# Computer Architecture: Cache Coherence

Prof. Onur Mutlu
Carnegie Mellon University

# Readings: Cache Coherence

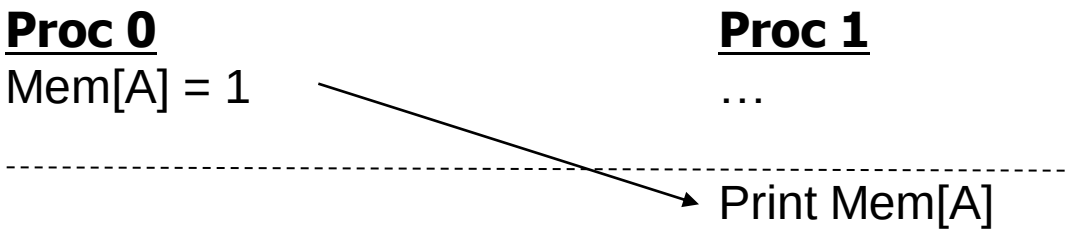
- Required
  - Culler and Singh, *Parallel Computer Architecture*
    - Chapter 5.1 (pp 269 – 283), Chapter 5.3 (pp 291 – 305)
  - P&H, *Computer Organization and Design*
    - Chapter 5.8 (pp 534 – 538 in 4th and 4th revised eds.)
  - Papamarcos and Patel, "A low-overhead coherence solution for multiprocessors with private cache memories," ISCA 1984.
  - Laudon and Lenoski, "The SGI Origin: a ccNUMA highly scalable server," ISCA 1997.

- Recommended
  - Censier and Feautrier, "A new solution to coherence problems in multicache systems," IEEE Trans. Comput., 1978.
  - Goodman, "Using cache memory to reduce processor-memory traffic," ISCA 1983.
  - Lenoski et al, "The Stanford DASH Multiprocessor," IEEE Computer, 25(3):63-79, 1992.
  - Martin et al, "Token coherence: decoupling performance and correctness," ISCA 2003.
  - Baer and Wang, "On the inclusion properties for multi-level cache hierarchies," ISCA 1988.

# Cache Coherence

# Shared Memory Model

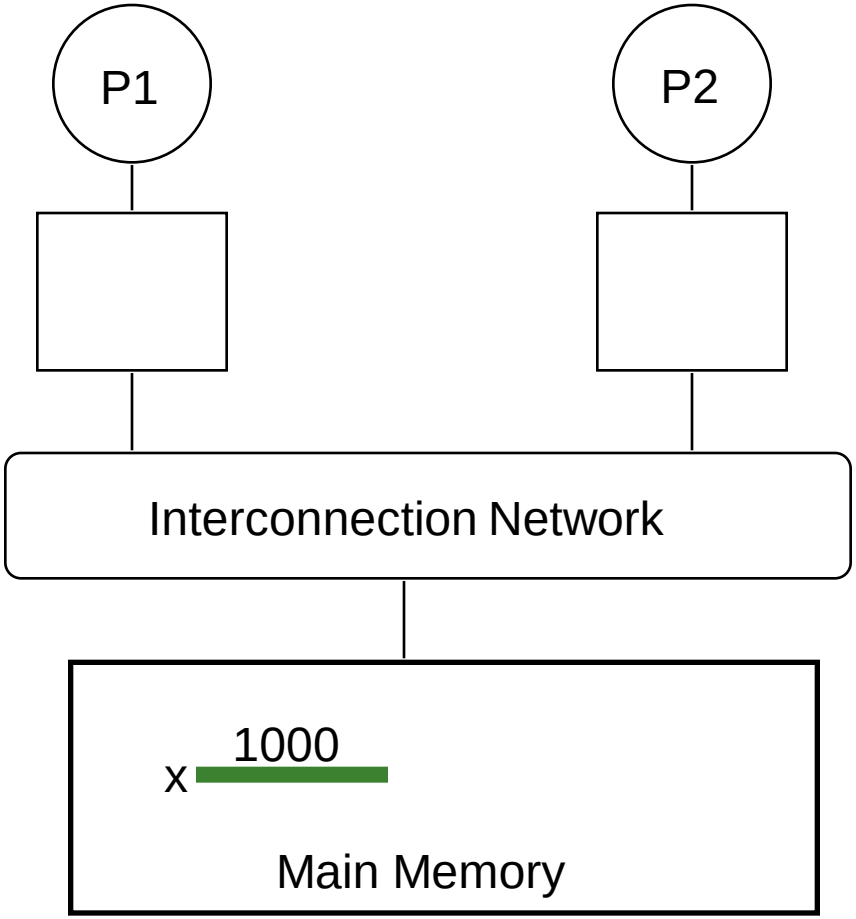
- Many parallel programs communicate through *shared memory*
- Proc 0 writes to an address, followed by Proc 1 reading
  - This implies communication between the two



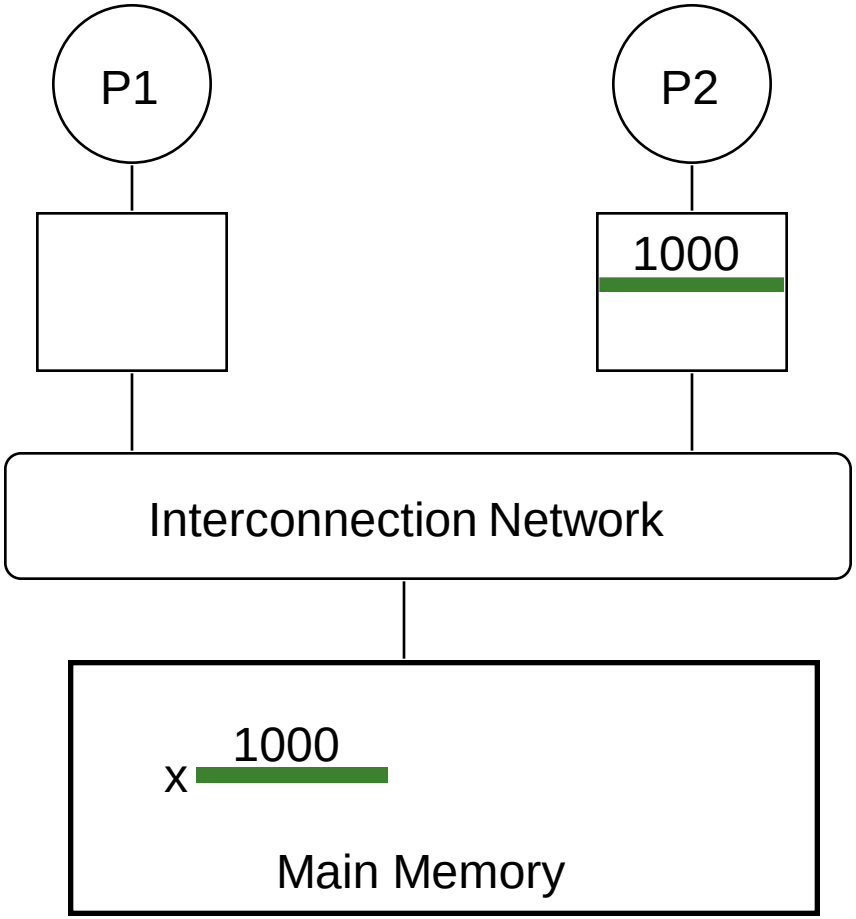

- Each read should receive the value last written by anyone
  - This requires synchronization (what does last written mean?)
- What if Mem[A] is cached (at either end)?

# Cache Coherence

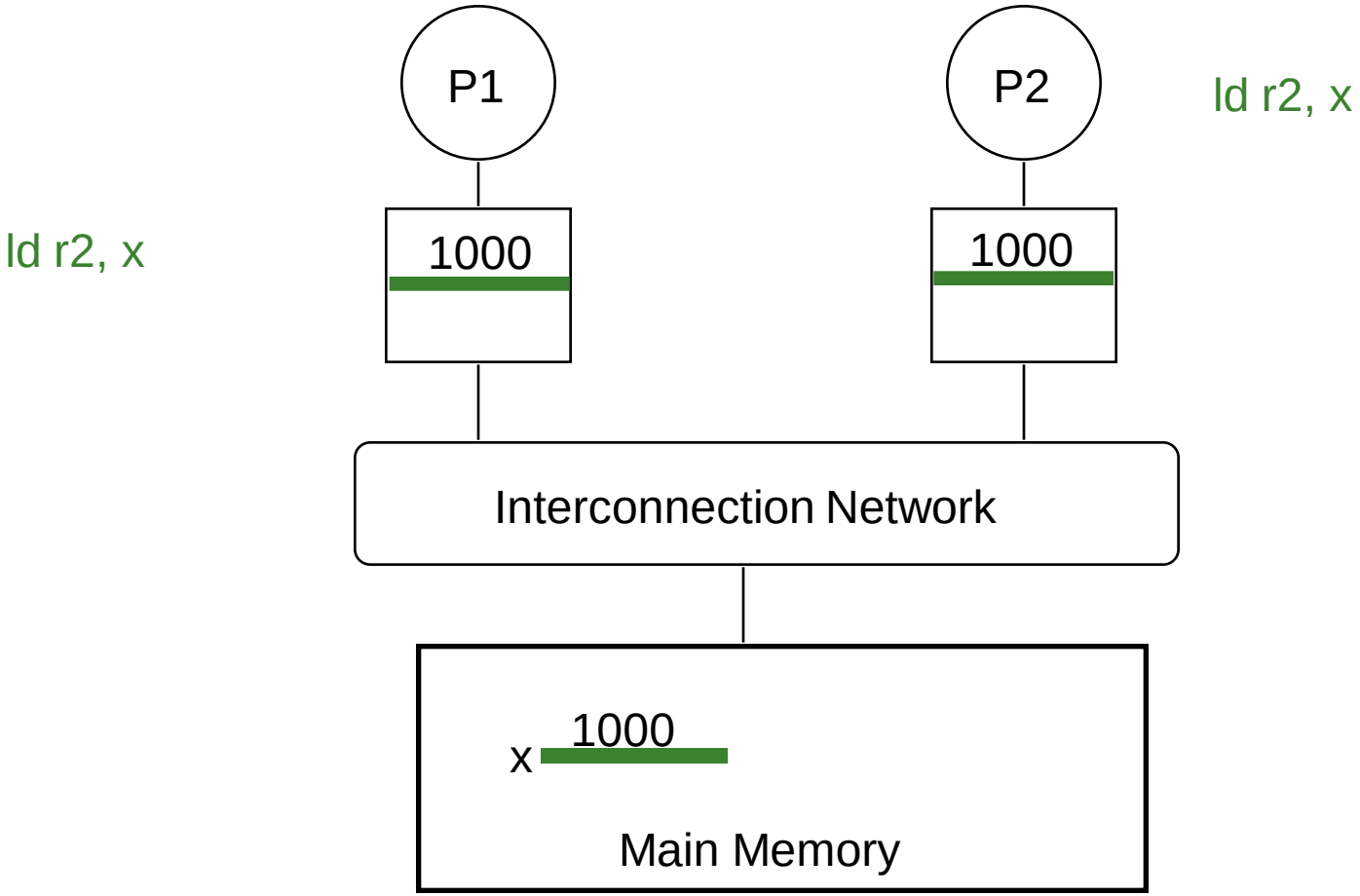
- Basic question: If multiple processors cache the same block, how do they ensure they all see a consistent state?



