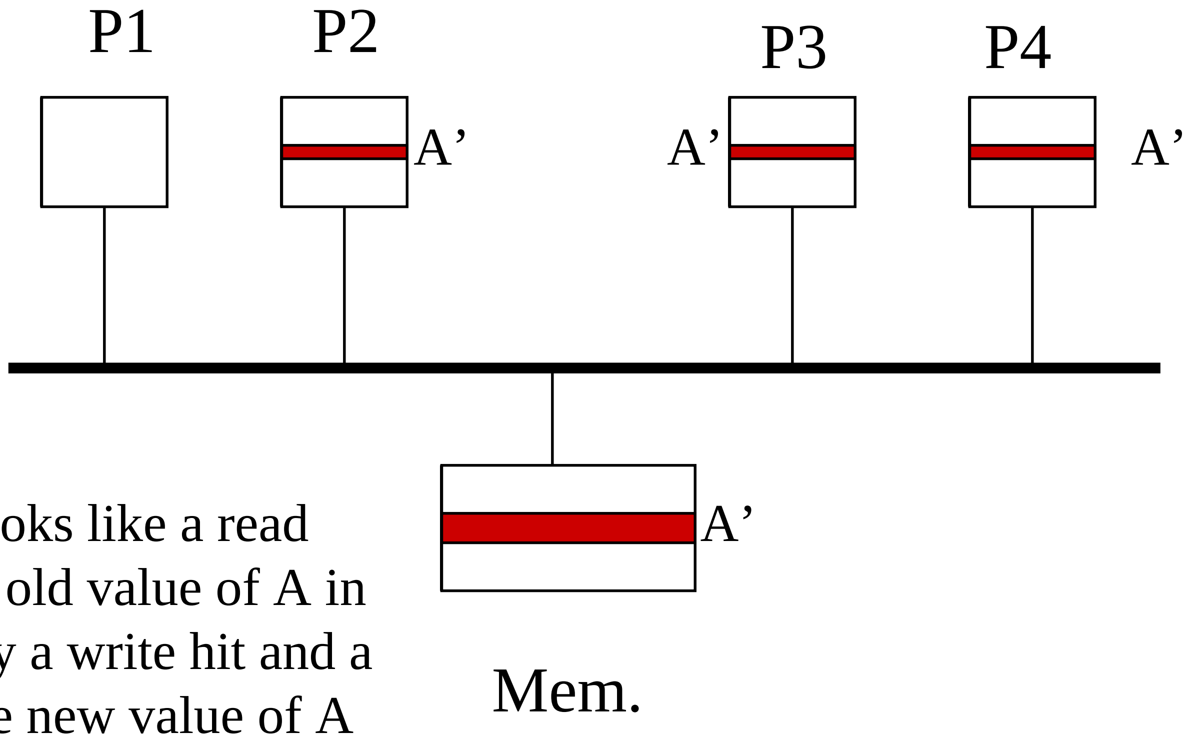


Cache coherence (shared-bus)

- Now P2 wants to write A
- Two choices:
  - Broadcast the new value of A on the bus; value of A snooped by cache of P3: Write-update (or write broadcast) protocol (resembles write-through). Memory is also updated.
  - Broadcast an invalidation message with the address of A; the address snooped by cache of P3 which invalidates its copy of A: Write-invalidate protocols. Note that the copy in memory is not up-to-date any longer (resembles write-back).
- If instead of P2 wanting to write A, we had a write miss in P4 for A, the same two choices of protocol apply.

Write-update

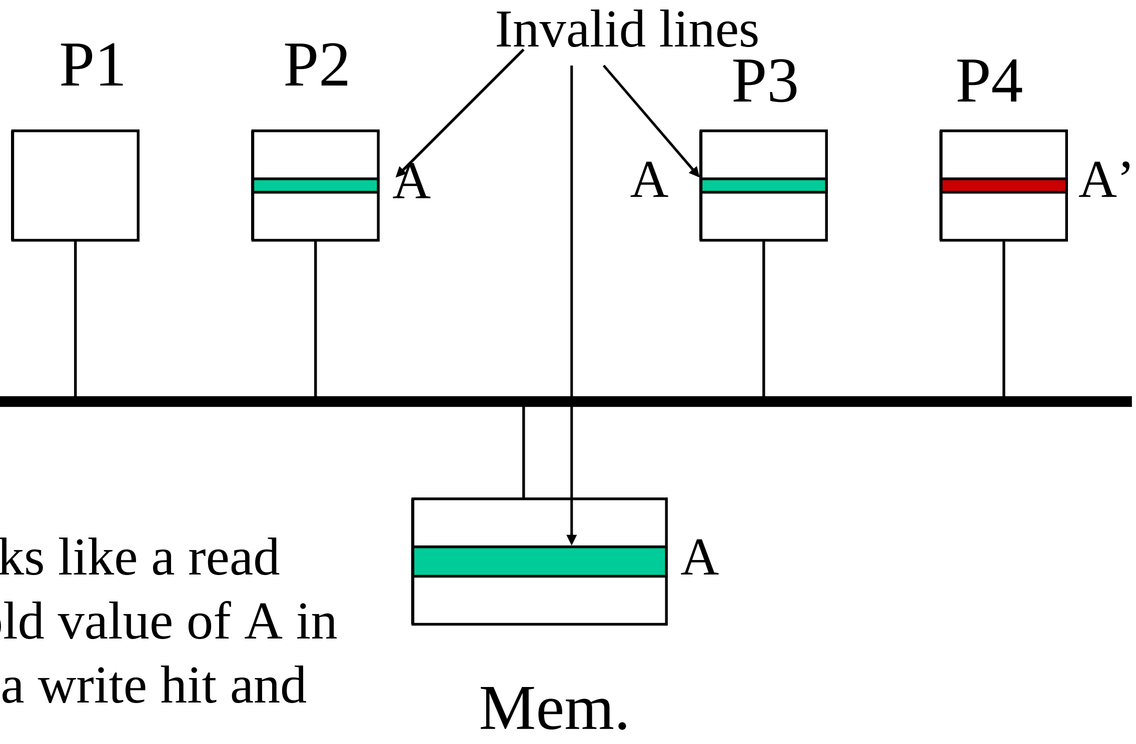
P2 and P3 have read line A; P4 has a write miss on an element of line A



