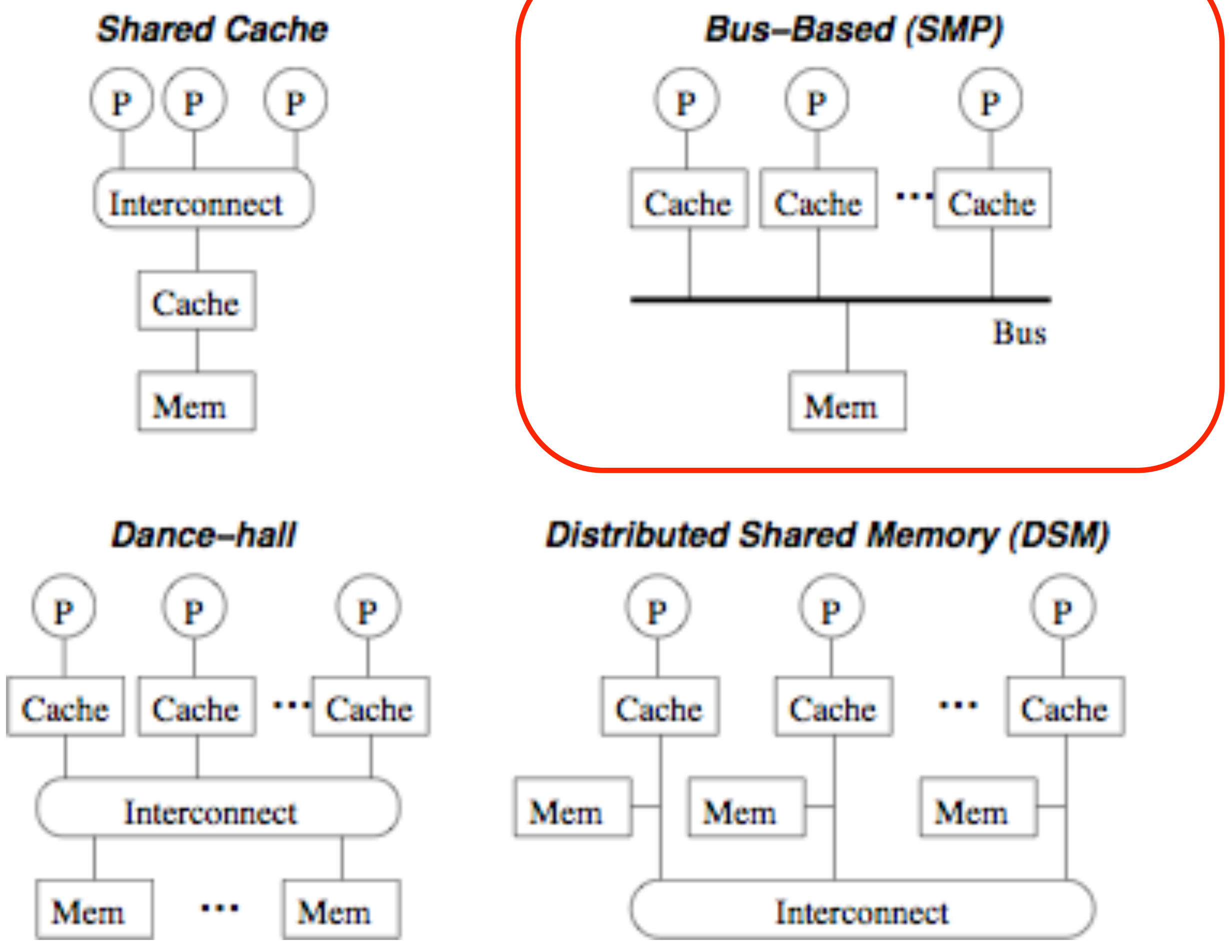
Chapter 8
Bus-Based Coherent Multiprocessors

Module 8.1 – Intro to Bus-Based Multiprocessor

Shared Memory Multiproc Organization



Focus: Bus-based, Centralized Memory

- Shared cache: popular in SMT and early multicores (Power4, Sparc, Pentium, Itanium)
  - Small number of cores makes a shared cache feasible
  - No coherence problem since there is only one copy of data
  - Produces contention and negative interference
    - Thread starvation, priority inversion, thread-mix dependent throughput
  - Higher hit and miss latencies due bandwidth and size pressure
- Bus-based: popular in SMP (symmetric multiprocessors)
- Dancehall: becoming popular in multicore
- Distributed memory
  - To be discussed later
- Organization not exclusive of one another - any combination of them may be employed

Bus-Based Multi Processor Systems

- Built on top of two fundamentals of uniprocessor sys.
  - Bus transactions
  - Cache line finite state machine
- Uniprocessor bus transaction:
  - Three phases: arbitration, command/address, data transfer
  - All devices observe addresses, one is responsible
- Uniprocessor cache states:
  - Every cache line has a finite state machine
  - In WT+write no-allocate: Valid, Invalid states
  - WB: Valid, Invalid, Modified (“Dirty”)
- Multiprocessors extend both these somewhat to implement coherence

BUS

- Single set of wires shared among multiple devices
- Contains *control lines* (for passing command and identifying data) and *data lines* (for passing data)
- Advantages vs. other interconnections
  - Low cost (cost does not scale with system size)
  - Versatile
  - New devices can be added easily
- Disadvantages
  - Communication bottleneck
- Bus speed
  - Max speed determined by length, number of devices
  - E.g. CPU/Memory Bus vs. I/O Bus
    - Memory bus: devices known → can be made shorter → faster
    - I/O bus: devices unknown, must be compatible across different systems → standardized and slower

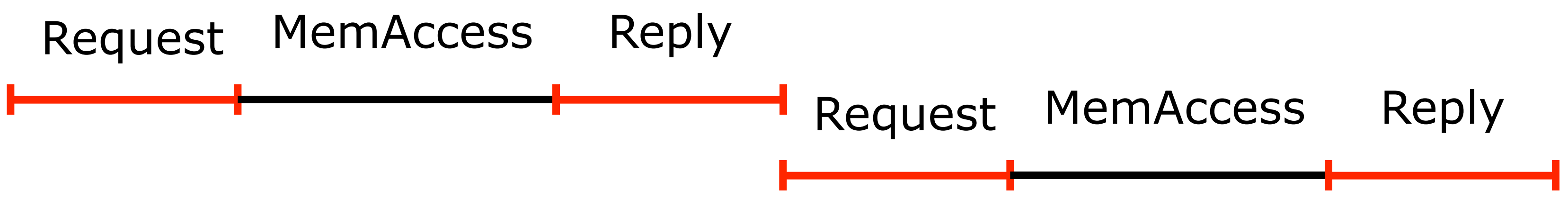
- Bus Transaction: sending address + receiving/sending data
  - Read transaction
  - Write transaction
- Bus timing:
  - Synchronous: all devices are controlled by the same clock signal
    - The timing of read/write/etc. is pre-determined
    - Bus must be short otherwise it has to have low frequency due to clock skew problem
  - Asynchronous: devices have independent clock signals
    - The timing for read/write/etc. must be established with handshaking protocol

Design Choices

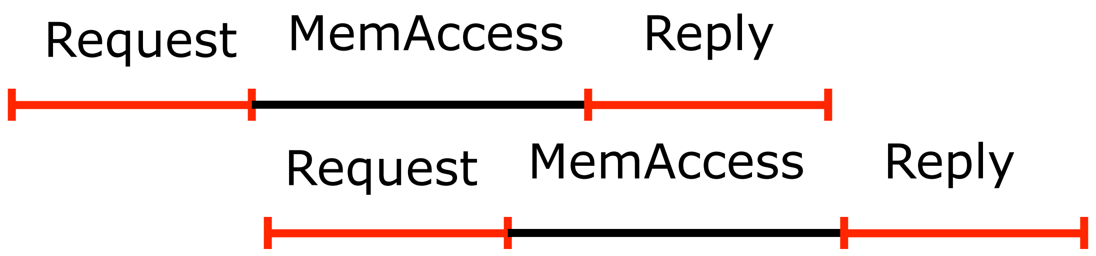
- Separate or shared address/data lines?
  - Separate: expensive
  - Multiplexed: longer latency
- Width of data line?
  - Wider: needs more chip pins (for off-chip buses)
- Transfer size?
  - Single word: simple
  - Multiple words: less overhead
- Transfer prioritization:
  - Critical word first: send the requested word first
- How many bus masters?
  - One: no bus arbitration logic needed
  - Multiple: requires bus arbitration, priority scheme
- Synchronous vs. Asynchronous
  - Sync: simpler, fast, have to be short (to mitigate clock skew), all devices at same speed
  - Typically, sync for Mem Bus, Async for IO bus

Timing: Atomic vs. Split Transaction

- Atomic



- Split Transaction



More attractive when MemAccess latency grows

Split Transaction vs. Atomic Bus

- Decoupling Request and Reply
- Transactions now have to be tagged
- Pending transactions buffer
- Requires Arbitration
- Adv:
  - Higher Bandwidth
  - Can tolerate higher latency
- Disadv:
  - More complex
  - Thus, higher latency

Bus Arbitration

- Deciding which device should be granted an access to the bus
- Necessary when there are multiple bus masters (e.g. multiple processors, split-transaction)
- Types
  - Daisy chain: bus grant line connected from highest priority to lowest priority devices
  - Centralized arbitration: each device has a request line, arbiter grants bus to selected device (e.g. PCI)
  - Distributed self-selection: each device examines the bus and decides whether to go ahead or not
  - Distributed with collision detection: optimistically places request, backs off and retries if a collision is detected (e.g. Ethernet)

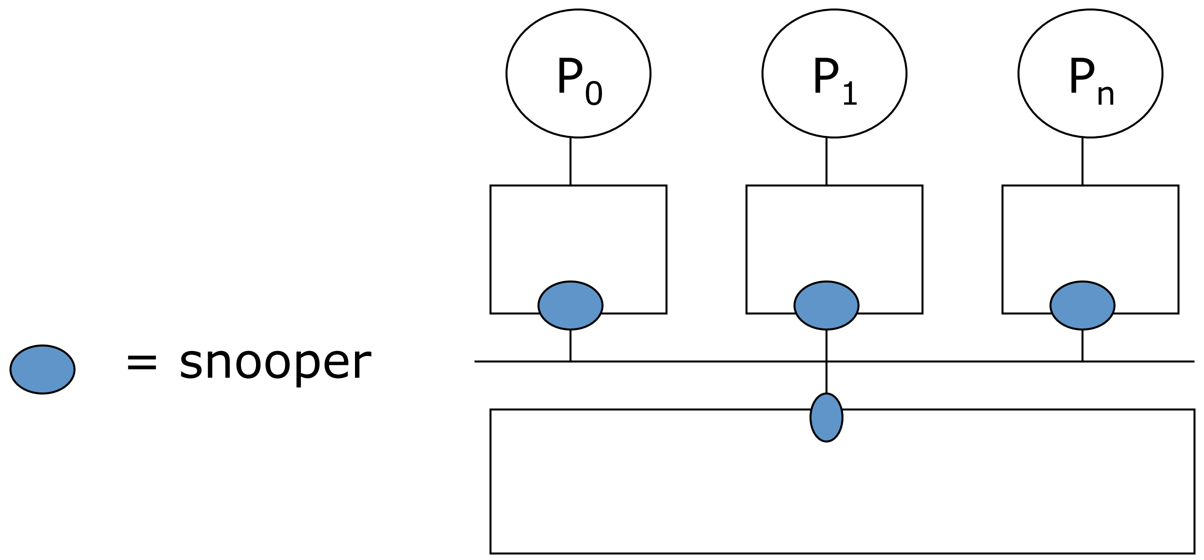
Providing Cache Coherence

- *Processor-Side:*
  - Add a snooper to each node to snoop all bus transactions
  - Caches (via cache controllers) react by changing line states
- *Memory-Side:*
  - Memory controller (MC) also snoops bus transactions
  - MC reacts by deciding whether to read line from memory and return it

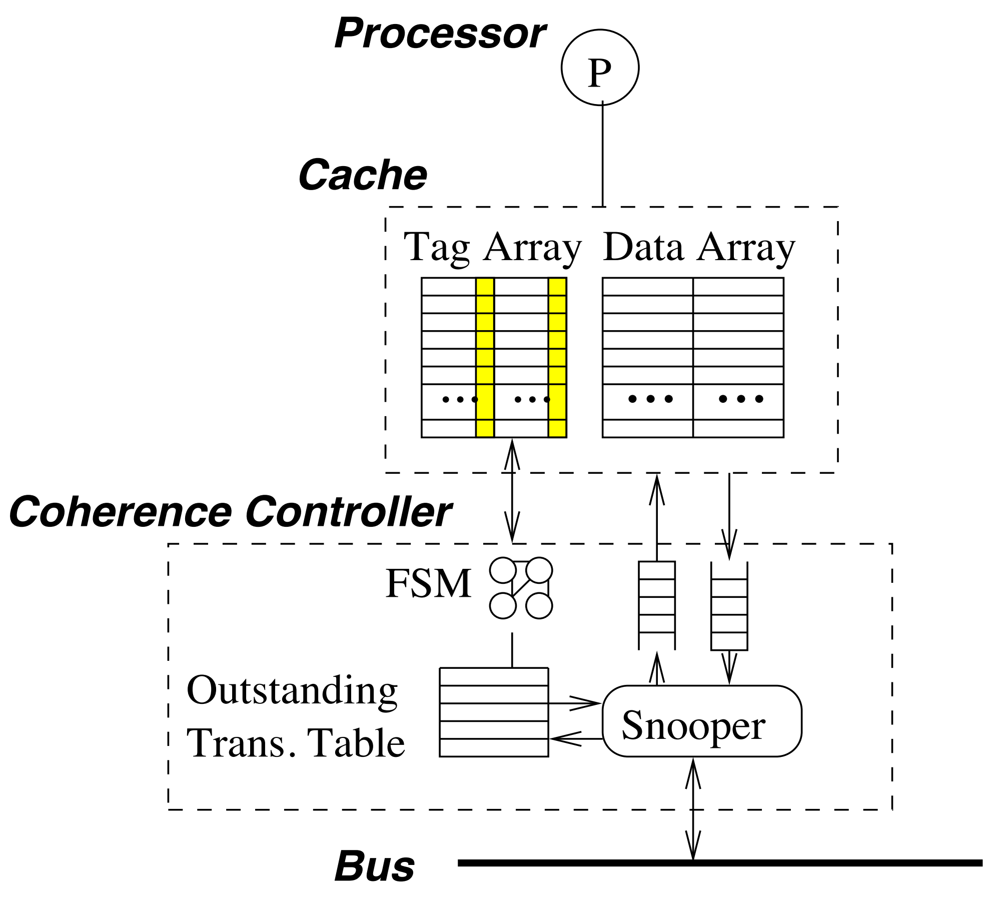
Coherence Correctness Requirements:

- Write propagation: on a snooped write,
  - ***Update***: snooped value updates local copy
  - ***Invalidate***: local copy invalidated to force future read to re-fetch the line
- Write serialization
  - Bus provides global ordering of writes
  - Each snooper reacts in bus order

Where Snoopers are Integrated



Architecture



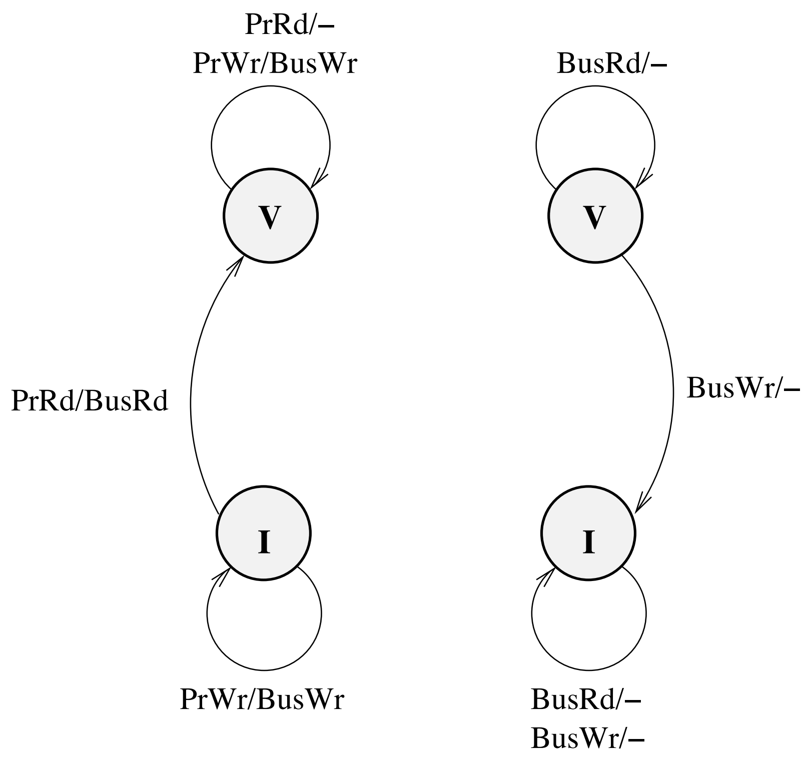
- Coherence controller is added
- Snooper snoops all bus transactions
- If a bus transaction is relevant, an action is taken
- Finite State Machine (FSM) determines the new state
- Outstanding transaction table keeps transactions that are pending (not completed yet)

Module 8.2 – Coherence for Write-Through Caches and MSI Protocol

Coherence Protocol for Write-Through Caches

- Processor transactions:
  - PrRd
  - PrWr
- Snooped bus transactions
  - BusRd
  - BusWr
- Cache state
  - Valid
  - Invalid
- Assumptions
  - Write no-allocate
  - Write invalidate

Write-through State Transition Diagram



- Key: write invalidates all other caches
- Therefore, we have:
  - Modified line: exists as V in only 1 cache
  - Clean line: exists as V in at least 1 cache
  - Invalid state represents invalidated line or not present in the cache

Problem with Write-Through Caches

- High bandwidth requirement due to each write causing a write propagation
- What if we use write-back caches?
- Write hits stop at the cache $\rightarrow$ eliminate most bus write transactions

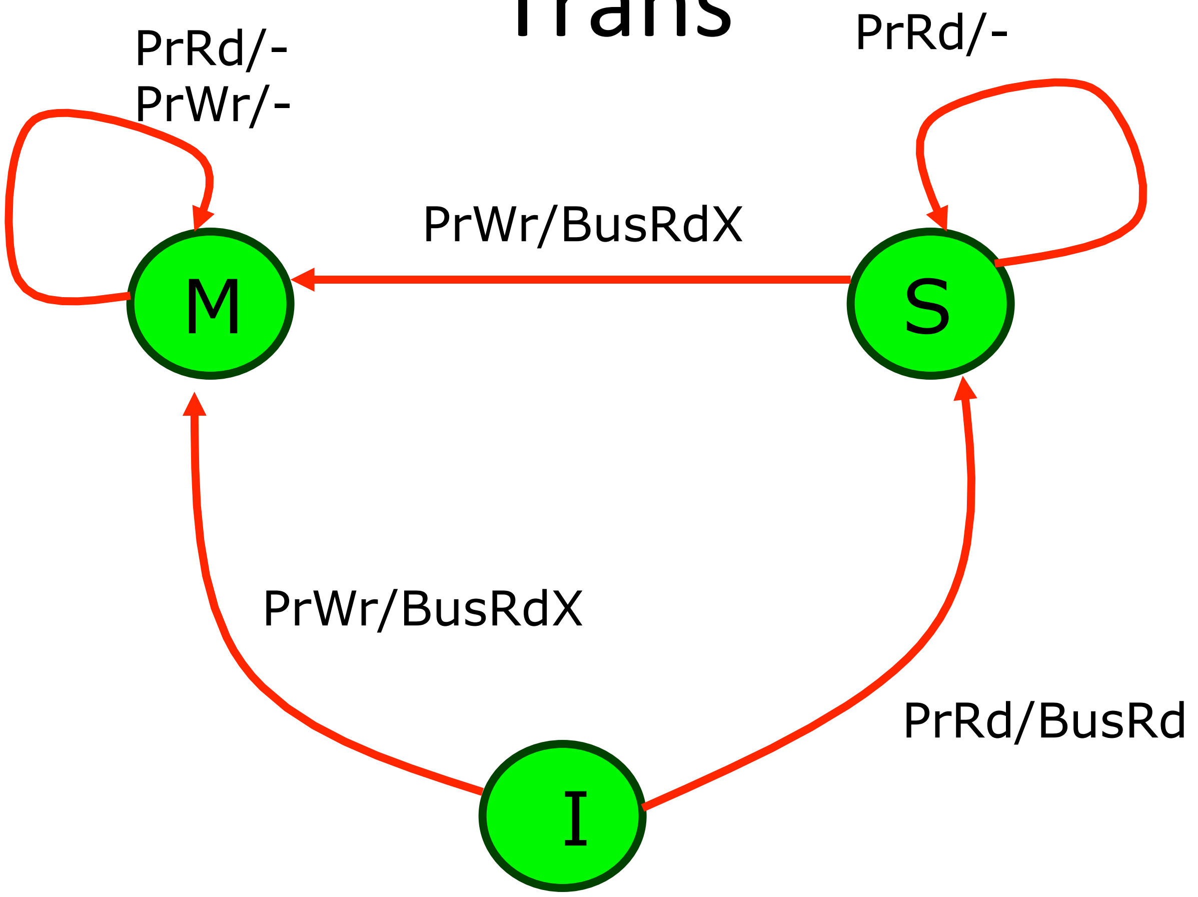
Basic MSI Writeback Inval Protocol

- States
  - Invalid (I)
  - Shared (S): one or more copies, and memory copy is up-to-date
  - Dirty or Modified (M): only one copy
- 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 (instead of BusWr)
  - Flush: updates memory
- Actions
  - Update state, perform bus transaction, flush value onto bus

