Cache Coherence

CSE 661 – Parallel and Vector Architectures

Muhamed Mudawar

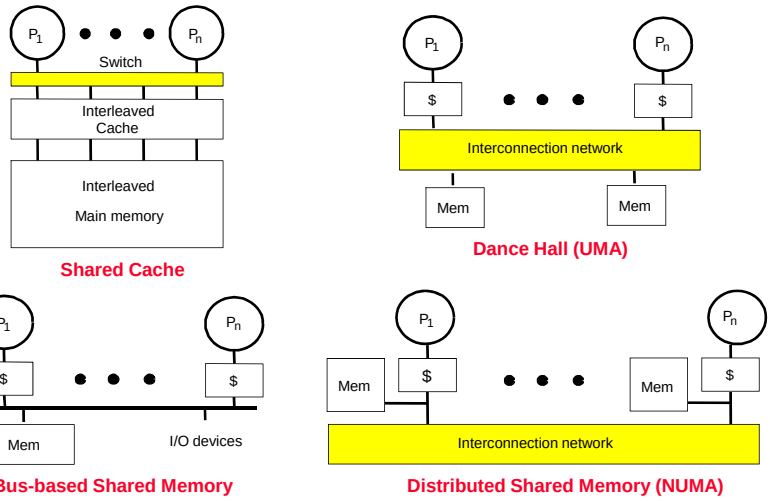
Computer Engineering Department

King Fahd University of Petroleum and Minerals

Outline of this Presentation

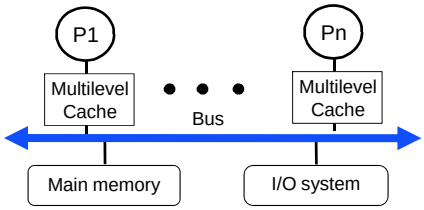
- Shared Memory Multiprocessor Organizations
- Cache Coherence Problem
- Cache Coherence through Bus Snooping
  - 2-state Write-Through Invalidation Protocol
- Design Space for Snooping Protocols
  - 3-state (MSI) Write-Back Invalidation Protocol
  - 4-state (MESI) Write-Back Invalidation Protocol
  - 4-state (Dragon) Write-Back Update Protocol

Shared Memory Organizations



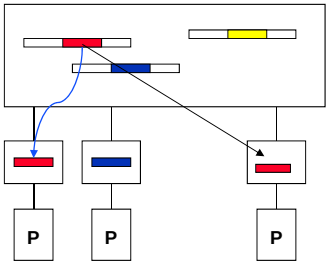
Bus-Based Symmetric Multiprocessors

- Symmetric access to main memory from any processor
- Dominate the server market
  - Building blocks for larger systems
- Attractive as throughput servers and for parallel programs
  - Uniform access via loads/stores
  - Automatic data movement and coherent replication in caches
  - Cheap and powerful extension to uniprocessors
  - Key is extension of memory hierarchy to support multiple processors



Caches are Critical for Performance

- Reduce average **latency**
  - Main memory access costs from 100 to 1000 cycles
  - Caches can reduce latency to few cycles
- Reduce average **bandwidth** and demand to access main memory
  - Reduce access to shared bus or interconnect
- Automatic **migration** of data
  - Data is moved closer to processor
- Automatic **replication** of data
  - Shared data is replicated upon need
  - Processors can share data efficiently
- But private caches create a problem

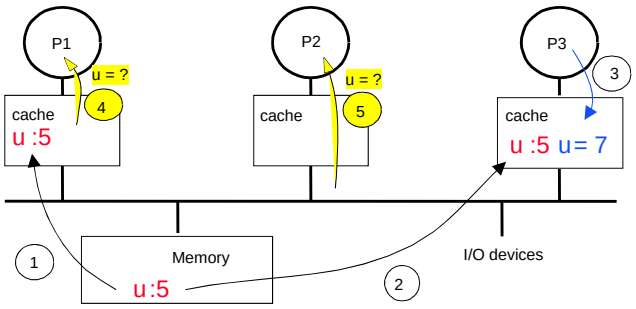


Cache Coherence

- What happens when loads & stores on **different** processors to **same** memory location?
- Private processor caches create a problem
  - Copies of a variable can be present in multiple caches
  - A write by one processor may NOT become visible to others
  - Other processors keep accessing **stale** value in their caches