A write miss looks like a read miss (bring the old value of A in P4) followed by a write hit and a broadcast of the new value of A

Write-invalidate

P2 and P3 have read line A; P4 has a write miss on an element of line A



A write miss looks like a read miss (bring the old value of A in P4) followed by a write hit and an invalidation

Snoopy Cache Coherence Protocols

- Associate states with each cache line; for example:
  - Invalid (I)
  - Shared (S) (or Clean) one or more copies are up to date
  - Modified (M) (or Dirty) exists in only one cache
- Fourth state (and sometimes more) for performance purposes
  - MOESI protocols: E stands for Exclusive and O for Ownership

State Transitions for a Given Cache Line

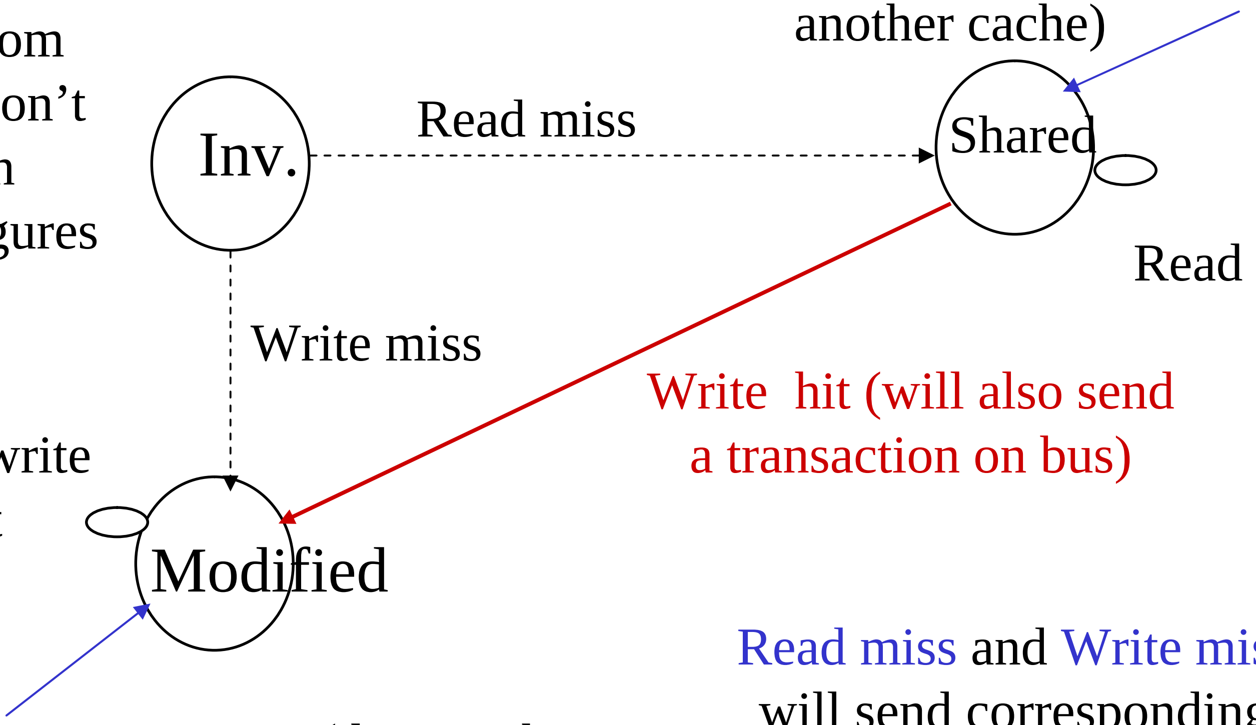
- Those incurred as answers to processor associated with the cache
  - Read miss, write miss, write on shared line
- Those incurred by snooping on the bus as result of other processor actions, e.g.,
  - Read miss by Q might make P’s line transit from M to S
  - Write miss by Q might make P’s line transit from M or S to I (write invalidate protocol)

Basic Write-invalidate Protocol (write-back write-allocate caches)

- Needs 3 states associated with each cache line
  - Invalid
  - Shared (read only – can be shared)
  - Modified (only valid copy in the system)
- Need to decompose state transitions into those:
  - Induced by the processor attached to the cache
  - Induced by snooping on the bus

Basic 3 State Protocol: Processor Actions

Transitions from Invalid state won’t be shown in forthcoming figures



Read miss (data might come from mem. or from another cache)

Inv.

