

# CSC 2224: Parallel Computer Architecture and Programming Memory Consistency & Cache Coherence

Prof. Gennady Pekhimenko

University of Toronto

Fall 2019

*The content of this lecture is adapted from the lectures of  
Onur Mutlu @ CMU*

# No class Oct. 29 week

- I am traveling to SOSIP 2019

# Reviews: Memory Consistency

- **Review:**
  - Lamport, “How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs,” IEEE Transactions on Computers, 1979 (less than 2 pages)
  - Boehm et al., “Foundations of the C++ Concurrency Memory Model”, PLDI 2008 (11 pages)

# Memory Ordering in Multiprocessors

# Ordering of Operations

- Operations: A, B, C, D
  - In what order should the hardware execute (and report the results of) these operations?
- A contract between programmer and microarchitect
  - Specified by the ISA
- Preserving an “expected” (more accurately, “agreed upon”) order simplifies programmer’s life
  - Ease of debugging; ease of state recovery, exception handling
- Preserving an “expected” order usually makes the hardware designer’s life difficult
  - Especially if the goal is to design a high performance processor: Load-store queues in out of order execution

# Single Processor Ordering

- Specified by the von Neumann model
- Sequential order
  - Hardware **executes** the load and store operations **in the order specified by the sequential program**
- Out-of-order execution does not change the semantics
  - Hardware **retires (reports to software the results of)** the load and store operations **in the order specified by the sequential program**
- Advantages: 1) Architectural state is precise within an execution. 2) Architectural state is consistent across different runs of the program ☐ Easier to debug programs
- Disadvantage: Preserving order adds overhead, reduces performance

# Dataflow Processor Ordering

- A memory operation executes when its operands are ready
- Ordering specified only by data dependencies
- Two operations can be executed and retired in any order if they have no dependency
- Advantage: Lots of parallelism ☐ high performance
- Disadvantage: Order can change across runs of the same program ☐ Very hard to debug

# MIMD Processor Ordering

- Each processor's memory operations are in sequential order with respect to the "thread" running on that processor (assume each processor obeys the von Neumann model)
- Multiple processors execute memory operations concurrently
- How does the memory see the order of operations from all processors?
  - In other words, what is the ordering of operations across different processors?



# Why Does This Even Matter?

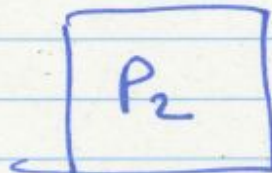
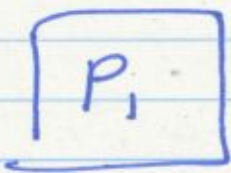
- Ease of debugging
  - It is nice to have the same execution done at different times have the same order of memory operations
- Correctness
  - Can we have incorrect execution if the order of memory operations is different from the point of view of different processors?
- Performance and overhead
  - Enforcing a strict “sequential ordering” can make life harder for the hardware designer in implementing performance enhancement techniques (e.g., OoO execution, caches)

# Protecting Shared Data

- Threads are not allowed to update shared data concurrently
  - For correctness purposes
- Accesses to shared data are encapsulated inside *critical sections* or protected via *synchronization constructs* (locks, semaphores, condition variables)
- Only one thread can execute a critical section at a given time
  - Mutual exclusion principle
- A multiprocessor should provide the *correct* execution of *synchronization primitives* to enable the programmer to protect shared data

# Supporting Mutual Exclusion

- Programmer needs to make sure mutual exclusion (synchronization) is correctly implemented
  - We will assume this
  - But, correct parallel programming is an important topic
  - Reading: Dijkstra, “[Cooperating Sequential Processes](#),” 1965.
- Programmer relies on hardware primitives to support correct synchronization
- If hardware primitives are not correct (or unpredictable), programmer’s life is tough
- If hardware primitives are correct but not easy to reason about or use, programmer’s life is still tough



# Protecting Shared Data

$F_1 = \emptyset$



A  $F_1 = 1$

B IF ( $F_2 == \emptyset$ ) THEN  
  { Critical section }

$F_1 \neq \emptyset$

ELSE  
  { ... }

$F_2 = \emptyset$



X  $F_2 = 1$

Y IF ( $F_1 == \emptyset$ ) THEN  
  { Critical section }

$F_2 = \emptyset$

ELSE  
  { ... }

Only  $P_1$  or  
 $P_2$  should  
be in this  
section  
at any given  
time,  
not both

Assume  $P_1$  is in critical section.

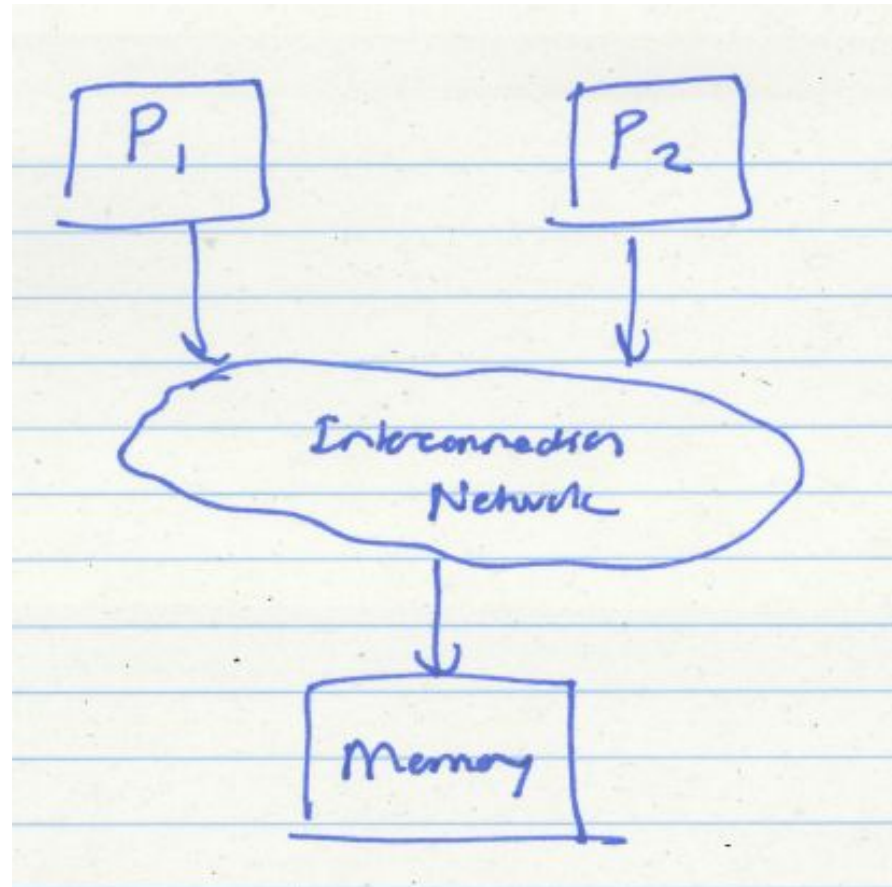
Intuitively, it must have executed A,

which means  $F_1$  must be 1 (as A happens before B),

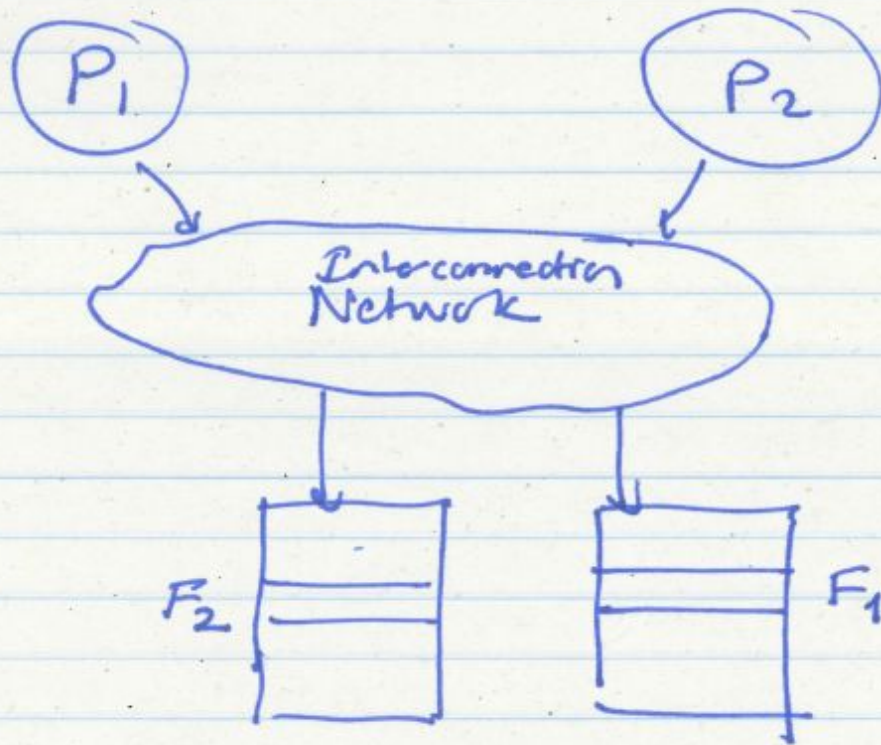
which means  $P_2$  should not enter the critical section.