  ⇒ **Cache coherence problem**
- Also in uniprocessors when I/O operations occur
  - Direct Memory Access (DMA) between I/O device and memory
  - DMA device reads **stale** value in memory when processor updates cache
  - Processor reads **stale** value in cache when DMA device updates memory

Example on Cache Coherence Problem



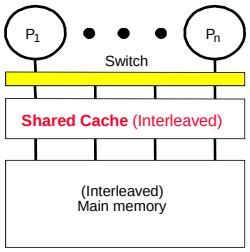
- Processors see **different** values for *u* after event 3
- With write back caches …
  - Processes accessing main memory may see **stale** (old incorrect) value
  - Value written back to memory depends on sequence of cache flushes
- Unacceptable to programs, and frequent!

What to do about Cache Coherence?

- Organize the memory hierarchy to make it go away
  - Remove private caches and use a shared cache
    - A switch is needed ⇒ added cost and latency
    - Not practical for a large number of processors
- Mark segments of memory as **uncacheable**
  - Shared data or segments used for I/O are not cached
  - Private data is cached only
  - We loose performance
- Detect and take actions to eliminate the problem
  - Can be addressed as a basic hardware design issue
  - Techniques solve both multiprocessor as well as I/O cache coherence

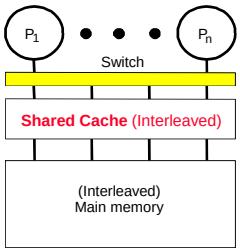
Shared Cache Design: Advantages

- Cache placement identical to single cache
  - Only one copy of any cached block
  - No coherence problem
- Fine-grain sharing
  - Communication latency is reduced when sharing cache
  - Attractive to Chip Multiprocessors (CMP), latency is few cycles
- Potential for positive interference
  - One processor prefetches data for another
- Better utilization of total storage
  - Only one copy of code/data used
- Can share data within a block
  - Long blocks without false sharing



Shared-Cache Design: Disadvantages

- Fundamental bandwidth limitation
  - Can connect only a small number of processors
- Increases latency of all accesses
  - Crossbar switch
  - Hit time increases
- Potential for negative interference
  - One processor flushes data needed by another
- Share second-level (L2) cache:
  - Use private L1 caches but make the L2 cache shared
  - Many L2 caches are shared today



Intuitive Coherent Memory Model

- Caches are supposed to be transparent
- What would happen if there were no caches?
  - All reads and writes would go to main memory
  - Reading a location should return **last value written** by any processor
- What does **last value written** mean in a multiprocessor?
  - All operations on a **particular location** would be **serialized**
  - All processors would **see the same access order** to a particular location
    - If they bother to read that location
- Interleaving among memory accesses from different processors
  - Within a processor ⇒ program order on a given memory location
  - Across processors ⇒ only constrained by explicit synchronization

Formal Definition of Memory Coherence

- A memory system is coherent if there exists a serial order of memory operations on each memory location X, such that …

  1. A read by any processor P to location X that follows a write by processor Q (or P) to X returns the **last written value** if no other writes to X occur between the two accesses

  2. Writes to the same location X are **serialized**; two writes to same location X by any two processors are seen in the same order by all processors

- Two properties
  - **Write propagation**: writes become visible to other processors
  - **Write serialization**: writes are seen in the same order by all processors

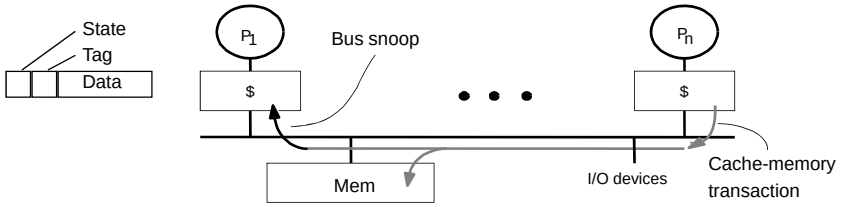
Hardware Coherency Solutions

- Bus Snooping Solution
  - Send all requests for data to all processors
  - Processors snoop to see if they have a copy and respond accordingly
  - Requires broadcast, since caching information is in processors
  - Works well with bus (natural broadcast medium)
  - Dominates for small scale multiprocessors (most of the market)
- Directory-Based Schemes
  - Keep track of what is being shared in one logical place
  - Distributed memory ⇒ distributed directory
  - Send point-to-point requests to processors via network
  - Scales better than Snooping and avoids bottlenecks
  - Actually existed before snooping-based schemes

