Cache Coherence Using a Bus

Built on top of two fundamentals of uniprocessor systems

- Bus transactions
- State transition diagram in 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 (“dirty”)

Multiprocessors extend both these somewhat to implement coherence

Snooping-based Coherence

Basic Idea

Transactions on bus are visible to all processors

Processors or their representatives can snoop (monitor) bus and take action on relevant events (e.g. change state)

Implementing a Protocol

Cache controller now receives inputs from both sides:

- Requests from processor, bus requests/responses from snooper

In either case, takes zero or more actions

- Updates state, responds with data, generates new bus transactions

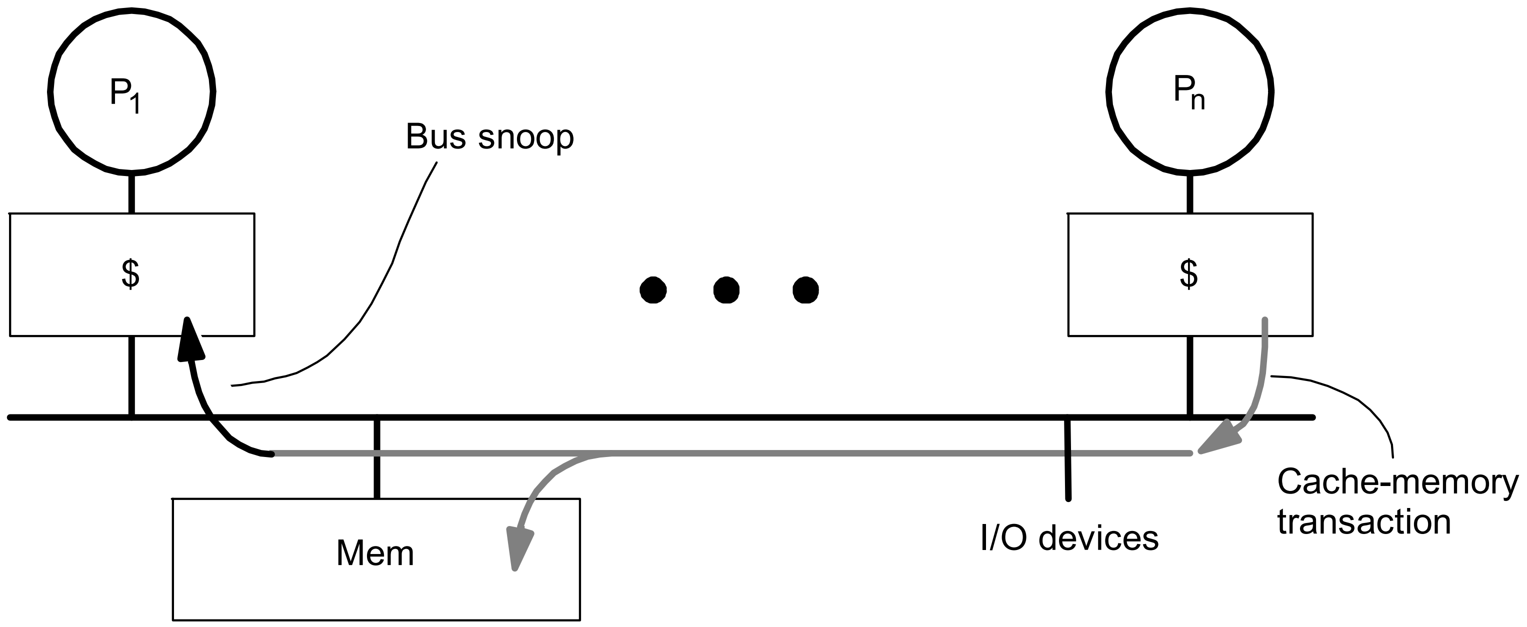
Protocol is distributed algorithm: cooperating state machines