# A Question

- Can the two processors be in the critical section at the same time given that they both obey the von Neumann model?
- Answer: yes



An Incorrect Result (due to an implementation that does not provide sequential consistency)



time 0:  $P_1$  executes A  
(set  $F_1 = 1$ ) st  $F_1$  complete  
A is sent to memory (from  $P_1$ 's view)

$P_2$  executes X  
(set  $F_2 = 1$ ) st  $F_2$  complete  
X is sent to memory (from  $P_2$ 's view)

# Both Processors in Critical Section

time 0:  $P_1$  executes A  
(set  $F_1 = 1$ ) st  $F_1$  complete  
A is sent to memory (from  $P_1$ 's view)

$P_2$  executes X  
(set  $F_2 = 1$ ) st  $F_2$  complete  
X is sent to memory (from  $P_2$ 's view)

time 1:  $P_1$  executes B  
(test  $F_2 == 0$ ) ld  $F_2$  started  
B is sent to memory

$P_2$  executes Y  
(test  $F_1 == 0$ ) ld  $F_1$  started  
Y is sent to memory

time 50: Memory sends back to  $P_1$   
 $F_2 (0)$  ld  $F_2$  complete

Memory sends back to  $P_2$   
 $(F_1 (0))$  ld  $F_1$  complete

time 51:  $P_1$  is in critical section  
~~execute~~

$P_2$  is in critical section

time 100: Memory completes A  
 $F_1 = 1$  in memory  
(too late!)

Memory completes ~~X~~  
 $F_2 = 1$  in memory  
(too late!)

# What happened?

## P<sub>1</sub>'s view of mem. ops

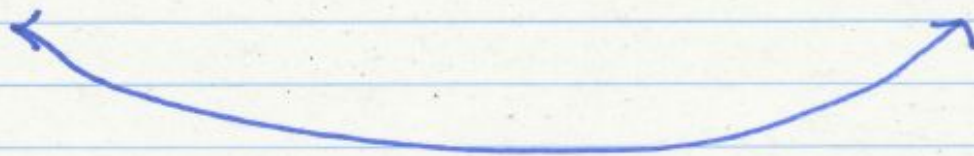
A (F<sub>1</sub>=1)  
B (test F<sub>2</sub>=0)  
X (F<sub>2</sub>=1)

## P<sub>2</sub>'s view

X (F<sub>2</sub>=1)  
Y (test<sub>2</sub> F<sub>1</sub>=0)  
A (F<sub>1</sub>=1)

B executed before X

Y executed before A



Problem!

These two processors did  
not see the same order  
of operations in memory



# How Can We Solve The Problem?

- Idea: **Sequential consistency**
- **All processors see the same order of operations to memory**
- i.e., all memory operations happen in an order (called the global total order) that is consistent across all processors
- Assumption: within this global order, each processor's operations appear in sequential order with respect to its own operations.

# Sequential Consistency

Lamport, “[How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs](#),” IEEE Transactions on Computers, 1979

- A multiprocessor system is sequentially consistent if:
  - the result of any execution is the same as if the operations of all the processors were executed in some sequential order

AND

- the operations of each individual processor appear in this sequence in the order specified by its program
- This is a memory ordering model, or memory model
  - Specified by the ISA

# Programmer's Abstraction

- Memory is a switch that services one load or store at a time from any processor
- All processors see the currently serviced load or store at the same time
- Each processor's operations are serviced in program order

# Sequentially Consistent Operation

- Potential correct global orders (all are correct):
- A B X Y
- A X B Y
- A X Y B
- X A B Y
- X A Y B
- X Y A B
- Which order (interleaving) is observed depends on implementation and dynamic latencies

# Consequences of Sequential Consistency

1. Within the same execution, all processors see the same global order of operations to memory
  - ❓ No correctness issue
  - ❓ Satisfies the “happened before” intuition
2. Across different executions, different global orders can be observed (each of which is sequentially consistent)
  - ❓ Debugging is still difficult (as order changes across runs)

# Issues with Sequential Consistency?

- Nice abstraction for programming, but two issues:
  - Too conservative ordering requirements
  - Limits the aggressiveness of performance enhancement techniques
- Is the total global order requirement too strong?
  - Do we need a global order across all operations and all processors?
  - How about a global order only across all stores?
    - Total store order memory model; unique store order model
  - How about enforcing a global order only at the boundaries of synchronization? Relaxed/Acquire-release consistency model

# Issues with Sequential Consistency?

Performance enhancement techniques that could make SC implementation difficult

- Out-of-order execution
  - Loads happen out-of-order with respect to each other and with respect to independent stores
- Caching
  - A memory location is now present in multiple places
  - Prevents the effect of a store to be seen by other processors

# Weaker Memory Consistency

- The ordering of operations is important when the order affects operations on shared data ☒ i.e., when processors need to synchronize to execute a “program region”
- Weak consistency
  - Idea: Programmer specifies regions in which memory operations do not need to be ordered
  - “Memory fence” instructions delineate those regions
    - All memory operations before a fence must complete before the fence is executed
    - All memory operations after the fence must wait for the fence to complete
    - Fences complete in program order
  - All synchronization operations act like a fence



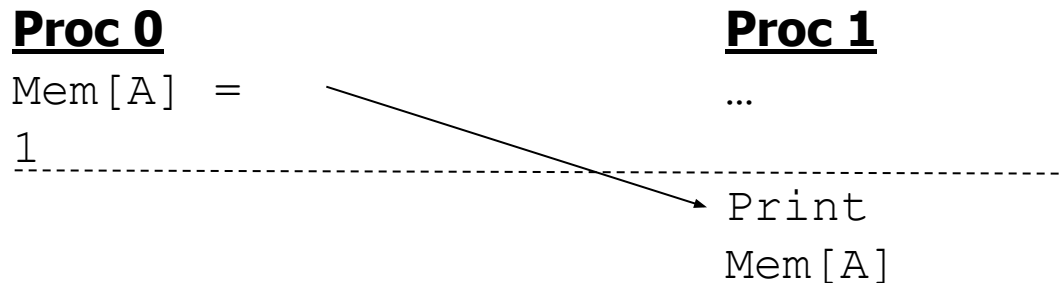
# Tradeoffs: Weaker Consistency

- Advantage
  - No need to guarantee a very strict order of memory operations
    - ☐ Enables the hardware implementation of performance enhancement techniques to be **simpler**
    - ☐ Can be **higher performance** than stricter ordering
- Disadvantage
  - More **burden on the programmer** or software (need to get the “fences” correct)
- Another example of the programmer-microarchitect tradeoff

# Cache Coherence

# Shared Memory Model

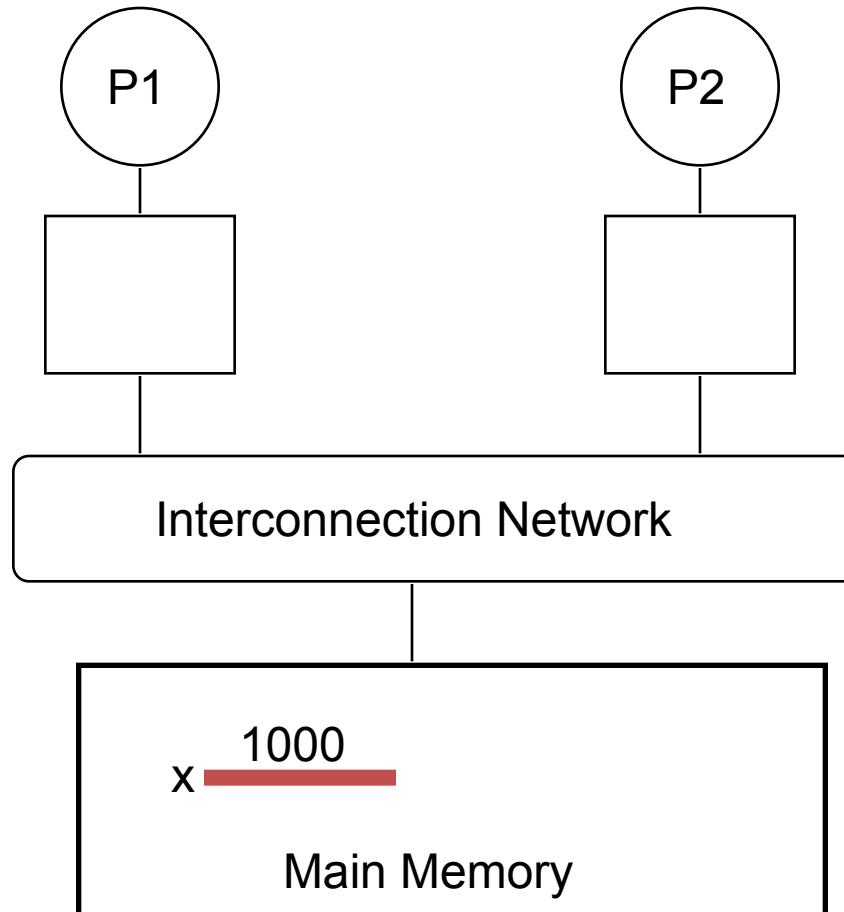
- Many parallel programs communicate through *shared memory*
- Proc 0 writes to an address, followed by Proc 1 reading
  - This implies communication between the two