Cache Coherence Using a Bus

- Built on top of two fundamentals of uniprocessor systems
  - Bus transactions
  - State transition diagram in a cache
- Uniprocessor bus transaction
  - Three phases: arbitration, command/address, data transfer
  - All devices observe addresses, one is responsible
- Uniprocessor cache states
  - Effectively, every block is a finite state machine
  - Write-through, write no-allocate has two states: Valid, Invalid
  - Writeback caches have one more state: Modified (or Dirty)
- Multiprocessors extend both to implement coherence

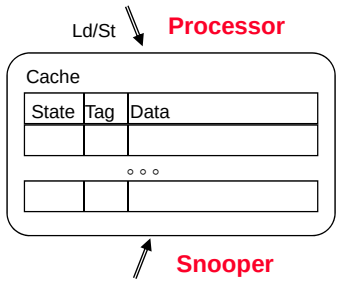
Snoopy Cache-Coherence Protocols



- Bus is a broadcast medium & caches know what they have
  - Transactions on bus are visible to all caches
- Cache controllers **snoop** all transactions on the shared bus
  - Relevant transaction if for a block it contains
  - Take action to ensure coherence
    - Invalidate, update, or supply value
  - Depends on state of the block and the protocol

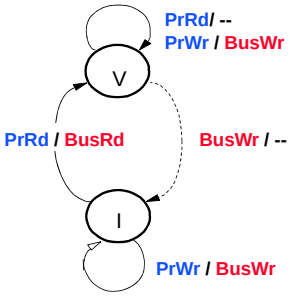
Implementing a Snooping Protocol

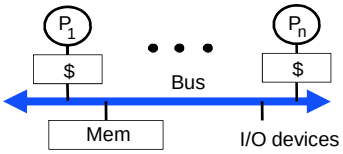
- Cache controller receives inputs from two sides:
  - Requests from processor (load/store)
  - Bus requests/responses from snooper
- Controller takes action in response to both inputs
  - Updates state of blocks
  - Responds with data
  - Generates new bus transactions
- Protocol is a distributed algorithm
  - Cooperating state machines and actions
- Basic Choices
  - Write-through versus Write-back
  - Invalidate versus Update



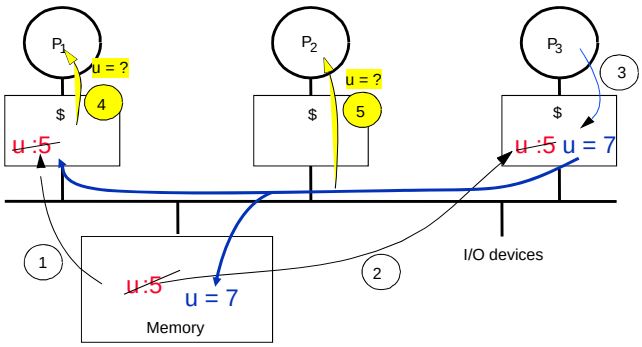
Write-through Invalidate Protocol

- Two states per block in each cache
  - States similar to a uniprocessor cache
  - Hardware state bits associated with blocks that are in the cache
  - Other blocks can be seen as being in invalid (not-present) state in that cache
- Writes invalidate all other caches
  - No local change of state
  - Multiple simultaneous readers of block, but write invalidates them





Example of Write-through Invalidate



- At step 4, an attempt to read *u* by P_1 will result in a cache miss
  - Correct value of *u* is fetched from memory
- Similarly, correct value of *u* is fetched at step 5 by P_2