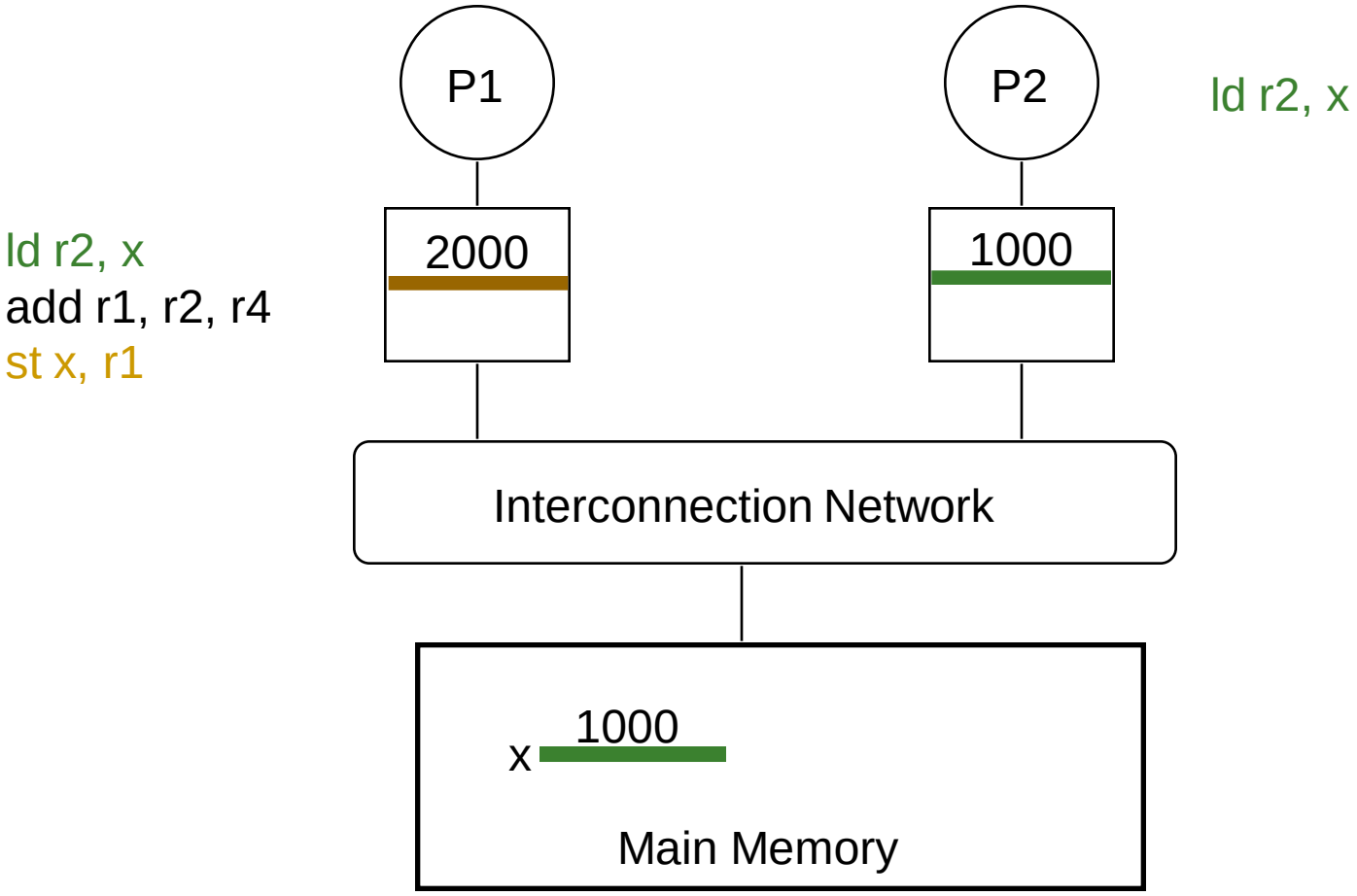
# The Cache Coherence Problem



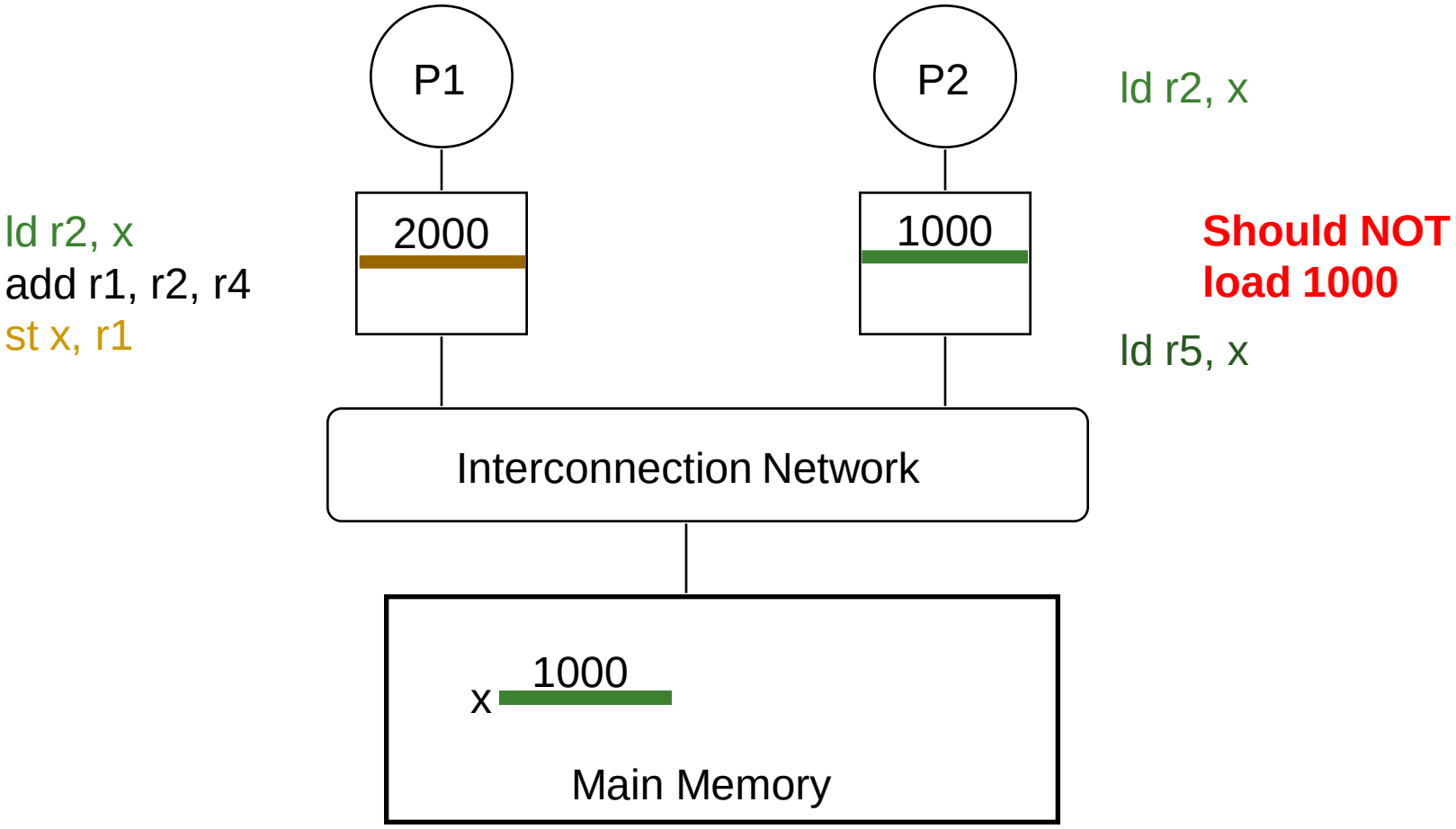
# The Cache Coherence Problem



# The Cache Coherence Problem



# The Cache Coherence Problem



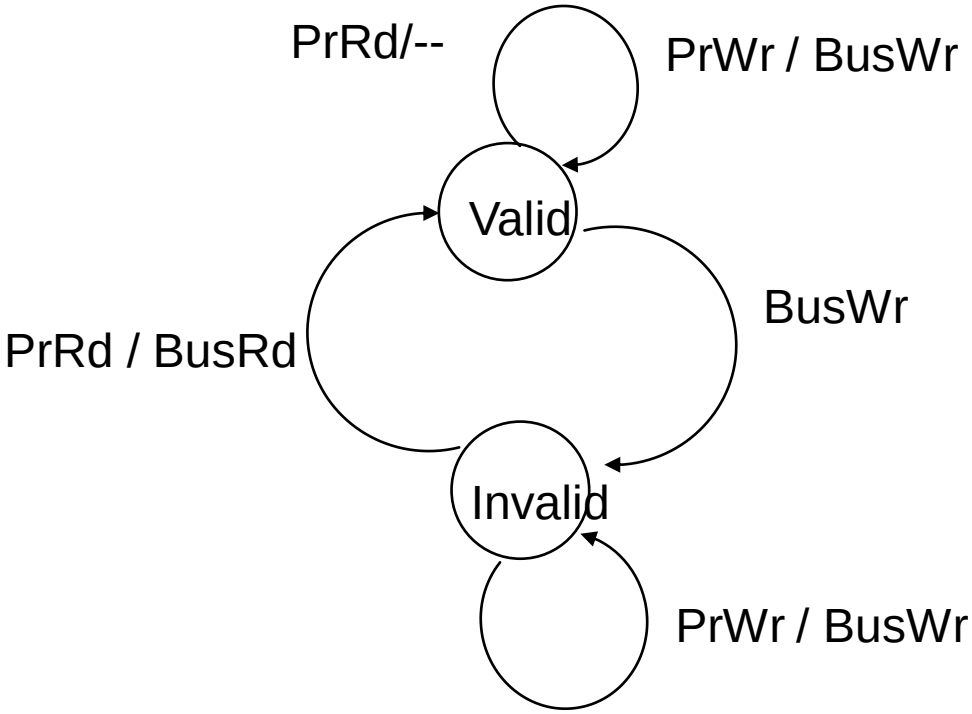
# Cache Coherence: Whose Responsibility?