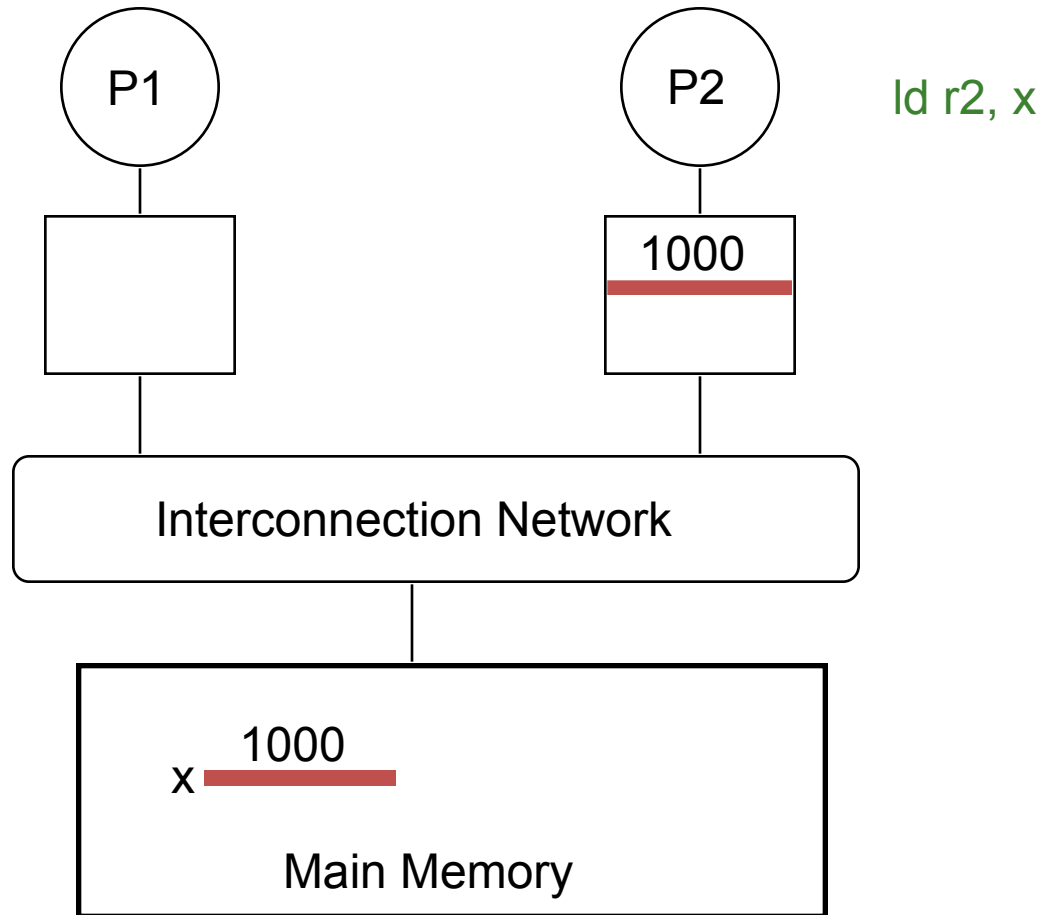
- Each read should receive the value last written by anyone
  - This requires synchronization (what does last written mean?)
- What if Mem[A] is cached (at either end)?

# Cache Coherence

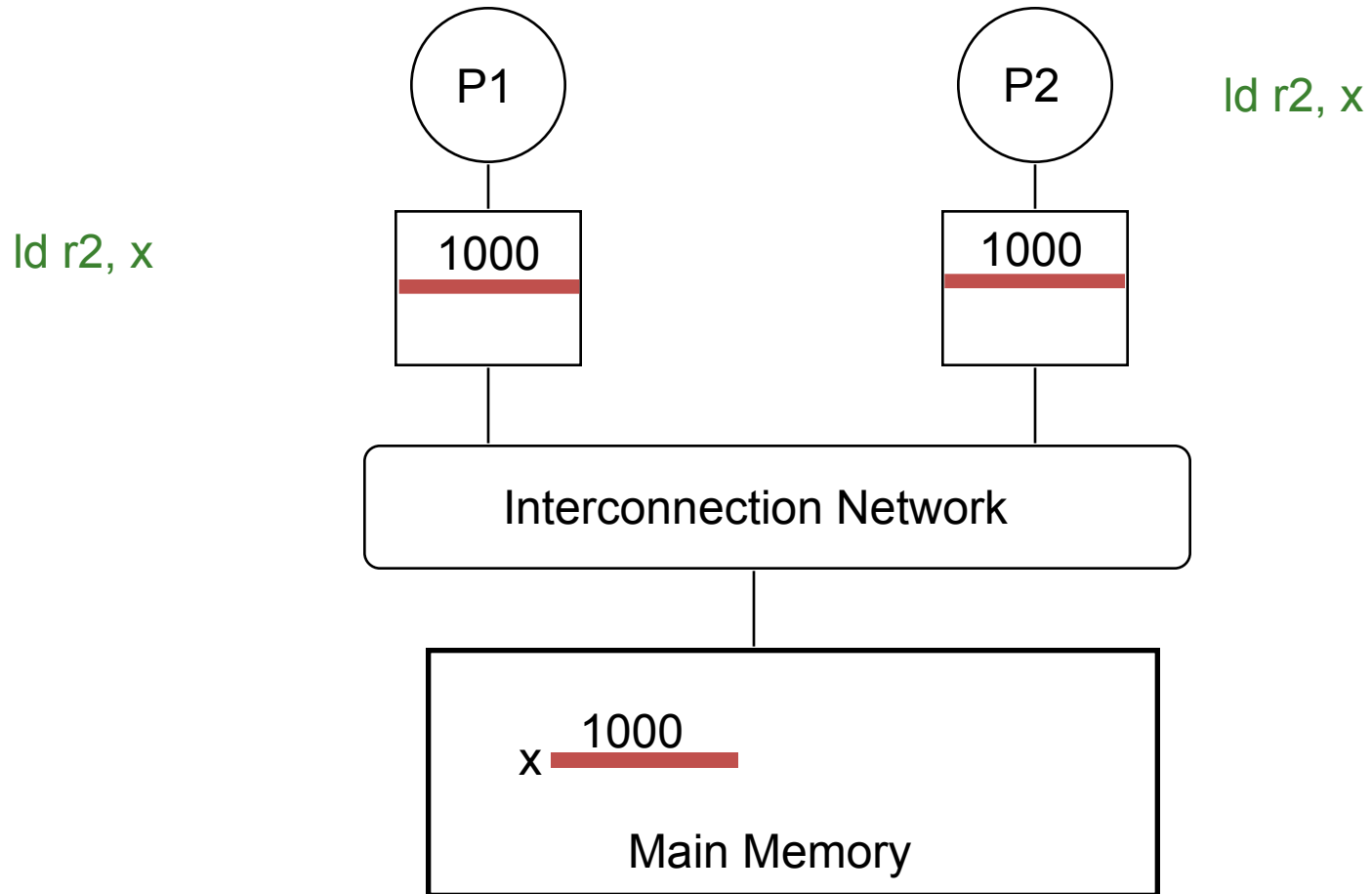
- Basic question: **If multiple processors cache the same block, how do they ensure they all see a consistent state?**