2-state Protocol is Coherent

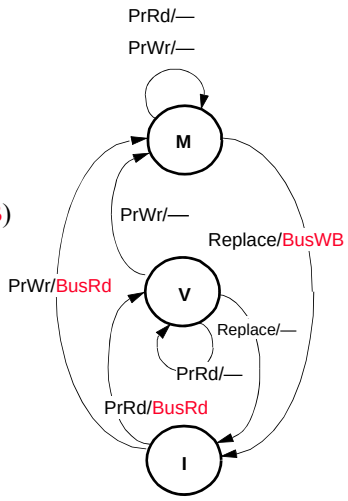
- Assume bus transactions and memory operations are atomic
  - All phases of one bus transaction complete before next one starts
  - Processor waits for memory operation to complete before issuing next
- Assume one-level cache
  - Invalidations applied during bus transaction
- All writes go to bus + atomicity
  - **Writes serialized** by order in which they appear on bus ⇒ **bus order**
  - Invalidations are performed by all cache controllers in bus order
- Read misses are serialized on the bus along with writes
  - Read misses are guaranteed to return the last written value
- Read hits do not go on the bus, however …
  - Read hit returns last written value by processor or by its last read miss

Write-through Performance

- Write-through protocol is simple
  - Every write is observable
- However, every write goes on the bus
  - Only one write can take place at a time in any processor
- Uses a lot of bandwidth!
- **Example:** 200 MHz dual issue, CPI = 1, 15% stores of 8 bytes
  - 0.15 * 200 M = 30 M stores per second per processor
  - 30 M stores * 8 bytes/store = 240 MB/s per processor
  - 1GB/s bus can support only about 4 processors before saturating
- Write-back caches absorb most writes as cache hits
  - But write hits don't go on bus – need more sophisticated protocols

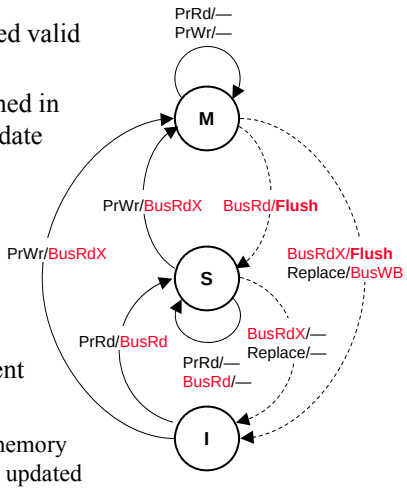
Write-back Cache

- Processor / Cache Operations
  - **PrRd**, **PrWr**, block **Replace**
- States
  - **Invalid**, **Valid** (clean), **Modified** (dirty)
- Bus Transactions
  - Bus Read (**BusRd**), Write-Back (**BusWB**)
  - Only cache-block are transfered
- Can be adjusted for cache coherence
  - Treat Valid as **Shared**
  - Treat Modified as **Exclusive**
- Introduce one new bus transaction
  - **Bus Read-eXclusive (BusRdX)**
  - For purpose of modifying (read-to-own)



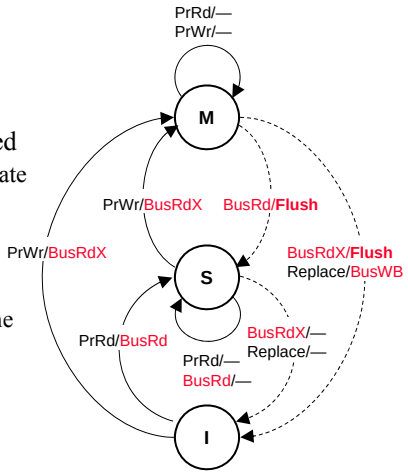
MSI Write-Back Invalidate Protocol

- Three States:
  - **Modified**: only this cache has a modified valid copy of this block
  - **Shared**: block is clean and may be cached in more than one cache, memory is up-to-date
  - **Invalid**: block is invalid
- Four bus transactions:
  - Bus Read: **BusRd** on a read miss
  - Bus Read Exclusive: **BusRdX**
    - Obtain exclusive copy of cache block
  - Bus Write-Back: **BusWB** on replacement
  - **Flush** on BusRd or BusRdX
    - Cache puts data block on the bus, not memory
      Cache-to-cache transfer and memory is updated