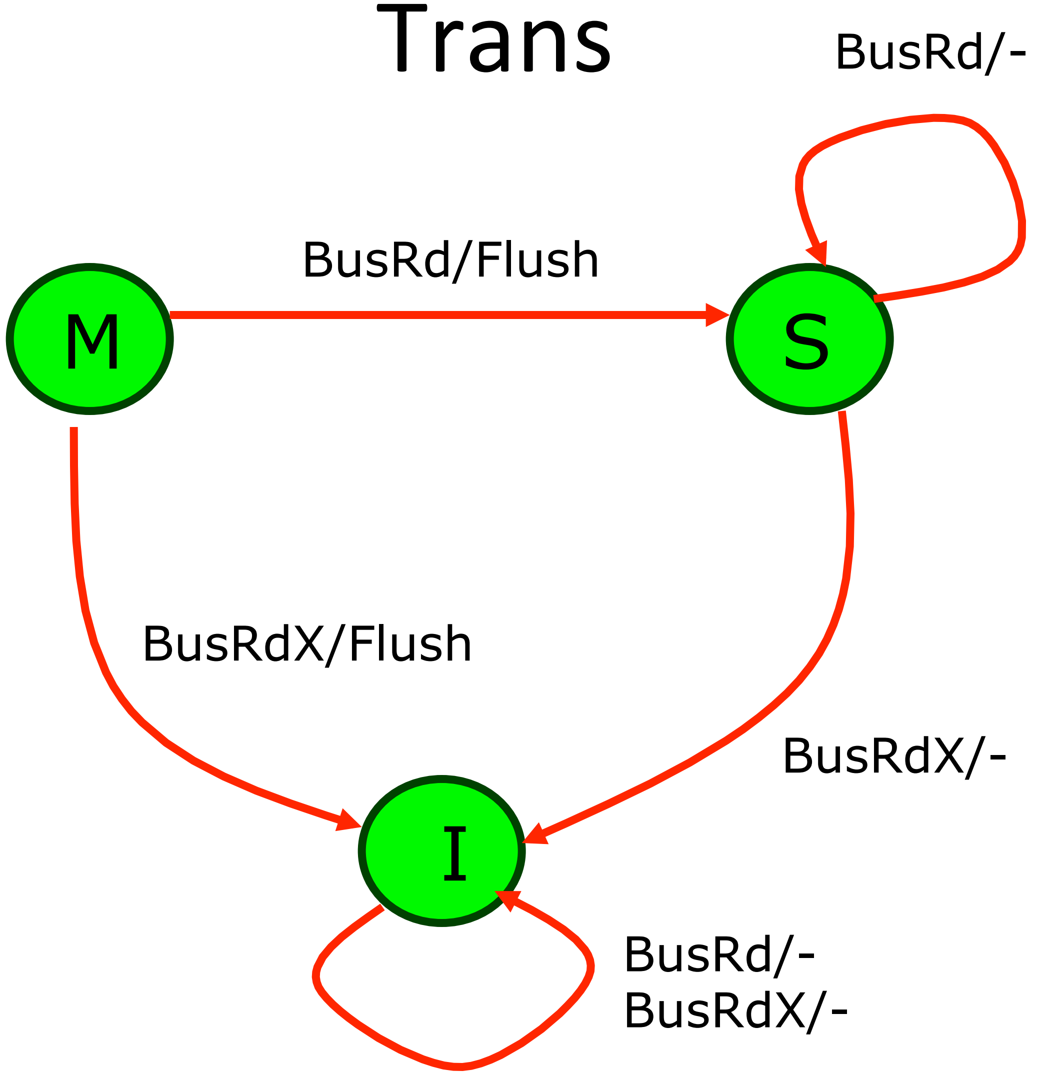
Definition

- Invalidation: Any -> I
- Intervention: Exclusive/Modified -> Shared

State Transition Diagram – Proc Init Trans

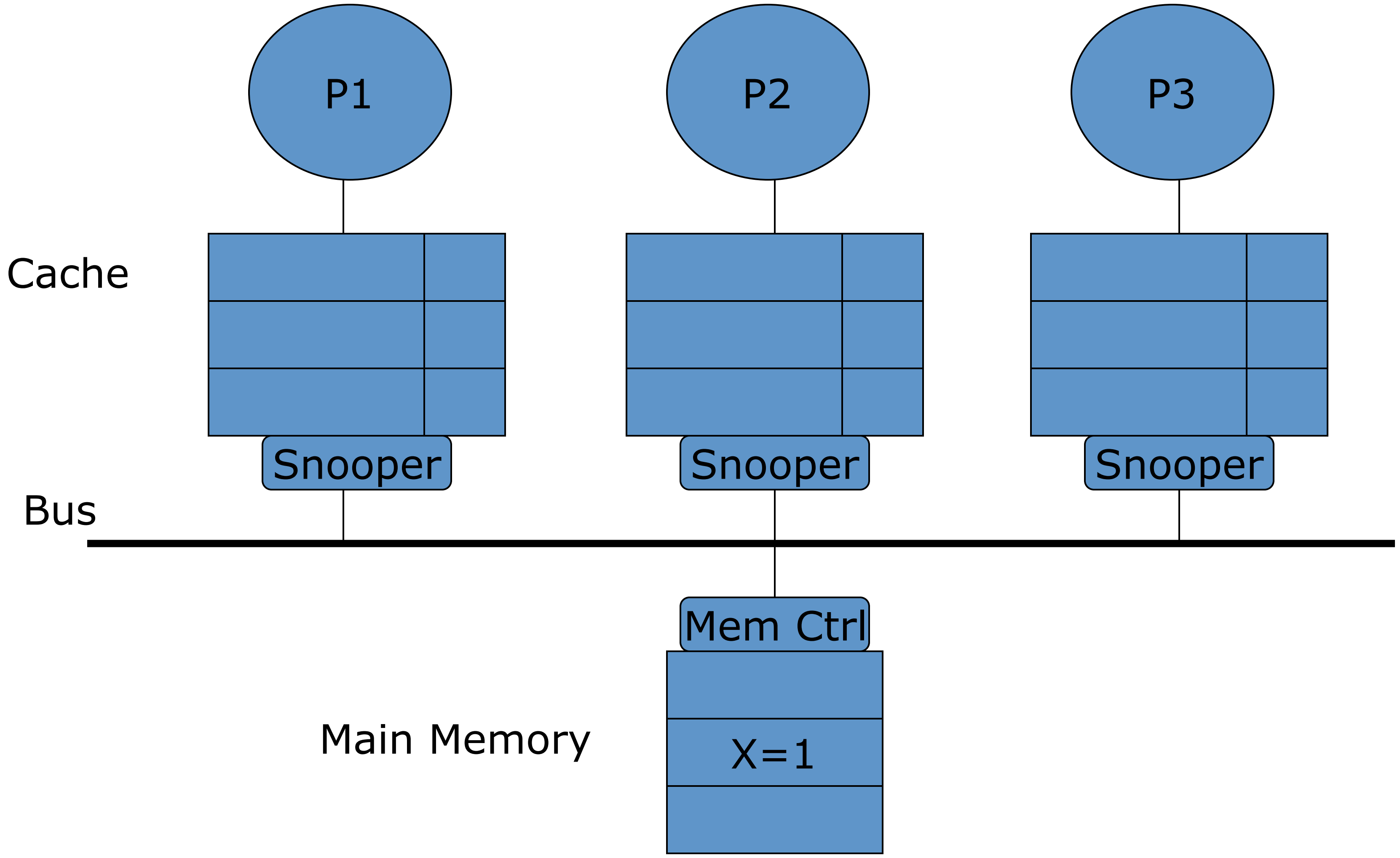


State Transition Diagram – Bus Init Trans

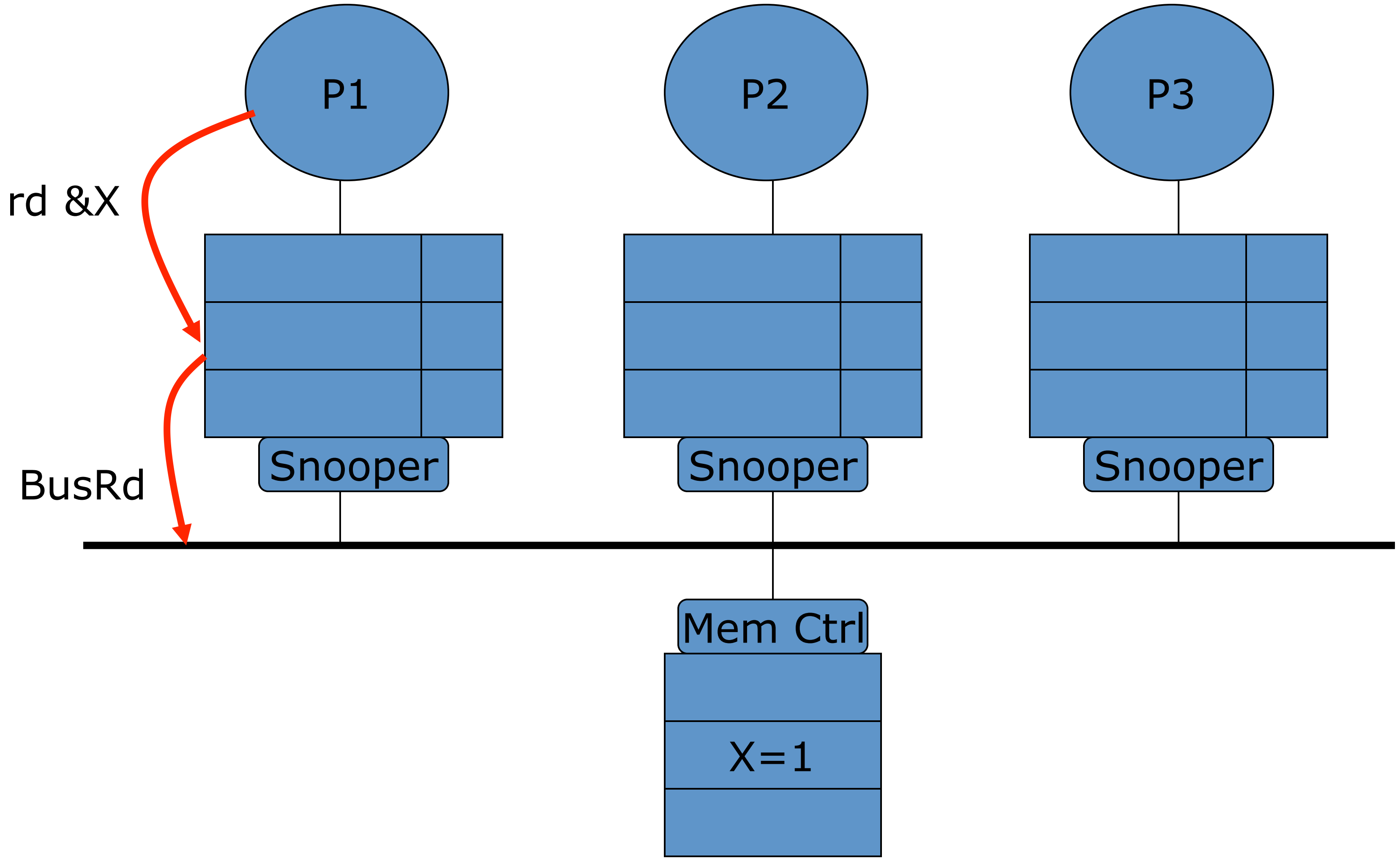


Thus, valid data must be supplied by memory

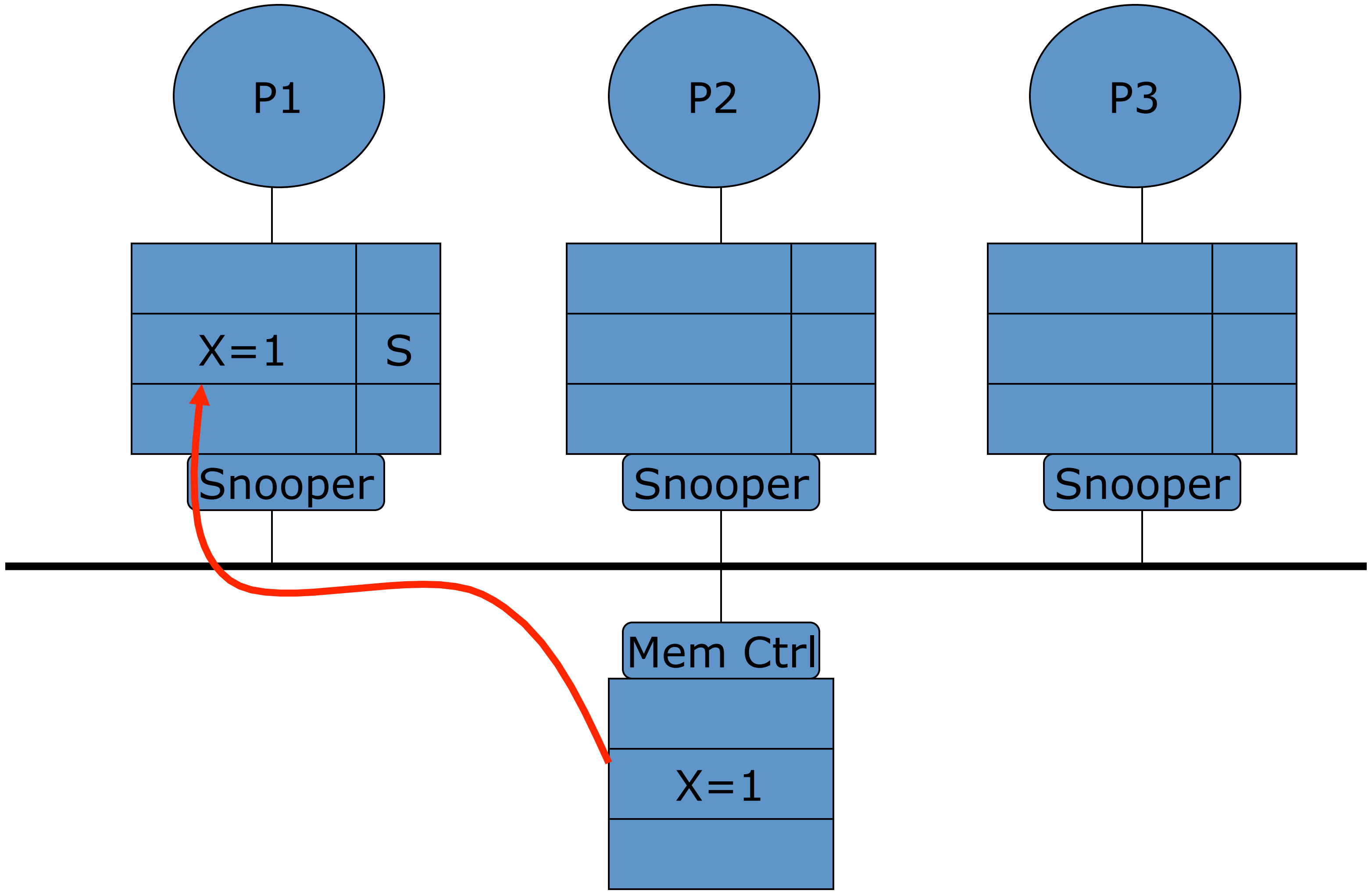
MSI Visualization



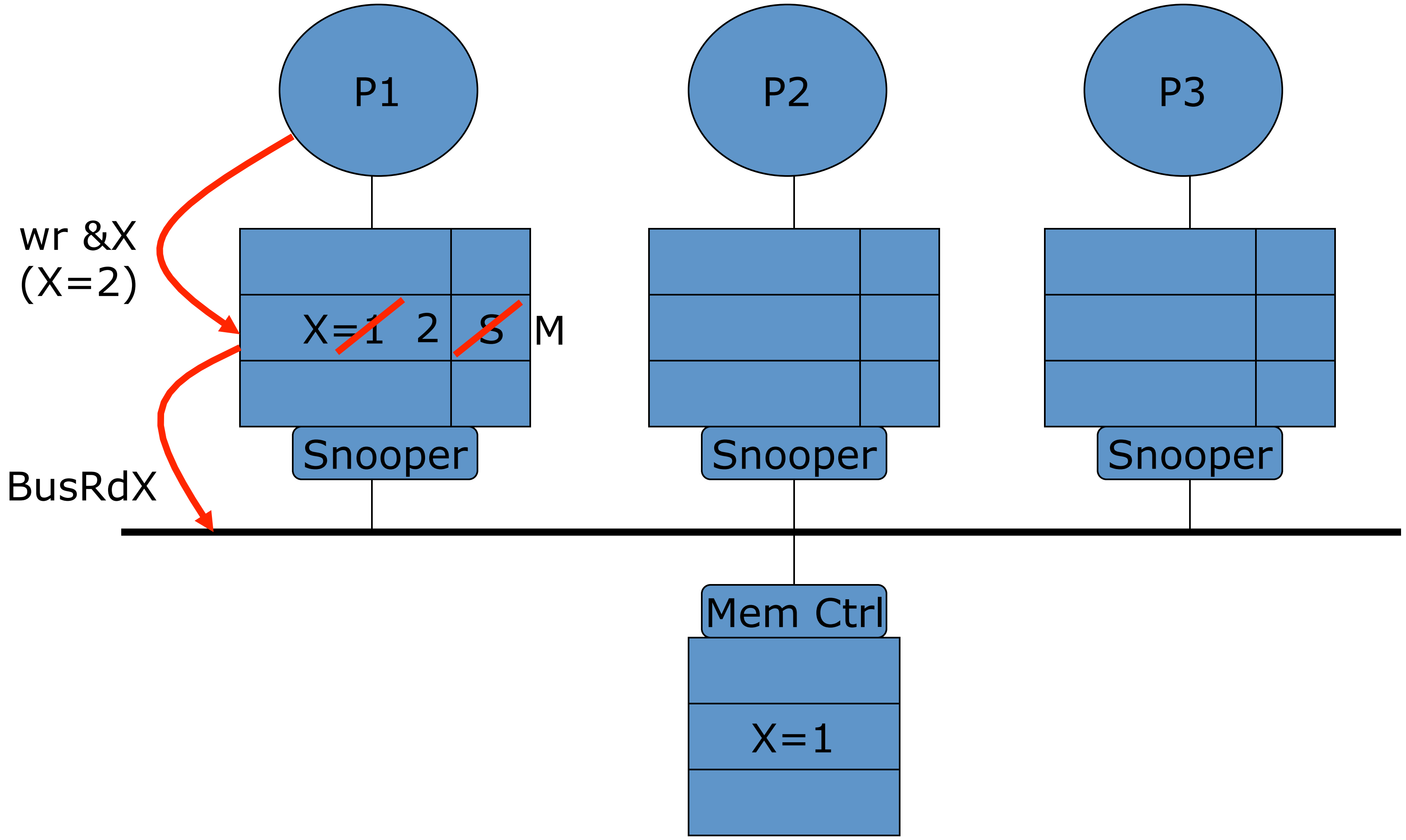
MSI Visualization



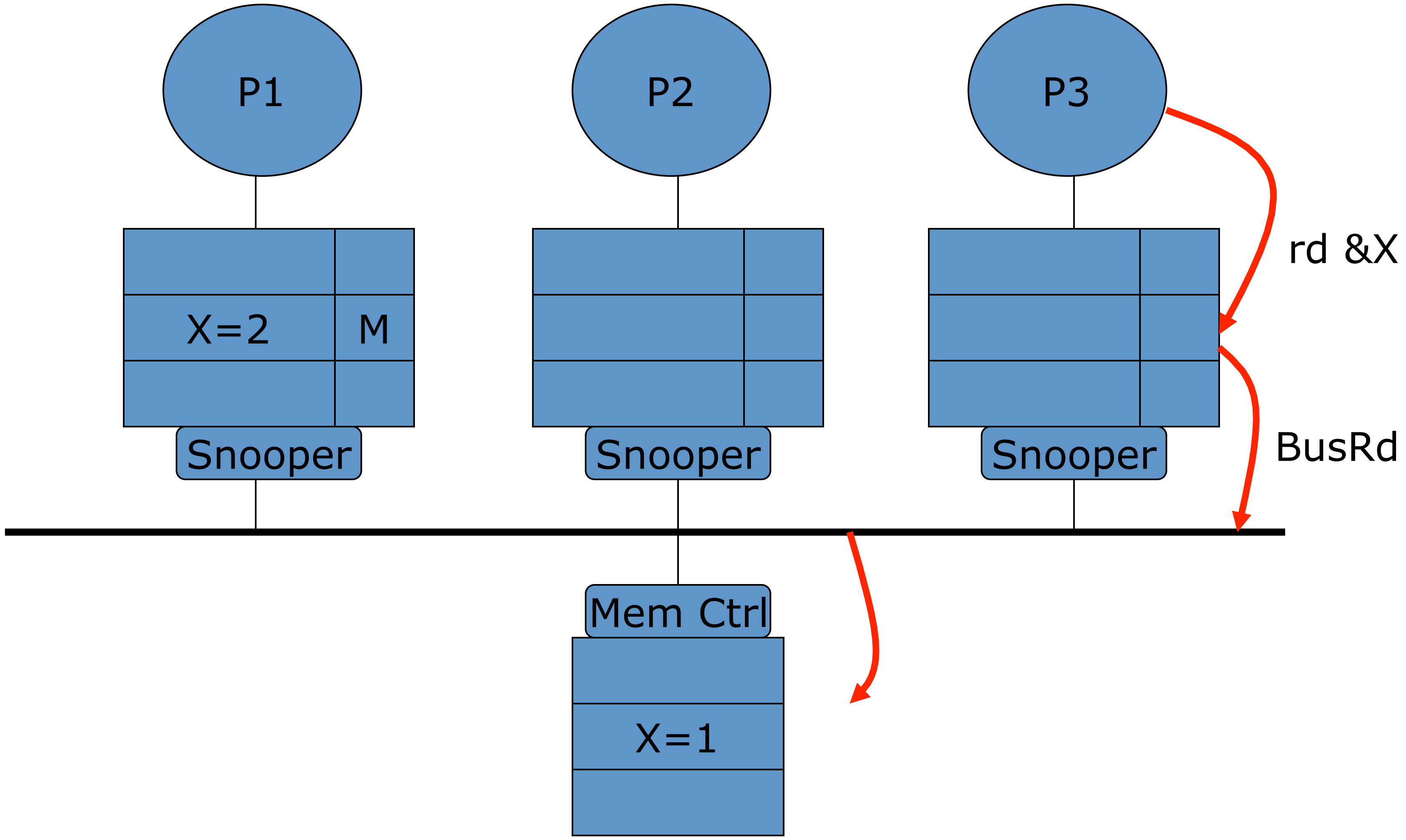
MSI Visualization



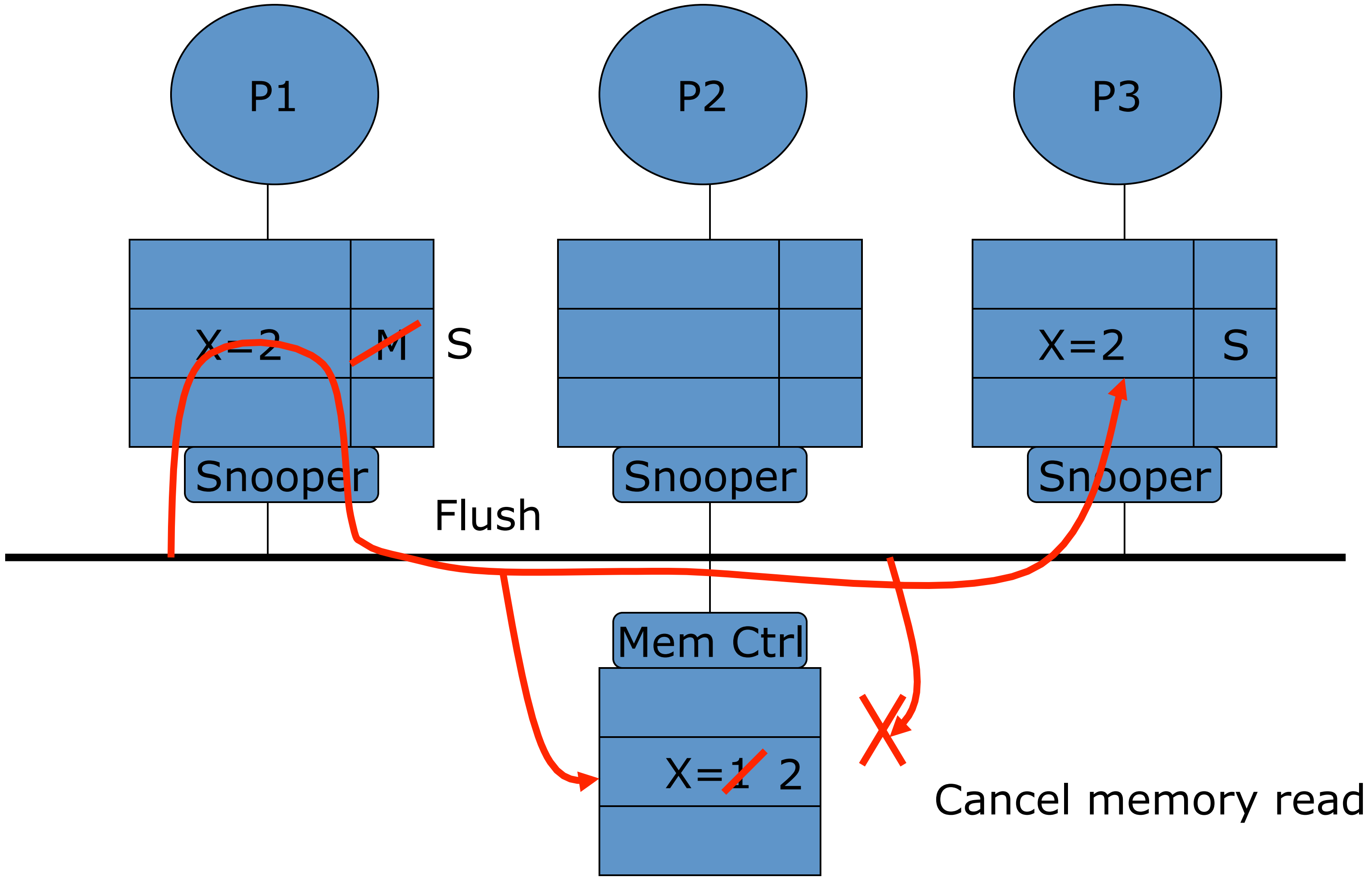
MSI Visualization



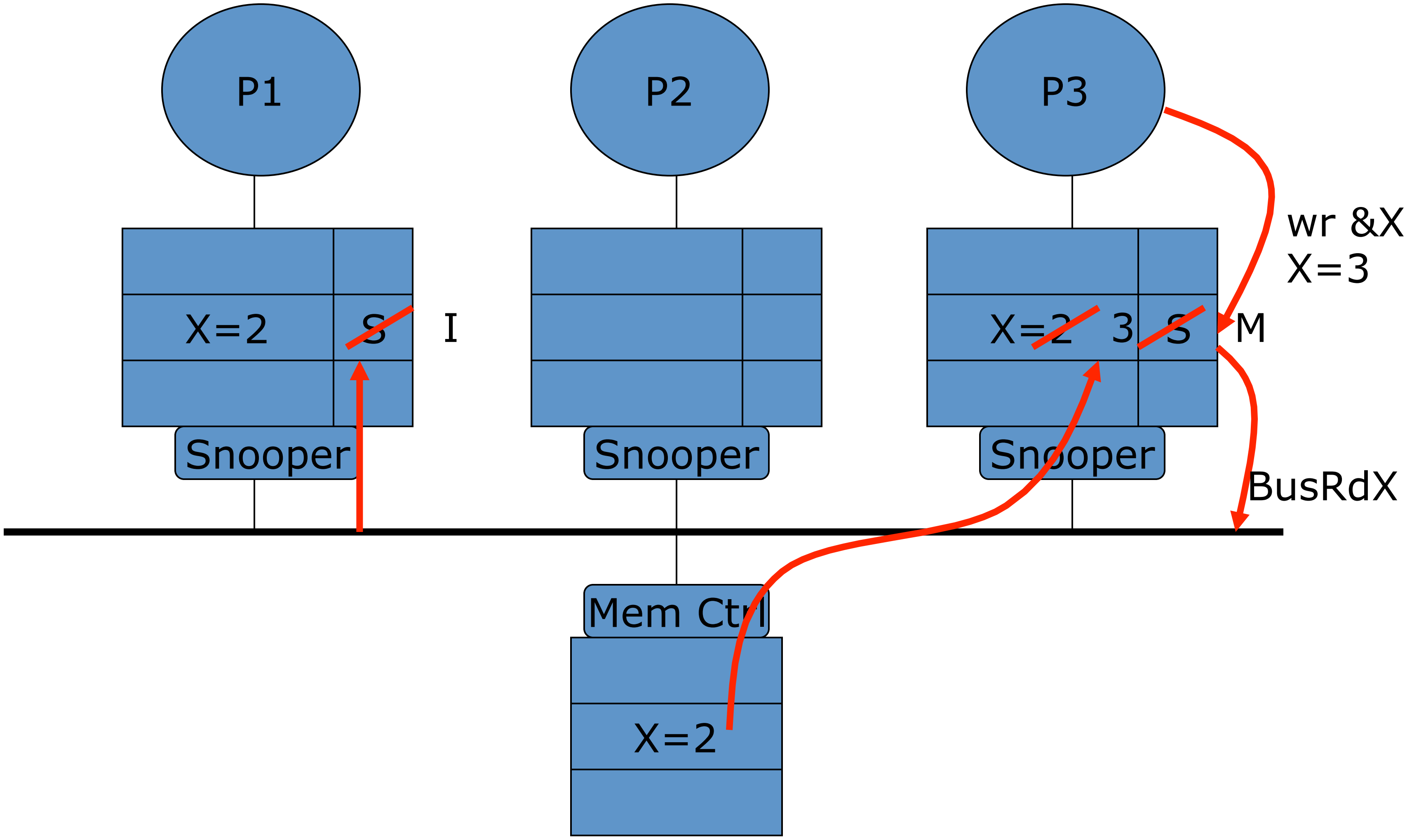
MSI Visualization



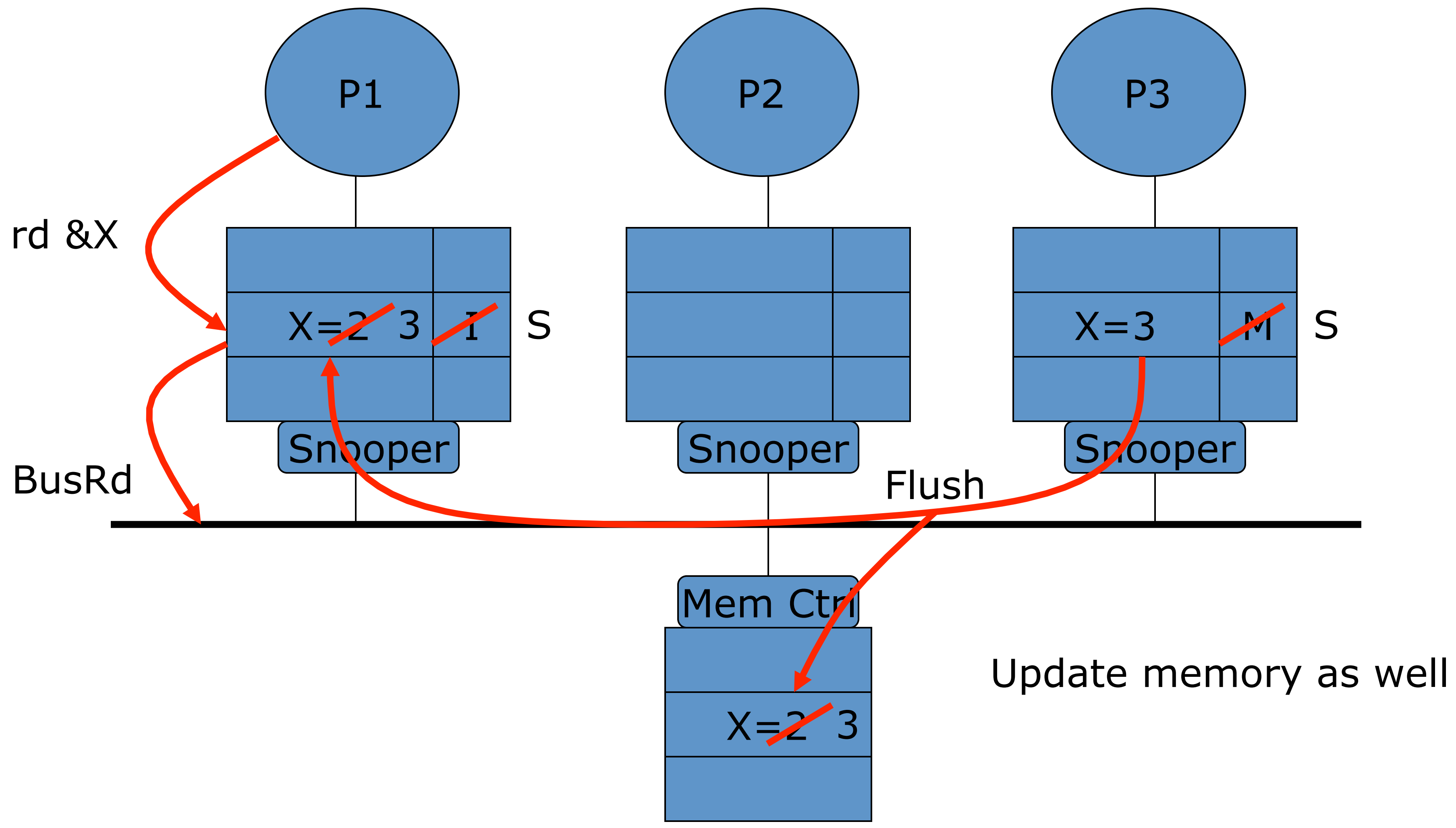
MSI Visualization



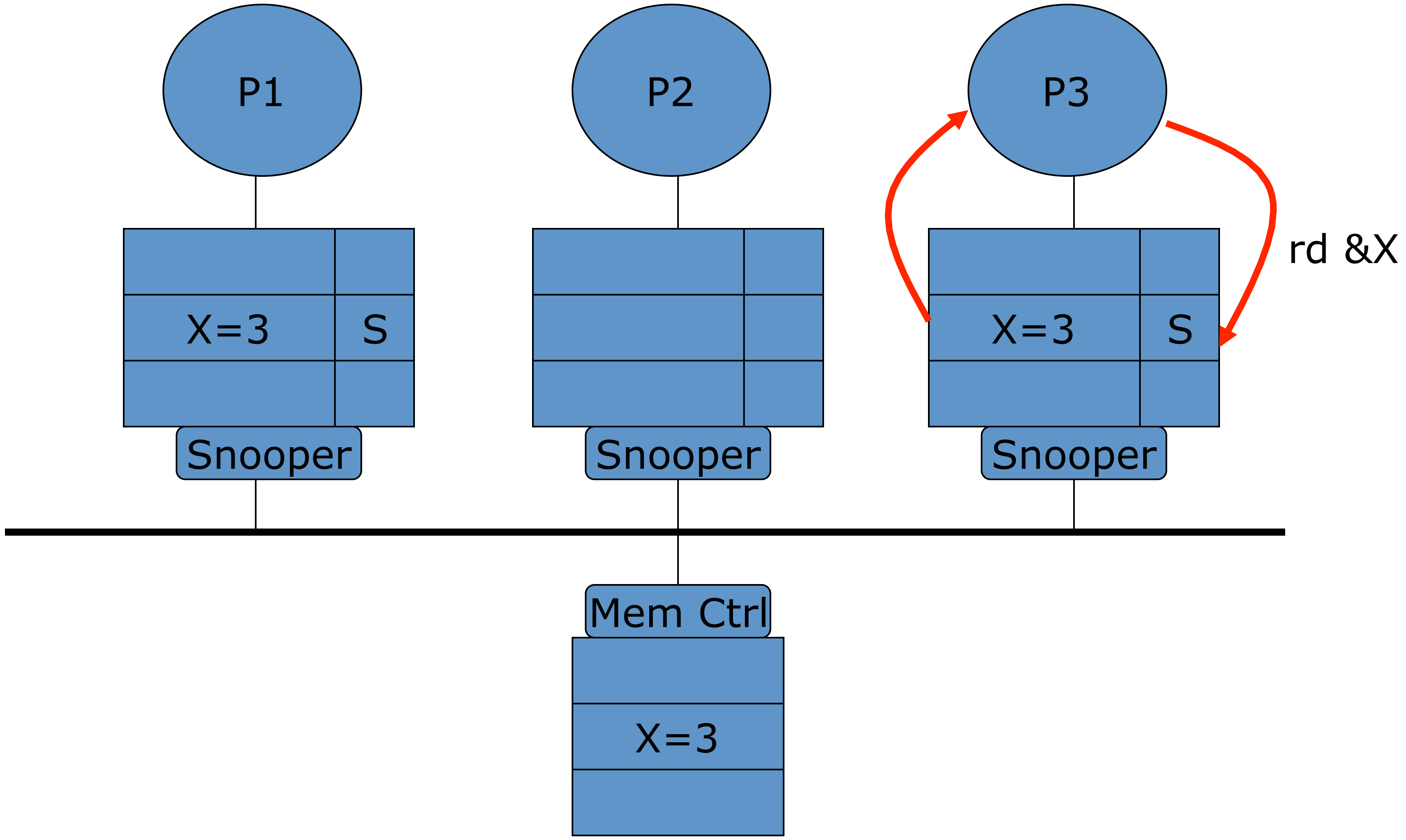
MSI Visualization



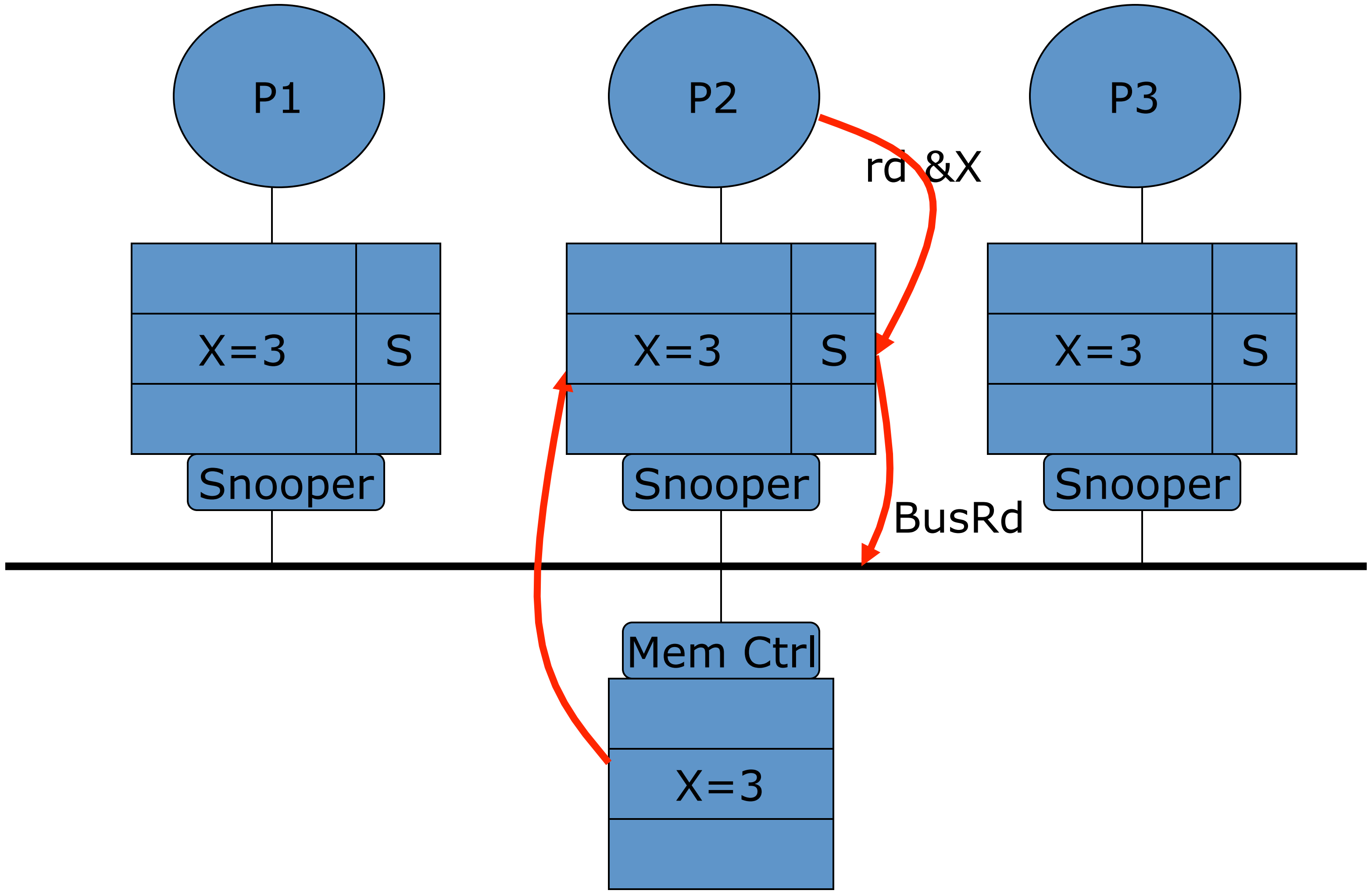
MSI Visualization



MSI Visualization



MSI Visualization



MSI Example

Assume:

- hit without bus transaction (1 cycle)
- hit/miss w/ bus transaction, no data or data from/to another cache (20 cycles)
- hit/miss w/ bus transaction, data from memory (40 cycles)

Proc Action	State @P1	State @P2	State @P3	Bus Action	Data Supplier	Cost
R1	S	-	-	BusRd	Mem	40
W1	M	-	-	BusRdX	Mem	40
R3	S	-	S	BusRd/ Flush	P1 cache	20
W3	I	-	M	BusRdX	Mem	40
R1	S	-	S	BusRd/ Flush	P3 cache	20
R3	S	-	S	-	-	1
R2	S	S	S	BusRd	Mem	40

Questions

- What are the actions at MC?

- Upon snooping BusRdX, if a block state is M, why do we need to Flush the block? Why bring a block that we want to overwrite anyway?

- Write to Shared block (already in cache):
  - Who supplies data on BusRdX?
  - What is a more efficient mechanism? BusUpgr vs. BusRdX

Answers

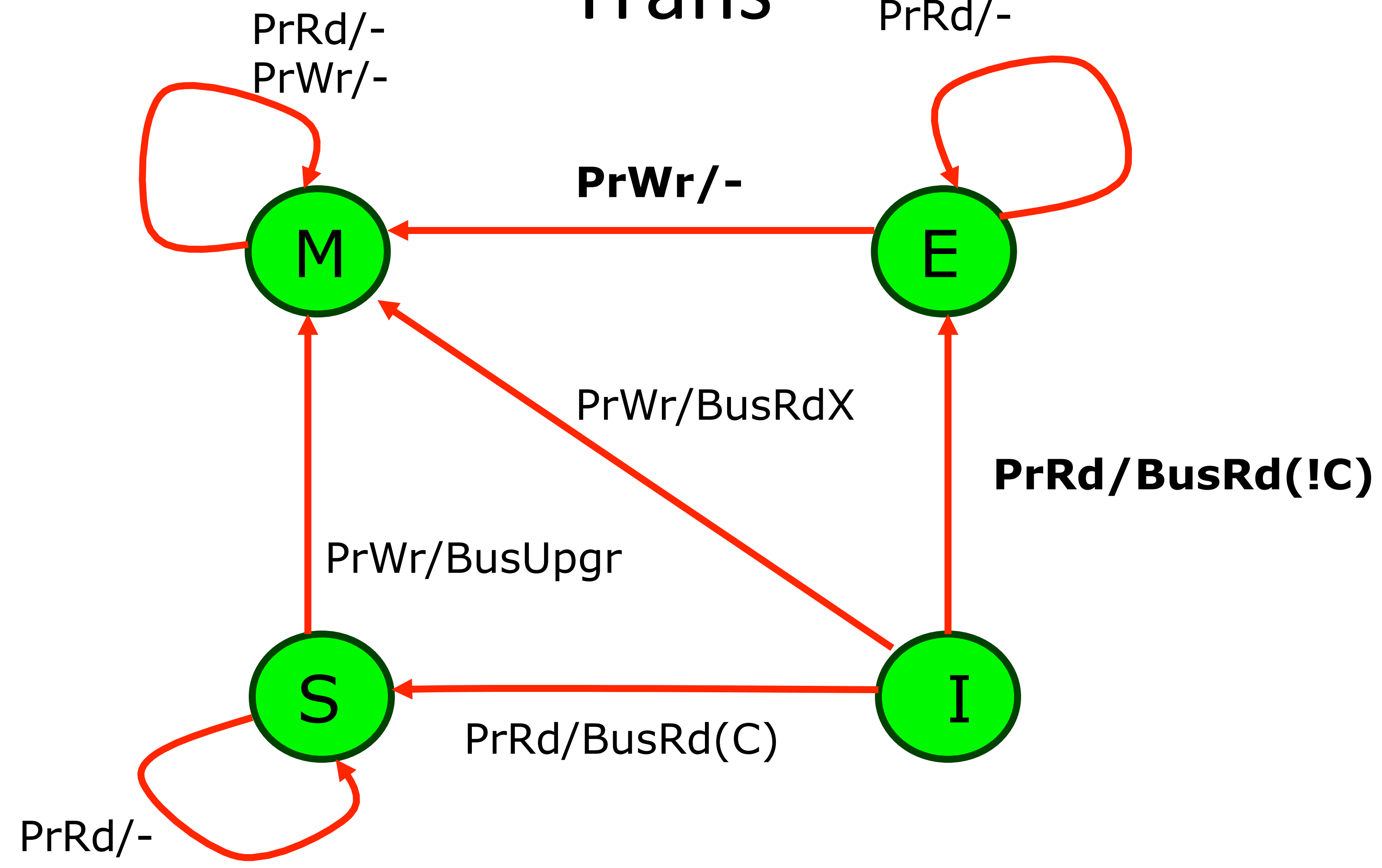
- What are the actions at MC?
  - Snooped a BusRd: read and supply data