- 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)

# Coherence: Update vs. Invalidate (II)

- **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

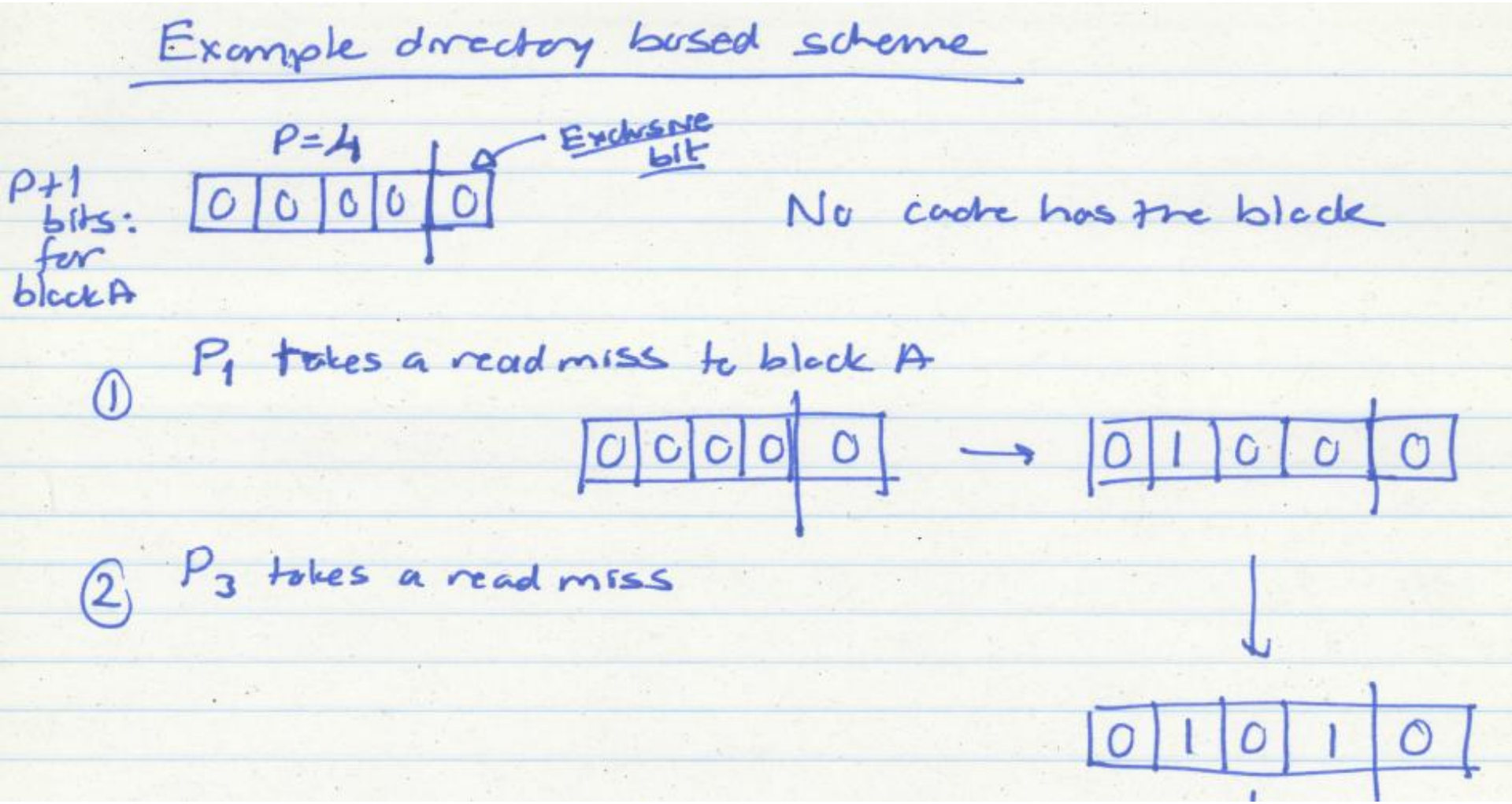
# Directory Based Cache Coherence

# 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 (I)



③ $P_2$ takes a write miss

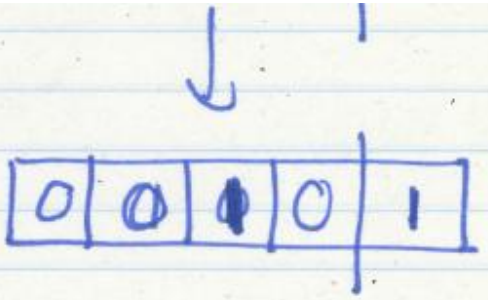
→ Invalidate $P_1$ & $P_3$'s caches

→ write request → $P_2$ has the exclusive copy of the block now. Set the Exclusive bit

→ $P_2$ can now update the block without notifying any other processor or the directory

→ $P_2$ needs to have a bit in its cache indicating it can perform exclusive updates to that block

→ private/exclusive bit per cache block

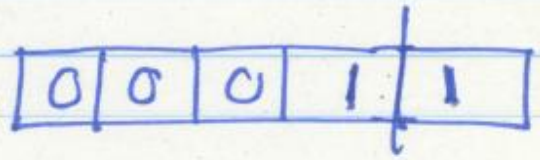



④ $P_3$ takes a write miss

→ Mem ~~Cntrller~~ Controller requests ~~true~~ block from $P_2$

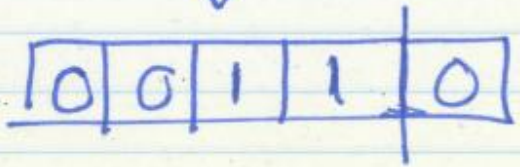
→ Mem Controller gives block to $P_3$

→ P2 invalidates its copy




⑤ $P_2$ takes a read miss

→ $P_3$ supplies it

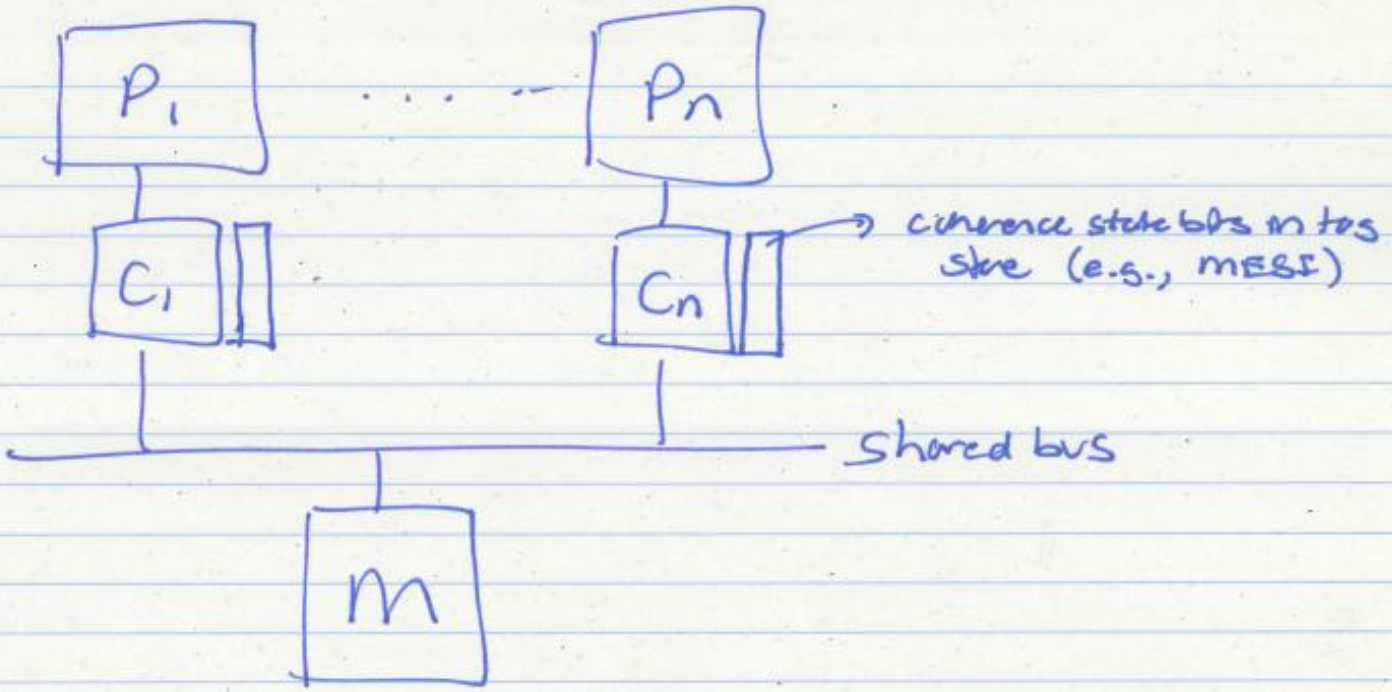


# Snoopy Cache Coherence

# 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

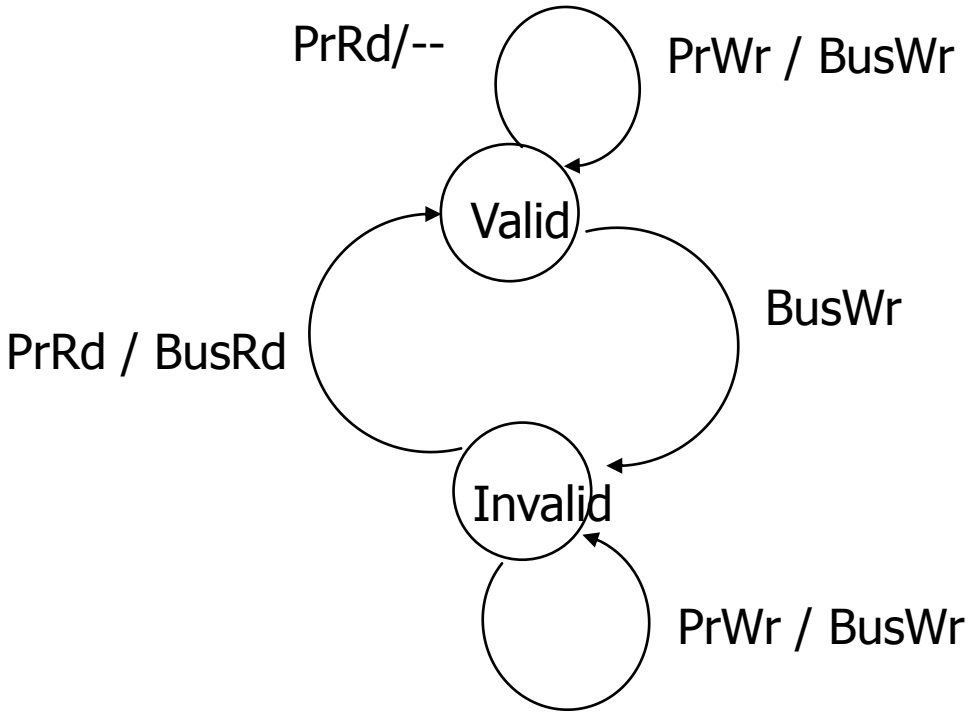
Each Cache observes its own processor & the bus
- Changes the state of the cached block based on observed actions by processor & the bus

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

Bus actions to a block (comms from another processor): BR (Bus Read)
BW (Bus Write)
or BRx (Bus Read Exclusive)

# A Simple Snoopy Cache Coherence Protocol

- Caches "snoop" (observe) each other's write/read operations
- A simple protocol:



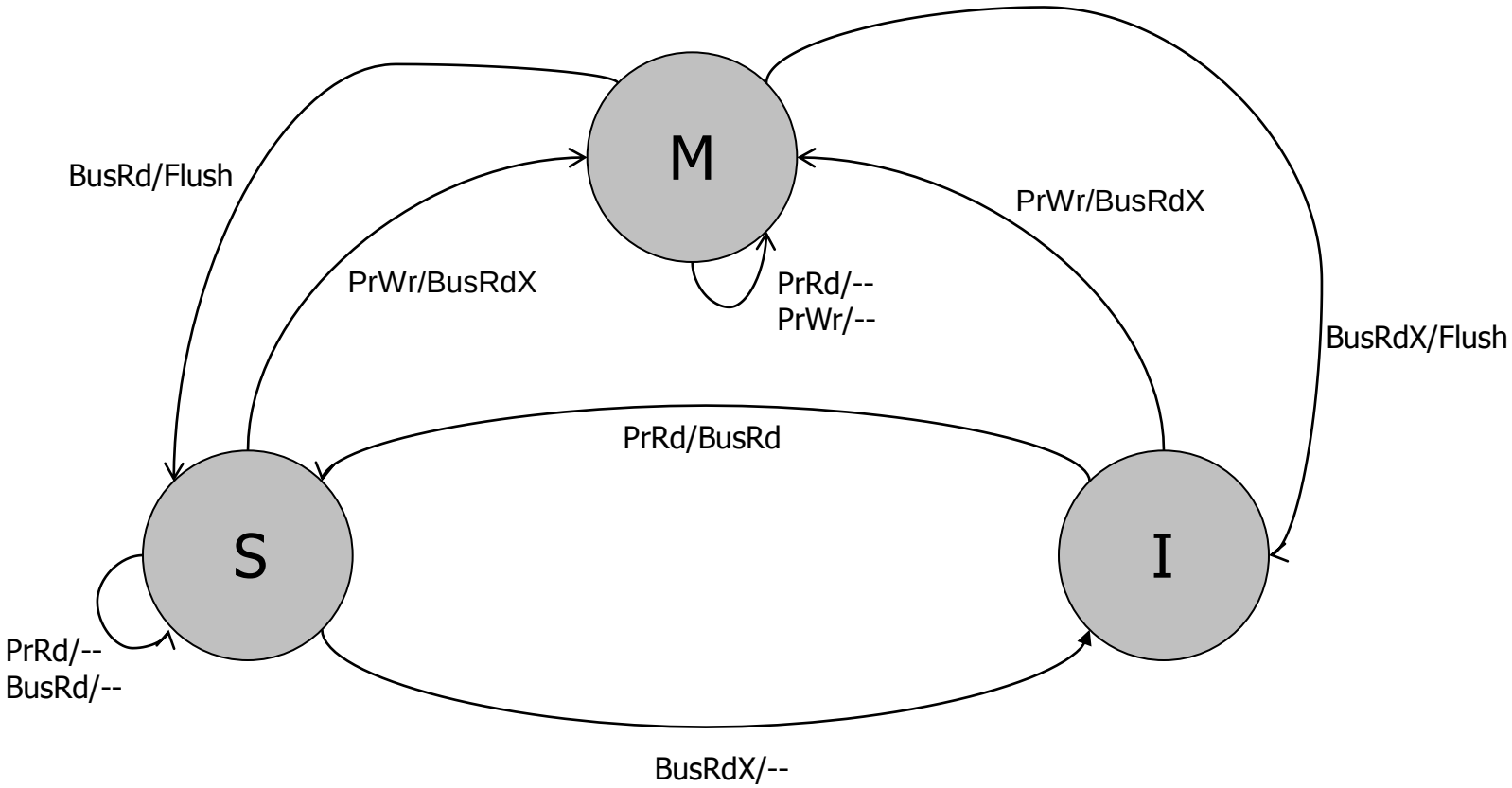

- Write-through, no-write-allocate cache
- Actions: PrRd, PrWr, BusRd, BusWr

# 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

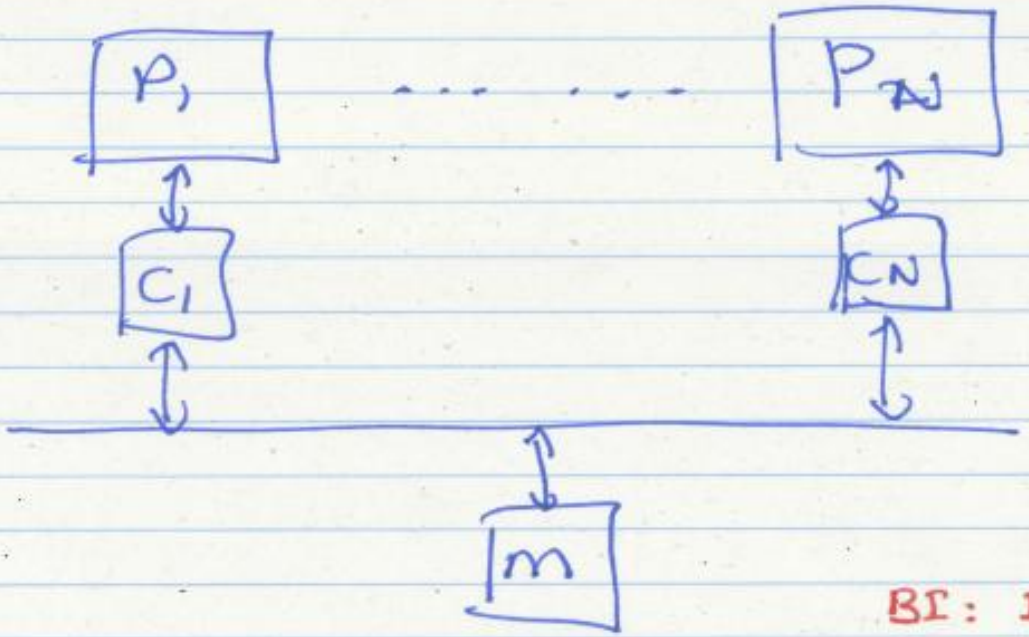
- Block is placed into the *exclusive* state if, during *BusRd*, no other cache had it
  - Wired-OR "shared" signal on bus can determine this: snooping caches assert the signal if they also have a copy

- Silent transition *Exclusive→Modified* is possible on write

  - MESI is also called the *Illinois protocol*
  - Papamarcos and Patel, "A low-overhead coherence solution for multiprocessors with private cache memories," ISCA 1984.

Papamarcos & Patel, ISCA 1984

Illinois Protocol



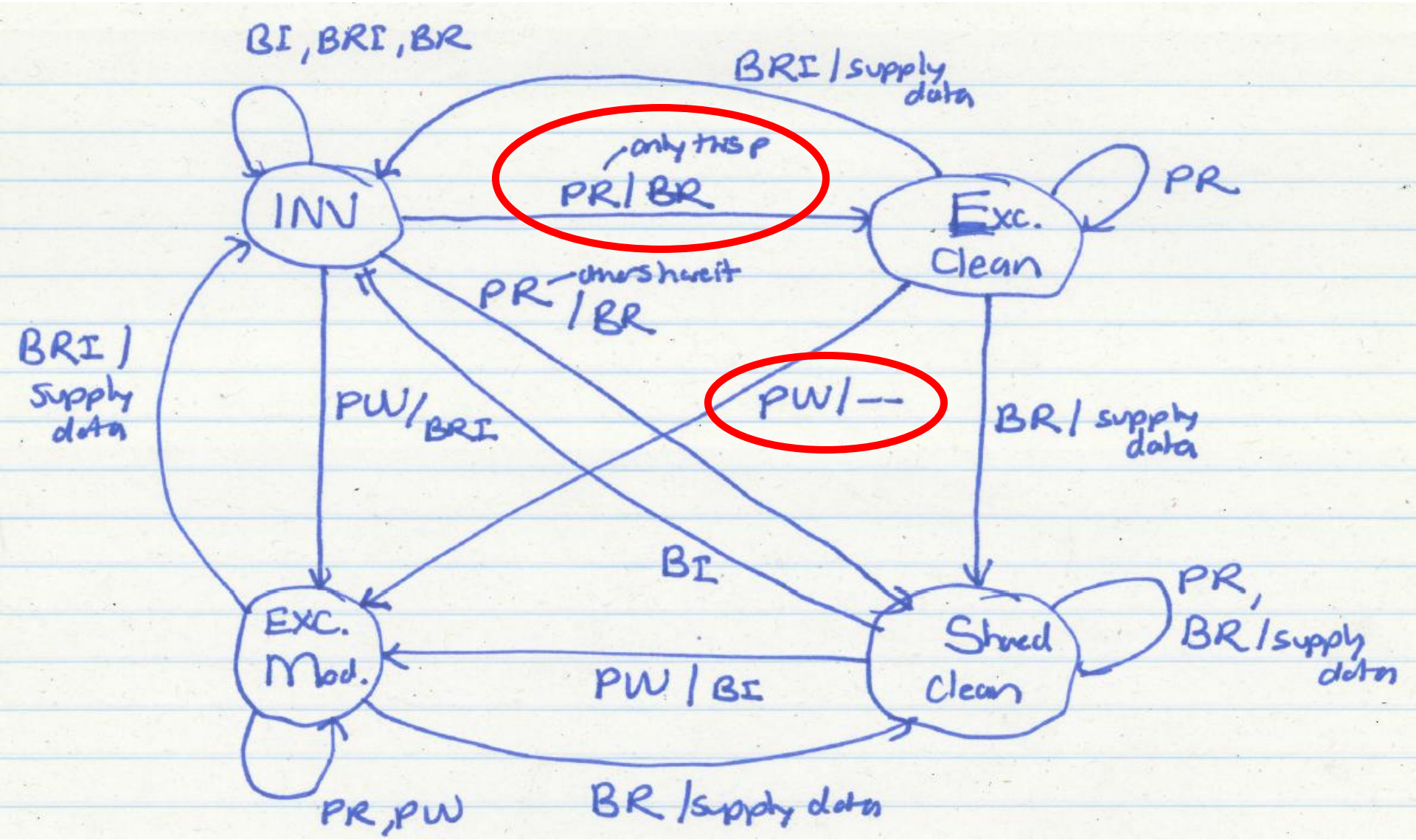

BI: Invalidate, but already have the data (do not supply it)