Read miss

Shared

Read hit

Write miss

Write hit (will also send a transaction on bus)

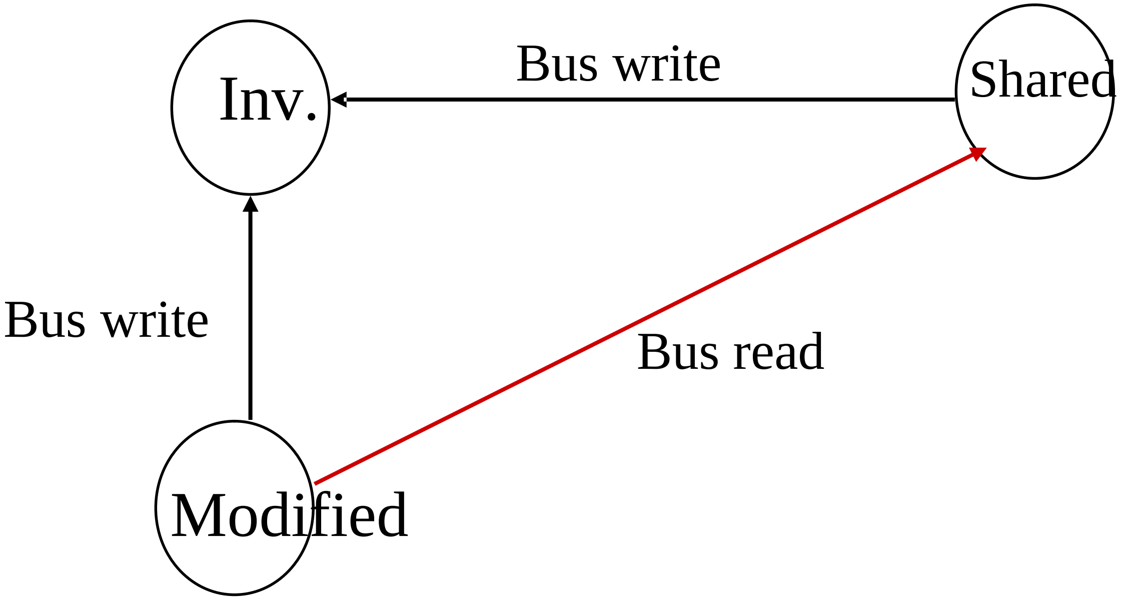
Read/write hit

Modified

Write miss (data might come from mem. or from another cache)

Read miss and Write miss will send corresponding transactions on the bus

Basic 3 State Protocol: Transitions from Bus Snooping



Snoopy protocol implementation

- Simple 3-state fsm?
- Yes but
  - Many more “internal states” because of write buffers, lock-up free caches, prefetching, split-transaction bus etc.
  - Example: split-transaction bus. Caches A and B have line L in state I and cache C has it in state S. Both A and B want to write L at the same time.
  - Split-transaction means for A and B (in this case) “Request to read” and for C “Data transfer” But the 2 “Request for read” should not arrive at C before the “Data transfer”. Need for intermediate states