  - Snooped a BusRdX: read and supply data
  - Snooped a Flush: update main memory
- Upon snooping BusRdX, if a block state is M, why do we need to Flush the block? Why bring a block that we want to overwrite anyway?
  - Writes could be to different words in a block → we must get the latest copy
  - In some protocols, BusRdX = intention to write, so may be followed by a Read!
- Write to shared block (already in cache):
  - Who supplies data on BusRdX?
  - What is a more efficient mechanism? BusUpgr vs. BusRdX
    - (read more details in the book)

Module 8.3 – MESI and MOESI Protocols

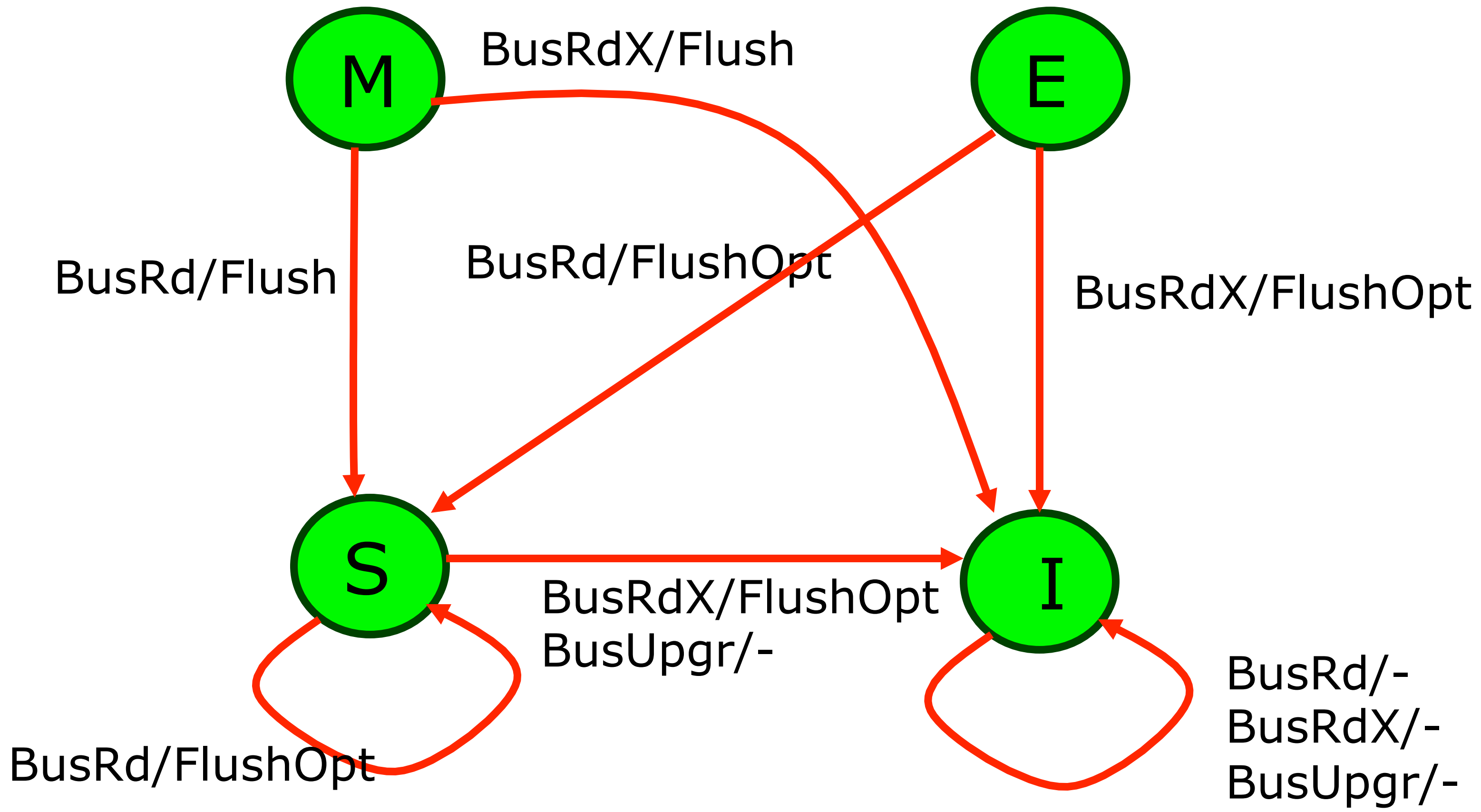
MESI (4-state) Invalidation Protocol

- Problem with MSI protocol
  - Rd, Wr sequence incurs 2 transactions
    - even when no one sharing (e.g., serial program!)
    - BusRd (I->S) followed by BusRdX or BusUpgr (S->M)
    - *In general, penalizing serial programs is unacceptable*
- Add *exclusive* state:
    - Invalid
    - modified (dirty)
    - shared (two or more caches may have copies)
    - Exclusive: (only this cache has *clean* copy, same value as in memory)
- How to decide I $\rightarrow$ E or I $\rightarrow$ S?
  - Need to check whether someone else has copy
  - a separate signal on bus: *wired-or* line asserted in response to BusRd. Call it C = "copy exists"

State Transition Diagram – Proc Init Trans



State Transition Diagram – Bus Init Trans

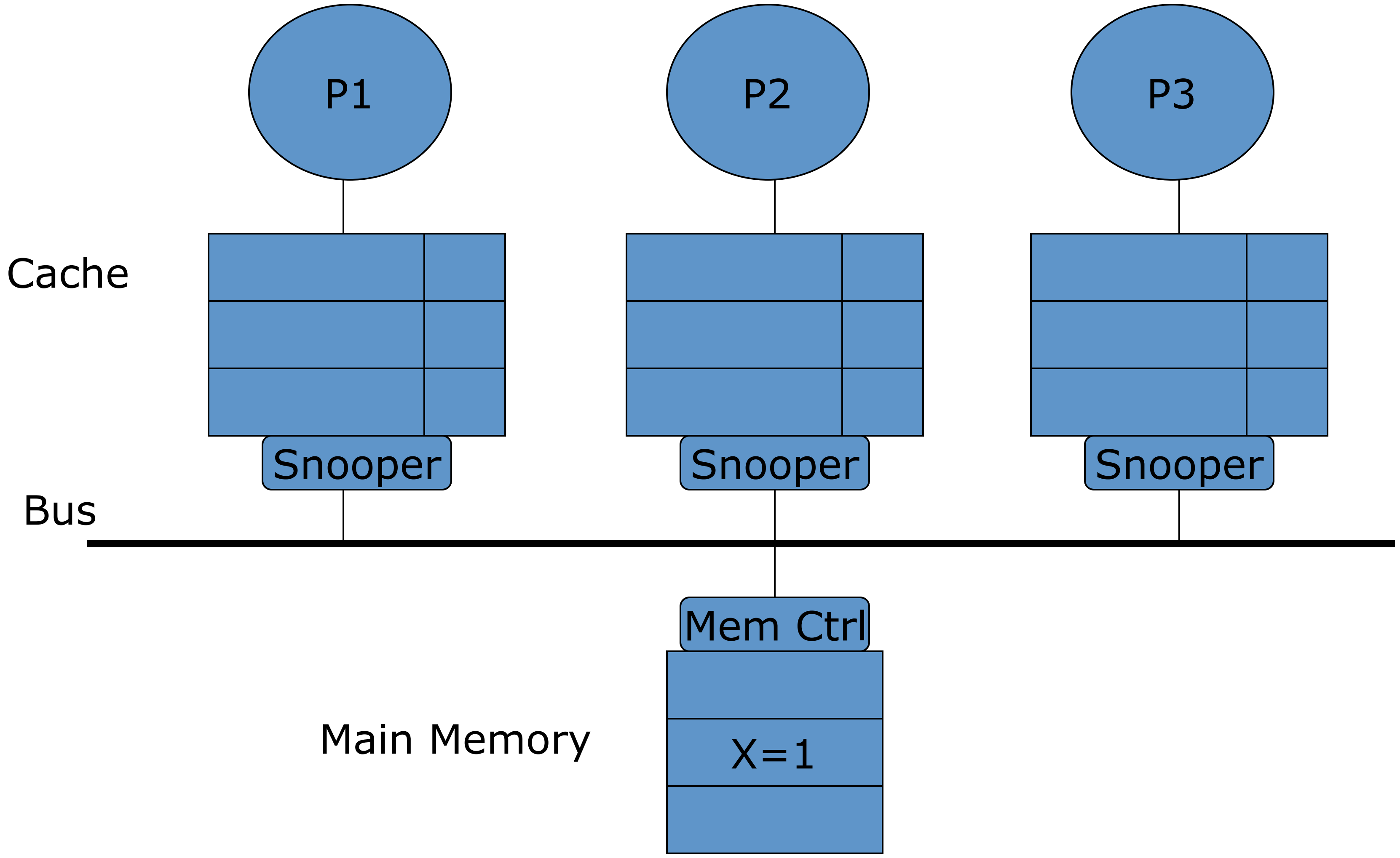


Flush vs. FlushOpt

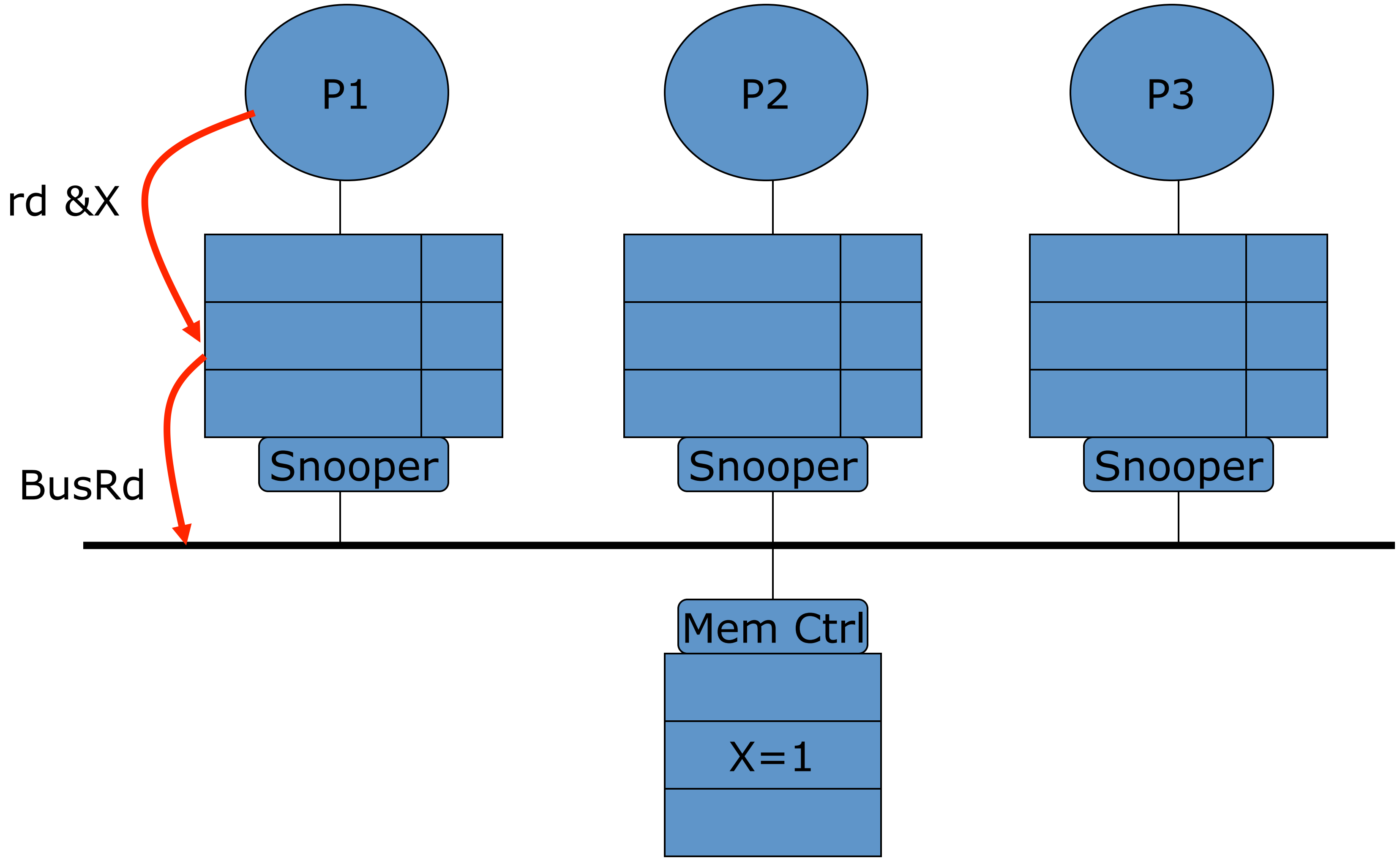
- Flush: mandatory
- FlushOpt:optional
  - Referred to as cache-to-cache transfer
  - Based on a premise that obtaining data from another cache is faster than obtaining it from the lower level memory
    - Premise not always valid