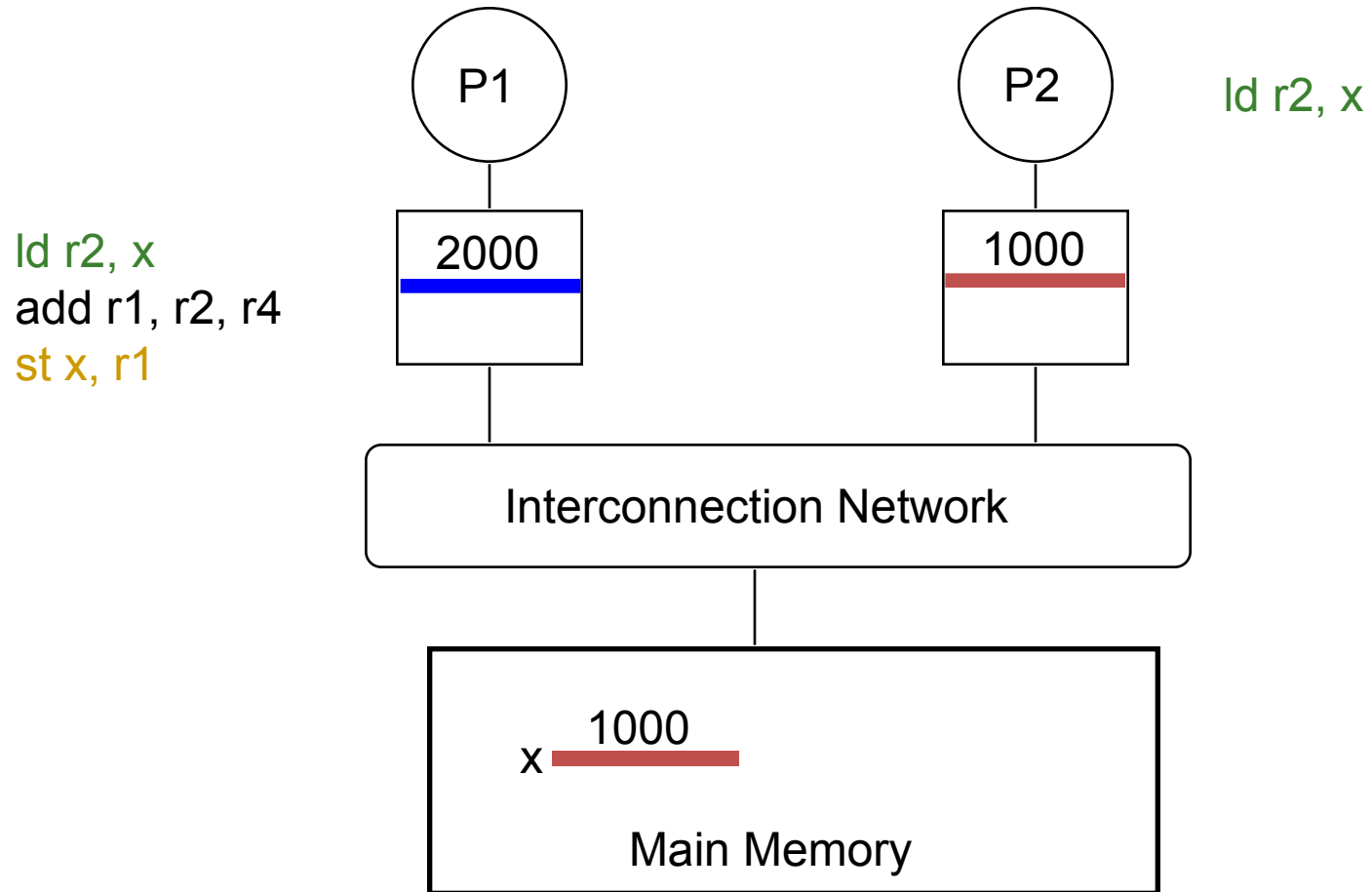
# The Cache Coherence Problem



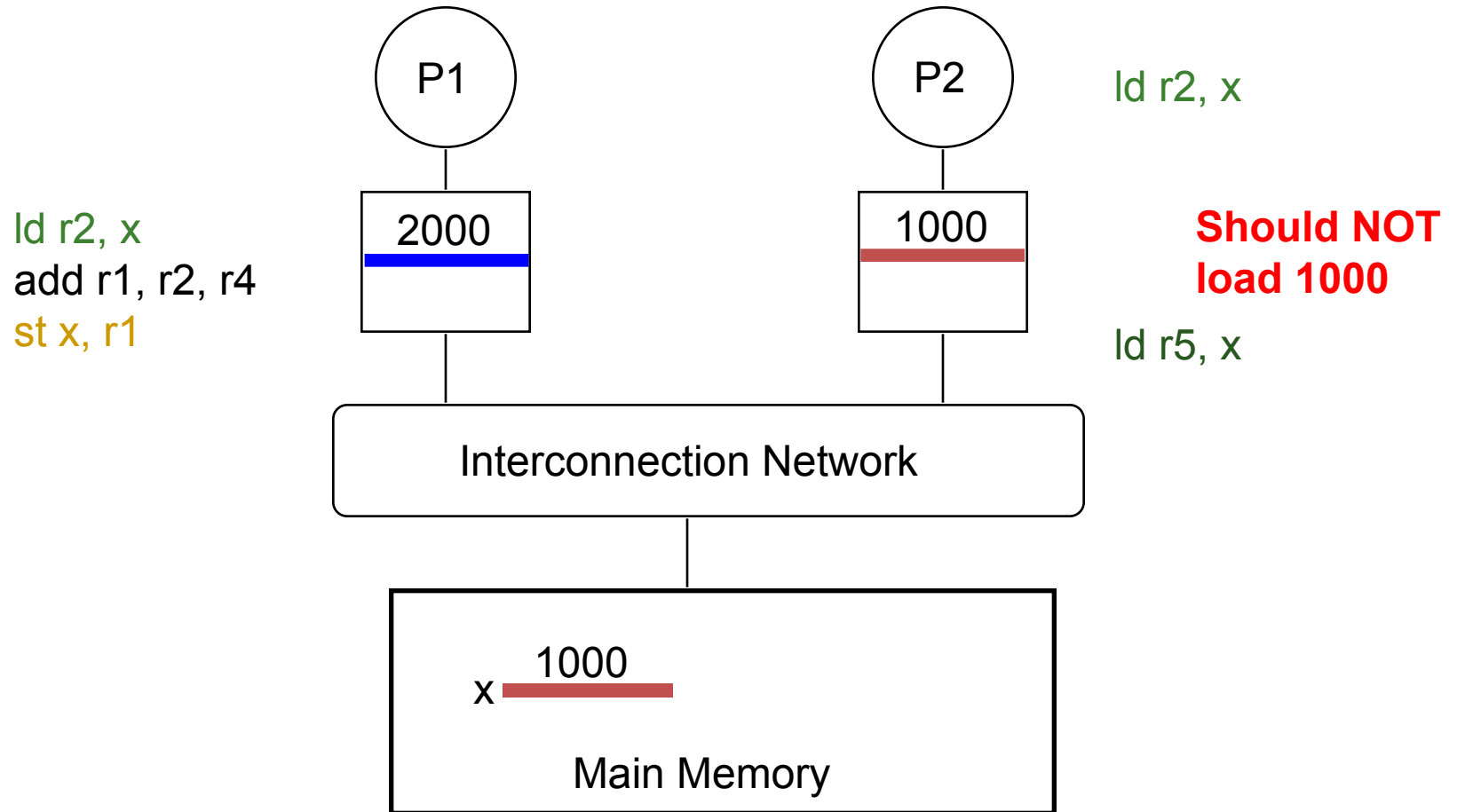
# The Cache Coherence Problem



# The Cache Coherence Problem



# The Cache Coherence Problem



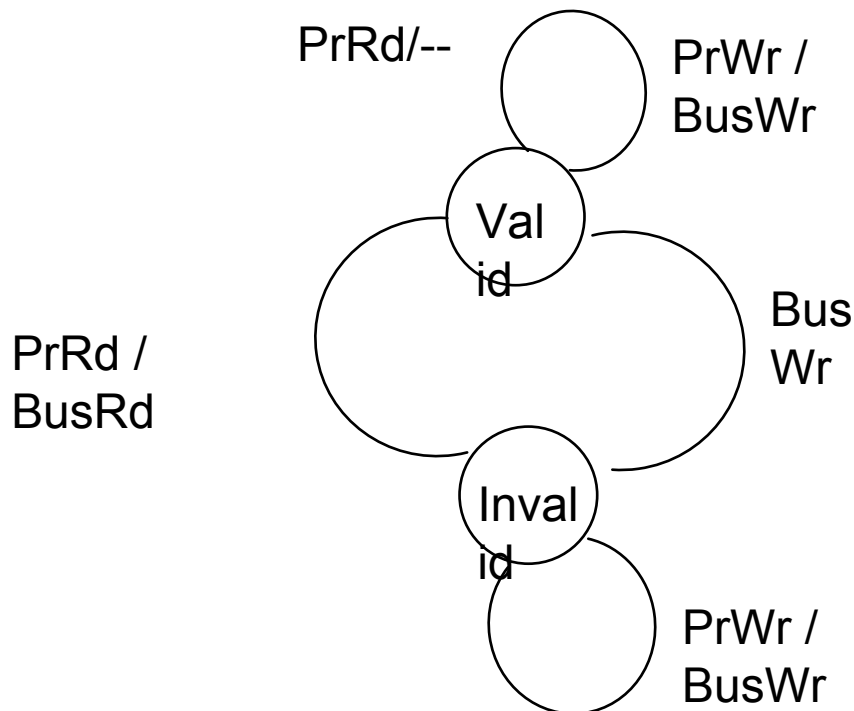


# Cache Coherence: Whose Responsibility?

- Software
  - Can the programmer ensure coherence if caches are invisible to software?
  - What if the ISA provided a cache flush instruction?
    - FLUSH-LOCAL A: Flushes/invalidates the cache block containing address A from a processor's local cache.
    - FLUSH-GLOBAL A: Flushes/invalidates the cache block containing address A from all other processors' caches.
    - FLUSH-CACHE X: Flushes/invalidates all blocks in cache X.
- Hardware
  - Simplifies software's job
  - One idea: Invalidate all other copies of block A when a processor writes to it

# A Very Simple Coherence Scheme

- Caches “snoop” (observe) each other’s write/read operations. If a processor writes to a block, all others invalidate it from their caches.
- A simple protocol:



- Write-through, no-write-allocate cache
- Actions: PrRd, PrWr, BusRd, BusWr

# (Non-)Solutions to Cache Coherence

- No hardware based coherence
  - Keeping caches coherent is software's responsibility
  - + Makes microarchitect's life easier
  - Makes average programmer's life much harder
    - need to worry about hardware caches to maintain program correctness?
  - Overhead in ensuring coherence in software
- All caches are shared between all processors
  - + No need for coherence
  - Shared cache becomes the bandwidth bottleneck
  - Very hard to design a scalable system with low-latency cache access this way