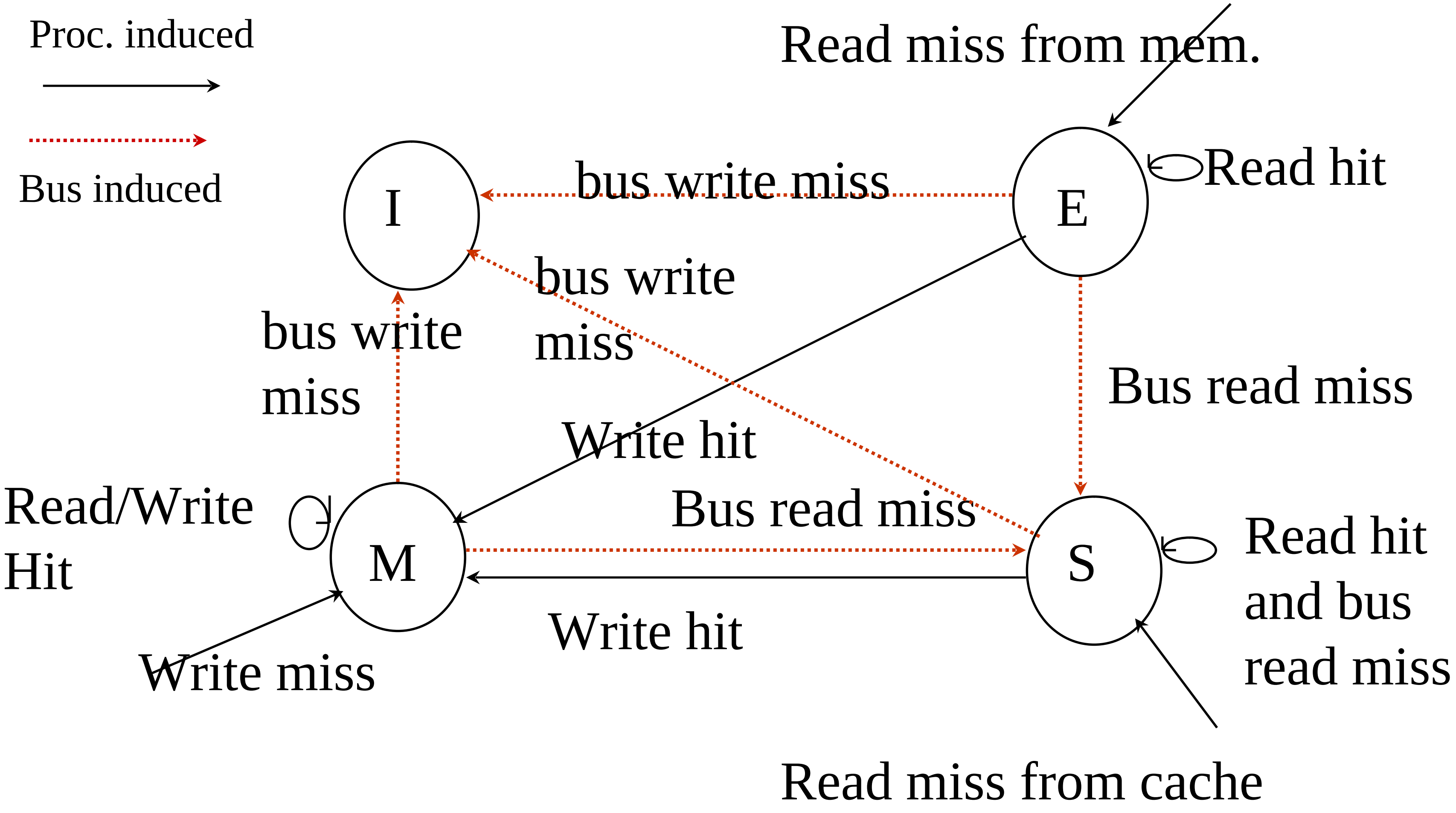
An Example of Write-invalidate Protocol: the Illinois Protocol

- States:
  - Invalid
  - (Valid)Exclusive (clean, only copy)
  - Shared (clean, possibly other copies)
  - Modified (modified, only copy)
  - In the MOESI notation, a MESI protocol