MESI Animation

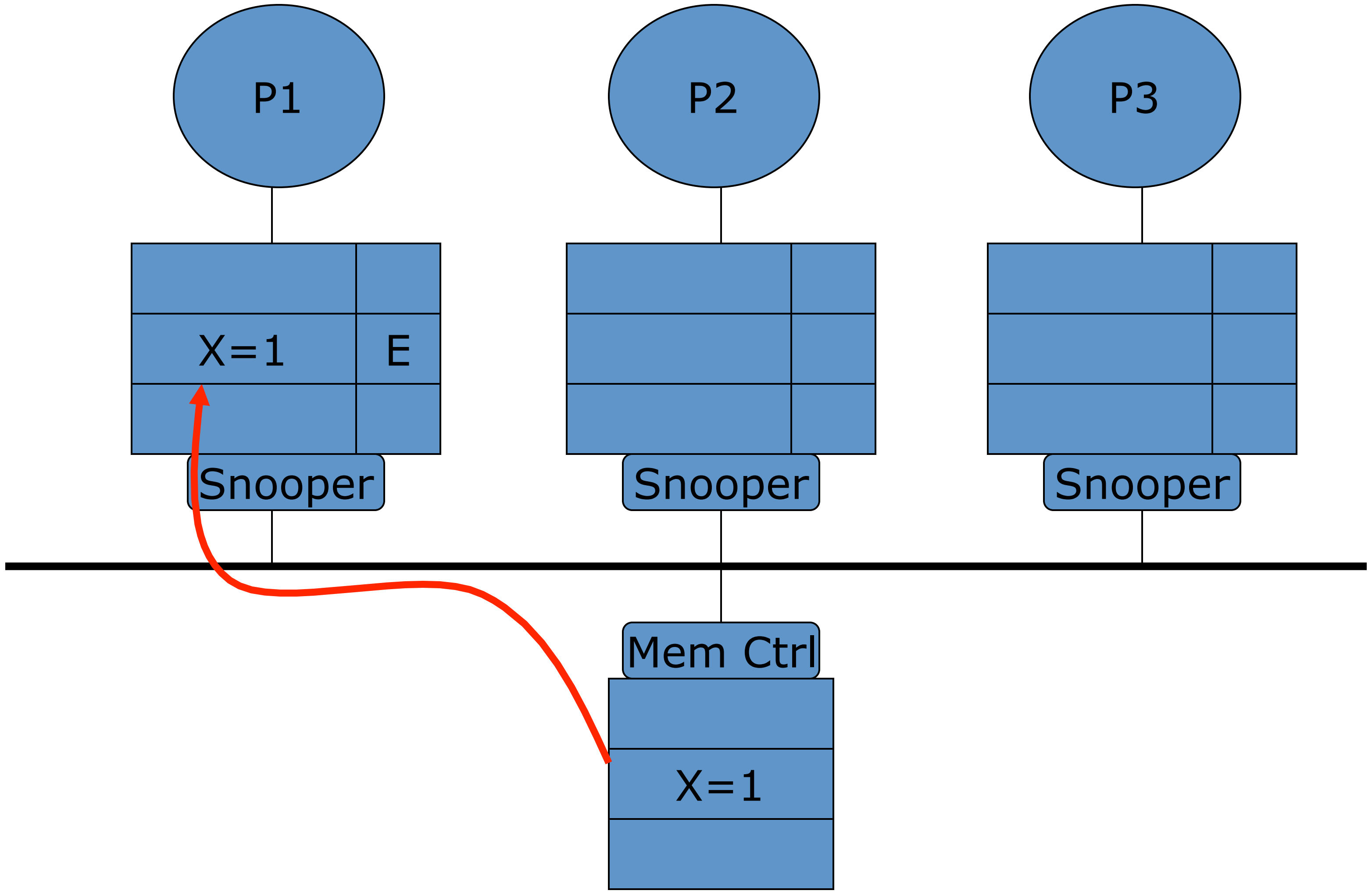
MESI Visualization



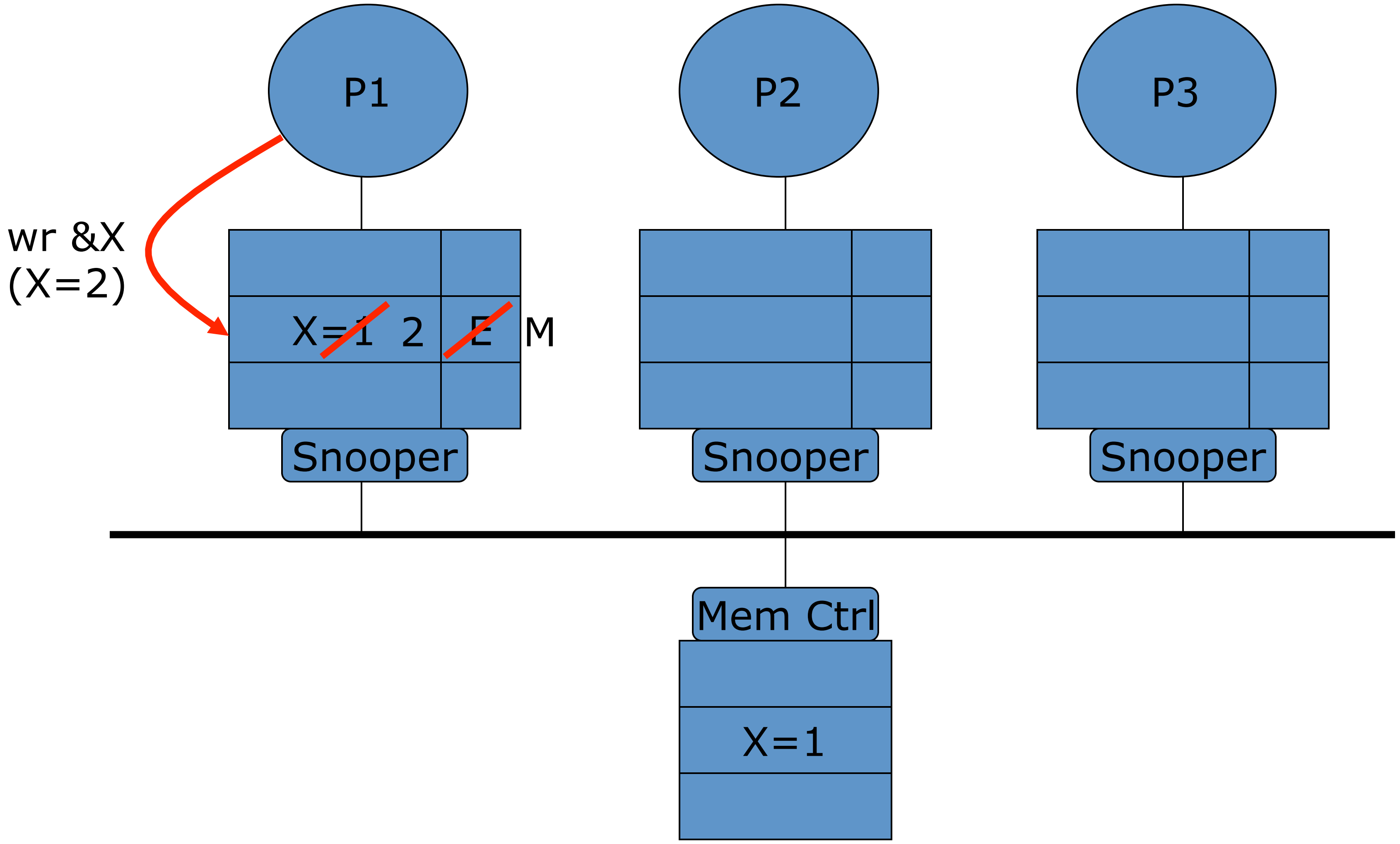
MESI Visualization



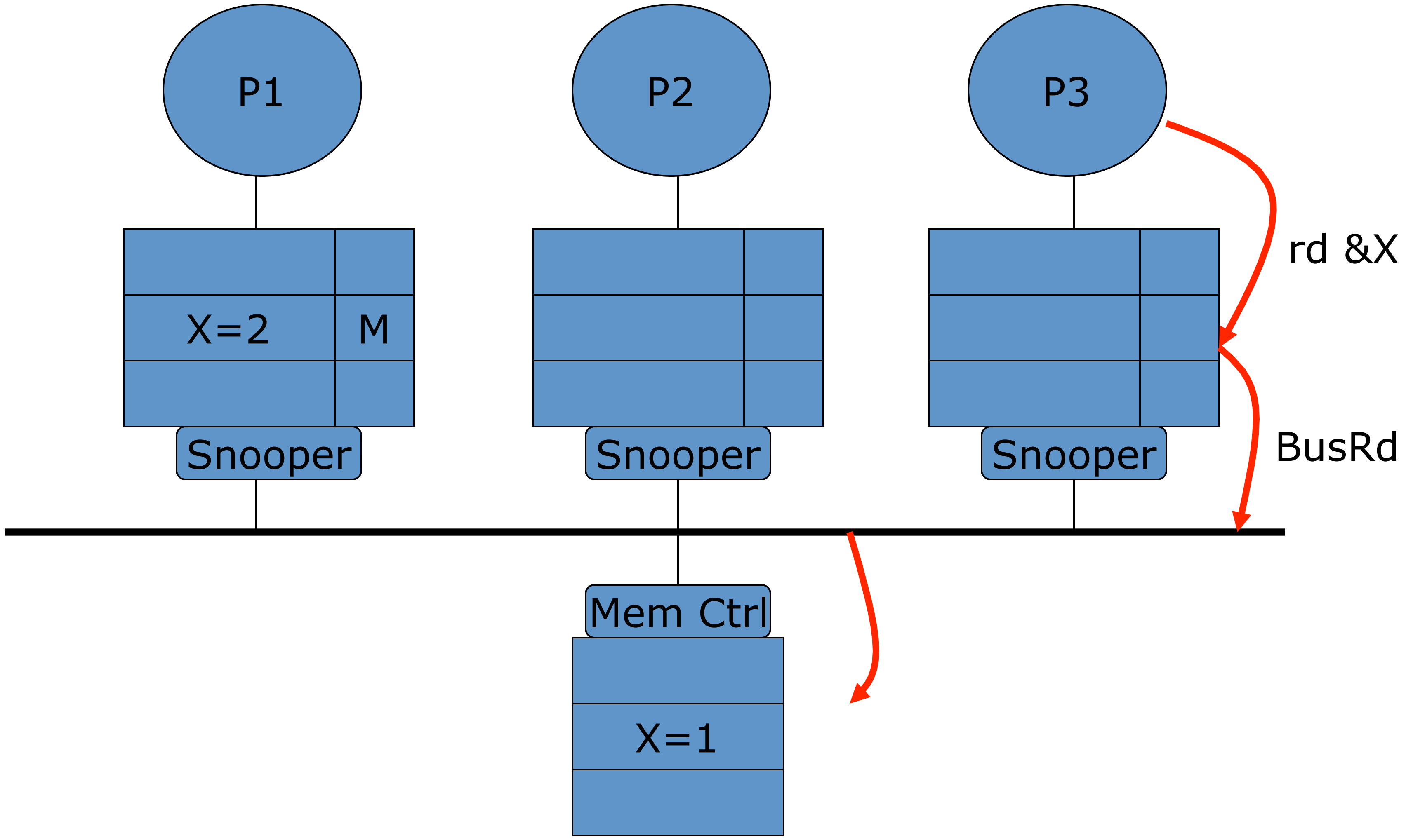
MESI Visualization



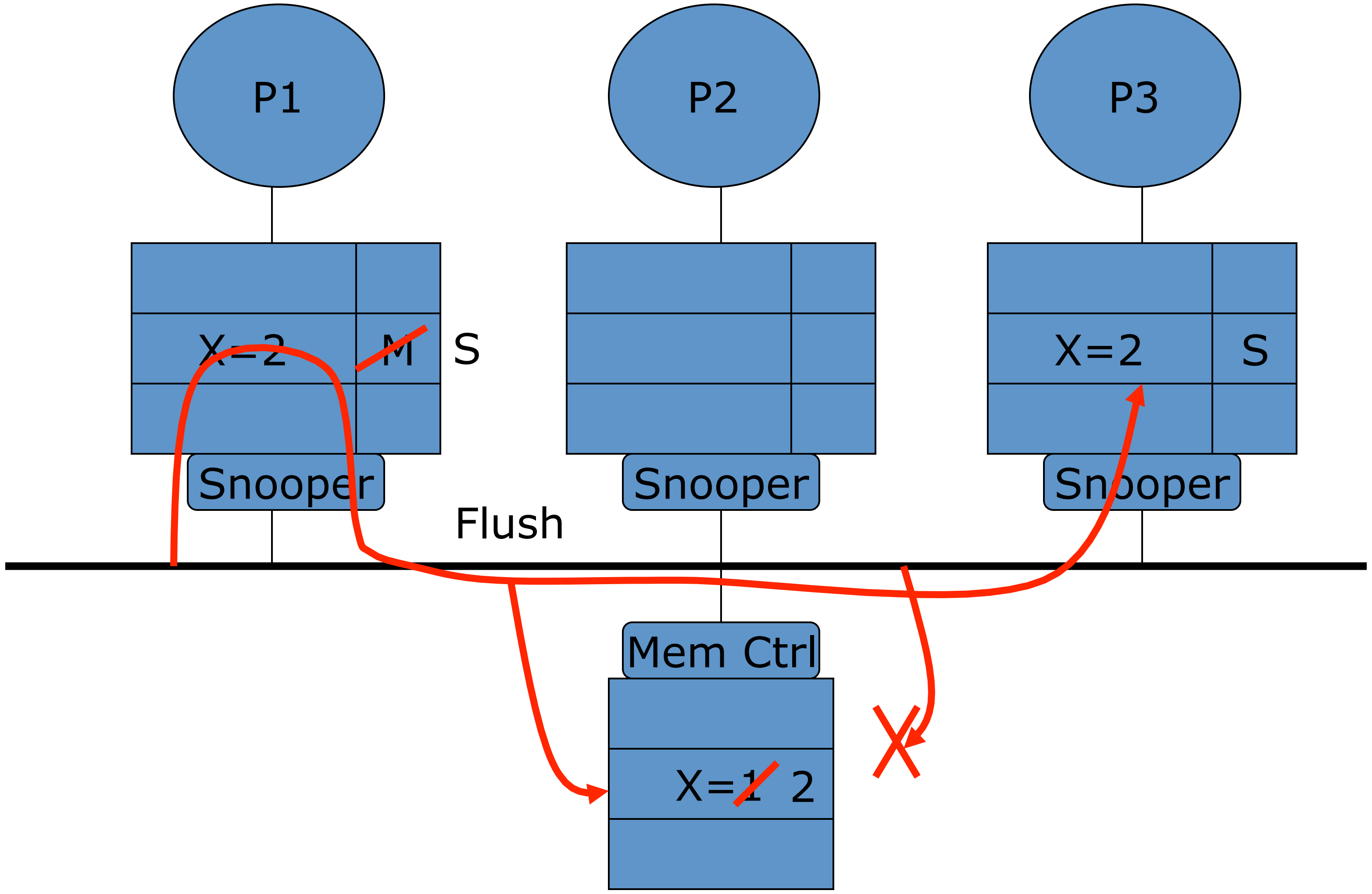
MESI Visualization



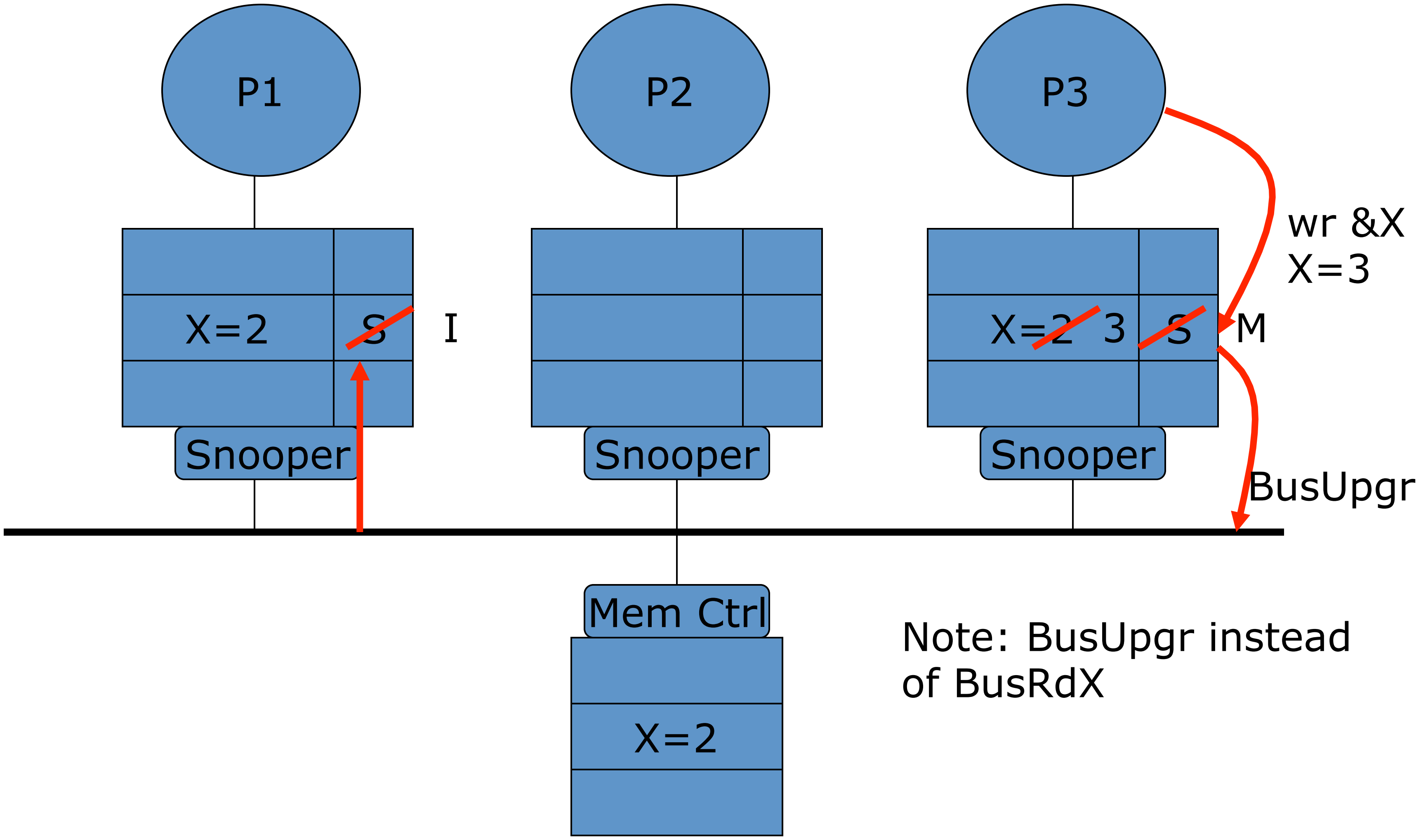
MESI Visualization



MESI Visualization

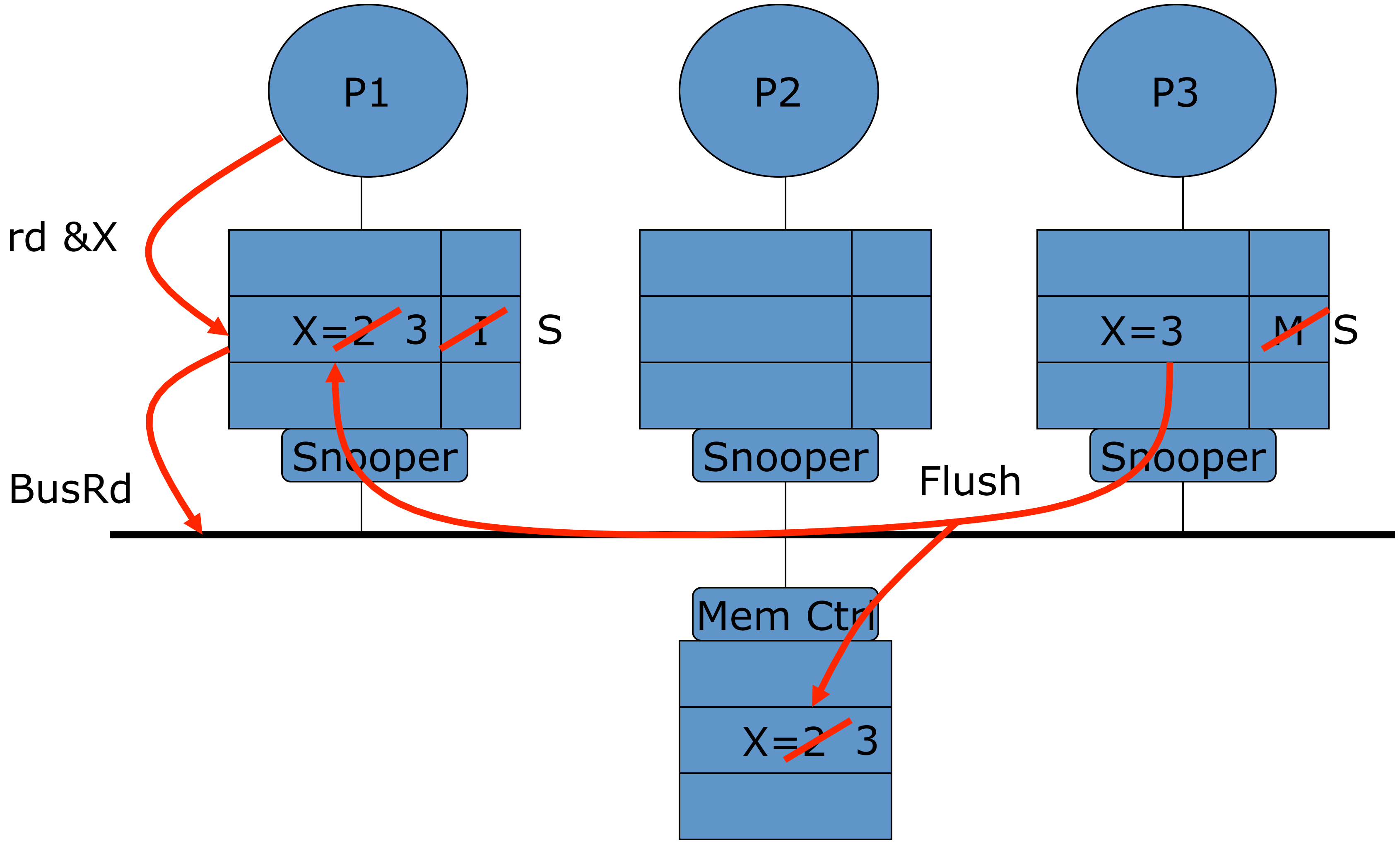


MESI Visualization

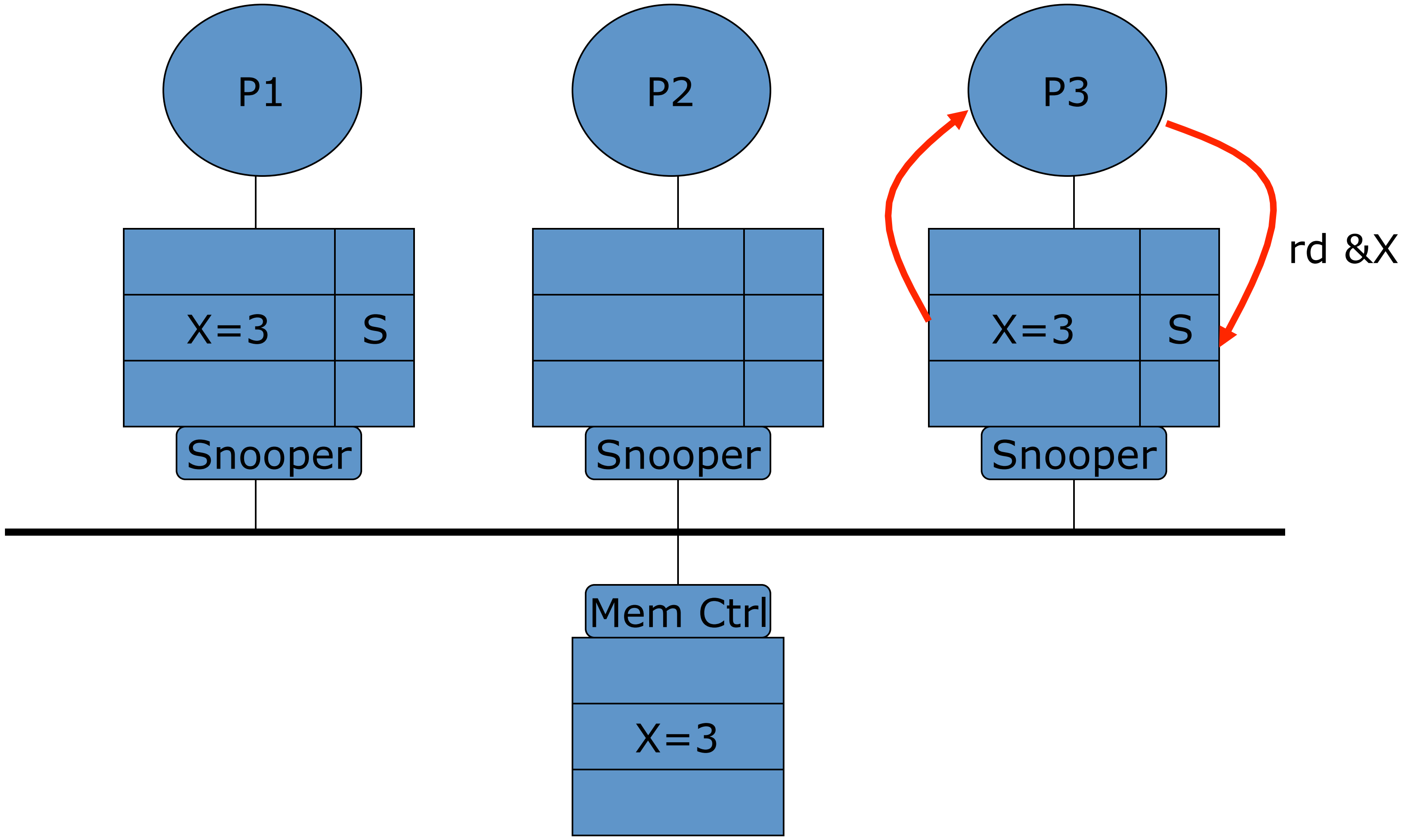


Note: BusUpgr instead of BusRdX

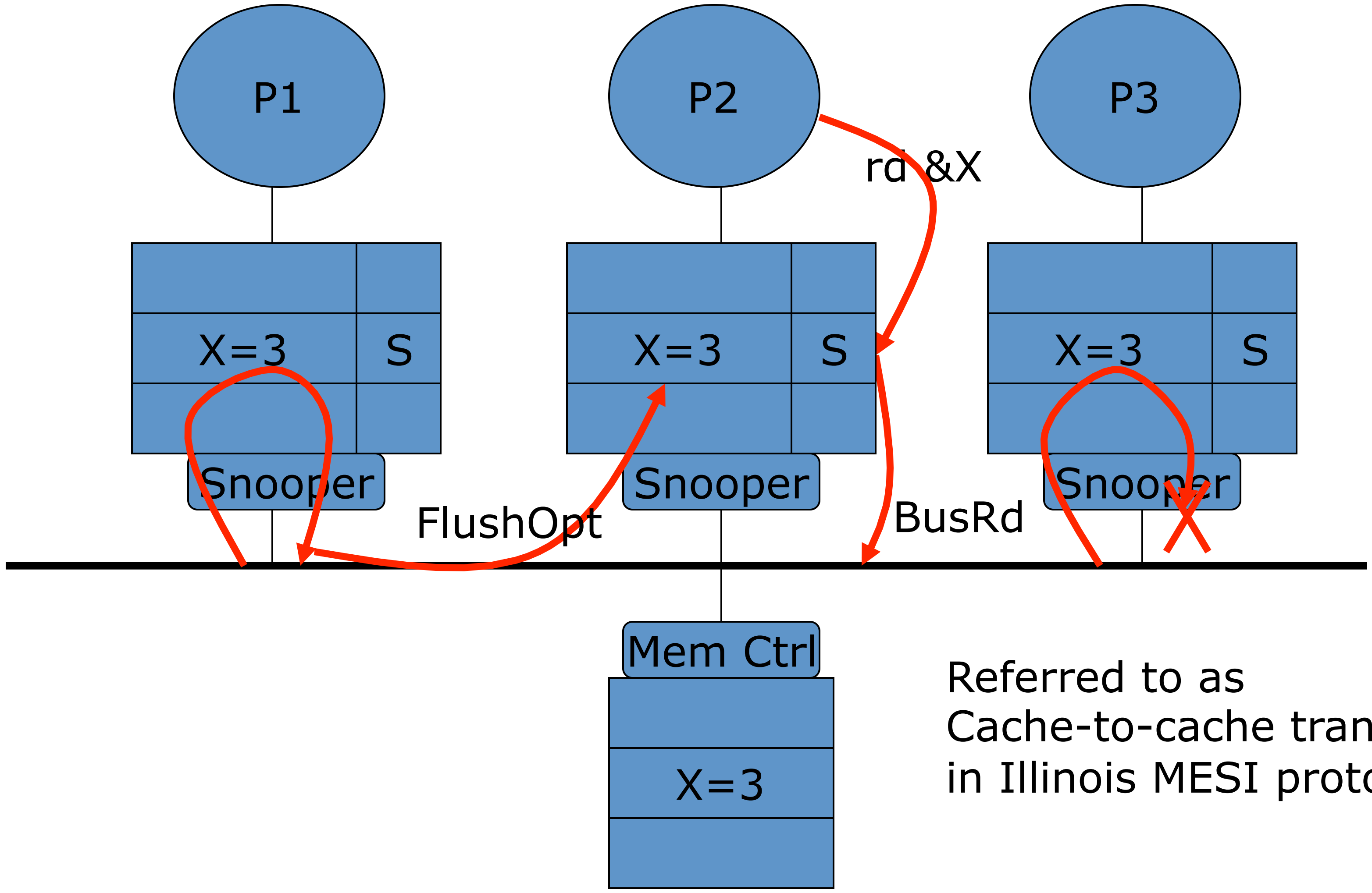
MESI Visualization



MESI Visualization



MESI Visualization



Referred to as
Cache-to-cache transfer
in Illinois MESI protocol

Illinois MESI Example

Assume:

- hit without bus transaction (1 cycle)
- hit/miss w/ bus transaction, no data or data from/to another cache (20 cycles)
- hit/miss w/ bus transaction, data from memory (40 cycles)

Proc Action	**State @P1**	**State @P2**	**State @P3**	**Bus Action**	**Data Supplier**	**Cost**
R1	**E**	-	-	BusRd	Mem	40
W1	M	-	-	-	-	1
R3	S	-	S	BusRd/Flush	P1's cache	20
W3	I	-	M	BusUpgr	-	20
R1	S	-	S	BusRd/Flush	P3's cache	20
R3	S	-	S	-	-	1
R2	S	S	S	BusRd/ FlushOpt	**P1/P3's cache**	20

Lower-level Protocol Choices

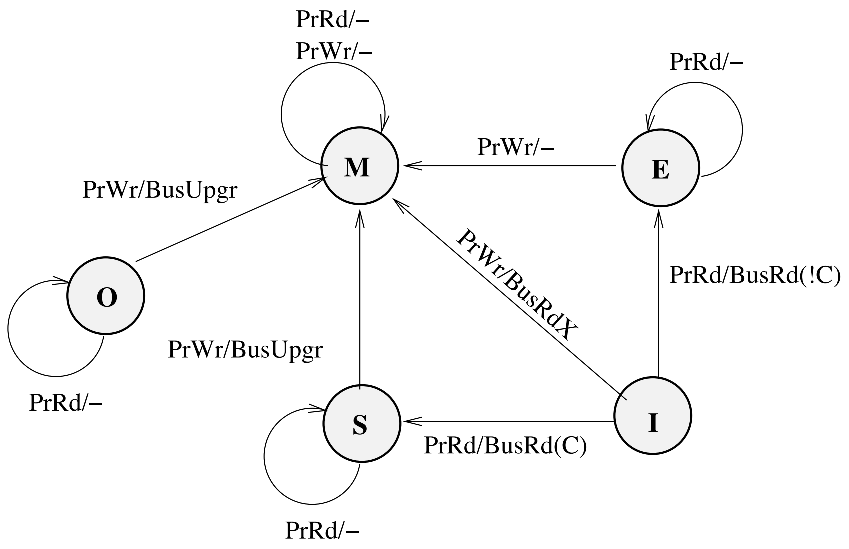
- Who supplies data on miss when not in M state: memory or cache?
- Original, *Illinois* MESI: cache
  - assume cache faster than memory (*FlushOpt)*
  - Not necessarily true
- Adds complexity
  - How does memory know it should supply data (must wait for caches)
  - Selection algorithm if multiple caches have valid data
- Valuable for distributed memory
  - May be cheaper to obtain from nearby cache than distant memory
  - Especially when constructed out of SMP nodes (Stanford DASH)

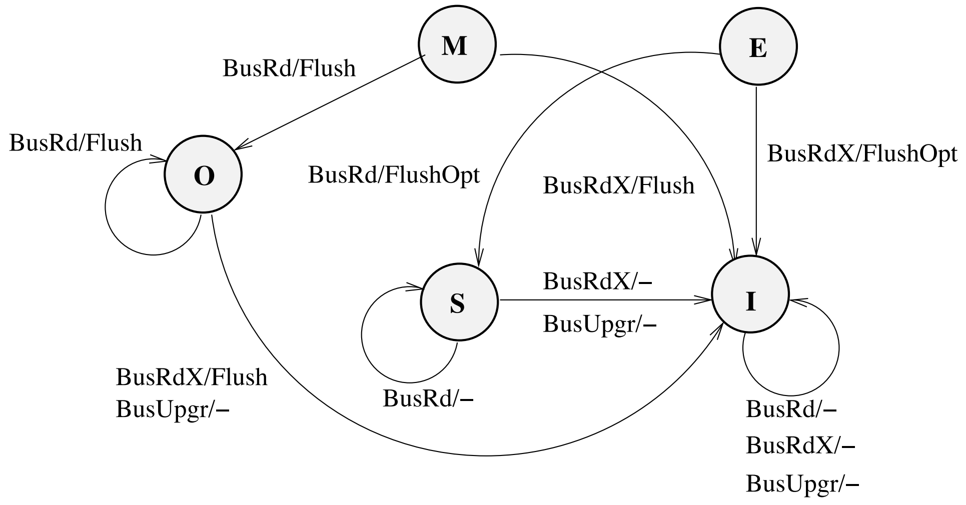
MOESI Protocol

- Problem with MESI
  - No dirty sharing: shared block must be clean
  - Too many write backs requiring high memory traffic
- Solution: allow dirty sharing
  - Dirty block can be shared
  - But, must remember to eventually write it back
  - Who should write it back? Choose one "owner", hence the "O" in MOESI

- Used in AMD processors, starting from K7 (Athlon)

MOESI Protocol

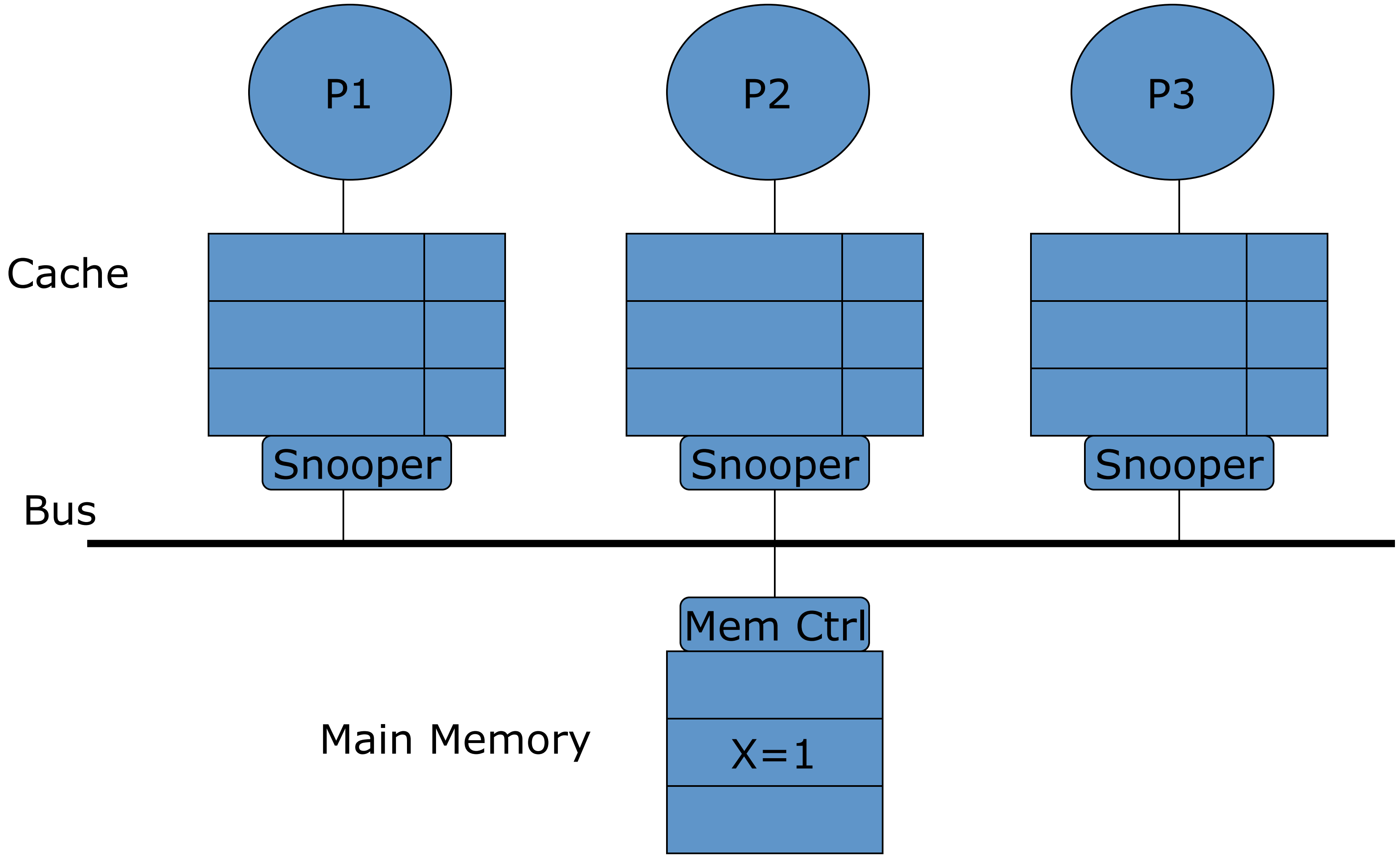




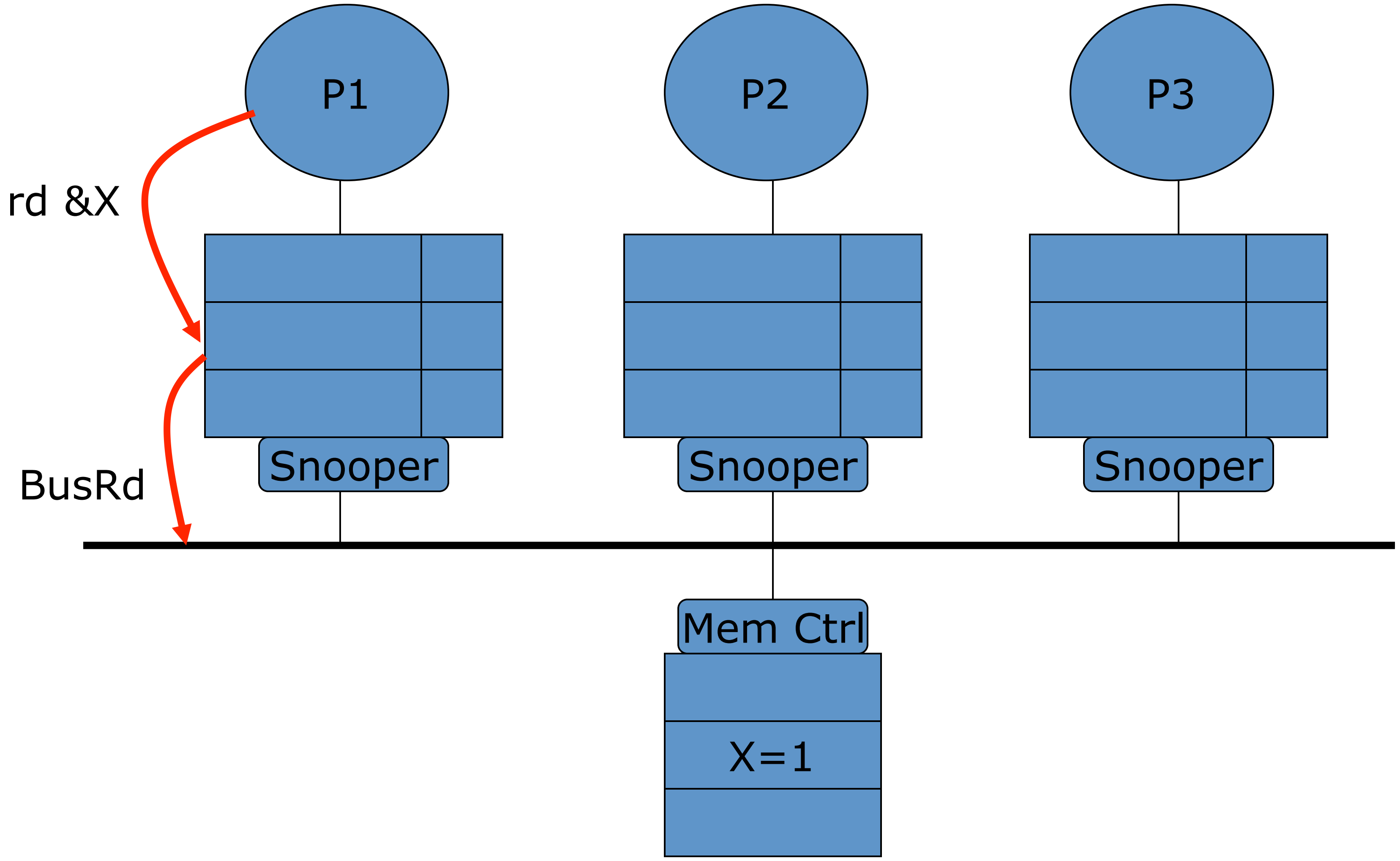
- The key is that when a dirty block is shared, one of them acts as the owner of the block
- Other caches can evict the block silently
  – Owner must write back to memory
- Can use for cache-to-cache sharing as well
  – When a read/write is snooped, other caches do not react
  – But owner supplies the block

MOESI Animation

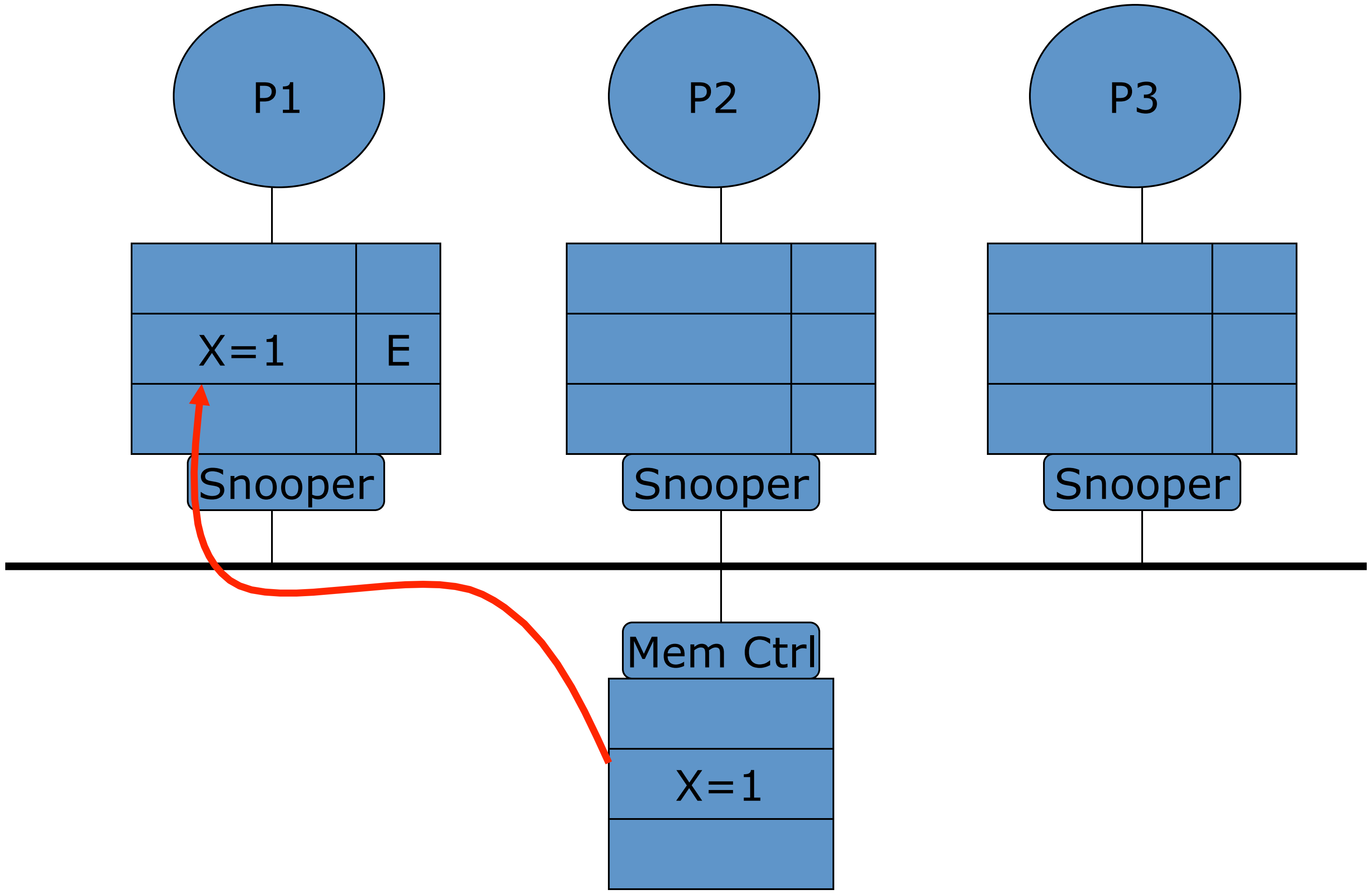
MOESI Protocol (used by AMD)