# Maintaining Coherence

- Need to guarantee that all processors see a consistent value (i.e., consistent updates) for the same memory location
- Writes to location A by P0 should be seen by P1 (eventually), and all writes to A should appear in some order
- Coherence needs to provide:
  - **Write propagation:** guarantee that updates will propagate
  - **Write serialization:** provide a consistent global order seen by all processors
- Need a global point of serialization for this store ordering

# Hardware Cache Coherence

- Basic idea:
  - A processor/cache broadcasts its write/update to a memory location to all other processors
  - Another cache that has the location either updates or invalidates its local copy

# Coherence: Update vs. Invalidate

- How can we *safely update replicated data*?
  - Option 1 (Update protocol): push an update to all copies
  - Option 2 (Invalidate protocol): ensure there is only one copy (local), update it
- **On a Read:**
  - If local copy isn't valid, put out request
  - (If another node has a copy, it returns it, otherwise memory does)

# Update vs. Invalidate (2)

- **On a Write:**

- Read block into cache as before

## **Update Protocol:**

- Write to block, and simultaneously broadcast written data to sharers
- (Other nodes update their caches if data was present)

## **Invalidate Protocol:**

- Write to block, and simultaneously broadcast invalidation of address to sharers
- (Other nodes clear block from cache)

# Update vs. Invalidate Tradeoffs

- Which do we want?
  - Write frequency and sharing behavior are critical
- **Update**
  - + If sharer set is constant and updates are infrequent, avoids the cost of invalidate-reacquire (broadcast update pattern)
  - If data is rewritten without intervening reads by other cores, updates were useless
  - Write-through cache policy ☐ bus becomes bottleneck
- **Invalidate**
  - + After invalidation broadcast, core has exclusive access rights
  - + Only cores that keep reading after each write retain a copy
  - If write contention is high, leads to ping-ponging (rapid mutual invalidation-reacquire)



# Two Cache Coherence Methods

- How do we ensure that the proper caches are updated?
- **Snoopy Bus** [Goodman ISCA 1983, Papamarcos+ ISCA 1984]
  - Bus-based, **single point of serialization for all requests**
  - Processors observe other processors' actions
    - E.g.: P1 makes “read-exclusive” request for A on bus, P0 sees this and invalidates its own copy of A
- **Directory** [Censier and Feautrier, IEEE ToC 1978]
  - **Single point of serialization *per block***, distributed among nodes
  - Processors make explicit requests for blocks
  - Directory tracks ownership (sharer set) for each block
  - Directory coordinates invalidation appropriately
    - E.g.: P1 asks directory for exclusive copy, directory asks P0 to invalidate, waits for ACK, then responds to P1

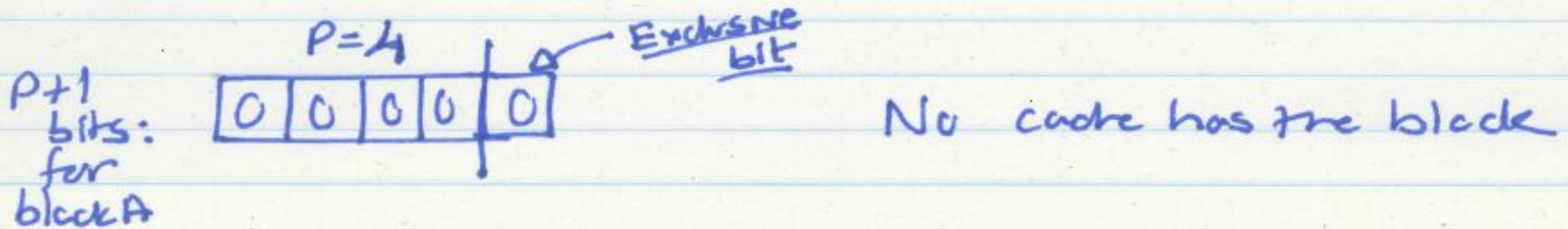
# Directory Based Cache Coherence

# Directory Based Coherence

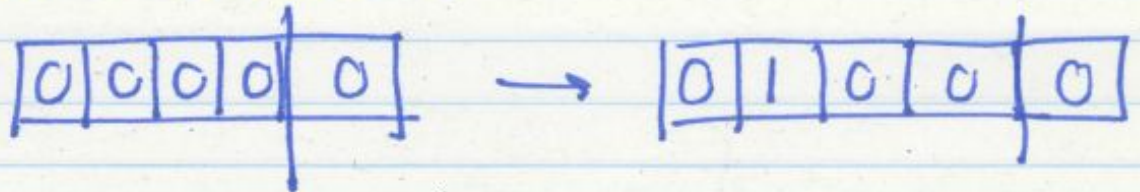
- Idea: A logically-central directory keeps track of where the copies of each cache block reside. Caches consult this directory to ensure coherence.
- An example mechanism:
  - For each cache block in memory, store  $P+1$  bits in directory
    - One bit for each cache, indicating whether the block is in cache
    - Exclusive bit: indicates that a cache has the only copy of the block and can update it without notifying others
  - On a read: set the cache's bit and arrange the supply of data
  - On a write: invalidate all caches that have the block and reset their bits
  - Have an “exclusive bit” associated with each block in each cache

# Directory Based Coherence Example

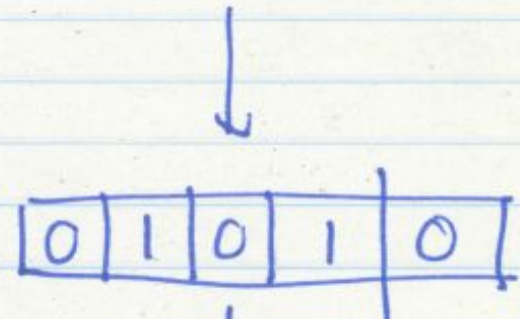
Example directory based scheme



①  $P_1$  takes a read miss to block A



②  $P_3$  takes a read miss



③ P<sub>2</sub> takes a write miss

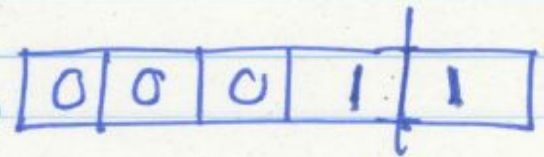
- invalidate P<sub>1</sub> & P<sub>3</sub>'s caches
- write request → P<sub>2</sub> has the exclusive copy of the block now. Set the Exclusive bit



- P<sub>2</sub> can now update the block without notifying any other processor or the directory
- P<sub>2</sub> needs to have a bit in its cache indicating it can perform exclusive updates to that block
  - private/exclusive bit per cache block

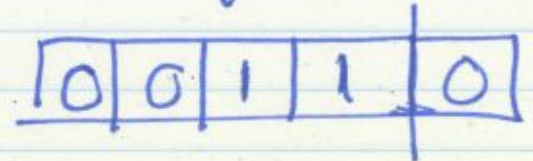
④ P<sub>3</sub> takes a write miss

- Mem ~~controller~~ Controller requests ~~the~~ block from P<sub>2</sub>
- Mem Controller gives block to P<sub>3</sub>
- P<sub>2</sub> invalidates its copy