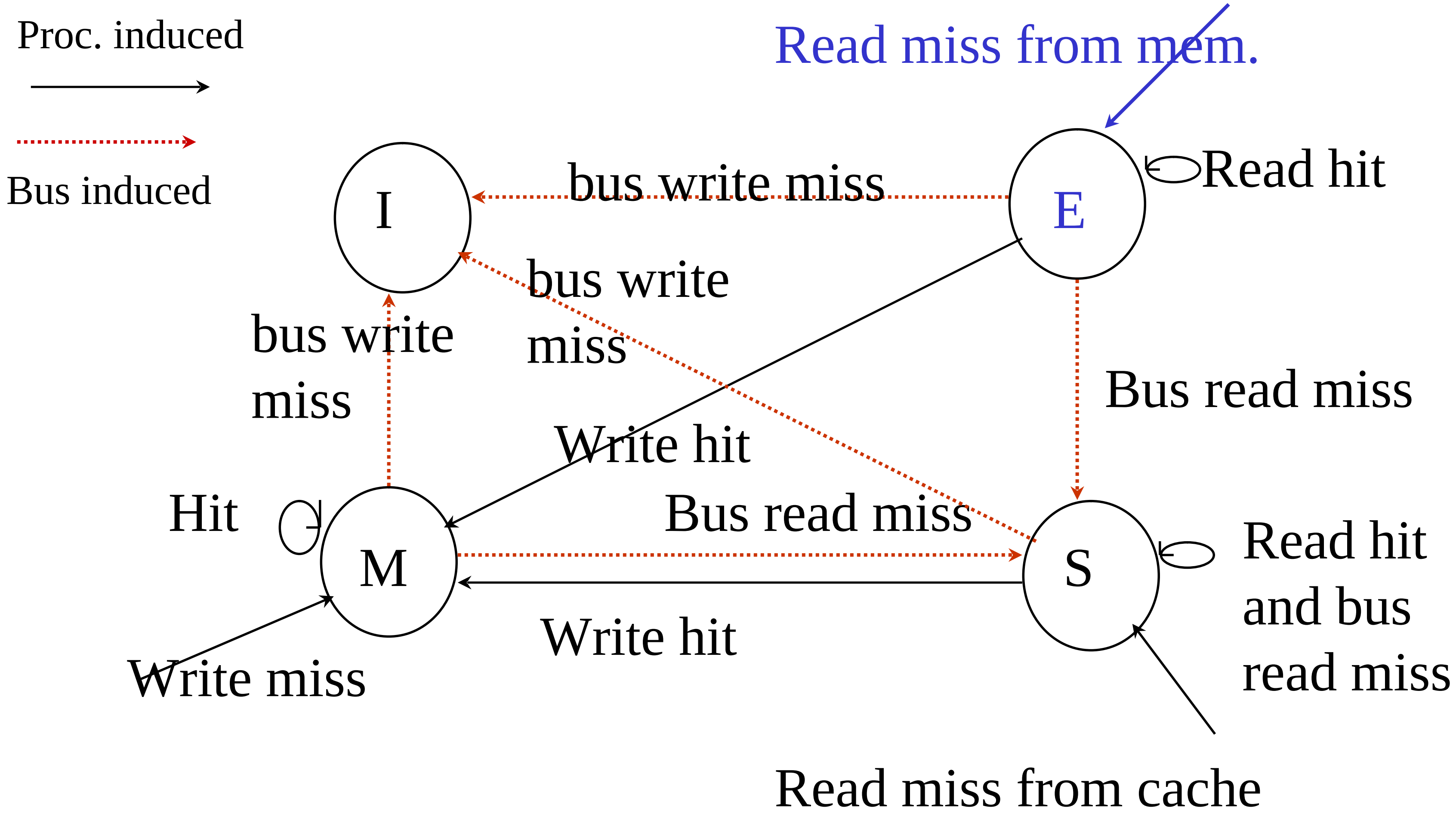
Illinois Protocol: Design Decisions

- The Exclusive state is there to enhance performance
  - On a write to a line in E state, no need to send an invalidation message (occurs often for private variables).
- On a read miss with no cache having the line in Modified state
  - Who sends the data: memory or cache (if any)?
    - Answer: cache for that particular protocol; other protocols might use the memory
  - If more than one cache, which one?
    - Answer: the first to grab the bus (tri-state devices)