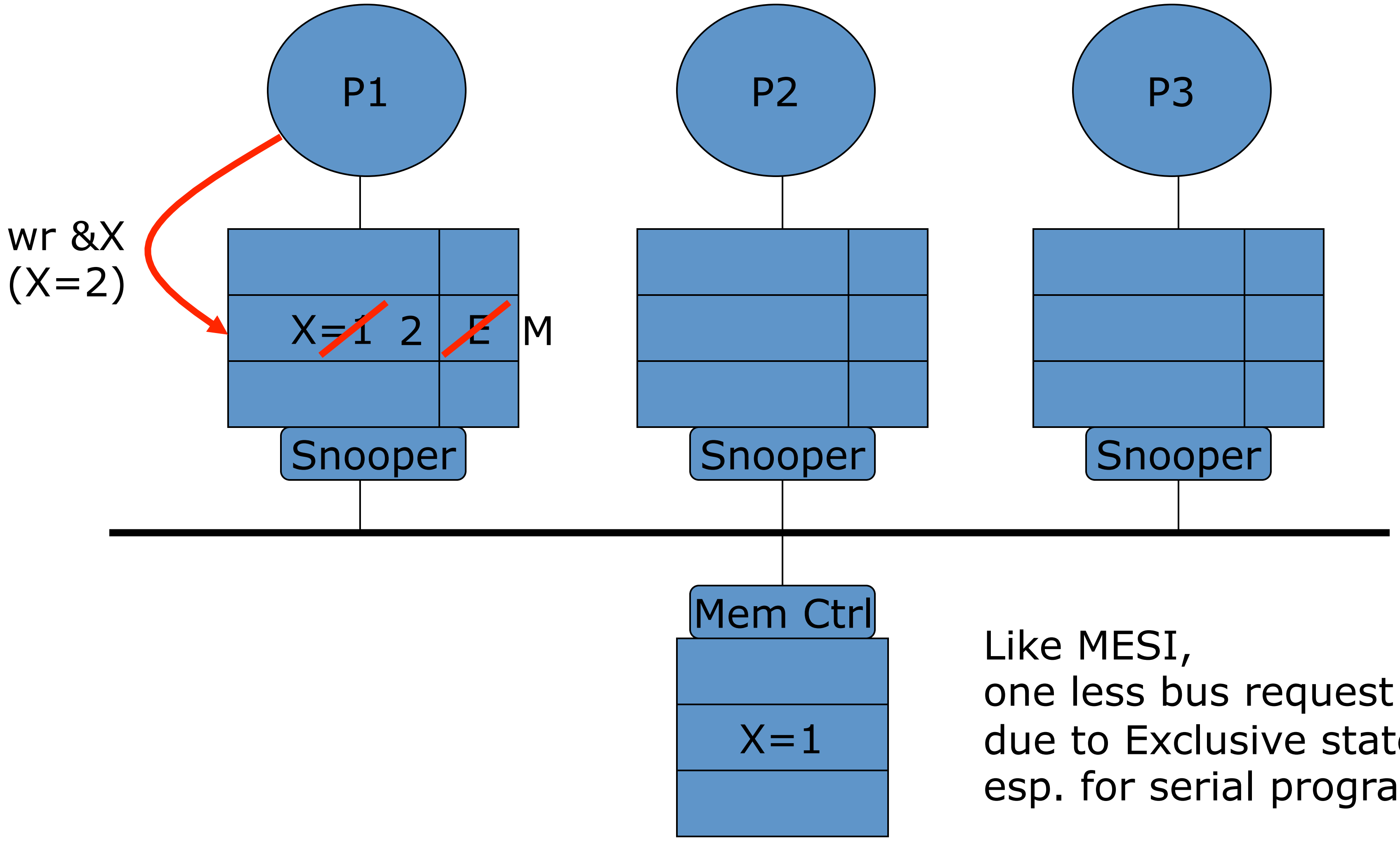
MOESI Visualization



MOESI Visualization

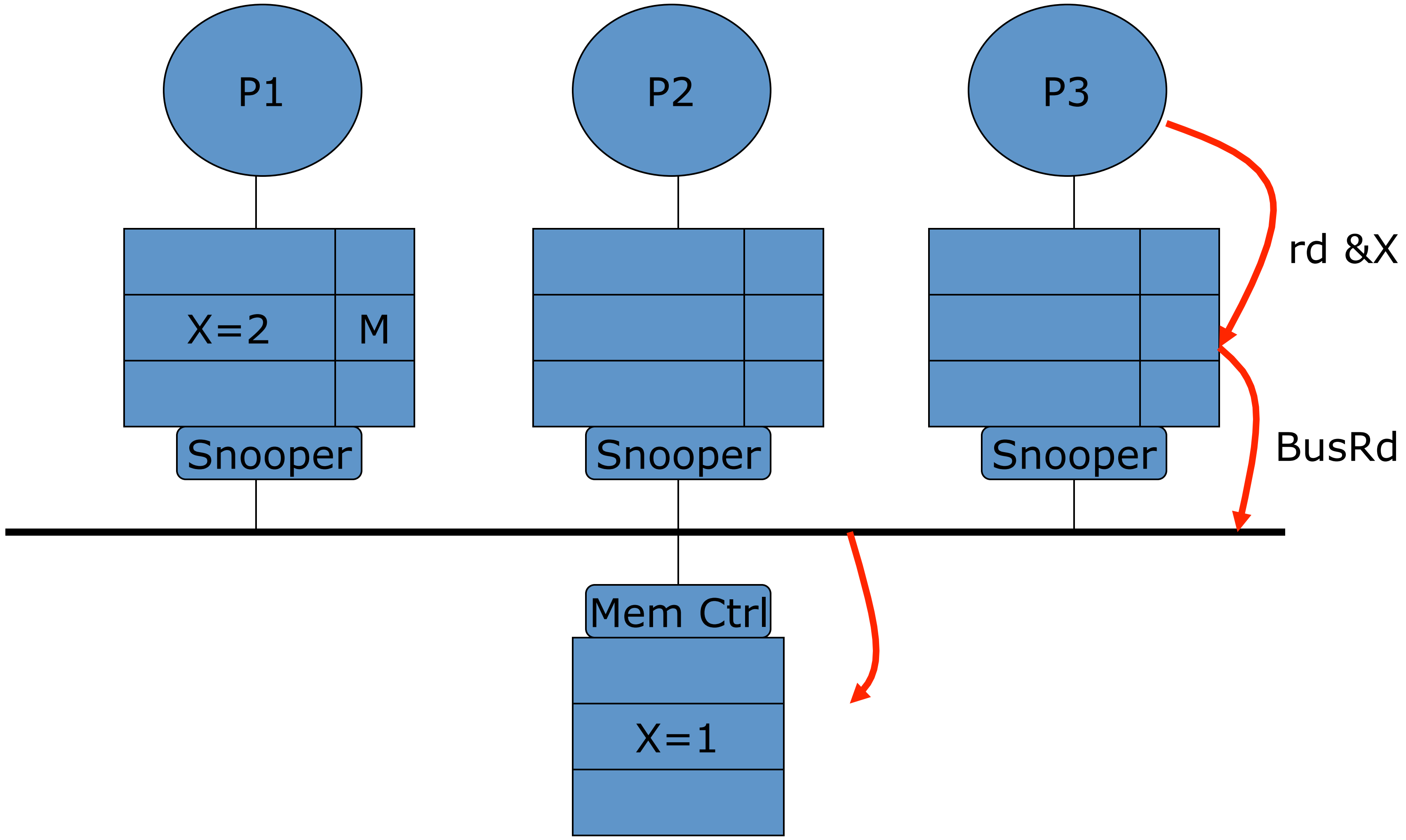


MOESI Visualization

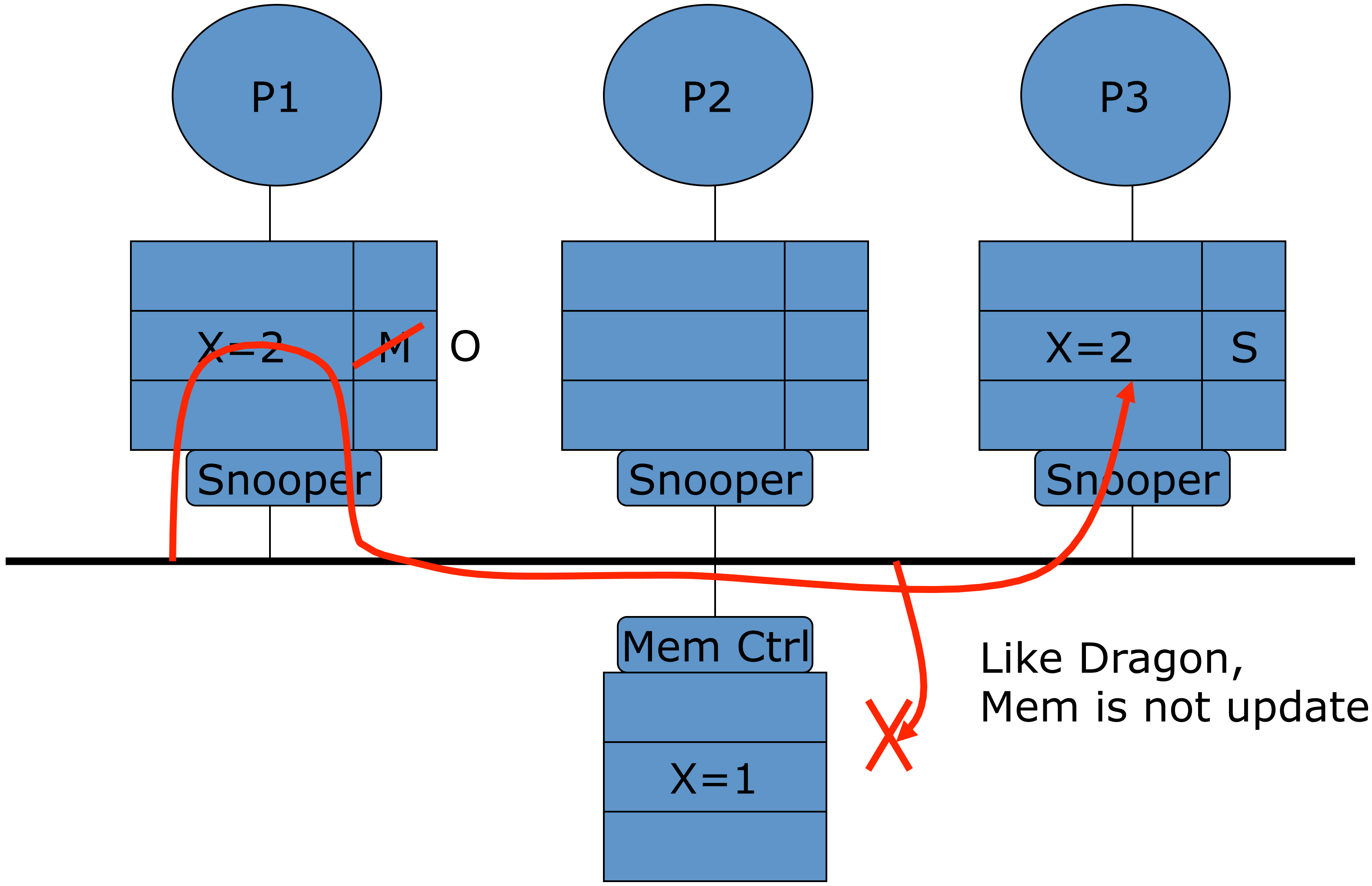


Like MESI,
one less bus request
due to Exclusive state,
esp. for serial programs

MOESI Visualization

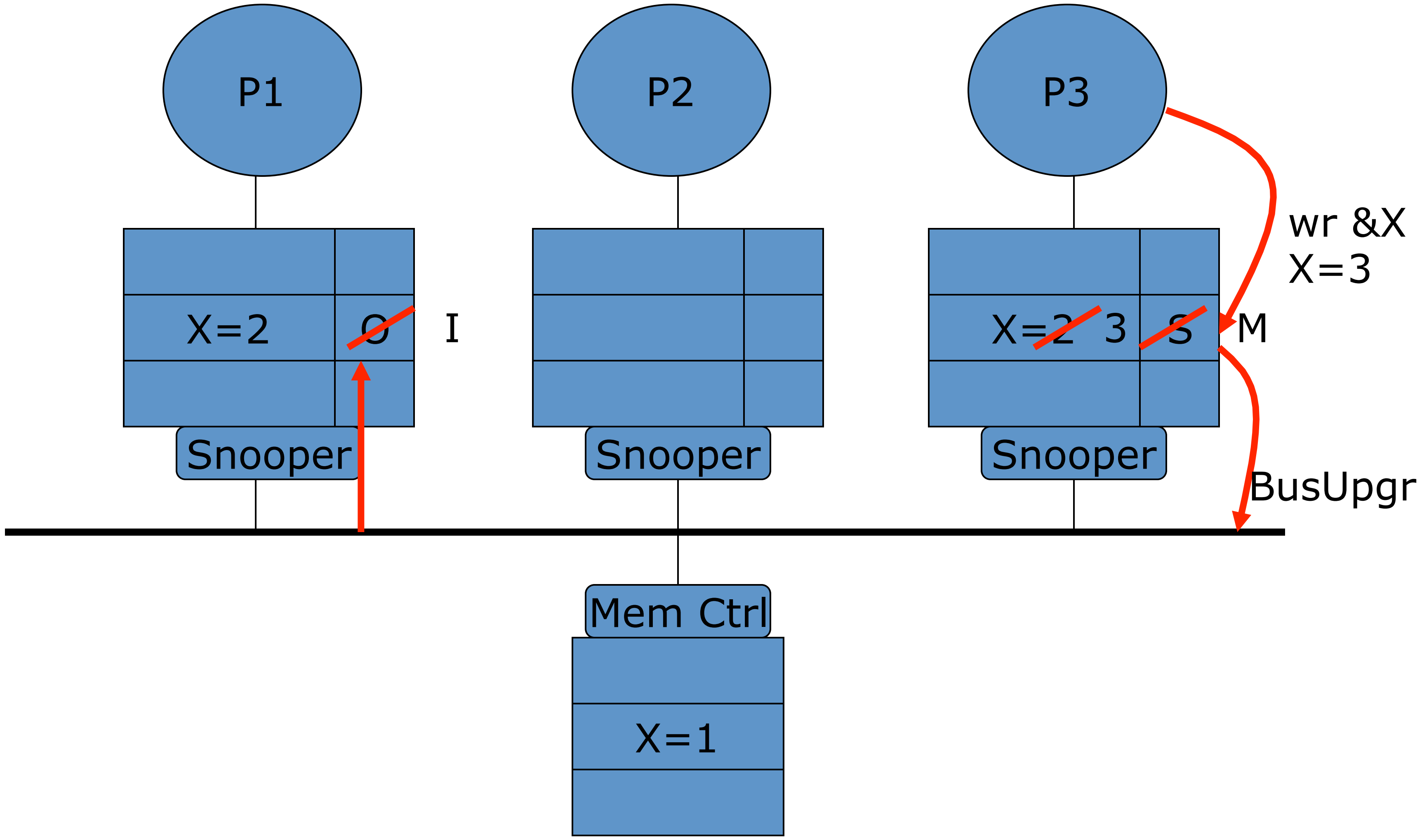


MOESI Visualization

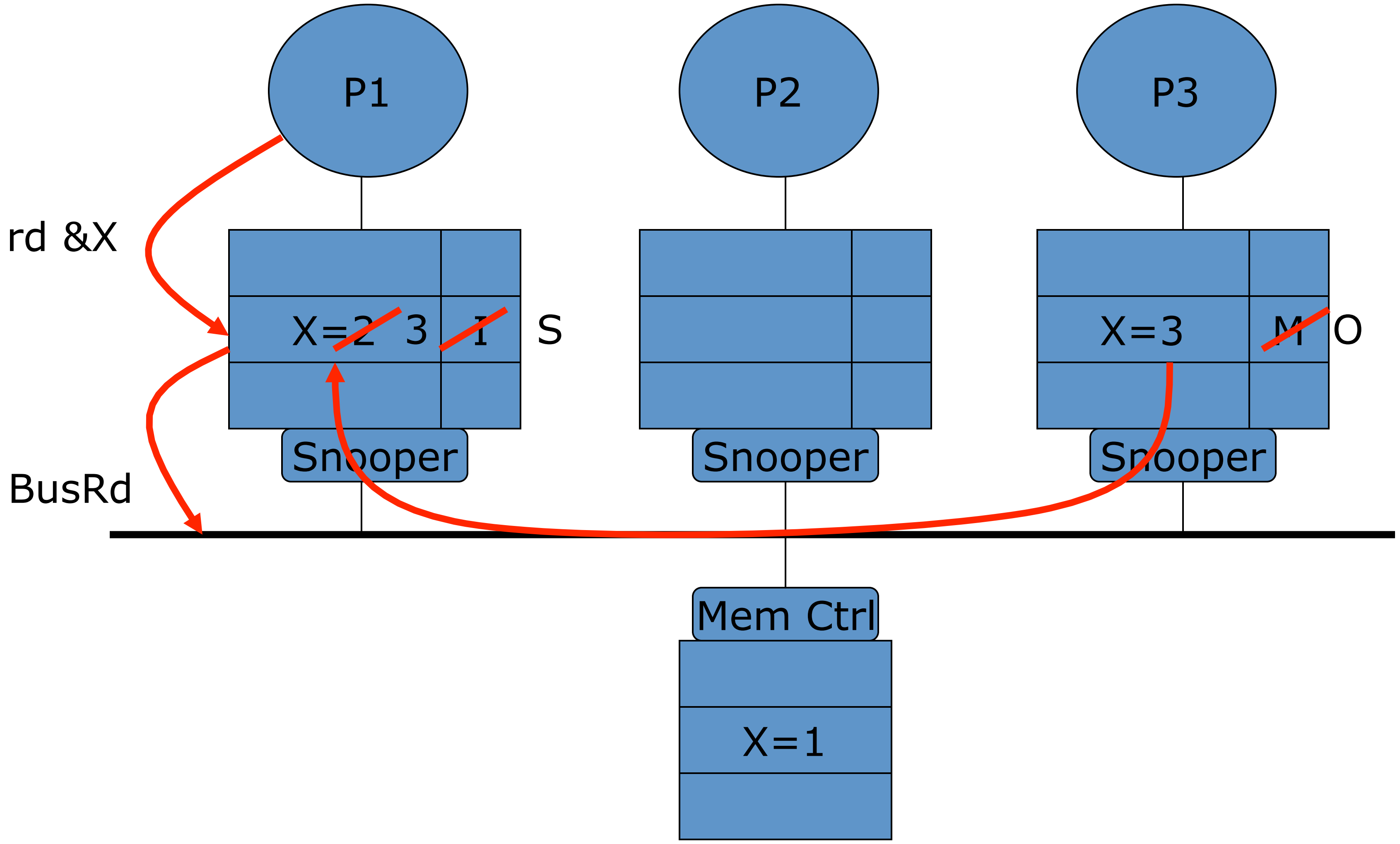


Like Dragon,
Mem is not updated

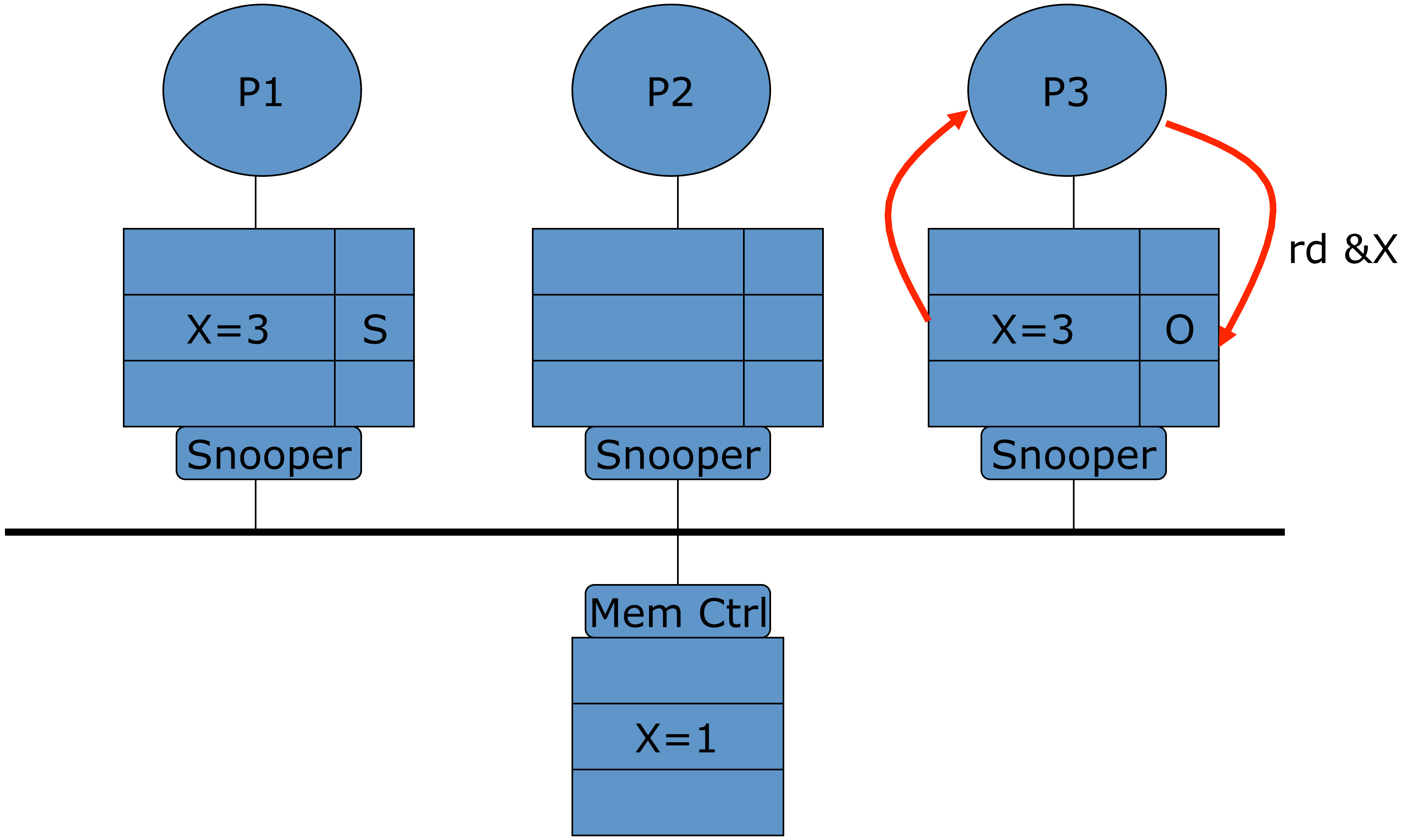
MOESI Visualization



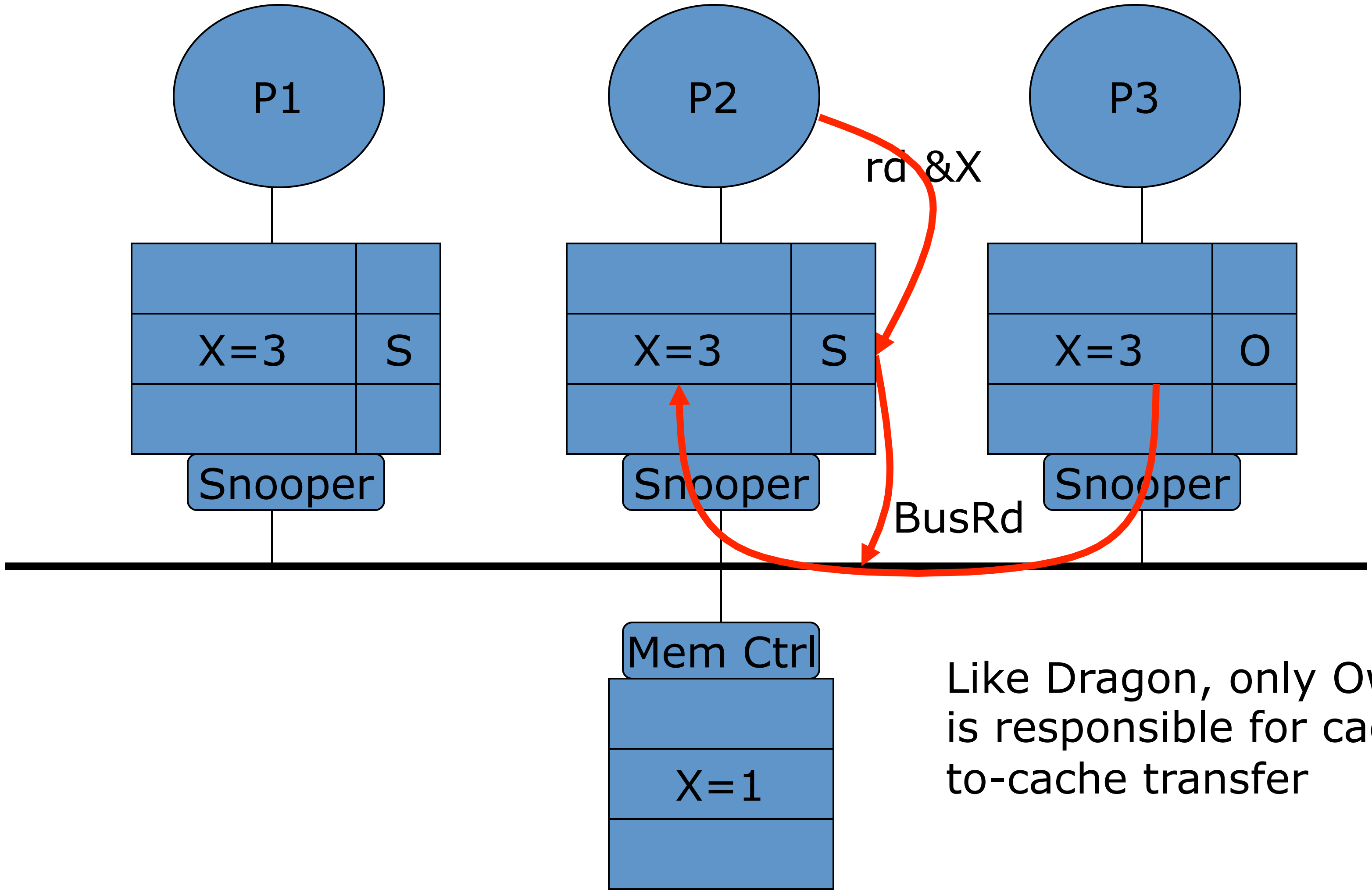
MOESI Visualization



MOESI Visualization

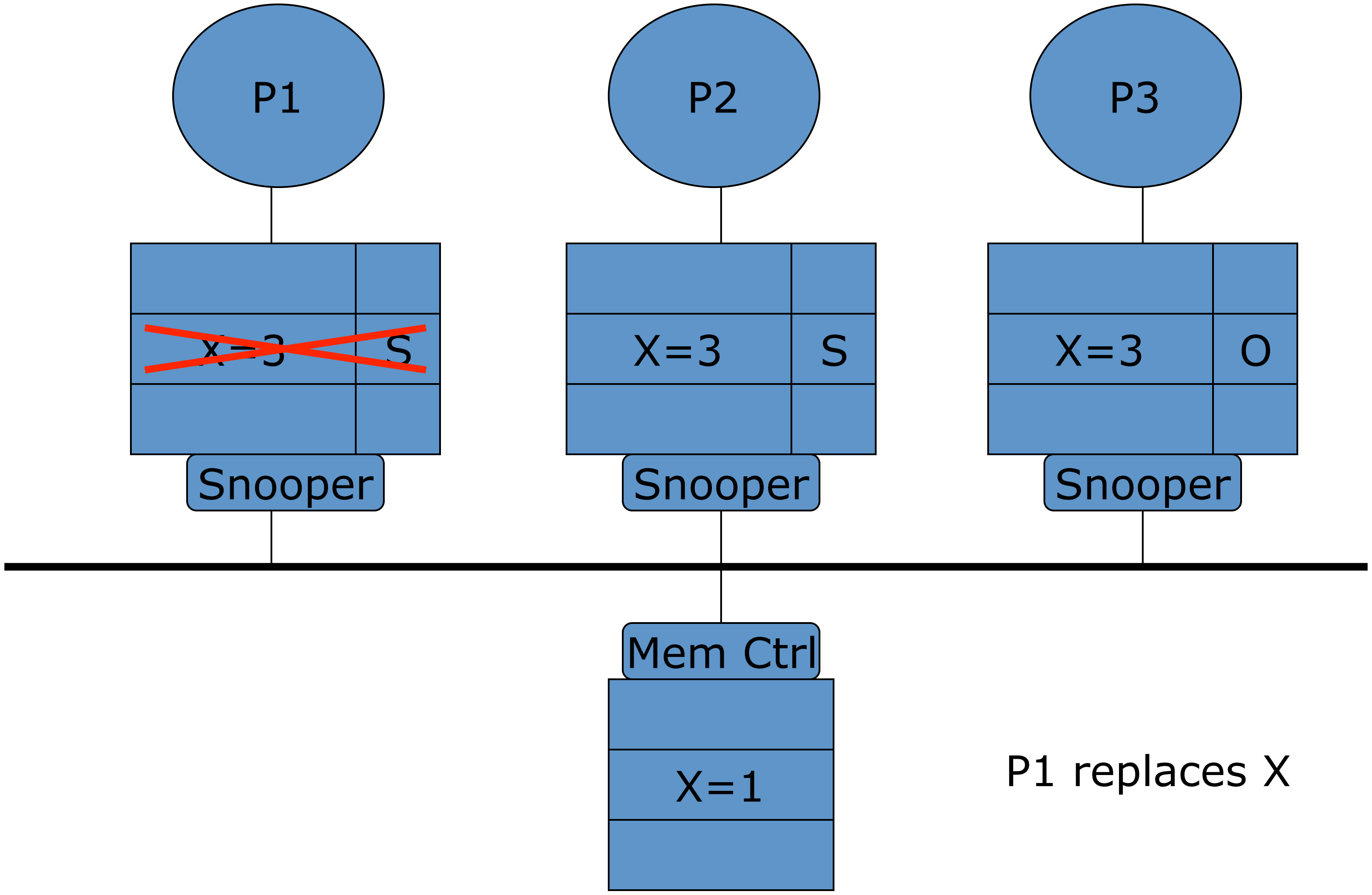


MOESI Visualization



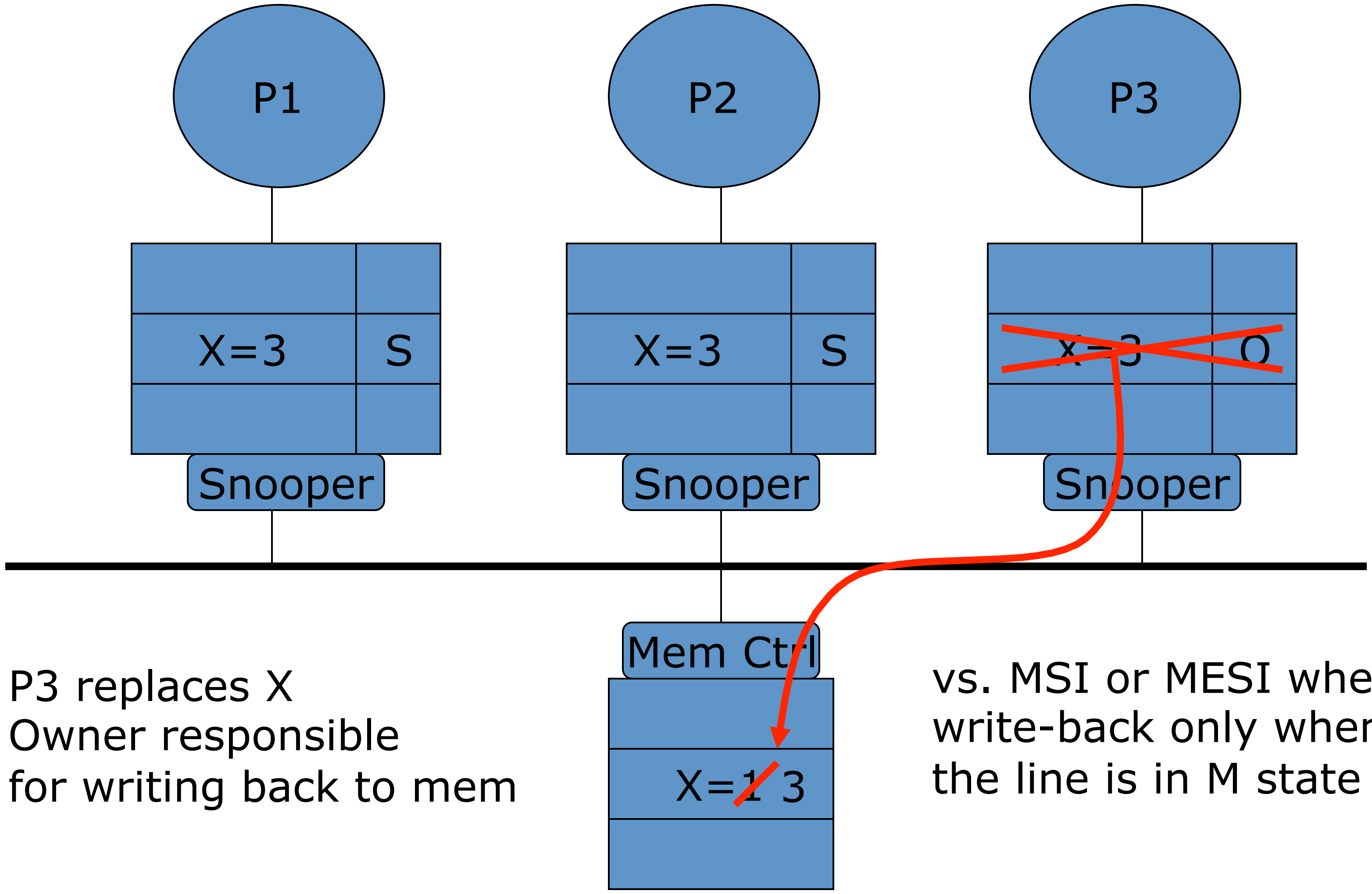
Like Dragon, only Owner is responsible for cache-to-cache transfer