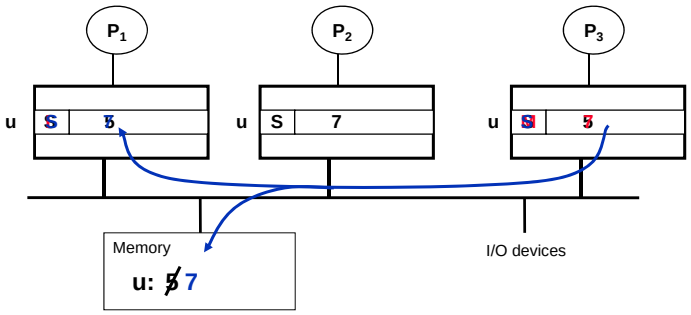


State Transitions in the MSI Protocol

- Processor Read
  - Cache miss ⇒ causes a Bus Read
  - Cache hit (S or M) ⇒ no bus activity
- Processor Write
  - Generates a BusRdX when not Modified
    - BusRdX causes other caches to invalidate
  - No bus activity when Modified block
- Observing a Bus Read
  - If Modified, flush block on bus
    - Picked by memory and requesting cache
    - Block is now shared
- Observing a Bus Read Exclusive
  - Invalidate block
  - Flush data on bus if block is modified



Example on MSI Write-Back Protocol



Processor Action	State P1	State P2	State P3	Bus Action	Data from
1. P1 reads u	S			BusRd	Memory
2. P3 reads u	S		S	BusRd	Memory
3. P3 writes u	I		M	BusRdX	Memory
4. P1 reads u	S		S	BusRd, Flush	P3 cache
5. P2 reads u	S	S	S	BusRd	Memory

Lower-level Design Choices

- Bus Upgrade (**BusUpgr**) to convert a block from state S to M
  - Causes invalidations (as BusRdX) but avoids reading of block
- When BusRd observed in state M: what transition to make?
  - M → S or M → I depending on expectations of access patterns
- Transition to state S
  - Assumption that I'll read again soon, rather than others will write
    - Good for mostly read data
- Transition to state I
  - So I don't have to be invalidated when other processor writes
  - Good for "migratory" data
    - I read and write, then another processor will read and write …
  - Sequent Symmetry and MIT Alewife use adaptive protocols
- Choices can affect performance of memory system

Satisfying Coherence

- Write propagation
  - A write to a shared or invalid block is made visible to all other caches
    - Using the Bus Read-exclusive (BusRdX) transaction
    - Invalidations that the Bus Read-exclusive generates
    - Other processors experience a cache miss before observing the value written
- Write serialization
  - All writes that appear on the bus (BusRdX) are serialized by the bus
    - Ordered in the same way for all processors including the writer
    - Write performed in writer's cache before it handles other transactions
  - However, not all writes appear on the bus
    - Write sequence to modified block must come from same processor, say *P*
    - Serialized within *P:* Reads by *P* will see the write sequence in the serial order
    - Serialized to other processors
      - Read miss by another processor causes a bus transaction
      - Ensures that writes appear to other processors in the same serial order

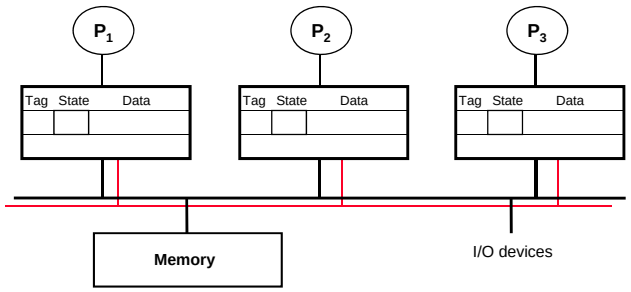
MESI Write-Back Invalidation Protocol

- Drawback of the MSI Protocol
  - Read/Write of a block causes 2 bus transactions
    - Read BusRd (I→S) followed by a write BusRdX (S→M)
    - This is the case even when a block is private to a process and not shared
    - Most common when using a multiprogrammed workload
- To reduce bus transactions, add an *exclusive* state
  - Exclusive state indicates that only this cache has clean copy
  - Distinguish between an exclusive clean and an exclusive modified state
  - A block in the exclusive state can be written without accessing the bus

Four States: MESI

- **M: Modified**
  - Only this cache has copy and is modified
  - Main memory copy is stale
- **E: Exclusive** or *exclusive-clean*
  - Only this cache has copy which is not modified
  - Main memory is up-to-date
- **S: Shared**
  - More than one cache may have copies, which are not modified
  - Main memory is up-to-date
- **I: Invalid**
- Know also as Illinois protocol
  - First published at University of Illinois at Urbana-Champaign
  - Variants of MESI protocol are used in many modern microprocessors