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

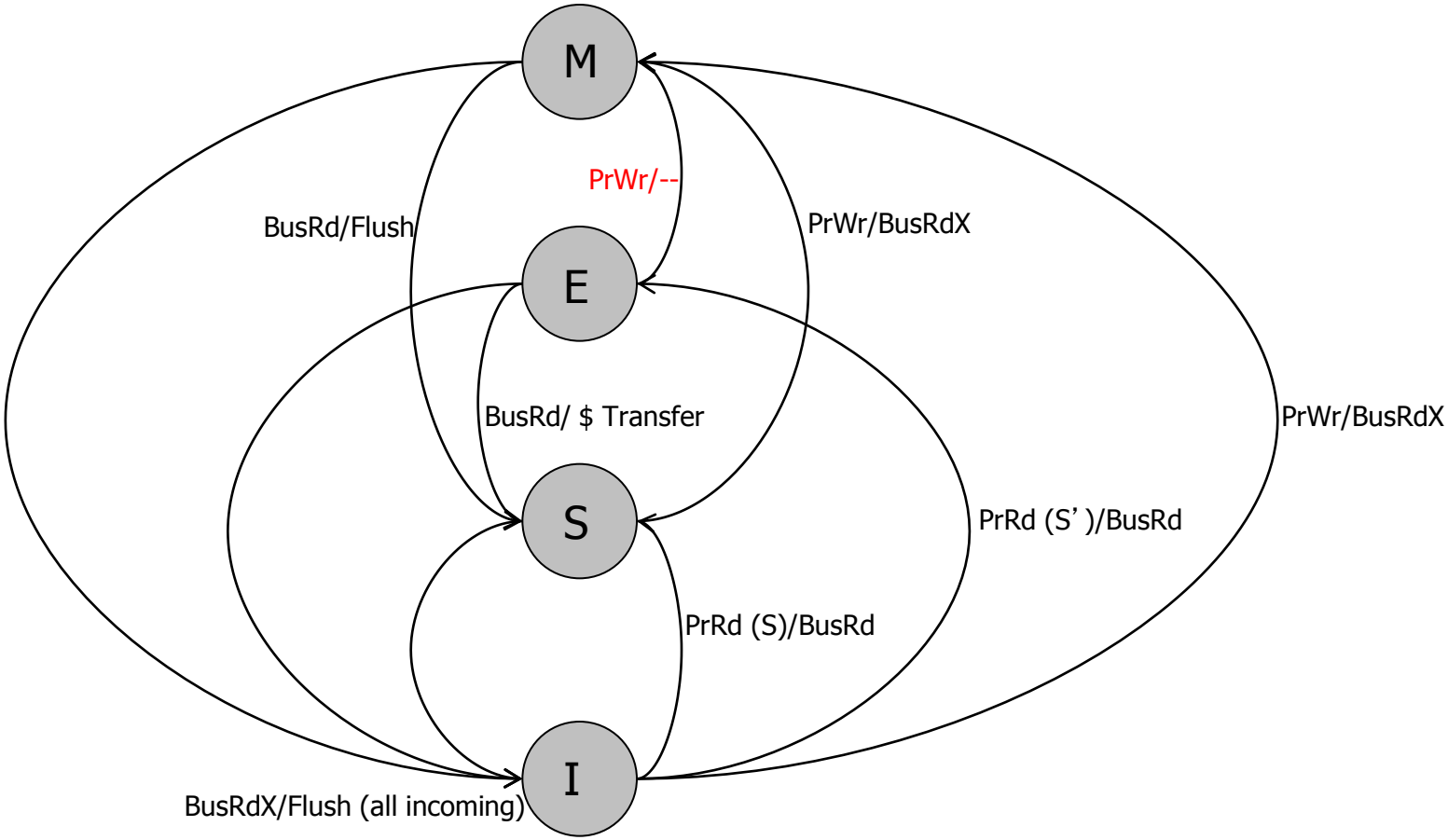
4 States

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

# MESI State Machine

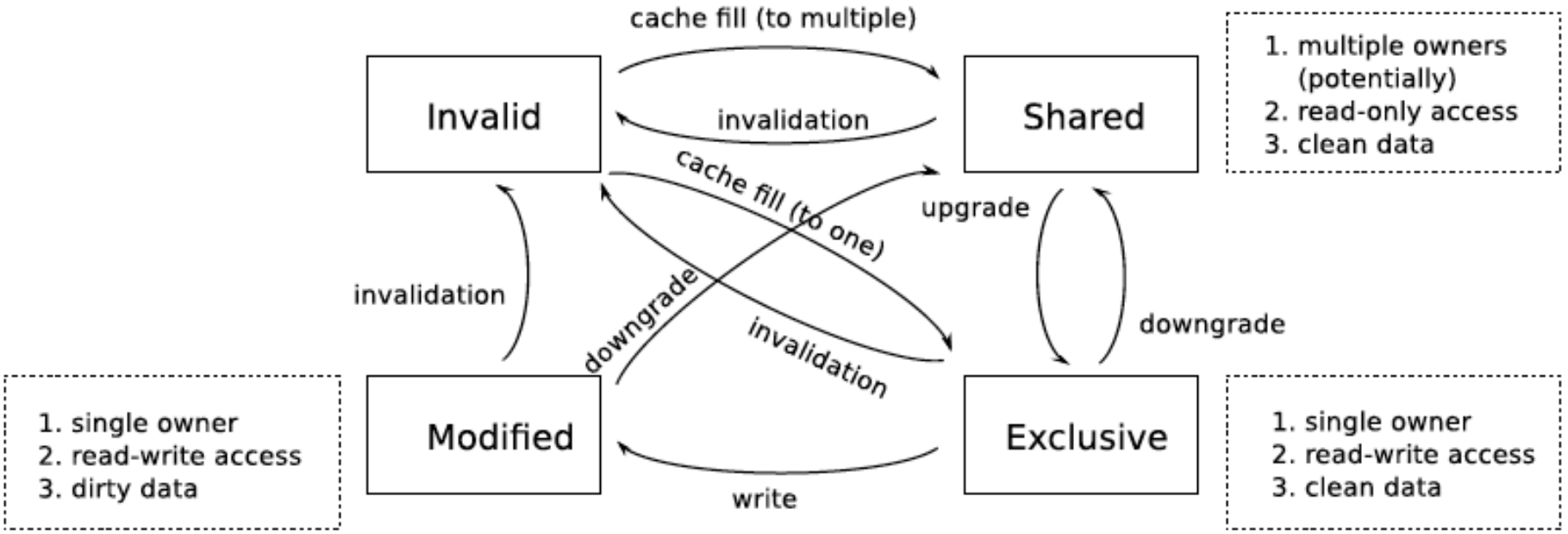


# MESI State Machine




[Culler/Singh96]

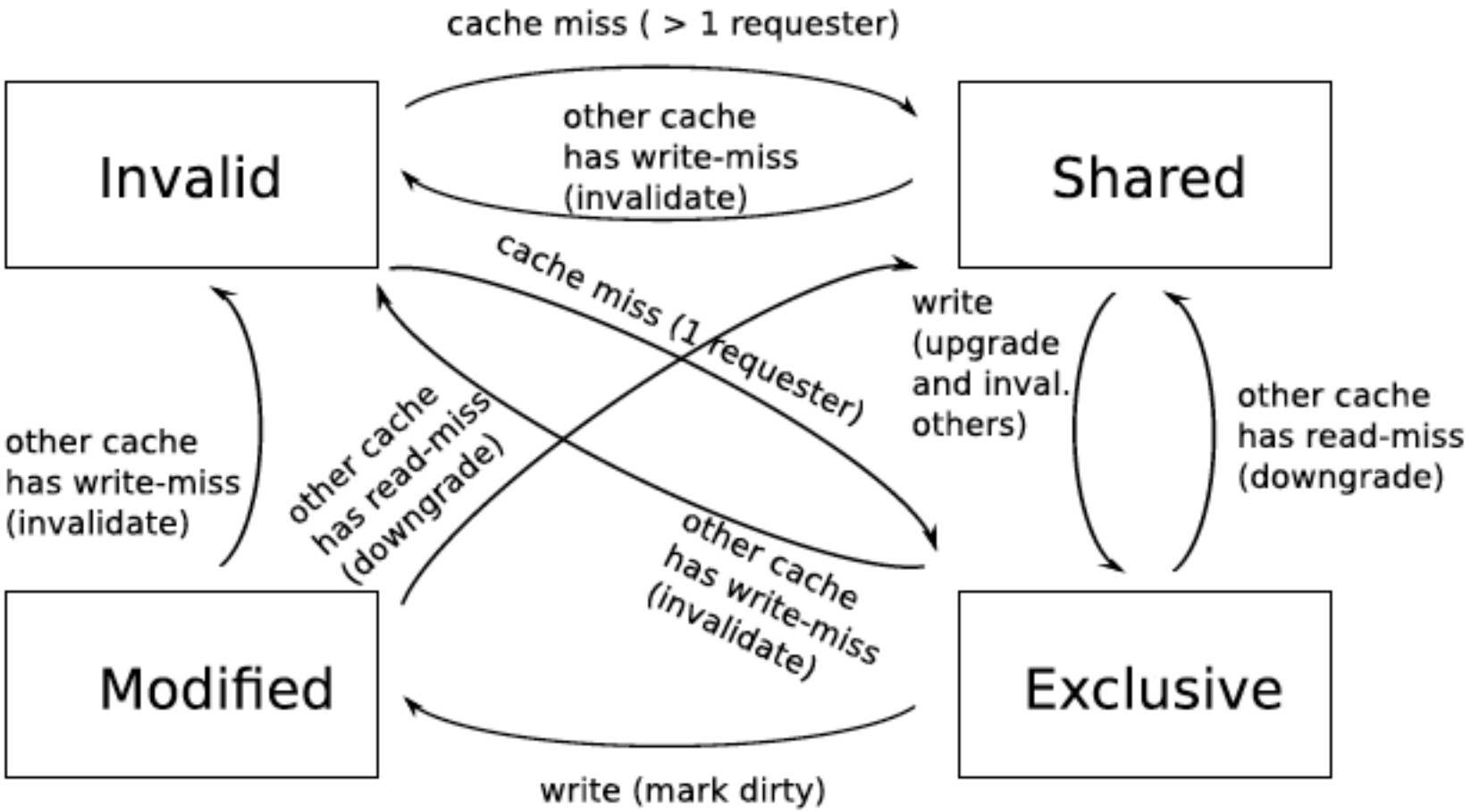
# MESI State Machine from 18-447 Lab 7



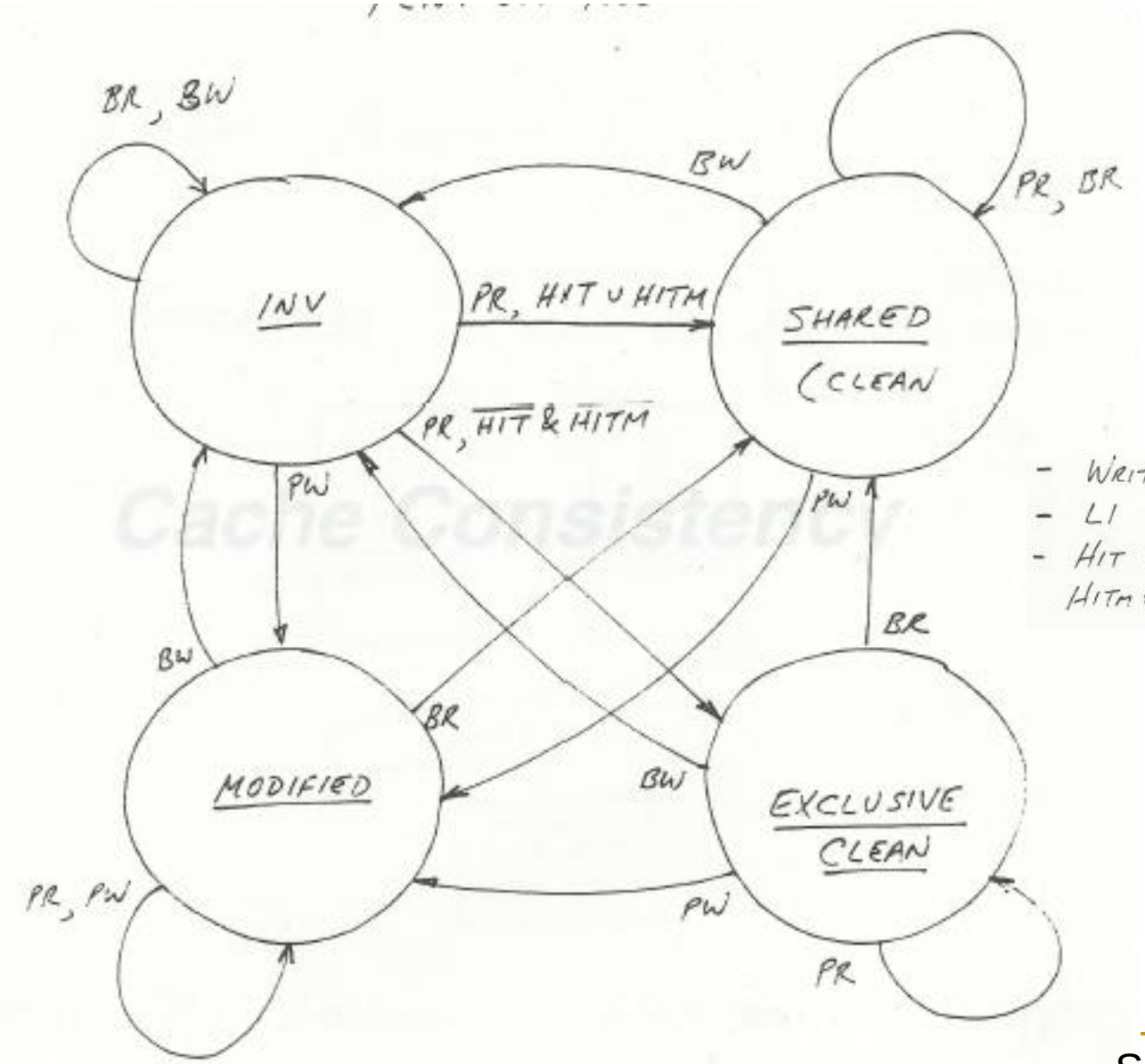

A transition from a single-owner state (Exclusive or Modified) to Shared is called a downgrade, because the transition takes away the owner's right to modify the data

A transition from Shared to a single-owner state (Exclusive or Modified) is called an upgrade, because the transition grants the ability to the owner (the cache which contains the respective block) to write to the block.