MOESI Visualization



P1 replaces X

MOESI Visualization



P3 replaces X
Owner responsible
for writing back to mem

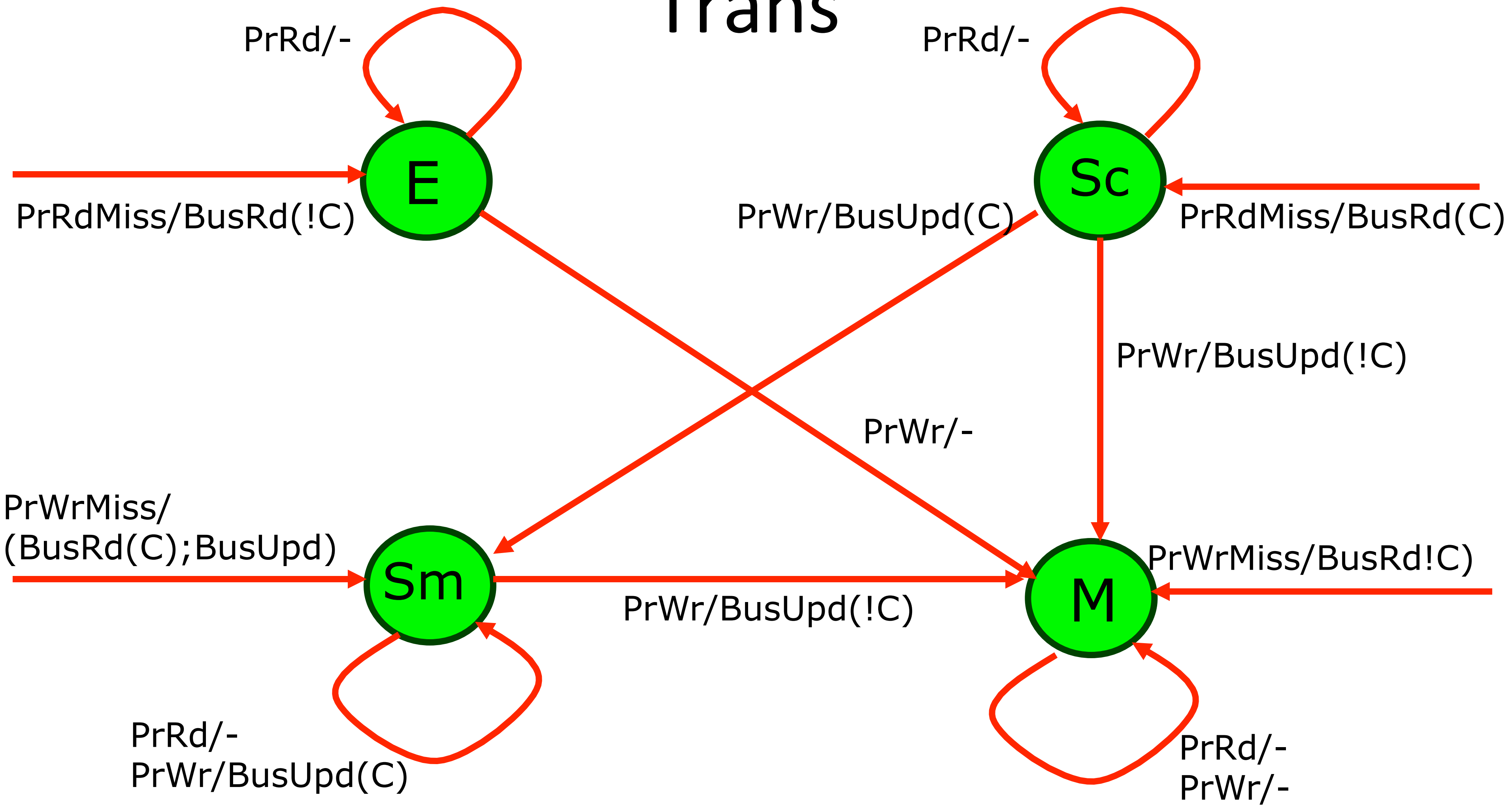
vs. MSI or MESI where
write-back only when
the line is in M state

Module 8.4 – Dragon Write Update Protocol and Coherence Protocol Performance

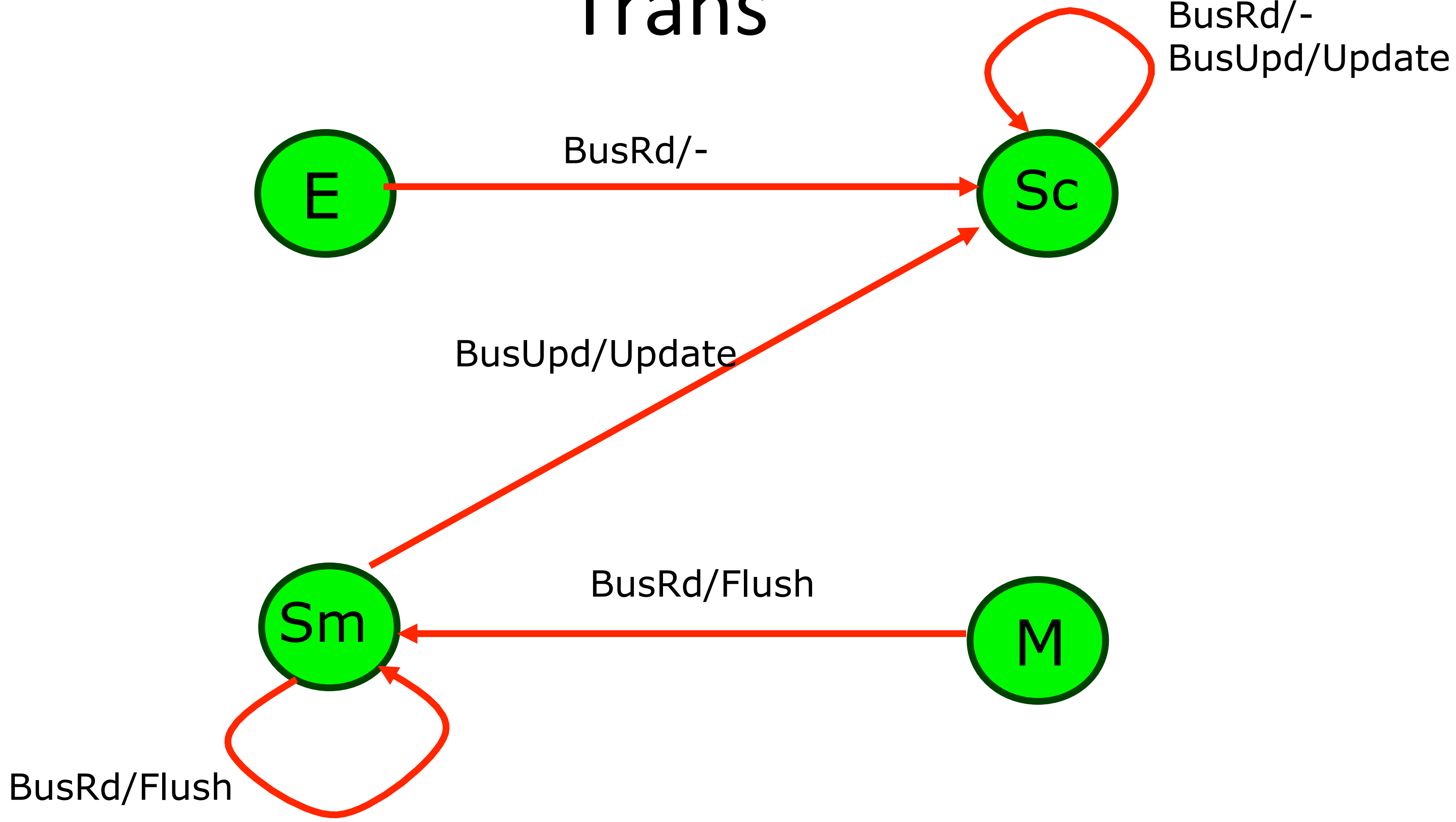
Dragon Update Protocol for WB Caches

- 4 states
  - **Exclusive-clean** (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 at most one Sm
  - Modified or dirty (M): I and, no one else
  - On replacement: Sc can silently drop, Sm has to flush
- 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 processor events: PrRdMiss, PrWrMiss
  - Introduced to specify actions when block not present in cache
- New bus transaction: BusUpd
  - Broadcasts single word written on bus; updates other relevant caches