- Set of states, state transition diagram, actions

Granularity of coherence is typically cache block

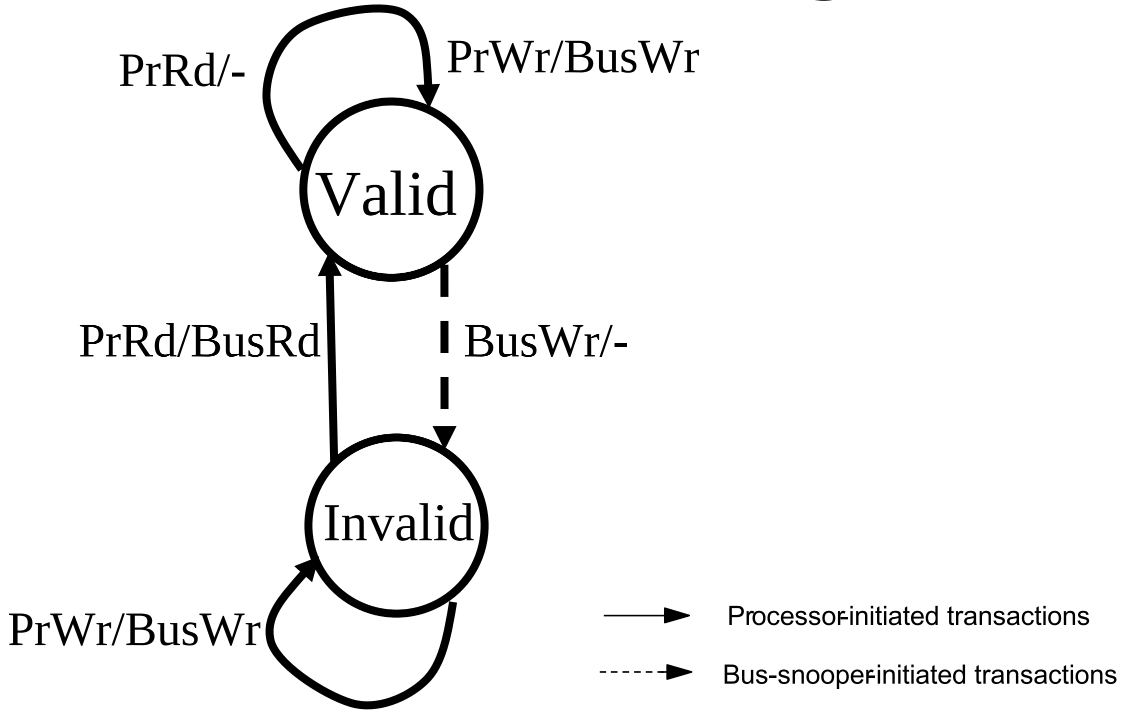
- Like that of allocation in cache and transfer to/from cache

Coherence with Write-through Caches



- Key extensions to uniprocessor: snooping, invalidating/updating caches
  - no new states or bus transactions in this case
  - invalidation- versus update-based protocols
- Write propagation: even in inval case, later reads will see new value
  - inval causes miss on later access, and memory up-to-date via write-through

Write-through Write-NoAllocate State Transition Diagram


PrRd/-
PrWr/BusWr
Valid
PrRd/BusRd
BusWr/-
Invalid
PrWr/BusWr
Processor-initiated transactions
Bus-snooper-initiated transactions

- Two states per block in each cache, as in uniprocessor
  - state of a block can be seen as *p*-vector
- Hardware state bits associated with only blocks that are in the cache
  - other blocks can be seen as being in invalid (not-present) state in that cache
- Write will invalidate all other caches (no local change of state)
  - can have multiple simultaneous readers of block,but write invalidates them

Is it Coherent?

Construct total order that satisfies program order, write serialization?

Assume atomic bus transactions and memory operations for now

- all phases of one bus transaction complete before next one starts
- processor waits for memory operation to complete before issuing next
- with one-level cache, assume invalidations applied during bus xaction
- (we’ll relax these assumptions in more complex systems later)

All writes go to bus + atomicity

- Writes serialized by order in which they appear on bus (*bus order*)
- Per above assumptions, invalidations applied to caches in bus order

How to insert reads in this order?