⑤ P<sub>2</sub> takes a read miss

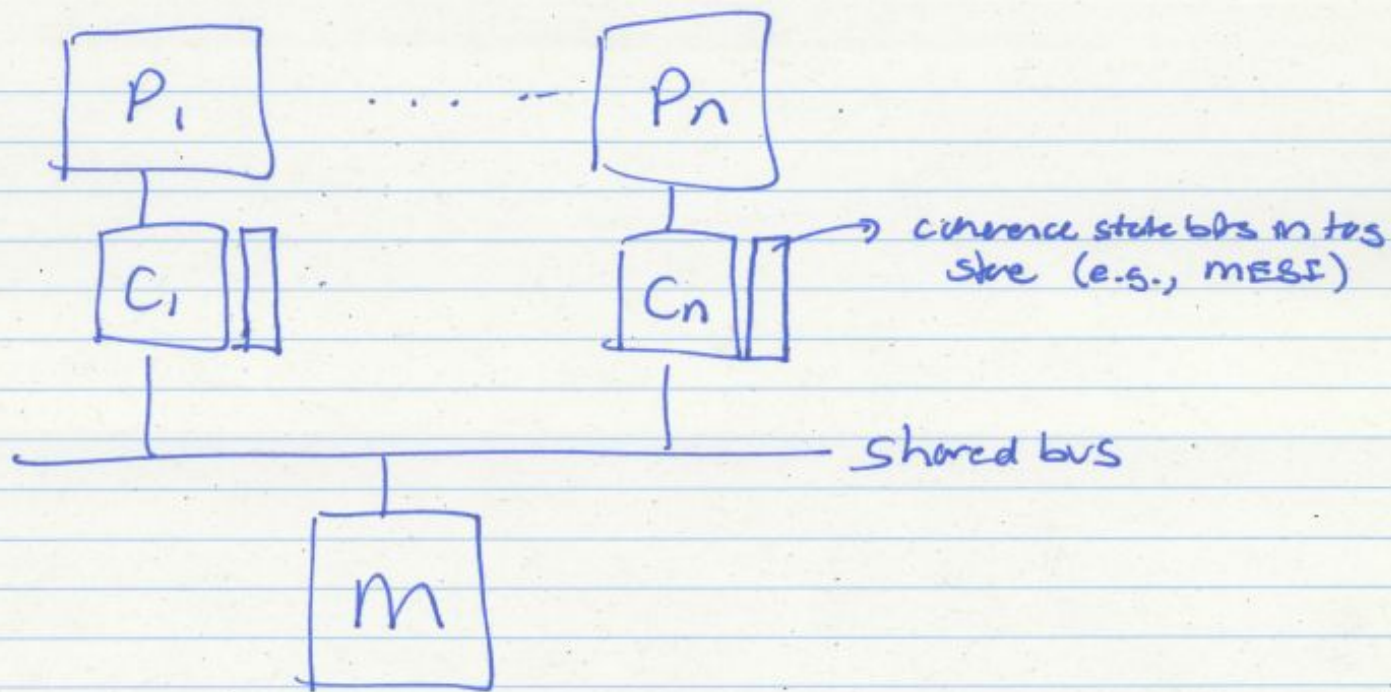
- P<sub>3</sub> supplies it



# Snoopy Cache Coherence

# Snoopy Cache Coherence

- Idea:
  - All caches “snoop” all other caches’ read/write requests and keep the cache block coherent
  - Each cache block has “coherence metadata” associated with it in the tag store of each cache
- Easy to implement if all caches share a common bus
  - Each cache broadcasts its read/write operations on the bus
  - Good for small-scale multiprocessors
  - What if you would like to have a 1000-node multiprocessor?



### SNOOPY CACHE

Each Cache observes its own processor & the bus  
 - Changes the state of the cached block based on observed actions by processor & the bus

Processor actions to a block: PR (Proc. Read)  
 RW (Proc. Write)

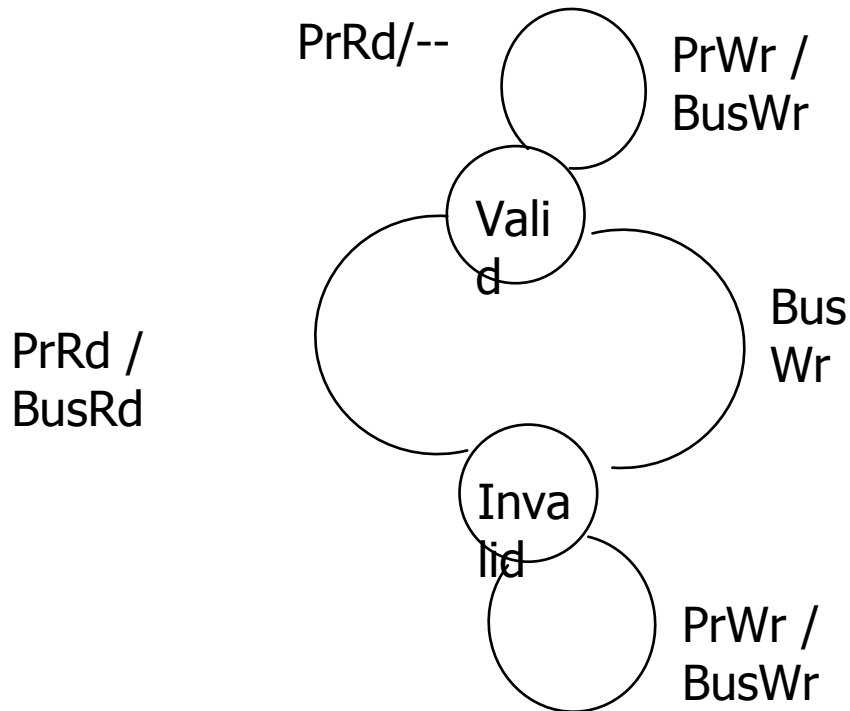
Bus actions to a block: BR (Bus Read)  
 (coming from another processor) BW (Bus Write)

or BRx (Bus Read Exclusive)



# A Simple Snoopy Protocol

- Caches “snoop” (observe) each other’s write/read operations
- A simple protocol:

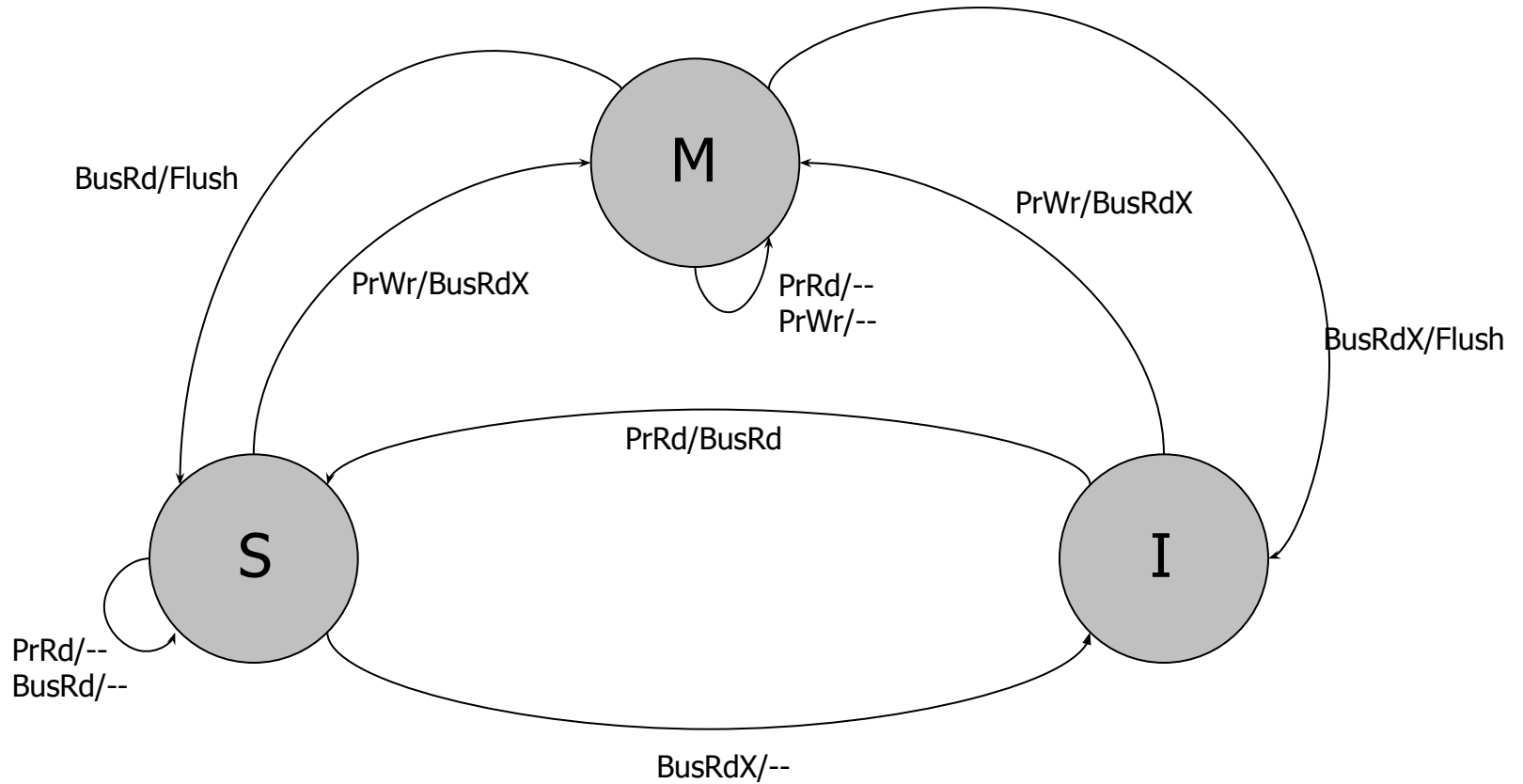


- Write-through, no-write-allocate cache
- Actions: PrRd, PrWr, BusRd, BusWr

# A More Sophisticated Protocol: MSI

- Extend single valid bit per block to three states:
  - **M**(odified): cache line is only copy and is dirty
  - **S**(hared): cache line is one of several copies
  - **I**(nvalid): not present  
Read miss makes a *Read* request on bus, transitions to **S**
- Write miss makes a *ReadEx* request, transitions to **M** state
- When a processor snoops *ReadEx* from another writer, it must invalidate its own copy (if any)
- **S**→**M** upgrade can be made without re-reading data from memory (via *Invalidations*)

# MSI State Machine



ObservedEvent/Action

[Culler/Singh96]