# MESI State Machine from 18-447 Lab 7



# Intel Pentium Pro




- Write Allocate
- L1 Can Have Data Not In L2
- Hit : Someone Has It Clean
  Hitm : Someone Has It Dirty

Slide credit: Yale Patt

# 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→S on a BusRd. Invalidate the copy and supply the modified block to the requesting processor directly without updating memory

- Idea 2: Transition from M→S, 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

# Revisiting 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

# Snoopy Cache vs. Directory Coherence

- **Snoopy Cache**
  - \+ Miss latency (critical path) is short: miss → 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):

    → single point of serialization (bus): *not scalable*

    → *need a virtual bus (or a totally-ordered interconnect)*

- **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)*

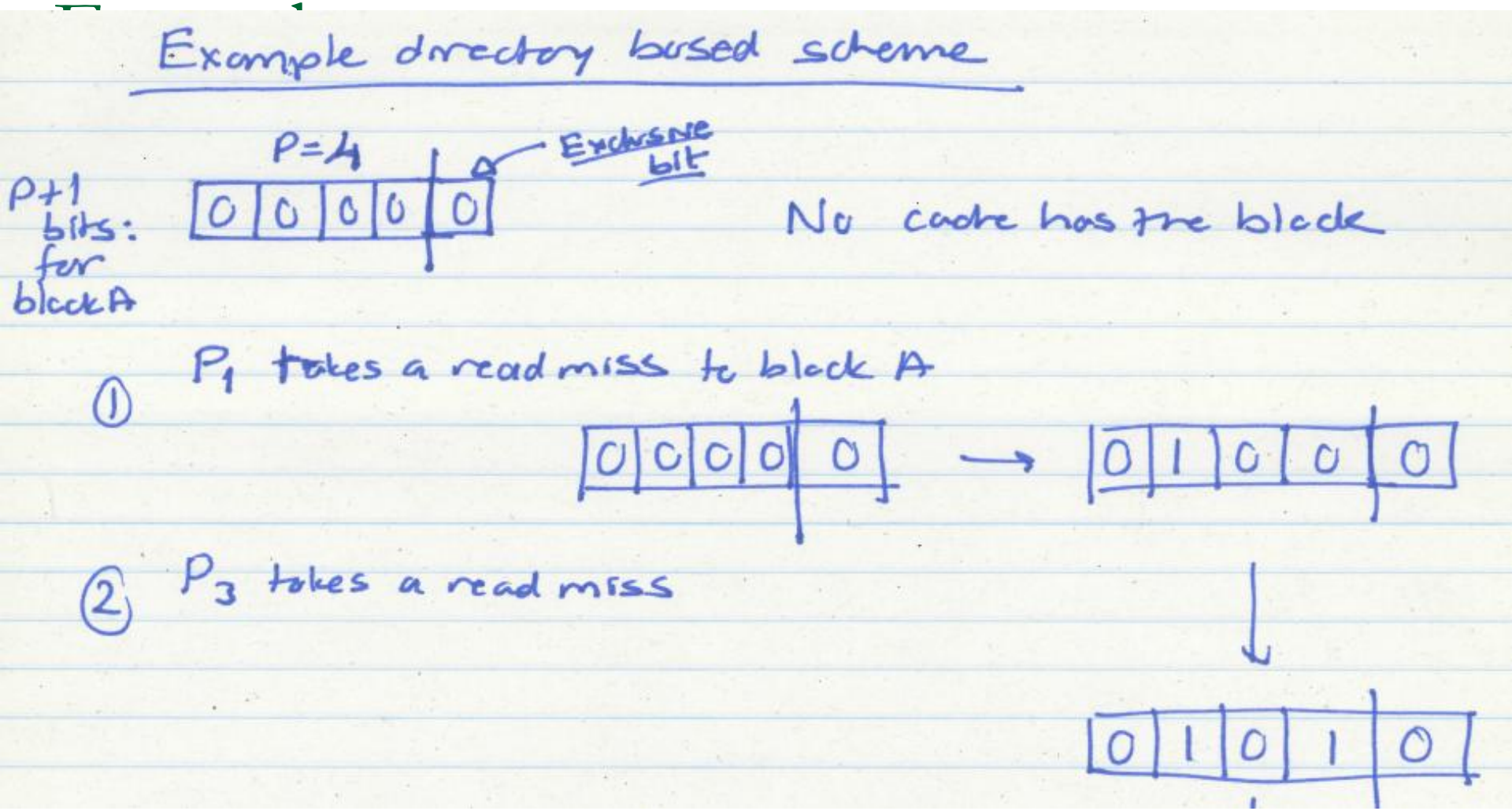
# Revisiting Directory-Based Cache Coherence

# Remember: 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 the cache that 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

# Remember: Directory Based Coherence



# Directory-Based Protocols

- Especially desirable when scaling the system past the capacity of a single bus
- Distributed, *but:*
  - Coherence still requires single point of serialization (for write serialization)
  - Serialization location can be different for every block (striped across nodes)

- We can reason about the protocol for a single block: one *server* (directory node), many *clients* (private caches)

- Directory receives *Read* and *ReadEx* requests, and sends *Invl* requests: invalidation is explicit (as opposed to snoopy buses)

# Directory: Data Structures

| | |
|---|---|
| 0x00 | Shared: {P0, P1, P2} |
| 0x04 | --- |
| 0x08 | Exclusive: P2 |
| 0x0C | --- |
| … | --- |

- Key operation to support is *set inclusion test*
  - False positives are OK: want to know which caches *may* contain a copy of a block, and spurious invalidations are ignored
  - False positive rate determines *performance*
- Most accurate (and expensive): full bit-vector
- Compressed representation, linked list, Bloom filters are all possible

# Directory: Basic Operations

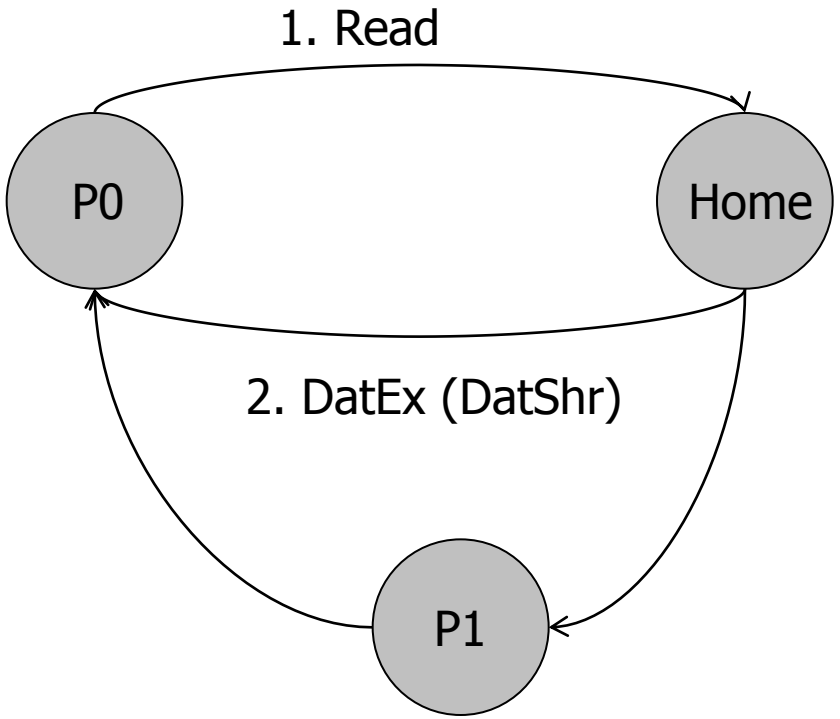
- Follow *semantics* of snoop-based system
  - but with explicit request, reply messages

- Directory:
  - Receives *Read, ReadEx, Upgrade* requests from nodes
  - Sends *Inval/Downgrade* messages to sharers if needed
  - Forwards request to memory if needed
  - Replies to requestor and updates sharing state

- Protocol design is flexible
  - Exact forwarding paths depend on implementation
  - For example, do cache-to-cache transfer?

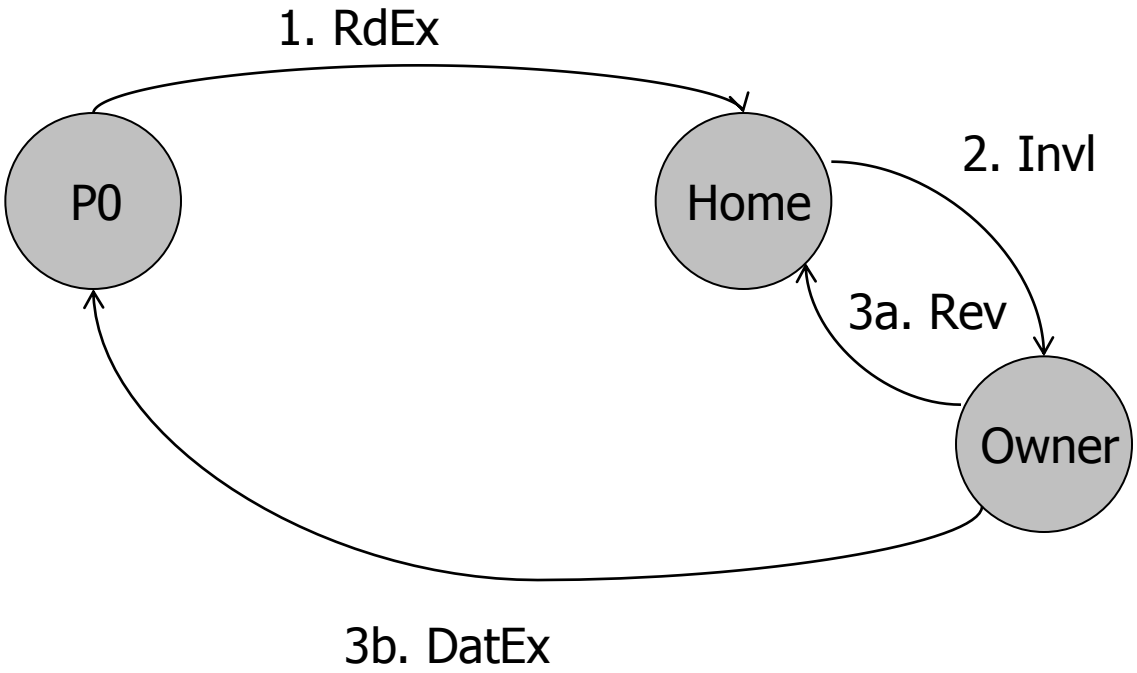
# MESI Directory Transaction: Read

P0 acquires an address for reading:

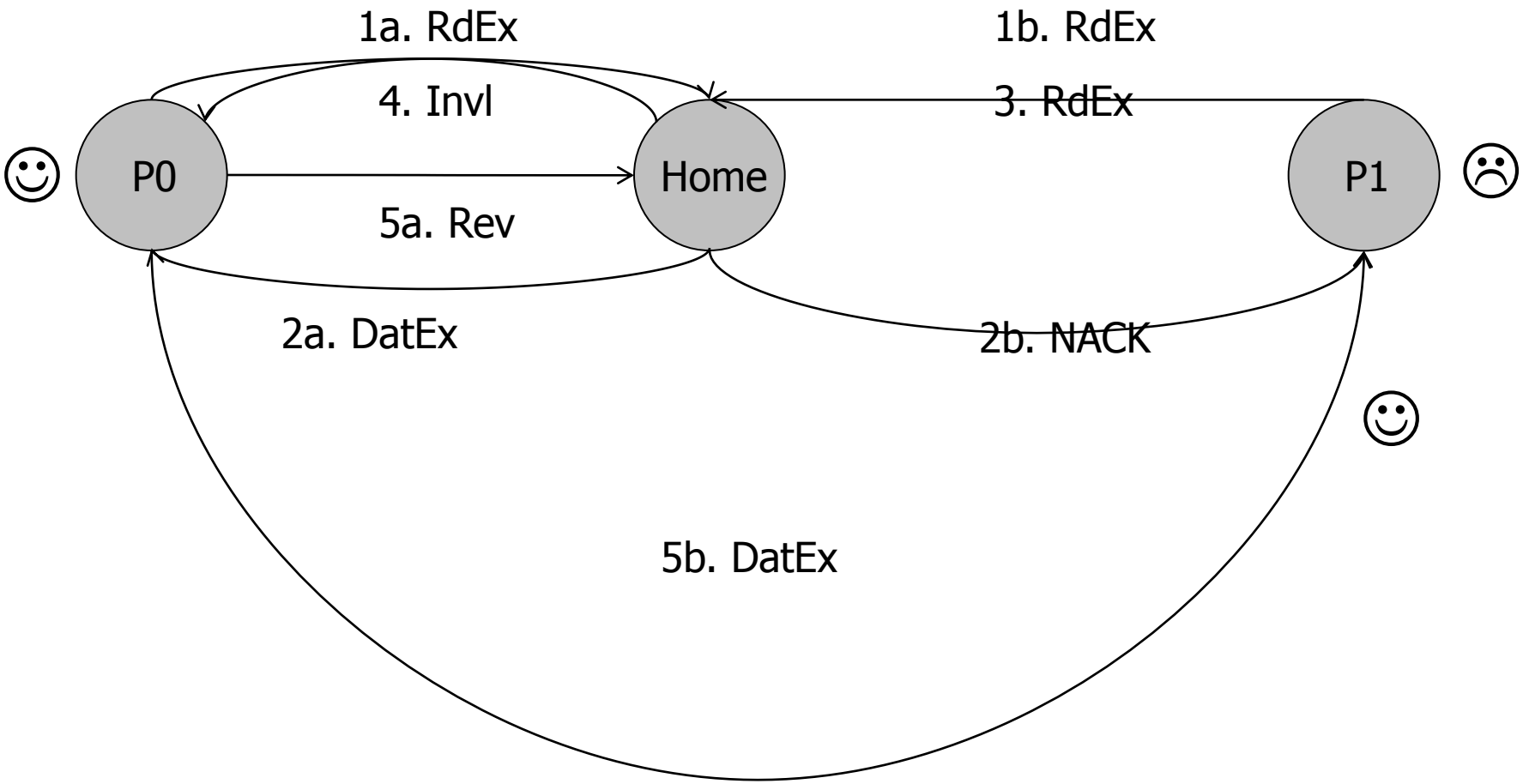



Culler/Singh Fig. 8.16

# RdEx with Former Owner



# Contention Resolution (for Write)



# Issues with Contention Resolution

- Need to escape race conditions by:
  - NACKing requests to busy (pending invalidate) entries
    - Original requestor retries
  - OR, queuing requests and granting in sequence
  - (Or some combination thereof)

- Fairness
  - Which requestor should be preferred in a conflict?
  - Interconnect delivery order, and distance, both matter

- Ping-ponging is a higher-level issue
  - With solutions like combining trees (for locks/barriers) and better shared-data-structure design

# Scaling the Directory: Some Questions

- How large is the directory?
- How can we reduce the access latency to the directory?
- How can we scale the system to thousands of nodes?
- Can we get the best of snooping and directory protocols?
  - Heterogeneity
  - E.g., token coherence [Martin+, ISCA 2003]

# Computer Architecture: Cache Coherence

Prof. Onur Mutlu
Carnegie Mellon University

# Backup slides

# Referenced Readings


- Papamarcos and Patel, "A low-overhead coherence solution for multiprocessors with private cache memories," ISCA 1984.
- Laudon and Lenoski, "The SGI Origin: a ccNUMA highly scalable server," ISCA 1997.
- Censier and Feautrier, "A new solution to coherence problems in multicache systems," IEEE Trans. Comput., 1978.
- Goodman, "Using cache memory to reduce processor-memory traffic," ISCA 1983.
- Lenoski et al, "The Stanford DASH Multiprocessor," IEEE Computer, 25(3):63-79, 1992.
- Martin et al, "Token coherence: decoupling performance and correctness," ISCA 2003.
- Baer and Wang, "On the inclusion properties for multi-level cache hierarchies," ISCA 1988.

# Other Recommended Readings (Research)

- Kelm et al., "WAYPOINT: scaling coherence to thousand-core architectures," PACT 2010.
- Kelm et al., "Cohesion: a hybrid memory model for accelerators," ISCA 2010.
- Martin et al, "Token coherence: decoupling performance and correctness," ISCA 2003.

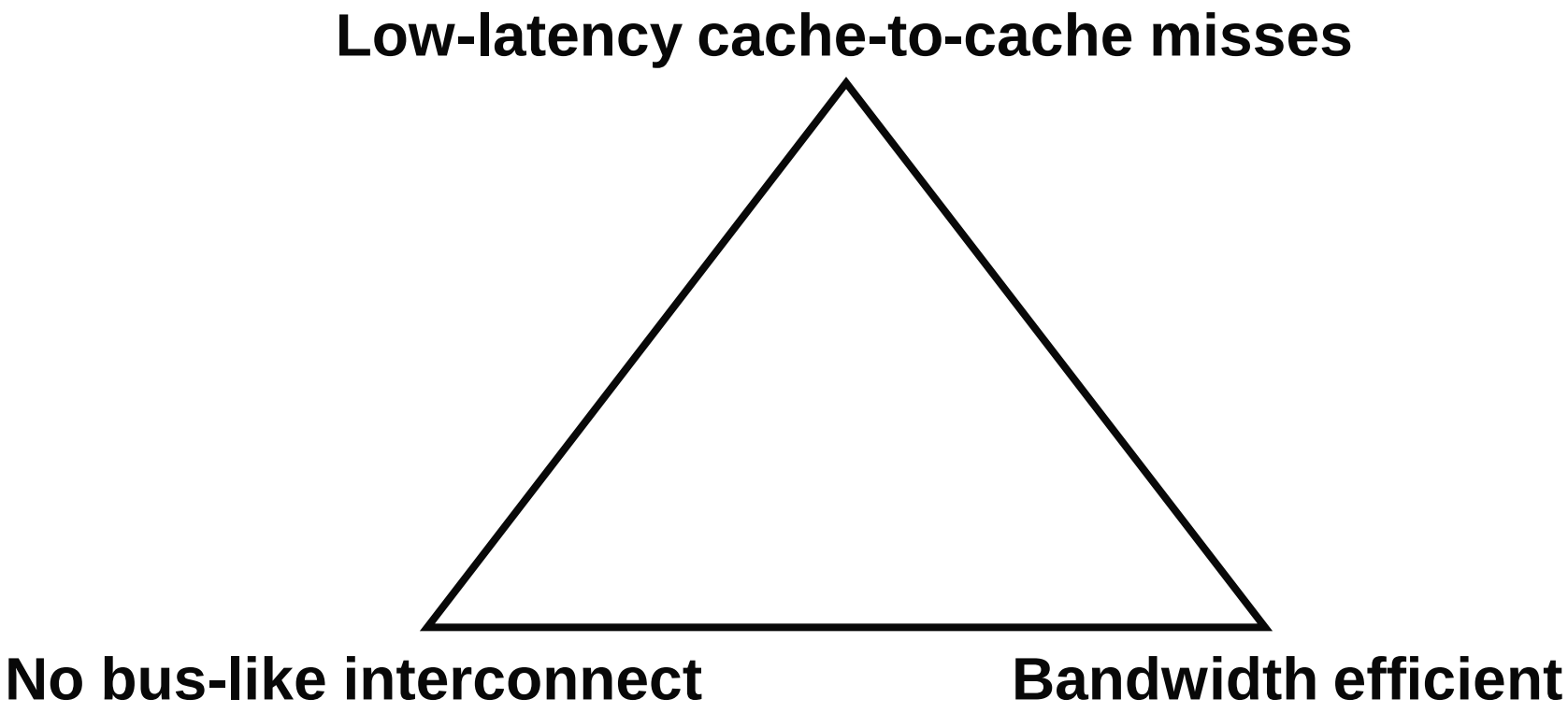
# Related Videos

- Multiprocessor Correctness and Cache Coherence
  - http://www.youtube.com/watch?v=U-VZKMgItDM
  - http://www.youtube.com/watch?v=6xEpbFVgnf8&list=PL5PHm2jkkXmidJOd59REog9jDnPDTG6IJ&index=33

# Related Exam Questions

- Question 5 in
  - http://www.ece.cmu.edu/~ece447/s13/lib/exe/fetch.php?media=final.pdf
- Question I-11 in
  - http://www.ece.cmu.edu/~ece447/s12/lib/exe/fetch.php?media=wiki:18447-final.pdf

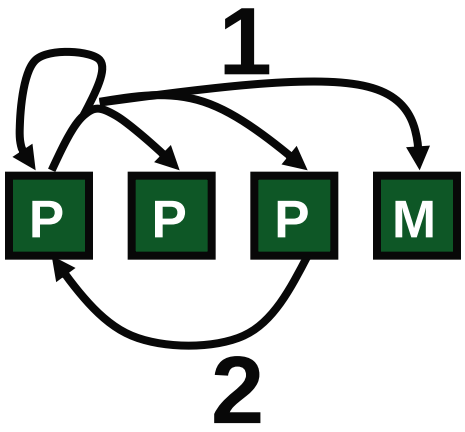
# Motivation: Three Desirable Attributes