- Important since processors see writes through reads, so determines whether write serialization is satisfied
- But read hits may happen independently and do not appear on bus or enter directly in bus order

Ordering Reads

Read misses: appear on bus, and will see last write in bus order

Read hits: do not appear on bus

- But value read was placed in cache by either
  - most recent write by this processor, or
  - most recent read miss by this processor
- Both these transactions appear on the bus
- So reads hits also see values as being produced in consistent bus order

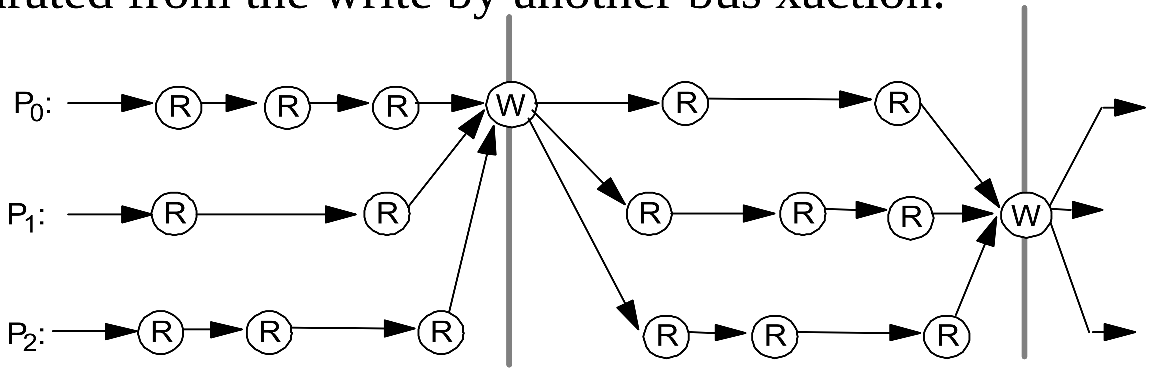
Determining Orders More Generally

A memory operation M2 is subsequent to a memory operation M1 if the operations are issued by the same processor and M2 follows M1 in program order.

Read is subsequent to write W if read generates bus xaction that follows that for W.

Write is subsequent to read or write M if M generates bus xaction and the xaction for the write follows that for M.

Write is subsequent to read if read does not generate a bus xaction and is not already separated from the write by another bus xaction.



- Writes establish a partial order
- Doesn't constrain ordering of reads, though bus will order read misses too
    - any order among reads between writes is fine, as long as in program order

SC in Write-through Example

Provides SC, not just coherence

Extend arguments used for coherence

- Writes and read misses to *all locations* serialized by bus into bus order
- If read obtains value of write W, W guaranteed to have completed
  - since it caused a bus transaction
- When write W is performed w.r.t. any processor, all previous writes in bus order have completed

Problem with Write-Through

High bandwidth requirements

- Every write from every processor goes to shared bus and memory
- 1GB/s bus can support only about 4 processors without saturating
- Write-through especially unpopular for SMPs

Write-back caches absorb most writes as cache hits

- Write hits don’t go on bus
- But now how do we ensure write propagation and serialization?
- Need more sophisticated protocols: large design space

Design Space for Snooping Protocols

No need to change processor, main memory, cache …

- Extend cache controller and exploit bus (provides serialization)

Focus on protocols for write-back caches

Dirty state now also indicates exclusive ownership

- Exclusive: only cache with a valid copy (main memory may be too)
- Owner: responsible for supplying block upon a request for it

Design space

- Invalidation versus Update-based protocols
- Set of states

Invalidation-based Protocols

Exclusive means can modify without notifying anyone else

- i.e. without bus transaction
- Must first get block in exclusive state before writing into it
- Even if already in valid state, need transaction, so called a write miss

Store to non-dirty data generates a *read-exclusive* bus transaction

- Tells others about impending write, obtains exclusive ownership
  - makes the write visible, i.e. write is performed
  - may be actually observed (by a read miss) only later
  - write hit made visible (performed) when block updated in writer's cache
- Only one RdX can succeed at a time for a block: serialized by bus

Read and Read-exclusive bus transactions drive coherence actions

- Writeback transactions also, but not caused by memory operation and quite incidental to coherence protocol
  - note: replaced block that is not in modified state can be dropped

Basic MSI Writeback Inval Protocol

States

- Invalid (I)
- Shared (S): one or more
- Dirty or Modified (M): one only

Processor Events:

- PrRd (read)
- PrWr (write)