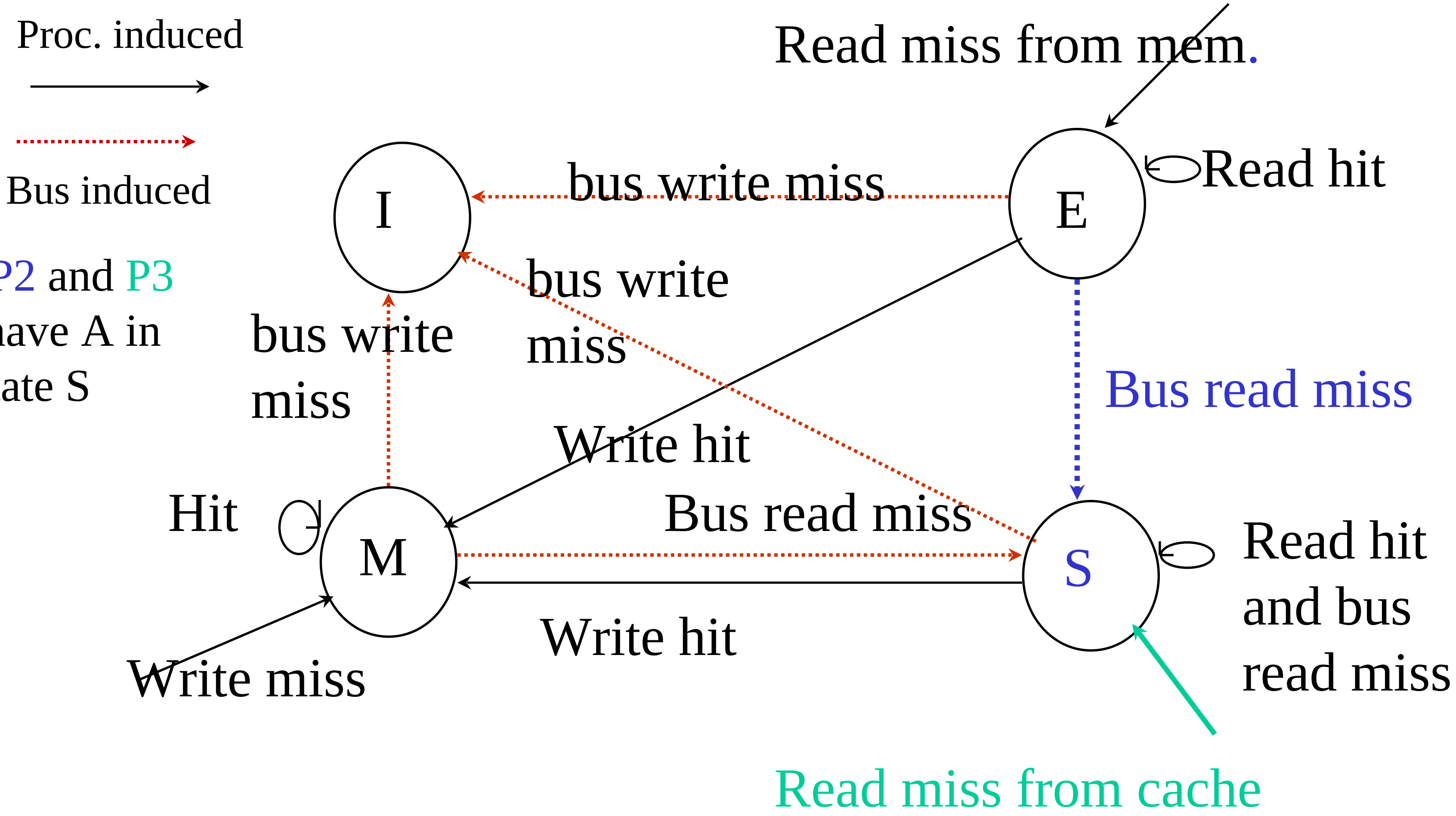
Illinois Protocol: State Diagram



Example: P2 reads A (A only in memory)

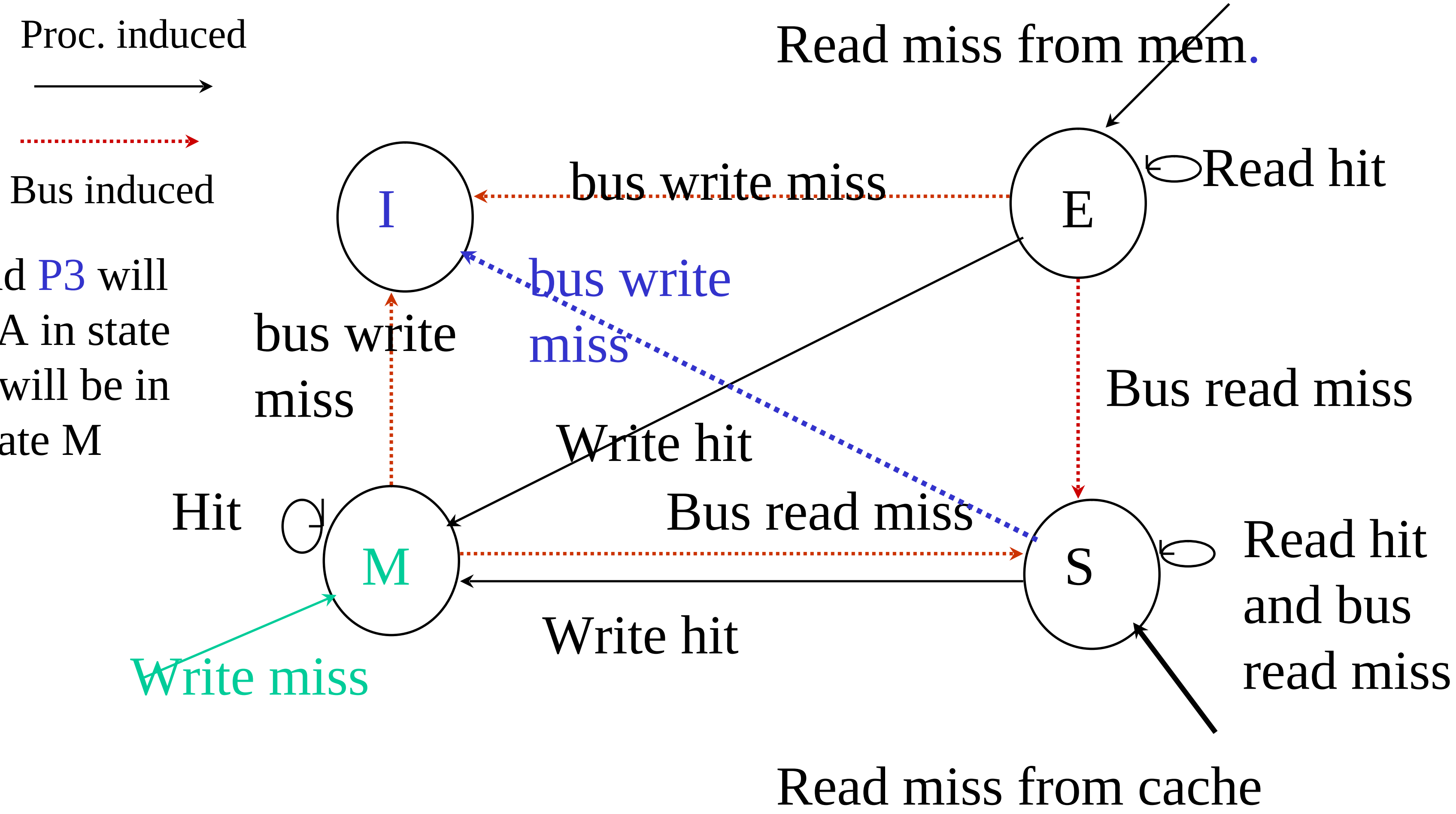


Example: P3 reads A (A comes from P2)