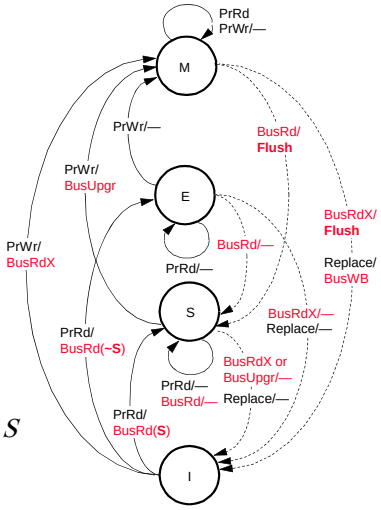
Hardware Support for MESI



- New requirement on the bus interconnect
  - Additional signal, called the shared signal *S*, must be available to all controllers
  - Implemented as a wired-OR line
- All cache controllers snoop on BusRd
  - Assert shared signal if block is present (state *S*, *E*, or *M*)
  - Requesting cache chooses between *E* and *S* states depending on shared signal

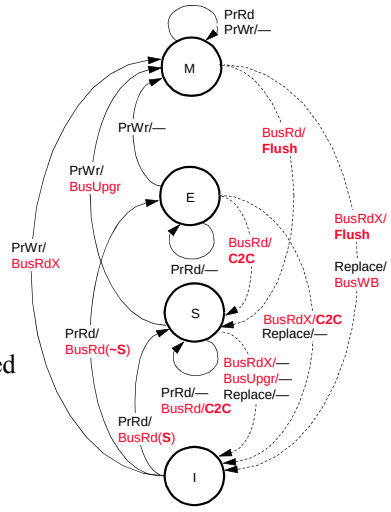
MESI State Transition Diagram

- Processor Read
  - Causes a BusRd on a read miss
  - BusRd(S) => shared line asserted
    - Valid copy in another cache
    - Goto state *S*
  - BusRd(~S) => shared line not asserted
    - No cache has this block
    - Goto state *E*
  - No bus transaction on a read hit
- Processor Write
  - Promotes block to state *M*
  - Causes BusRdX / BusUpgr for states *I* / *S*
    - To invalidate other copies
  - No bus transaction for states *E* and *M*



MESI State Transition Diagram – cont'd

- Observing a BusRd
  - Demotes a block from *E* to *S* state
    - Since another cached copy exists
  - Demotes a block from *M* to *S* state
    - Will cause modified block to be flushed
    - Block is picked up by requesting cache and main memory
- Observing a BusRdX or BusUpgr
  - Will invalidate block
  - Will cause a modified block to be flushed
- Cache-to-Cache (**C2C**) Sharing
  - Supported by original Illinois version
  - Cache rather than memory supplies data



MESI Lower-level Design Choices

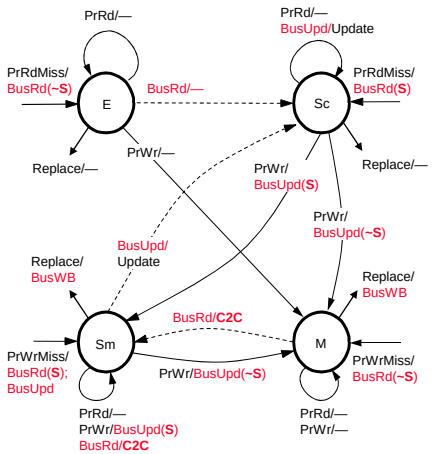
- Who supplies data on a BusRd/BusRdX when in E or S state?
  - Original, Illinois MESI: cache, since assumed faster than memory
- But cache-to-cache sharing adds complexity
  - Intervening is more expensive than getting data from memory
  - How does memory know it should supply data (must wait for caches)
  - Selection algorithm if multiple caches have shared data
- Flushing data on the bus when block is Modified
  - Data is picked up by the requesting cache and by main memory
  - But main memory is slower than requesting cache, so the block might be picked up only by the requesting cache and not by main memory
  - This requires a fifth state: **Owned** state ⇒ **MOESI** Protocol
  - Owned state is a Shared Modified state where memory is not up-to-date
    - The block can be shared in more than one cache but owned by only one