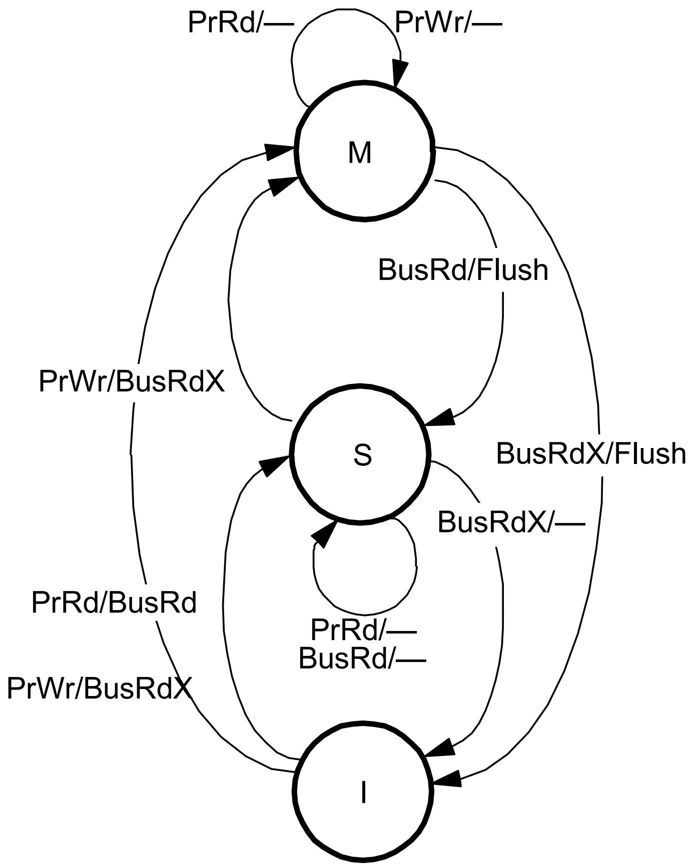
Bus Transactions

- BusRd: asks for copy with no intent to modify
- BusRdX: asks for copy with intent to modify
- BusWB: updates memory

Actions

- Update state, perform bus transaction, flush value onto bus

State Transition Diagram



- Write to shared block:
  - Already have latest data; can use upgrade (BusUpgr) instead of BusRdX
- Replacement changes state of two blocks: outgoing and incoming

Satisfying Sequential Consistency

1. Appeal to definition:
   - Bus imposes total order on bus xactions for all locations
   - Between xactions, procs perform reads/writes locally in program order
   - So any execution defines a natural partial order
     - M_j subsequent to M_i if (I) follows in program order on same processor, (ii) M_j generates bus xaction that follows the memory operation for M_i
   - In segment between two bus transactions, any interleaving of ops from different processors leads to consistent total order
   - In such a segment, writes observed by processor P serialized as follows
     - Writes from other processors by the previous bus xaction P issued
     - Writes from P by program order
2. Show sufficient conditions are satisfied
   - Write completion: can detect when write appears on bus
   - Write atomicity: if a read returns the value of a write, that write has already become visible to all others already (can reason different cases)