# The Problem with MSI

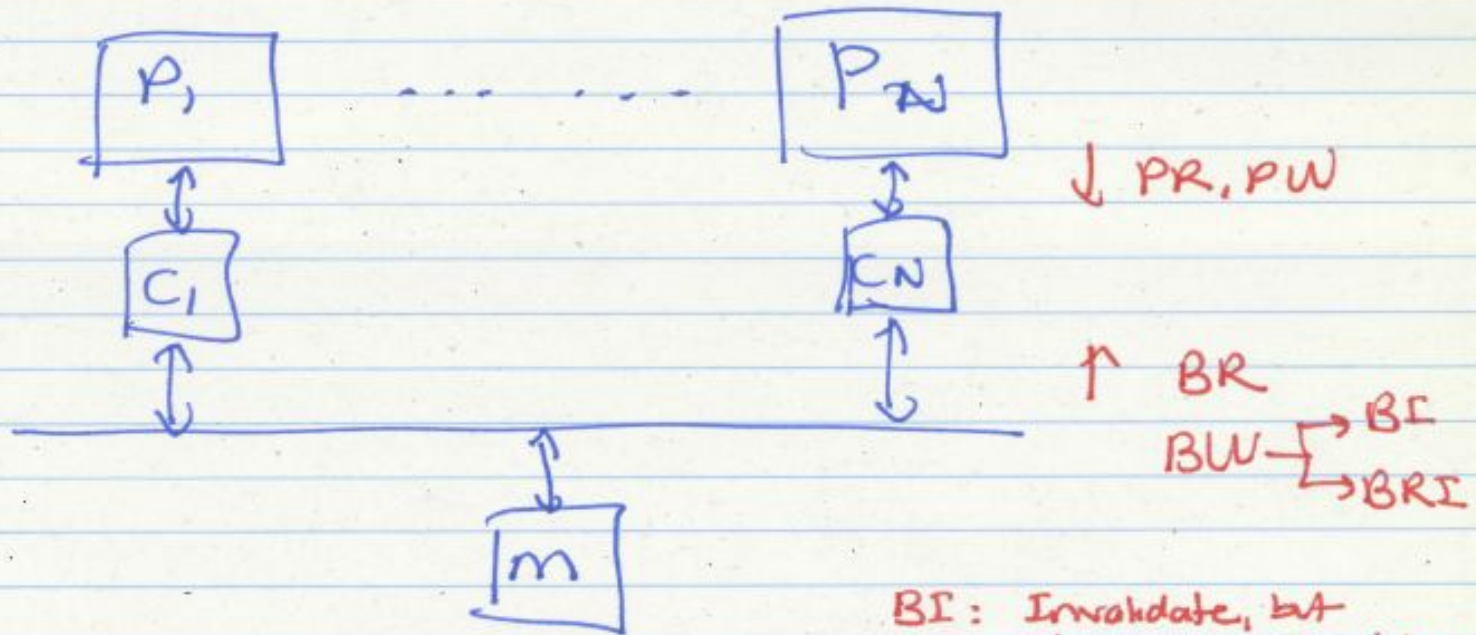
- A block is in no cache to begin with
- Problem: On a read, the block immediately goes to “Shared” state although it may be the only copy to be cached (i.e., no other processor will cache it)
- Why is this a problem?
  - Suppose the cache that read the block wants to write to it at some point
  - It needs to broadcast “invalidate” even though it has the only cached copy!
  - If the cache knew it had the only cached copy in the system, it could have written to the block without notifying any other cache ☹ saves unnecessary broadcasts of invalidations

# The Solution: MESI

- Idea: Add another state indicating that this is the only cached copy and it is clean.
  - *Exclusive* state
- Block is placed into the *exclusive* state if, during *BusRd*, no other cache had it
  - Wired-OR “shared” signal on bus can determine this: snooping caches assert the signal if they also have a copy
- Silent transition *Exclusive* → *Modified* is possible on write

Papamarcos & Patel, ISCA 1984

## Illinois Protocol



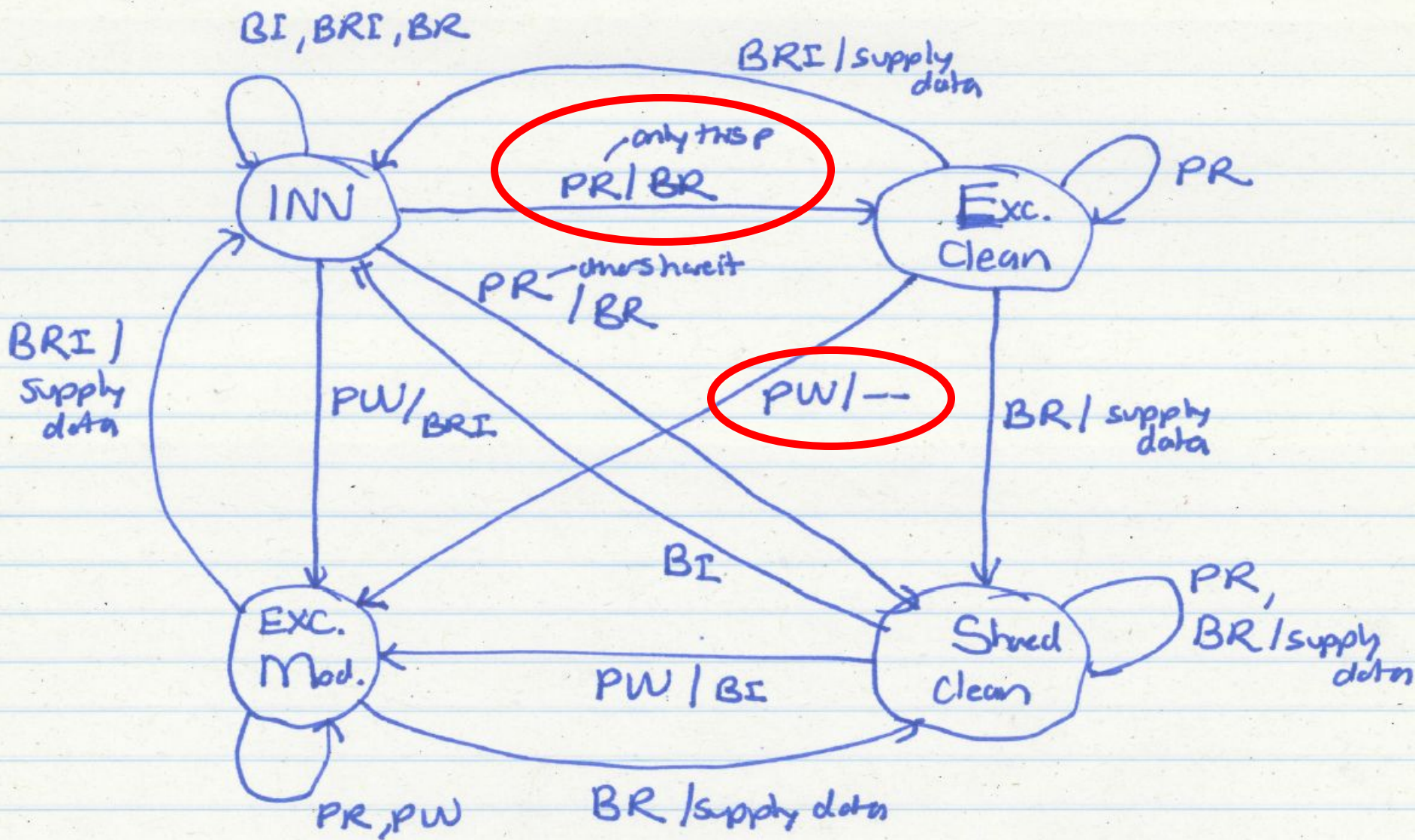
**BI:** Invalidate, but  
already have the data  
(do not supply it)