Both P2 and P3 will have A in state S

Example: P4 writes A (A comes from P2)



P2 and P3 will have A in state I; P4 will be in state M

Cache Parameters for Multiprocessors

- In addition to the 3 C's types of misses, add a 4th C: coherence misses
- As cache sizes increase, the misses due to the 3 C's decrease but coherence misses increase
- Shared data has been shown to have less spatial locality than private data; hence large line sizes could be detrimental
- Large line sizes induce more false sharing
  - P1 writes the first part of line A; P2 writes the second part. From the coherence protocol viewpoint, both look like "write A"

Performance of Snoopy Protocols

- Protocol performance depends on the length of a write run
- Write run: sequence of write references by 1 processor to a shared address (or shared line) uninterrupted by either access by another processor or replacement
  - Long write runs better to have write invalidate
  - Short write runs better to have write update
- There have been proposals to make the choice between protocols at run time
  - Competitive algorithms

What About Cache Hierarchies?

- Implement snoopy protocol at L2 (board-level) cache
- Impose multilevel inclusion property
  - Encode in L2 whether the line (or part of it if lines in L2 are longer than lines in L1) is in L1 (1 bit/line or subblock)
  - Disrupt L1 on bus transactions from other processors only if data is there, i.e., L2 shields L1 from unnecessary checks
  - Total inclusion might be expensive (need for large associativity) if several L1's share a common L2 (like in clusters). Instead use partial inclusion (i.e., possibility of slightly over invalidating L1)

Cache Coherence in NUMA Machines

- Snooping is not possible on media other than bus/ring
- Broadcast / multicast is not that easy
  - In Multistage Interconnection Networks (MINs), potential for message blocking is very large
  - In mesh-like networks, broadcast to every node is very inefficient
- How to enforce cache coherence
  - Having no caches (Tera MTA)
  - By software:disallow/limit caching of shared variables (Cray 3TD)
  - By hardware: having a data structure (a directory) that records the state of each line

Information Needed for Cache Coherence

- What information should the directory contain
  - At the very least whether a line is cached or not
  - Whether the cache copy – or copies – is shared (clean) or modified
- Where are the copies of the line
  - Directory structure associated with the line in memory
  - Linked list of all copies in the caches, including the one in memory

Full Directory

- Full information associated with each line in memory
- Entry in the directory: state vector associated with the line
  - For an *n* processor system, an (*n+1*) bit vector
  - Bit 0, clean/dirty
  - Bits 1-n: “location” vector ; Bit *i* set if *i*th cache has a copy
  - Protocol is write-invalidate
- Memory overhead:
  - For a 64 processor system, 65 bits / block
  - If a block is 64 bytes, overhead = 65 / (64 * 8), i.e., over 10%
  - This data structure is not scalable (but see later)

Home Node

- Definition
  - Home node: the node that contains the initial value of the line as determined by its physical address
  - Home node contains the directory entry for a line
  - Remote node: any other node
- On a cache miss (read and write), the request for data will be sent to the home node
- If a line has to be evicted from a cache, and it is dirty, its value should be written back in the home node

Basic protocol

- Assume write-back, write-allocate caches with a clean/dirty bit per line
- Read hit: Do nothing
- Write hit on dirty line: Do nothing

Basic Protocol – Read Miss on Uncached/clean Line

- Cache *i* has a read miss on an uncached line (state vector full of 0's)
  - The home node responds with the data
  - Add entry in directory (set clean and *i*th bit)
- Cache *i* has a read miss on a clean line (clean bit on in directory; at least one of the other bits on)
  - The home node responds with the data
  - Add entry in directory (set *i*th bit)

Basic Protocol – Read Miss on Dirty Line

- Cache *i* has a read miss on a dirty line
  - If dirty line is in home node, say node *j* (dirty and *j*th bits on) home node:
    - Updates memory (write back from its own cache *j*)
    - Changes the line encoding (dirty -> clean and set *i*th bit);
    - Sends data to cache *i* (1-hop)
  - If dirty line is not in home node but is in cache *k* (dirty and *k*th bits on) then the home node:
    - Asks cache *k* to send the line and updates memory
    - Change entry in directory (dirty -> clean and set *i*th bit);
    - Sends the data (2-hops)

Basic Protocol – Write Miss on Uncached/clean Block

- Cache *i* has a write miss on an uncached line (state vector full of 0's)
  - The home node responds with the data
  - Add entry in directory (set dirty and *i*th bits)
- Cache *i* has a write miss on a clean line (clean bit on; at least one of the other bits on)
  - Home node sends an invalidate message to all caches whose bits are on in the state vector (this is a series of messages)
  - The home node responds with the data
  - Change entry in directory (clean -> dirty and set *i*th bit)
- Note : the memory is not up-to-date