MESI (4-state) Invalidation Protocol

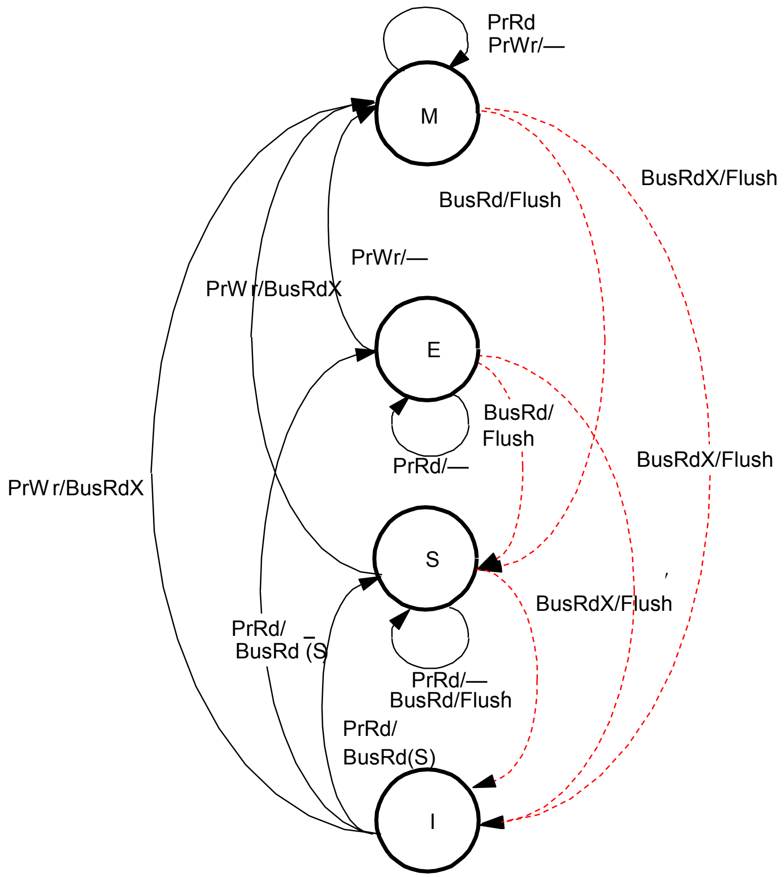
Problem with MSI protocol

- Reading and modifying data is 2 bus xactions, even if noone sharing
  - e.g. even in sequential program
  - BusRd (I->S) followed by BusRdX or BusUpgr (S->M)

Add *exclusive* state: write locally without xaction, but not modified

- Main memory is up to date, so cache not necessarily owner
- States
  - invalid
  - exclusive or *exclusive-clean* (only this cache has copy, but not modified)
  - shared (two or more caches may have copies)
  - modified (dirty)
- I -> E on PrRd if noone else has copy
  - needs "shared" signal on bus: wired-or line asserted in response to BusRd

MESI State Transition Diagram



- BusRd(S) means shared line asserted on BusRd transaction
- Flush’: if cache-to-cache sharing (see next), only one cache flushes data

Lower-level Protocol Choices

Who supplies data on miss when not in M state: memory or cache

Original, *lllinois* MESI: cache, since assumed faster than memory

- *Cache-to-cache sharing*

Not true in modern systems

- Intervening in another cache more expensive than getting from memory

Cache-to-cache sharing also adds complexity

- How does memory know it should supply data (must wait for caches)
- Selection algorithm if multiple caches have valid data

But valuable for cache-coherent machines with distributed memory

- May be cheaper to obtain from nearby cache than distant memory
- Especially when constructed out of SMP nodes (Stanford DASH)

Update-based Protocols

A write operation updates values in other caches

- New, update bus transaction

Dragon Write-back Update Protocol

4 states

- Exclusive-clean or exclusive (E): I and memory have it
- Shared clean (Sc): I, others, and maybe memory, but I’m not owner
- Shared modified (Sm): I and others but not memory, and I’m the owner
  - Sm and Sc can coexist in different caches, with only one Sm
- Modified or dirty (D): I and, no one else