**BRI:** Invalidate, but  
also need the data  
(supply it)

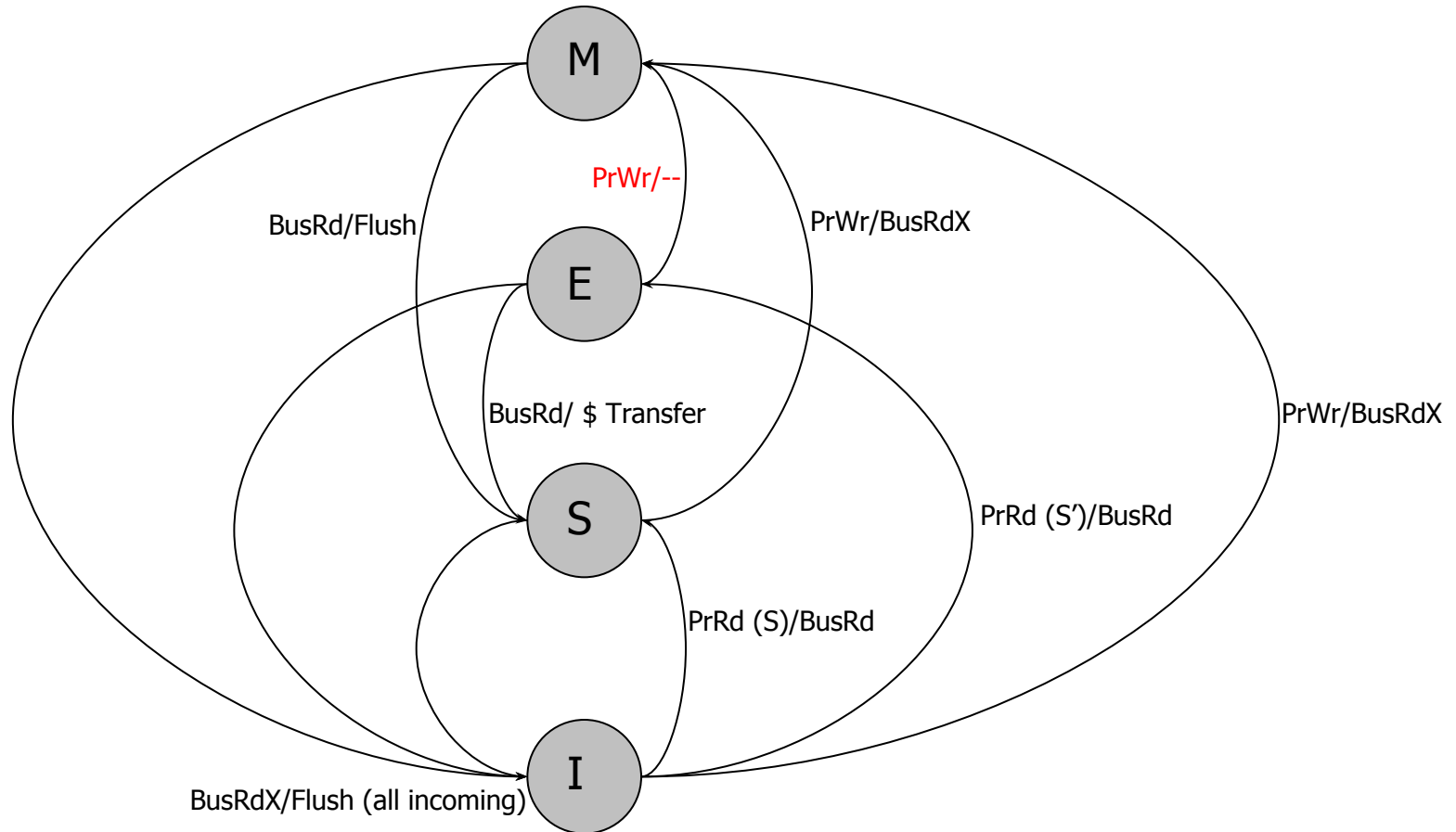
### 4 States

M: Modified (Exclusive copy, modified)  
E: Exclusive (" " , clean)  
S: Shared (Shared copy, clean)  
I: Invalid

# MESI State Machine



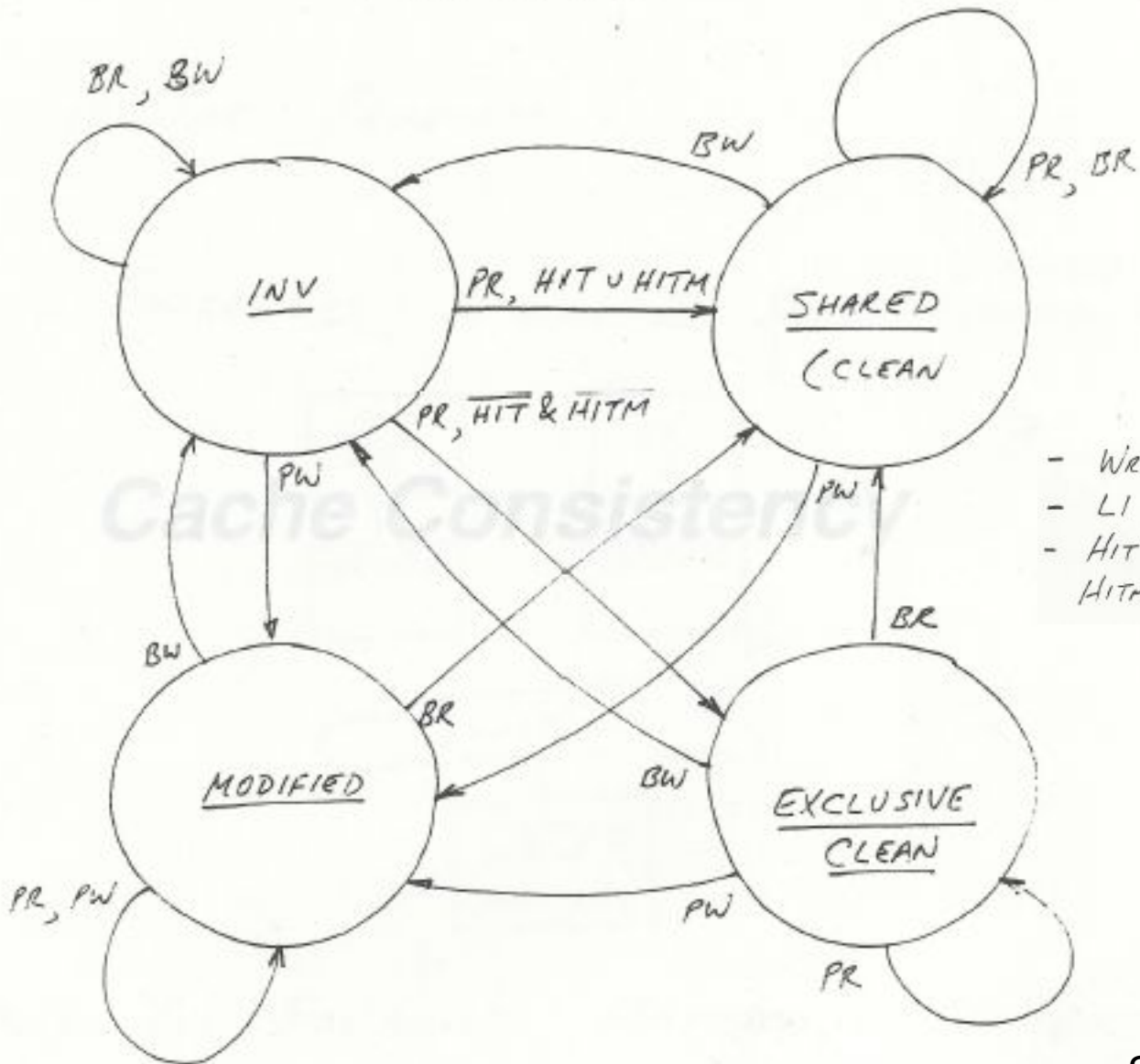
# MESI State Machine



[Culler/Singh96]



# Intel Pentium Pro



- WRITE ALLOCATE
- L1 CAN HAVE DATA NOT IN L2
- HIT : SOMEONE HAS IT CLEAN
- HITM : SOMEONE HAS IT DIRTY

# Snoopy Invalidation Tradeoffs

- Should a downgrade from M go to S or I?
  - S: if data is likely to be reused (before it is written to by another processor)
  - I: if data is likely to be not reused (before it is written to by another)
- Cache-to-cache transfer
  - On a BusRd, should data come from another cache or memory?
  - Another cache
    - may be faster, if memory is slow or highly contended
  - Memory
    - Simpler: no need to wait to see if cache has data first
    - Less contention at the other caches
    - Requires writeback on M downgrade
- Writeback on Modified->Shared: necessary?
  - One possibility: **Owner** (O) state (MOESI protocol)
    - One cache owns the latest data (memory is not updated)
    - Memory writeback happens when all caches evict copies

# The Problem with MESI

- Shared state requires the data to be clean
  - i.e., all caches that have the block have the up-to-date copy and so does the memory
- Problem: Need to write the block to memory when BusRd happens when the block is in Modified state
- Why is this a problem?
  - Memory can be updated unnecessarily ☹ some other processor may want to write to the block again while it is cached

# Improving on MESI

- Idea 1: Do not transition from M<sub>S</sub> on a BusRd. Invalidate the copy and supply the modified block to the requesting processor directly without updating memory
- Idea 2: Transition from M<sub>S</sub>, but designate one cache as the owner (O), who will write the block back when it is evicted
  - Now “Shared” means “Shared and potentially dirty”
  - This is a version of the MOESI protocol

# Tradeoffs in Sophisticated Cache Coherence Protocols

- The protocol can be optimized with more states and prediction mechanisms to
  - + Reduce unnecessary invalidates and transfers of blocks
- However, more states and optimizations
  - Are more difficult to design and verify (lead to more cases to take care of, race conditions)
  - Provide diminishing returns

# Snoopy Cache vs. Directory Coherence

## Snoopy Cache

- + Miss latency (critical path) is short: miss  $\Rightarrow$  bus transaction to memory
- + Global serialization is easy: bus provides this already (arbitration)
- + Simple: adapt bus-based uniprocessors easily
- Relies on broadcast messages to be seen by all caches (in same order):
  - $\Rightarrow$  single point of serialization (bus): *not scalable*
  - $\Rightarrow$  *need a virtual bus (or a totally-ordered interconnect)*

# Snoopy Cache vs. Directory Coherence

## Directory

- Adds indirection to miss latency (critical path): request → dir. → mem.
- Requires extra storage space to track sharer sets
  - Can be approximate (false positives are OK)
- Protocols and race conditions are more complex (for high-performance)
- + Does not require broadcast to all caches
- + Exactly as scalable as interconnect and directory storage  
*(much more scalable than bus)*

# Review: Memory Consistency

- Lamport, “How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs,” IEEE Transactions on Computers, 1979 (less than 2 pages)
- Boehm et al., “Foundations of the C++ Concurrency Memory Model”, PLDI 2008 (11 pages)



# CSC 2224: Parallel Computer Architecture and Programming Memory Consistency & Cache Coherence

Prof. Gennady Pekhimenko

University of Toronto

Fall 2019

*The content of this lecture is adapted from the lectures of  
Onur Mutlu @ CMU*