Basic Protocol – Write Miss on Dirty Block

- Cache *i* has a write miss on a dirty line
  - If dirty line is in home node, say node *j* (dirty and *j*th bits on) home node:
    - Updates memory (write back from its own cache *j*)
    - Changes the line encoding (clear *j*th bit and set *i*th bit);
    - Sends data to cache *i* (1-hop)
  - If dirty line is not in home node but is in cache *k* (dirty and *k*th bits on), then the home node:
    - Asks cache *k* to send the line and updates memory
    - Change entry in directory (clear *k*th bit and set *i*th bit);
    - Sends the data (2-hops)

Basic Protocol – Request to Write a Clean Block

- Cache *i* wants to write one of its lines which is clean
  - Known because clean/dirty bits also exist in the cache metadata
  - Perform as in write miss on a clean block except that the memory does not have to send the data

Basic Protocol - Replacing a Line

- What happens when a line is replaced
  - If dirty, it is of course written back and its state becomes a vector of 0’s
  - If clean could either “do nothing” but then encoding is wrong leading to possibly unneeded invalidations (and acks) or could send message and modify state vector accordingly (reset corresponding bit)
  - Acks are necessary to ensure correctness mostly if messages can be delivered out of order

The Most Economical (Memory-wise) Protocol

- Recall the minimal number of states needed
  - Not cached anywhere (i.e., valid in home memory)
  - Cached in one or more caches but not modified (clean)
  - Cached in one cache and modified (dirty)
- Simply encode the states (2-bit protocol) and perform broadcast invalidations (expensive because most often the data is not shared by many processors)
- Fourth state to enhance performance, say exclusive (E):
  - Cached in one cache only and still clean: no need to broadcast invalidations on a request for that cache to write its clean line. The cache metadata must include an Exclusive state also (set on reading a line that is not cached anywhere)

2-bit Protocol

- Differences with full directory protocol
  - Of course no bit setting in “location” vector
  - On a read miss to uncached line go to state exclusive (in directory and in cache)
  - On “request to write a clean line” from a cache that has the line in exclusive state, if the line is still in exclusive state in the directory, no need to broadcast invalidations
  - On a read miss to an exclusive line, change state to clean
  - On a write miss to clean and to exclusive line from another cache and read/write miss to dirty line, need to send a broadcast invalidate signal to all processors; in the case of dirty, the one with the copy of the line will send it back along with its ack.

Need for Partial Directories

- Full directory not scalable.
  - Location vector depends on number of processors
  - Might become too much memory overhead
- 2-bit protocol invalidations are costly
- Observation: Sharing is often limited to a small number of processors
  - Instead of full directory, have room for a limited number of processor id's.

Examples of Partial Directories

- Coarse bit-vector
  - Share a "location" bit among 2 or 4 or 8 processors etc.
  - Advantage: scalable since fixed amount of memory/line
- Dynamic pointer (many variations)
  - Directory for a block has 1 bit for local cache, one or more fields for a limited number of other caches, and possibly a pointer to a linked list in memory for overflow.
  - Need to "reclaim" pointers on clean replacements and/or to invalidate blindly if there is overflow
  - Protocols are $\mathrm{Dir}_i\mathrm{B}$ (i pointers and broadcast) or $\mathrm{Dir}_i\mathrm{NB}$ (i pointers and No Broadcast, i.e., forced invalidations)

Directories in the Cache -- The SCI Approach

- Copies of lines residing in various caches are linked via a doubly linked list
  - Doubly linked so that it is easy to insert/delete
- Header in the line's home node memory
  - Insertions "between" home node and new cache
- Economical in memory space
  - Proportional to cache space rather than memory space
- Invalidations can be lengthy (list traversal)

A Caveat about Cache Coherence Protocols

- They are more complex in the details than they look!
- Snoopy protocols
  - Writes are not atomic (first detect write miss and send request on the bus; then get line and write data -- only then should the line become dirty)
  - The cache controller must implement "pending states" for situations which would allow more than one cache to write data in a linek, or replace a dirty line, i.e., write in memory
  - Things become more complex for split-transaction buses
  - Things become even more complex for lock-up free caches (but it's manageable)

Subtleties in Directory Protocols

- No transaction is atomic.
- If they were treated as atomic, deadlock could occur
  - Assume line A from home node X is dirty in P1
  - Assume line B from home node Y is dirty in P2
  - P1 reads miss on B and P2 reads miss on A
  - Home node Y generates a “purge” for B in P2 and Home node X generates a “purge” for A in P1
  - Both P1 and P2 wait for their read misses and cannot answer the home node purges hence deadlock.
- So assume non-atomicity of transactions and allow only one in-flight transaction per line (nack any other while one is in progress)

Problems with Buffering

- Directory and cache controllers might have to send/receive many messages at the same time
  - Protocols must take into account finite amount of buffers
  - This leads to possibility of deadlocks
  - This is even more important for 2-bit protocol with lots of broadcasts
  - Solutions involve one or more of the following
    - separate networks for requests and replies so that requests don't block replies which free buffer space
    - each request reserves buffer room for its reply
    - use of nacks and of retries