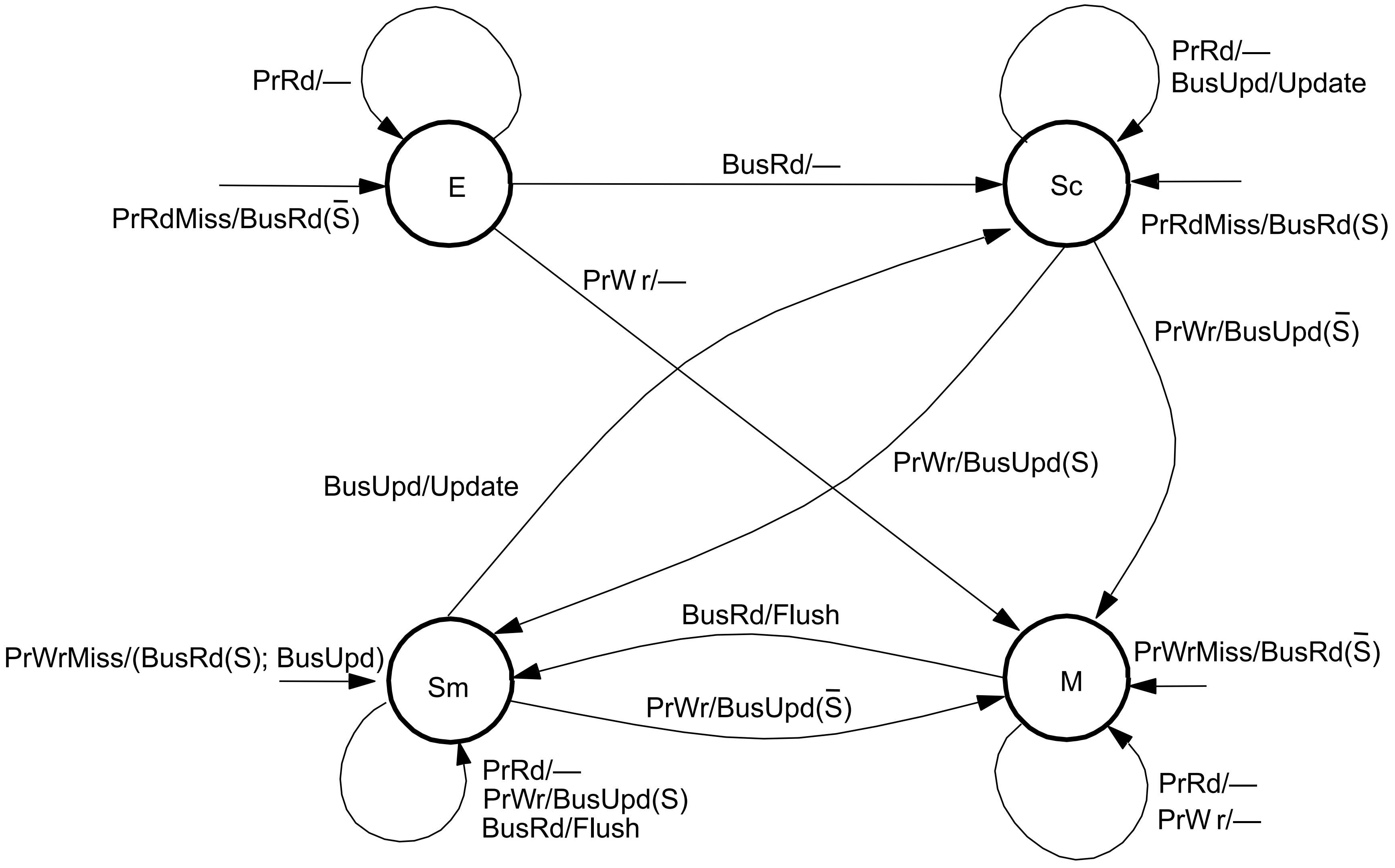
No invalid state

- If in cache, cannot be invalid
- If not present in cache, can view as being in not-present or invalid state

New bus transaction: BusUpd

- Broadcasts single word written on bus; updates other relevant caches

Dragon State Transition Diagram



Invalidate versus Update

Basic question of program behavior

- Is a block written by one processor read by others before it is rewritten?

Invalidation:

- Yes => readers will take a miss
- No => multiple writes without additional traffic
  - and clears out copies that won't be used again

Update:

- Yes => readers will not miss if they had a copy previously
  - single bus transaction to update all copies
- No => multiple useless updates, even to dead copies

Need to look at program behavior and hardware complexity

Invalidation protocols much more popular (more later)

- Some systems provide both, or even hybrid

Satisfying Coherence

Write propagation is clear

Write serialization?

- All writes that appear on the bus (BusRdX) ordered by the bus
  - Write performed in writer’s cache before it handles other transactions, so ordered in same way even w.r.t. writer
- Reads that appear on the bus ordered wrt these
- Write that don’t appear on the bus:
  - sequence of such writes between two bus xactions for the block must come from same processor, say P
  - in serialization, the sequence appears between these two bus xactions
  - reads by P will seem them in this order w.r.t. other bus transactions
  - reads by other processors separated from sequence by a bus xaction, which places them in the serialized order w.r.t the writes
  - so reads by all processors see writes in same order