Dragon Write-back Update Protocol

- Four states
  - **Exclusive-clean (E)**
    - My cache ONLY has the data block and memory is up-to-date
  - **Shared clean (Sc)**
    - My cache and other caches have data block and my cache is NOT owner
    - Memory MAY or MAY NOT be up-to-date
  - **Shared modified (Sm)**
    - My cache and other caches have data block and my cache is OWNER
    - Memory is NOT up-to-date
    - **Sm** and **Sc** can coexist in different caches, with only one cache in **Sm** state
  - **Modified (M)**
    - My cache ONLY has data block and main memory is NOT up-to-date
- No Invalid state
  - Blocks are never invalidated, but are replaced
  - Initially, cache misses are forced in each set to bootstrap the protocol

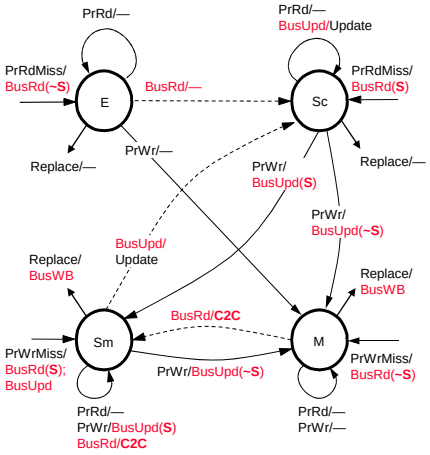
Dragon State Transition Diagram

- Cache Miss Events
  - **PrRdMiss, PrWrMiss**
    - Block is not present in cache
- New Bus Transaction
  - Bus Update: **BusUpd**
  - Broadcast single word on bus
  - Update other relevant caches
- Read Hit: no action required
- Read Miss: BusRd Transaction
  - Block loaded into E or Sc state
    - Depending on shared signal *S*
  - If block exists in another cache
    - If in M or Sm state then cache supplies data & changes state to Sm



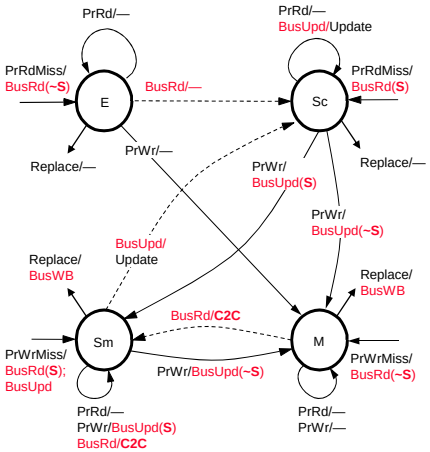
Dragon State Transition Diagram - cont'd

- Write Hit:
  - If Modified, no action needed
  - If Exclusive then
    - Make it Modified
    - No bus action needed
  - If shared (Sc or Sm)
    - Bus Update transaction
  - If any other cache has a copy
    - It asserts the shared signal *S*
    - Updates its block
    - Goto Sc state
  - Issuing cache goes to
    - Sm state if block is shared
    - M state if block is not shared



Dragon State Transition Diagram - cont'd

- Write Miss:
  - First, a BusRd is generated
  - Shared signal *S* is examined
  - If block is found is other caches
    - Block is loaded in Sm state
    - Bus update is also required
    - 2 bus transactions needed
  - If the block is not found
    - Block is loaded in M state
    - No Bus update is required
- Replacement:
  - Block is written back if modified
    - M or Sm state only



Dragon's Lower-level Design Choices

- Shared-modified state can be eliminated
  - Main memory is updated on every Bus Update transaction
    - DEC Firefly multiprocessor
  - However, Dragon protocol does not update main memory on Bus Update
    - Only caches are updated
    - DRAM memory is slower to update than SRAM memory in caches
- Should replacement of an Sc block be broadcast to other caches?
  - Allow last copy to go to E or M state and not to generate future updates
- Can local copy be updated on write hit before controller gets bus?
  - Can mess up write serialization
  - A write to a non-exclusive block must be "seen" (updated) in all other caches BEFORE the write can be done in the local cache