State Transition Diagram – Proc Init Trans

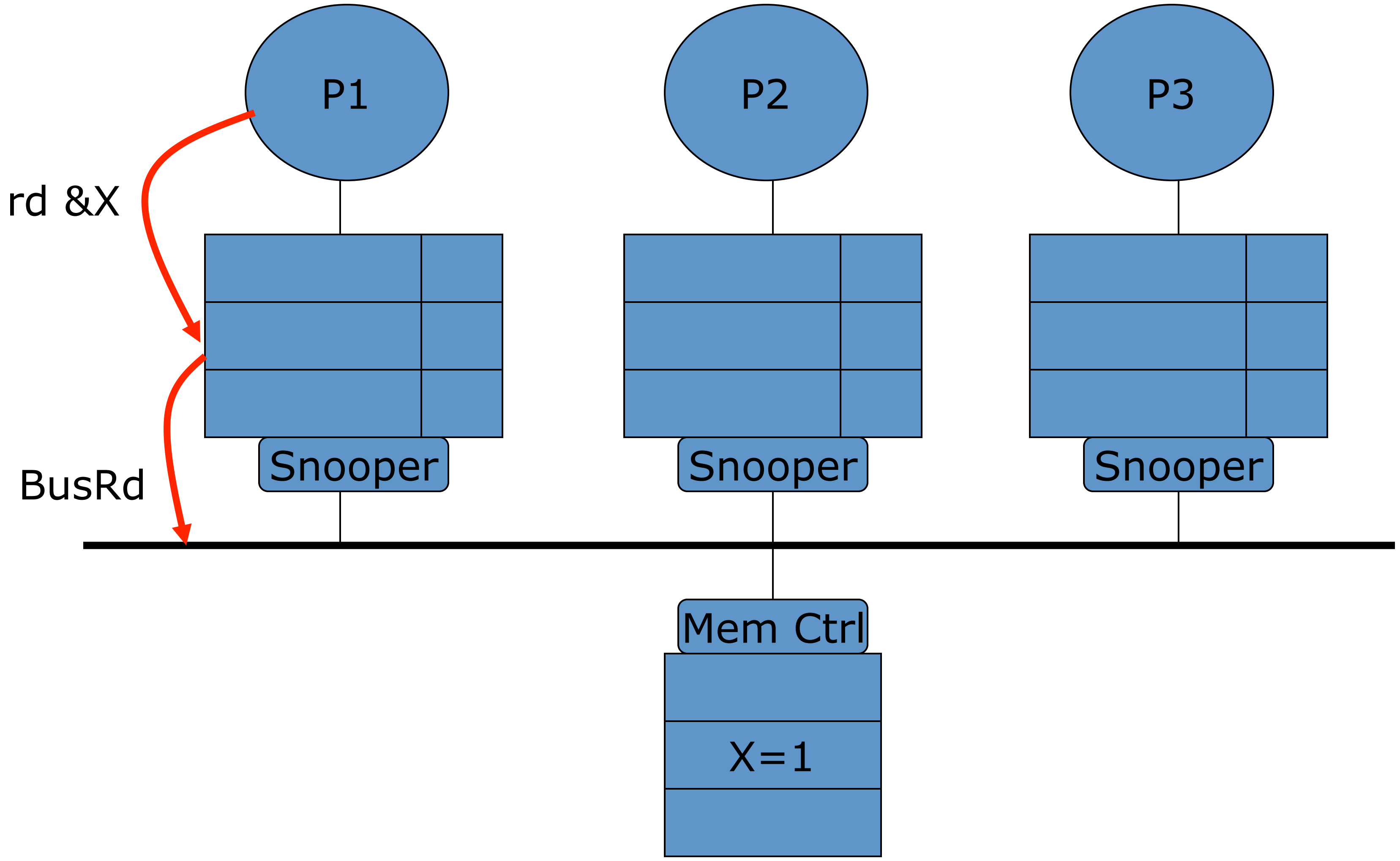


State Transition Diagram – Bus Init Trans

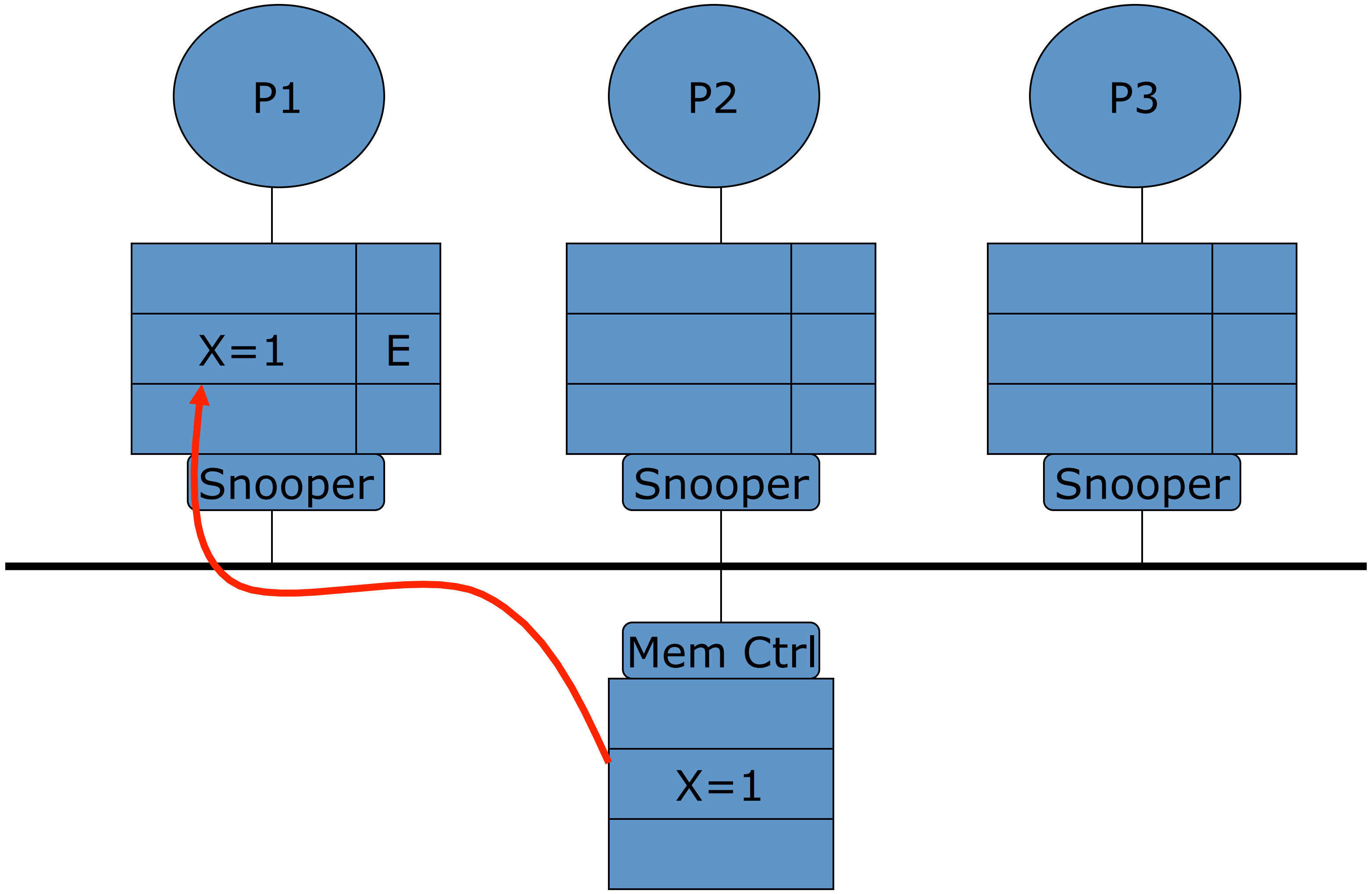


Dragon Animation

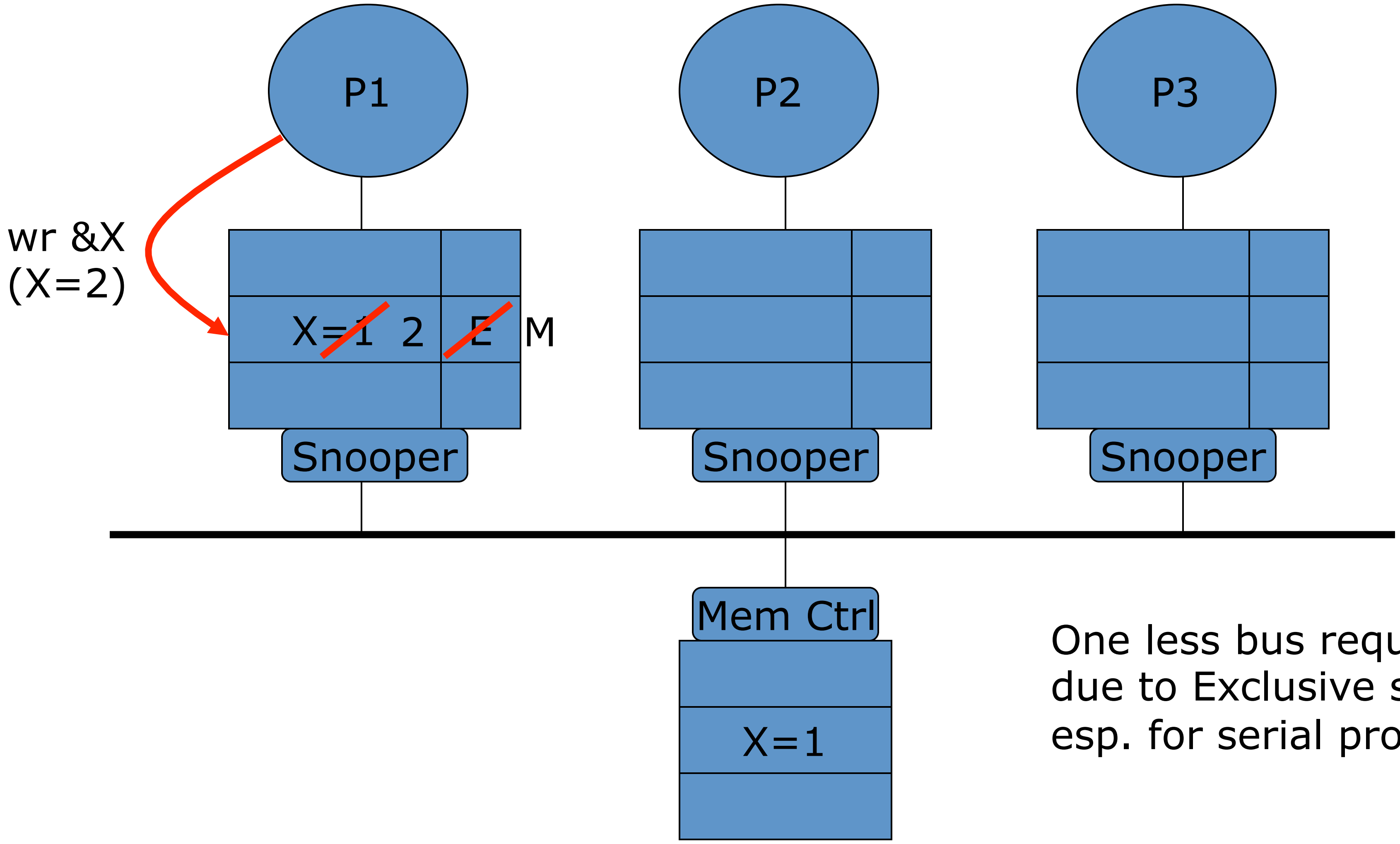
Dragon Visualization



Dragon Visualization

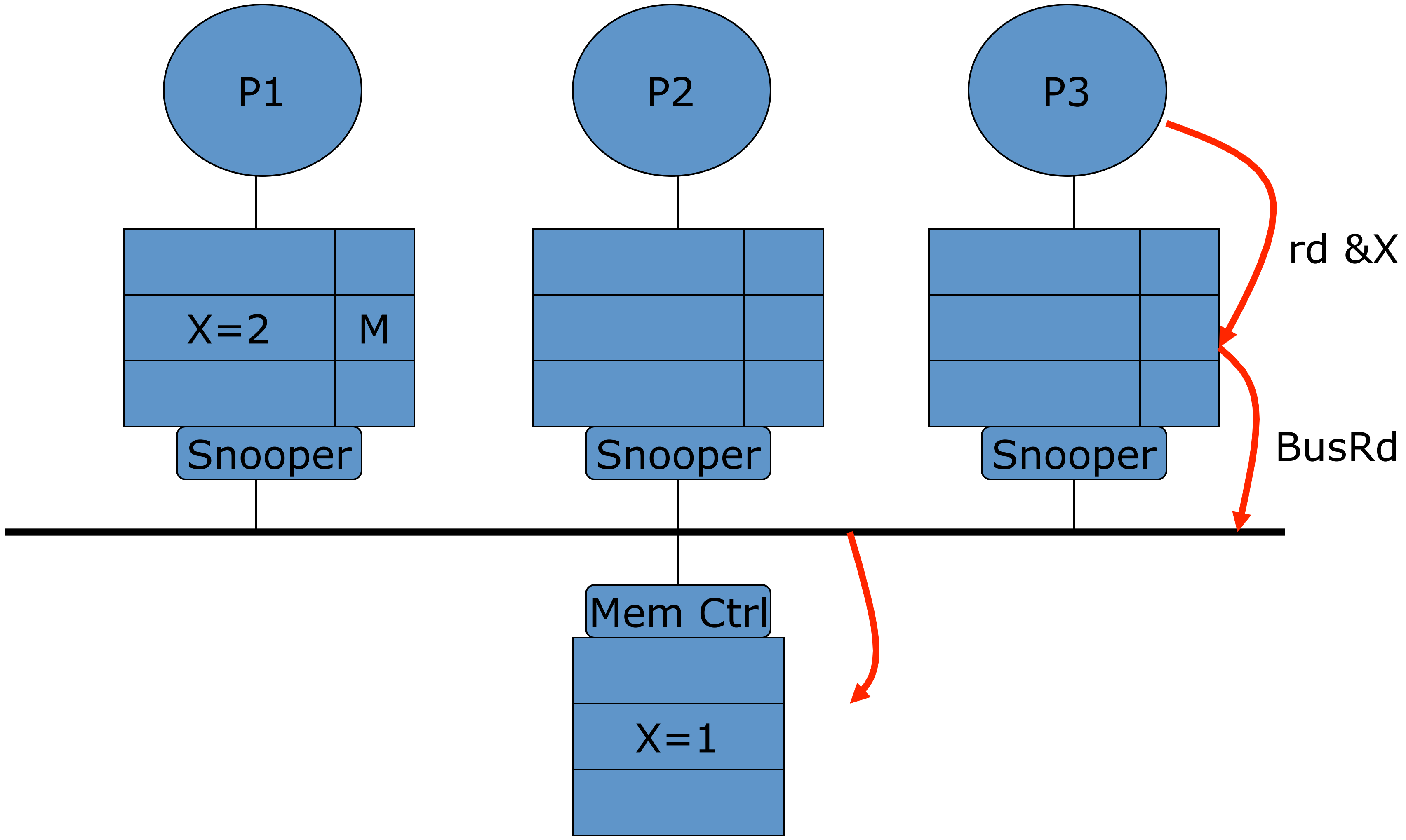


Dragon Visualization

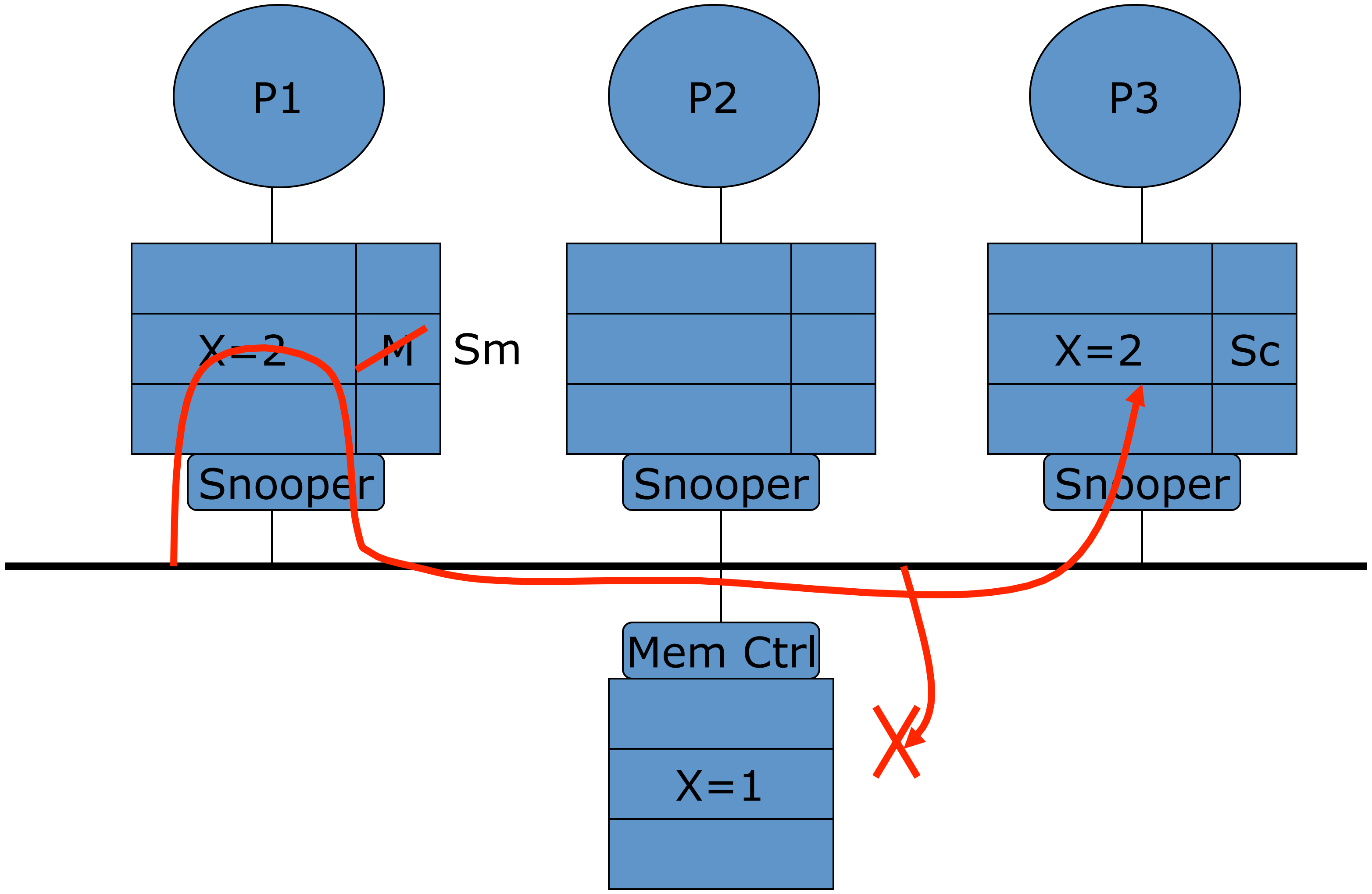


One less bus request due to Exclusive state, esp. for serial programs

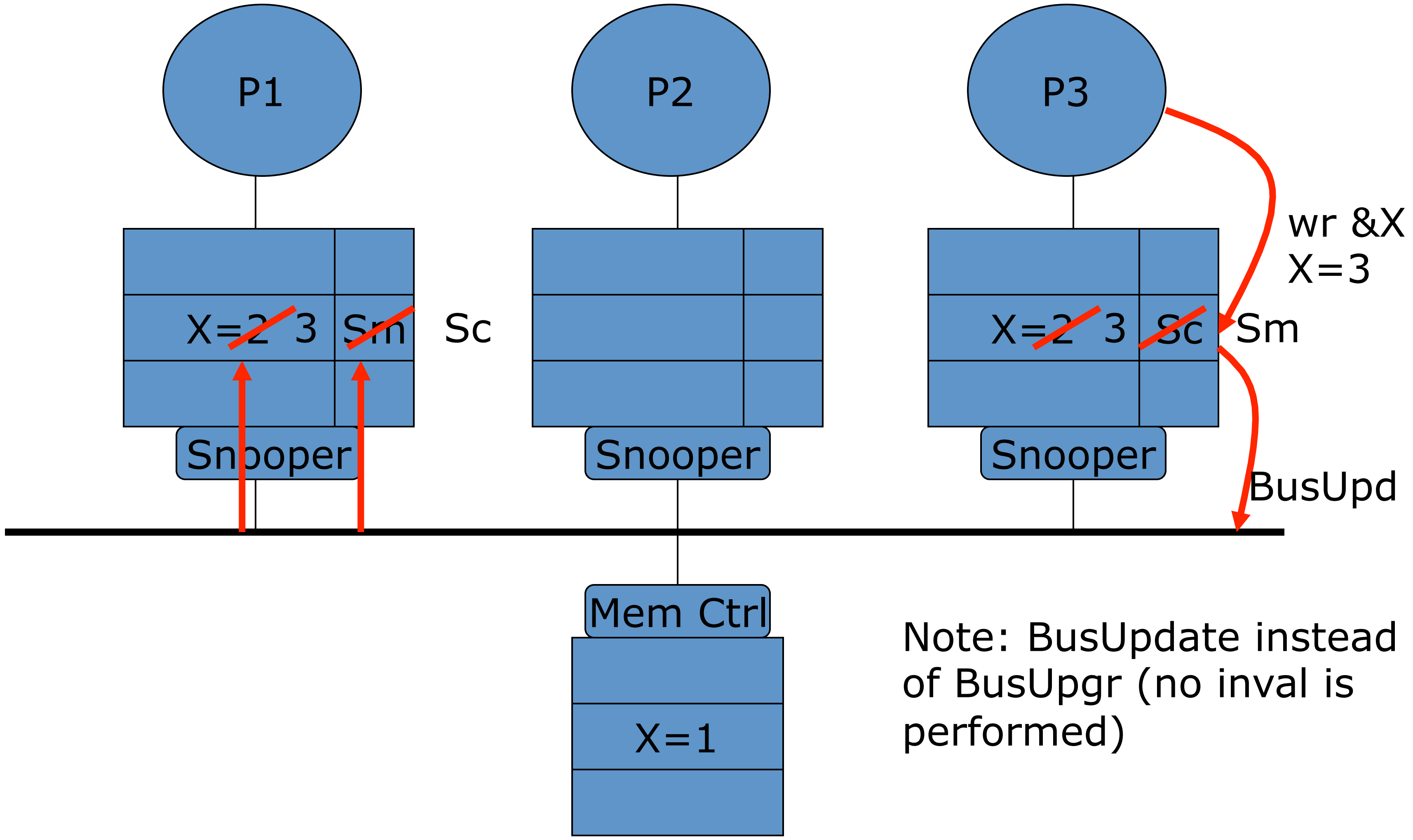
Dragon Visualization



Dragon Visualization

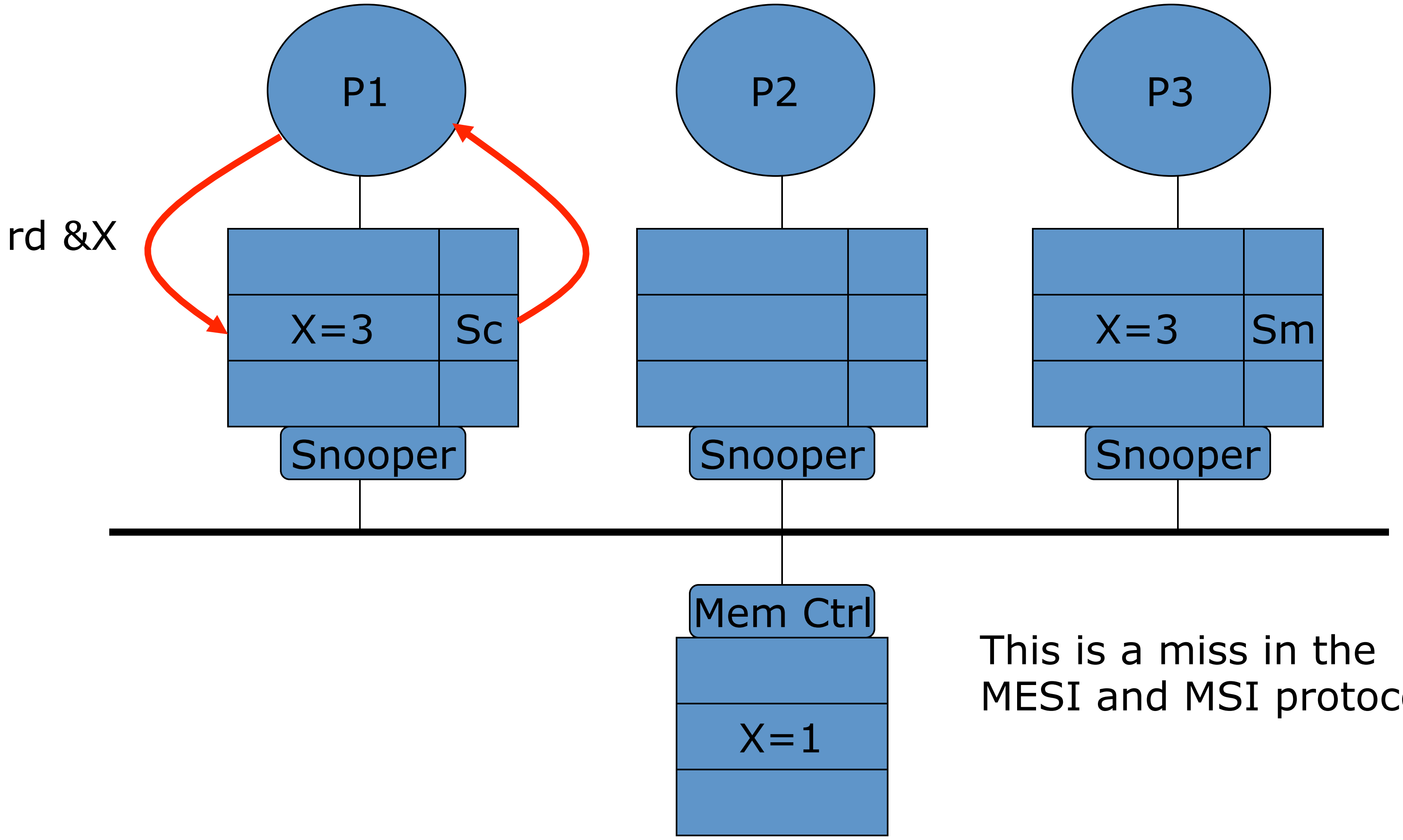


Dragon Visualization



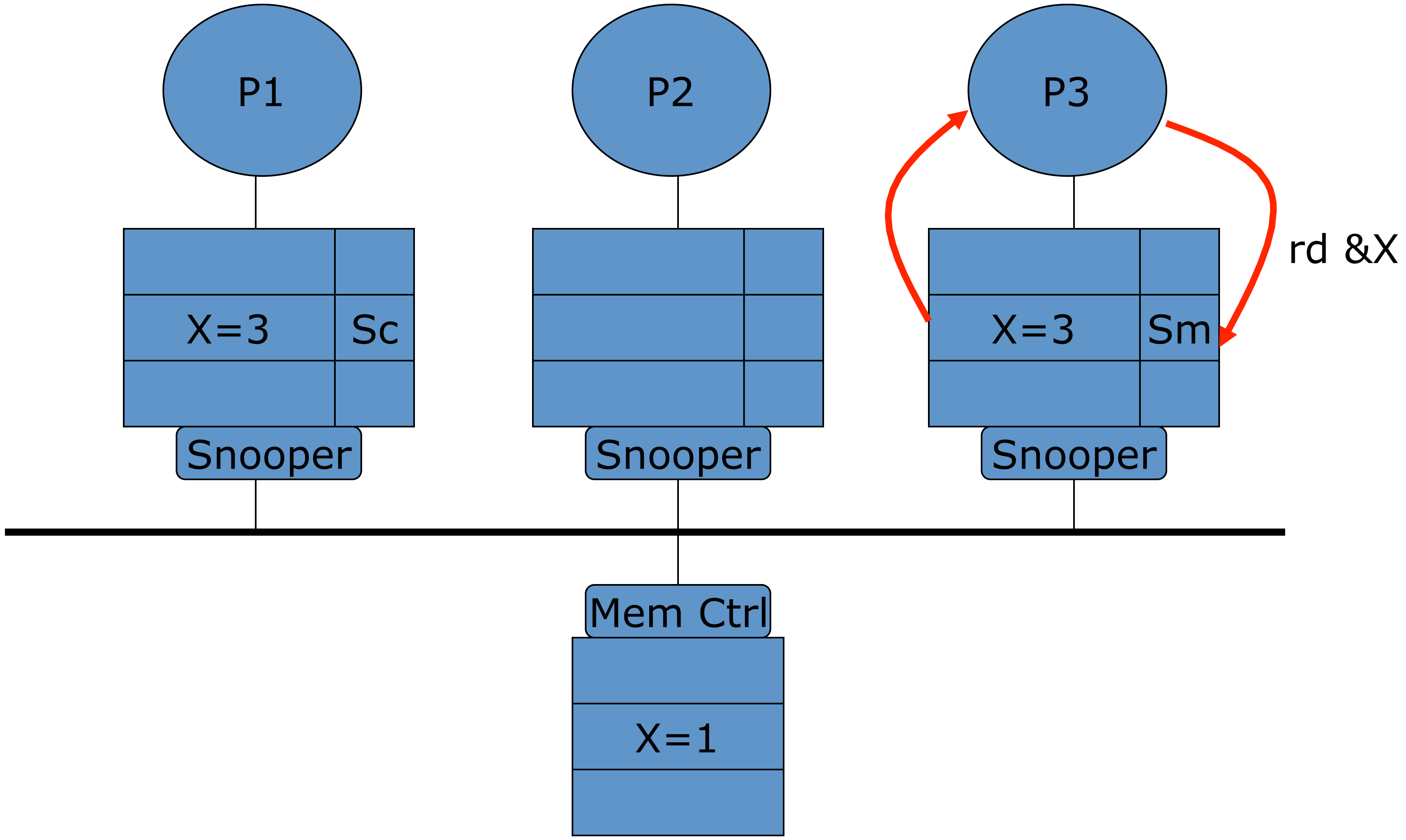
Note: BusUpdate instead of BusUpgr (no inval is performed)

Dragon Visualization

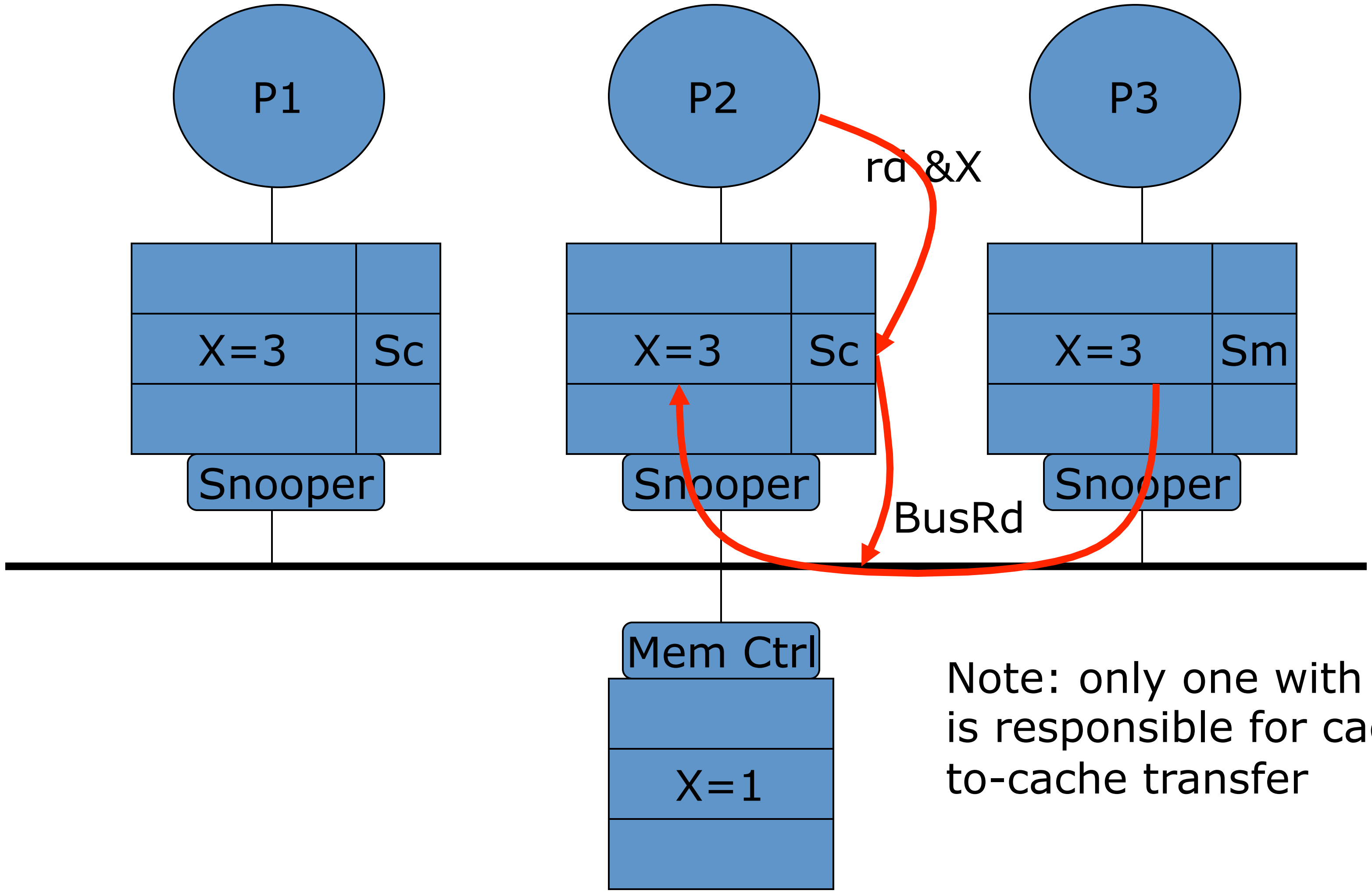


This is a miss in the MESI and MSI protocols

Dragon Visualization

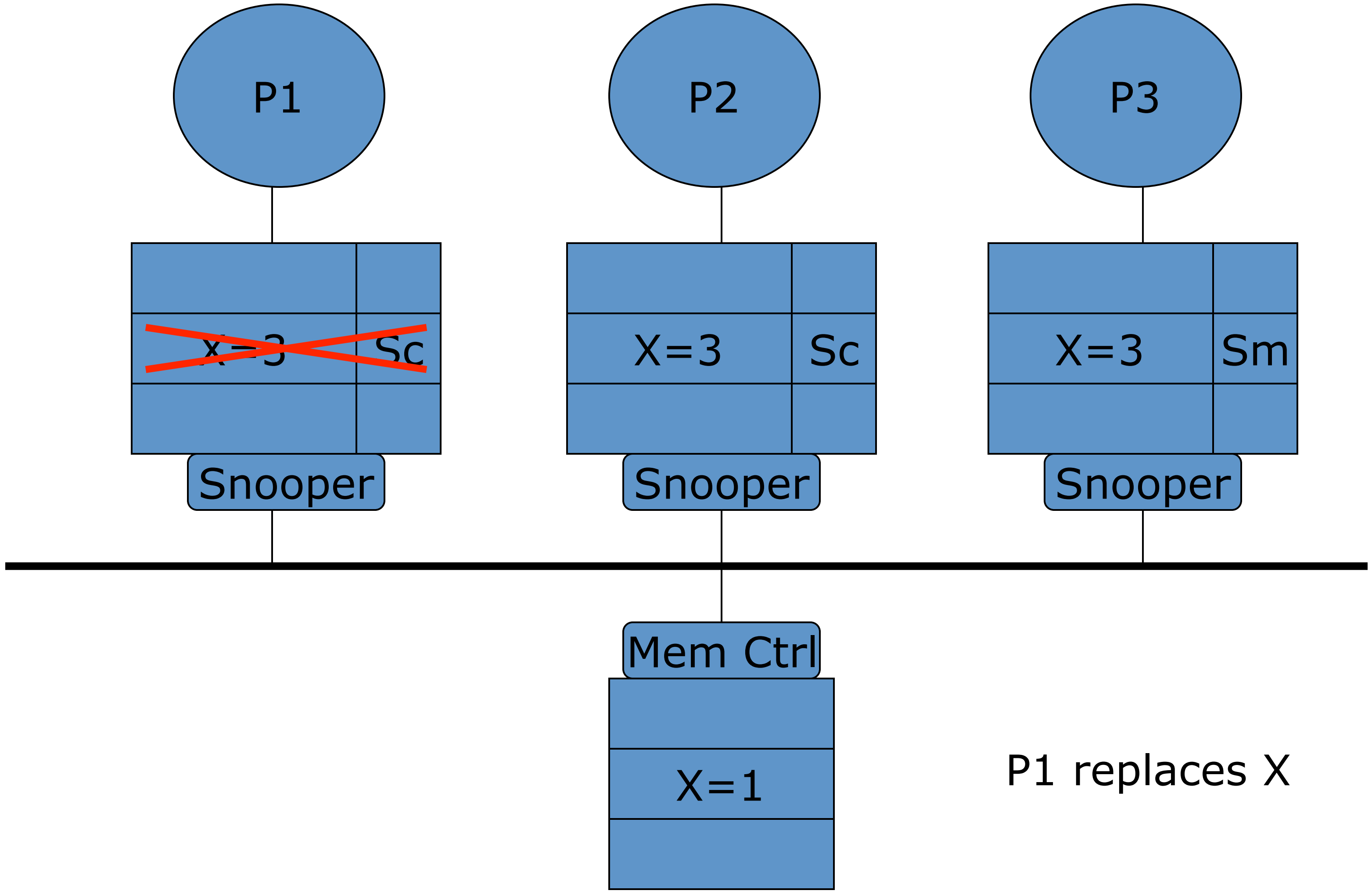


Dragon Visualization



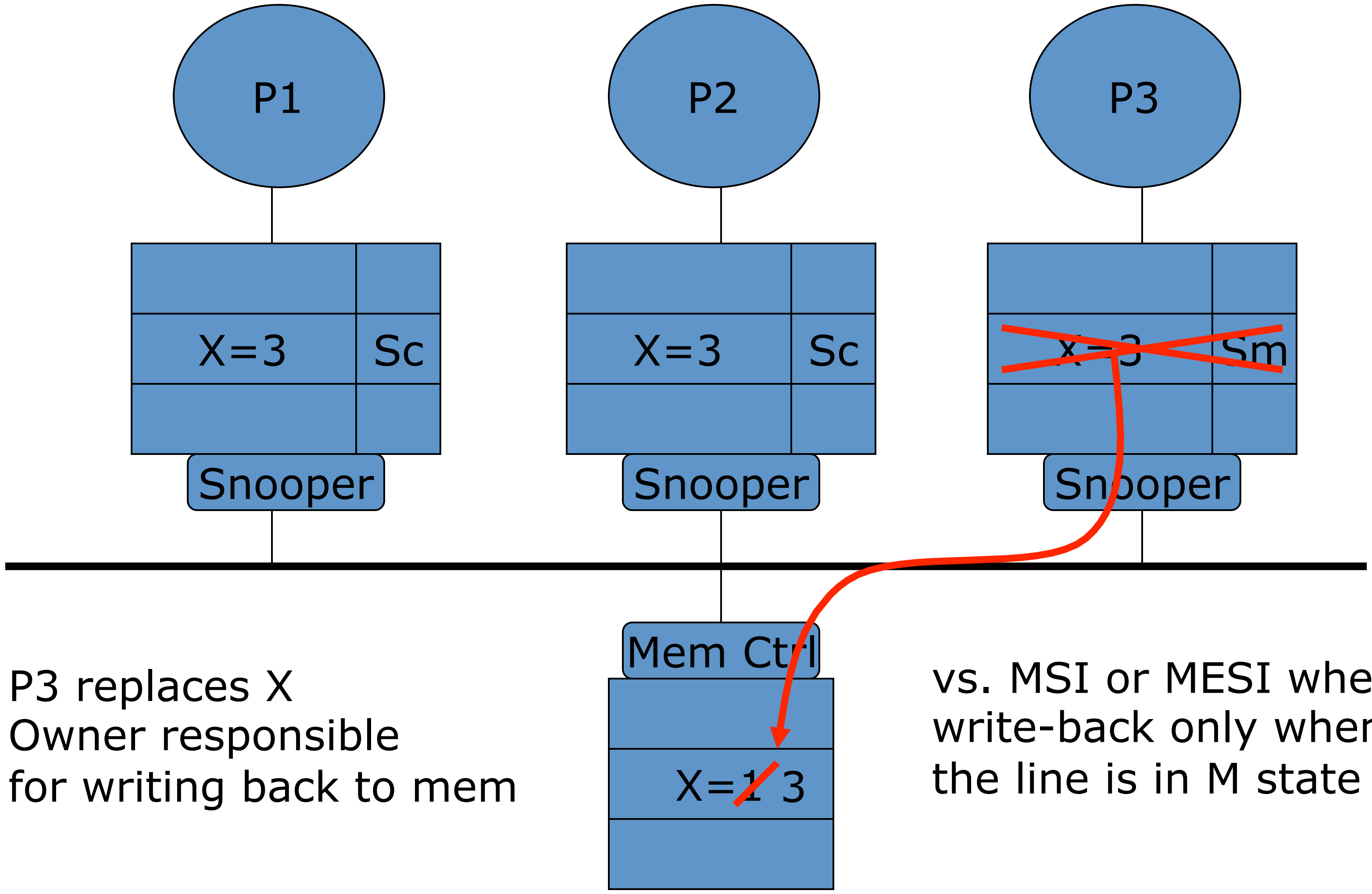
Note: only one with Sm is responsible for cache-to-cache transfer

Dragon Visualization



P1 replaces X

Dragon Visualization



P3 replaces X
Owner responsible
for writing back to mem

vs. MSI or MESI where
write-back only when
the line is in M state

Dragon Example

Assume:

- hit without bus transaction (1 cycle)
- hit/miss w/ bus transaction, no data or data from/to another cache (20 cycles)
- hit/miss w/ bus transaction, data from memory (40 cycles)

Proc Action	State P1	State P2	State P3	Bus Action	Data From	Cost
R1	E	-	-	BusRd	Mem	40
W1	M	-	-	-	-	1
R3	Sm	-	Sc	BusRd	P1's cache	20
W3	Sc	-	Sm	BusUpd	-	20
R1	Sc	-	Sm	-	-	1
R3	Sc	-	Sm	-	-	1
R2	Sc	Sc	Sm	BusRd	P3's cache	20

Lower-level Protocol Choices

- Can shared-modified state be eliminated?
  - If update memory as well on BusUpd transactions (DEC Firefly)
  - Dragon protocol doesn't (assumes DRAM memory slow to update)
- Should replacement of an Sc block be broadcast?
  - Would allow last copy to go to Exclusive state and not generate updates
  - Replacement bus transaction is not in critical path, later update may be
- Shouldn't update local copy on write hit before controller gets bus
  - Can mess up serialization
- Coherence, consistency considerations much like write-through case

Cache Coherence Performance

- Coherence misses
  - misses due to accesses to blocks that were invalidated due to coherence events
  - affects invalidation-based protocols (no invalidations in update-based protocols)
    - Update-based protocol increases conflict and capacity misses instead
- Types of coherence misses
  - True sharing: misses that occur when multiple threads share the same data item (byte/word)
  - False sharing: misses that occur when multiple threads share different data items located in a single cache block

Impact of Cache Parameters

Parameters	True Sharing Misses	False Sharing Misses
Larger cache size	increased	increased
Larger block size	decreased	increased
Larger associativity	unclear	unclear

Prefetching and Coherence Misses

- Prefetching to a dirty block causes downgrade at the owner, incurring several risks
  - The prefetched block may replace a more useful block (an inherent prefetching risk)
  - The prefetched block may be invalidated before it is used
  - A subsequent write by the owner is delayed (it must invalidate other copies first)

- Thus, timeliness of prefetching matters a great deal more in multiprocessor system. Performance may be hurt at both the cache that prefetches, and the cache that supplies the block.

Multi-level Caches

- Suppose each processor has its own L1 cache and L2 cache, and L2 caches are coherent
- Write propagation must be maintained between L1 and L2
- Upstream write propagation:
  - invalidation/intervention received by the L2 must be propagated to the L1 (if the L1 may have the block)
  - inclusion property cuts down the number of such upstream invalidation/intervention
- Downstream write propagation:
  - writes at L1 must be propagated to L2
    - through the use of write-through L1
    - or use explicit notification on writes
  - L2 must request flush from L1 if L1 uses write-back policy

Snoop filtering

- See Section 8.4.3 in the textbook