**Dictated by workload and technology trends**

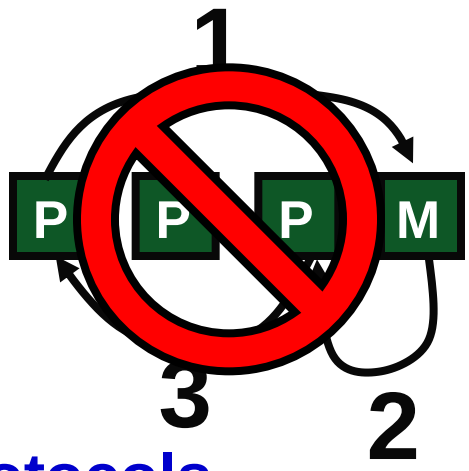
# Workload Trends

- Commercial workloads
  - Many cache-to-cache misses
  - Clusters of small multiprocessors




- Goals:
  - **Direct cache-to-cache misses (2 hops, not 3 hops)**
  - Moderate scalability

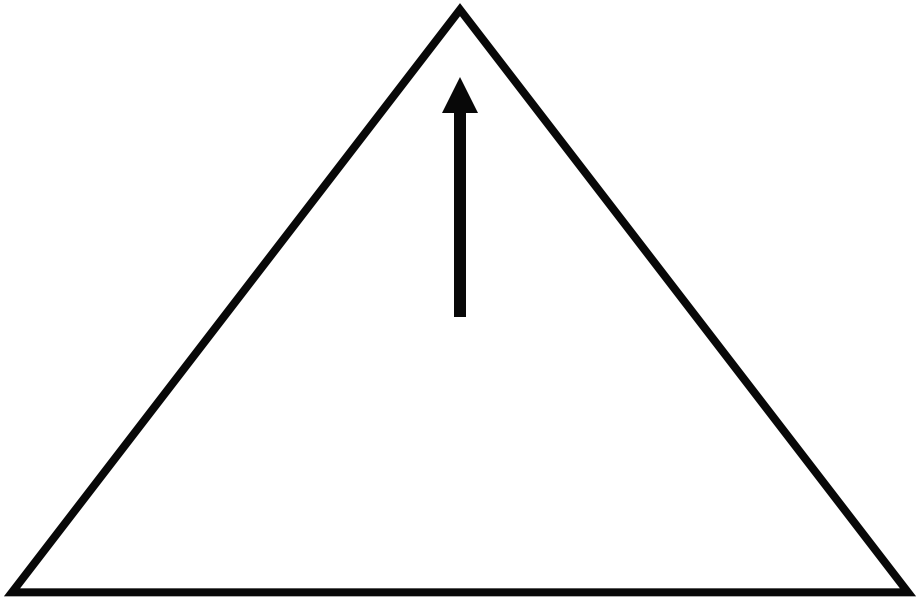
**Directory Protocol**




**Workload trends → snooping protocols**

# Workload Trends

**Low-latency cache-to-cache misses**



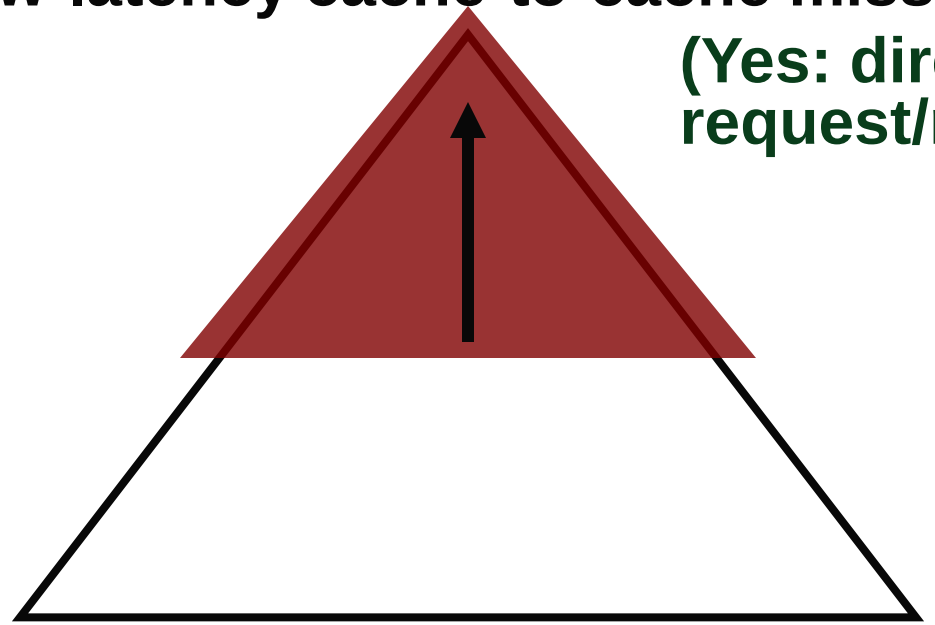
**No bus-like interconnect**

**Bandwidth efficient**

# Workload Trends  Snooping Protocols

**Low-latency cache-to-cache misses**

**(Yes: direct request/response)**



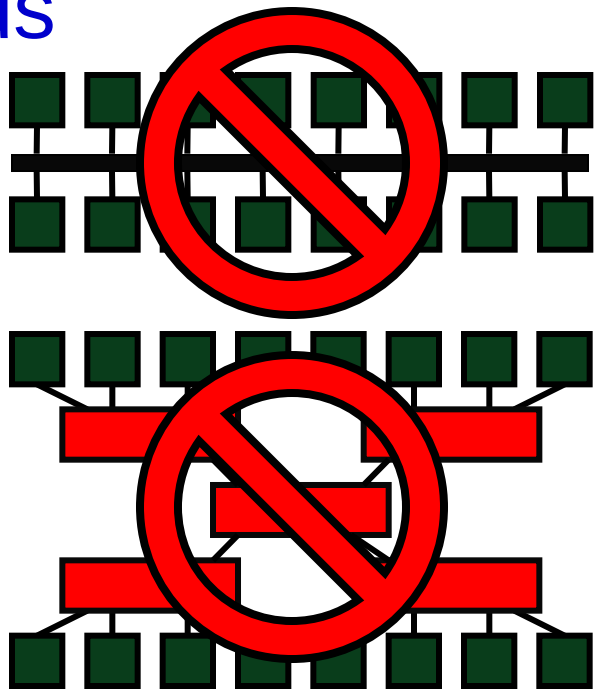
**No bus-like interconnect**

**(No: requires a "virtual bus")**

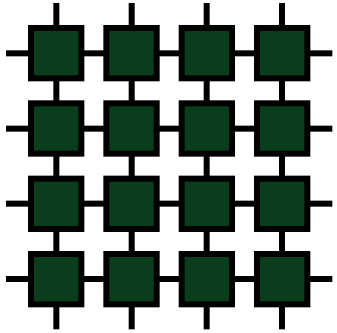
**Bandwidth efficient**

**(No: broadcast always)**

# Technology Trends



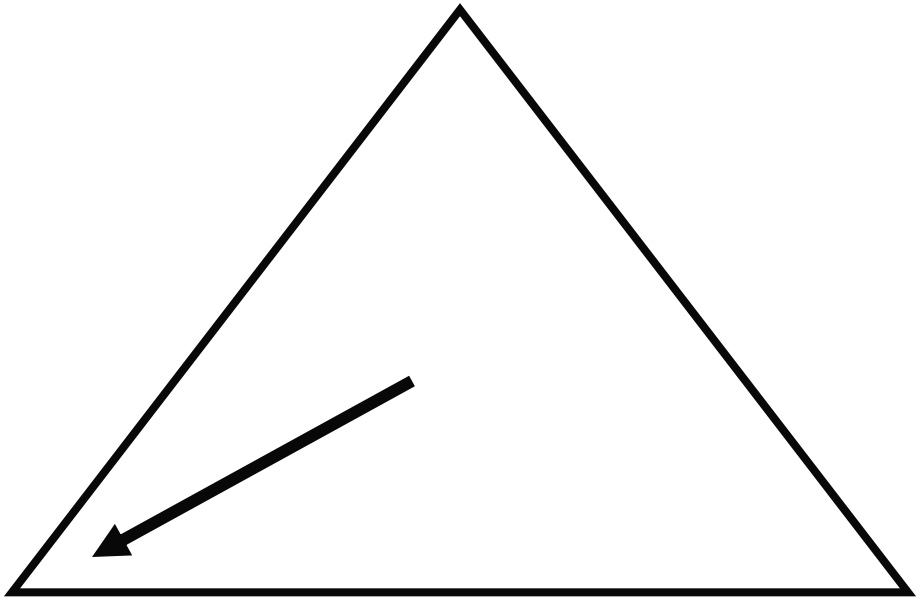
- High-speed point-to-point links
  - No (multi-drop) busses
- Increasing design integration
  - "Glueless" multiprocessors
  - Improve cost & latency
- Desire: low-latency interconnect
  - **Avoid "virtual bus" ordering**
  - **Enabled by directory protocols**



**Technology trends → unordered interconnects**

# Technology Trends

**Low-latency cache-to-cache misses**

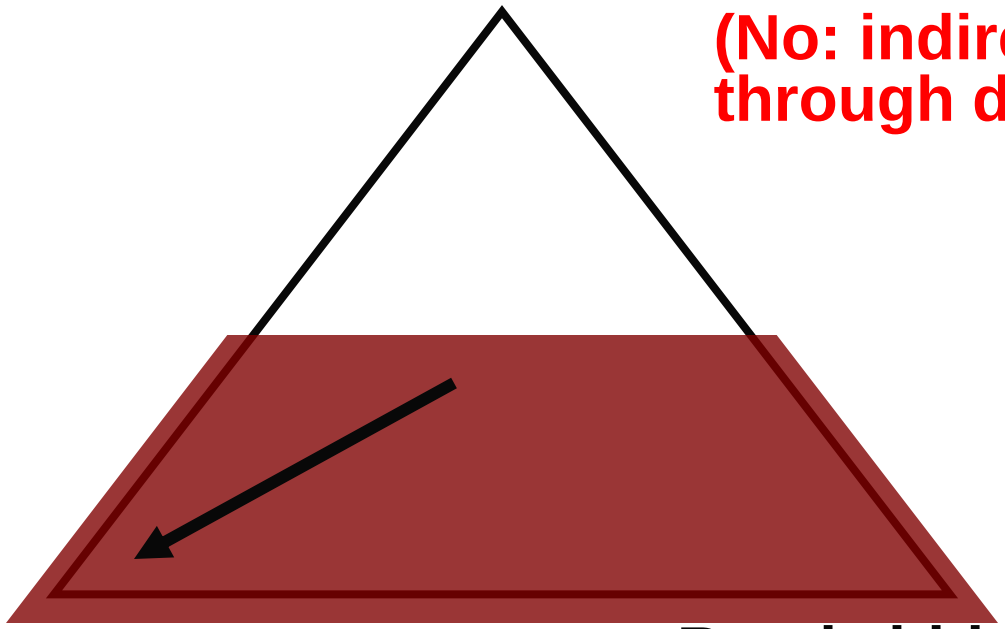


**No bus-like interconnect** **Bandwidth efficient**

# Technology Trends  Directory Protocols

**Low-latency cache-to-cache misses**

**(No: indirection through directory)**



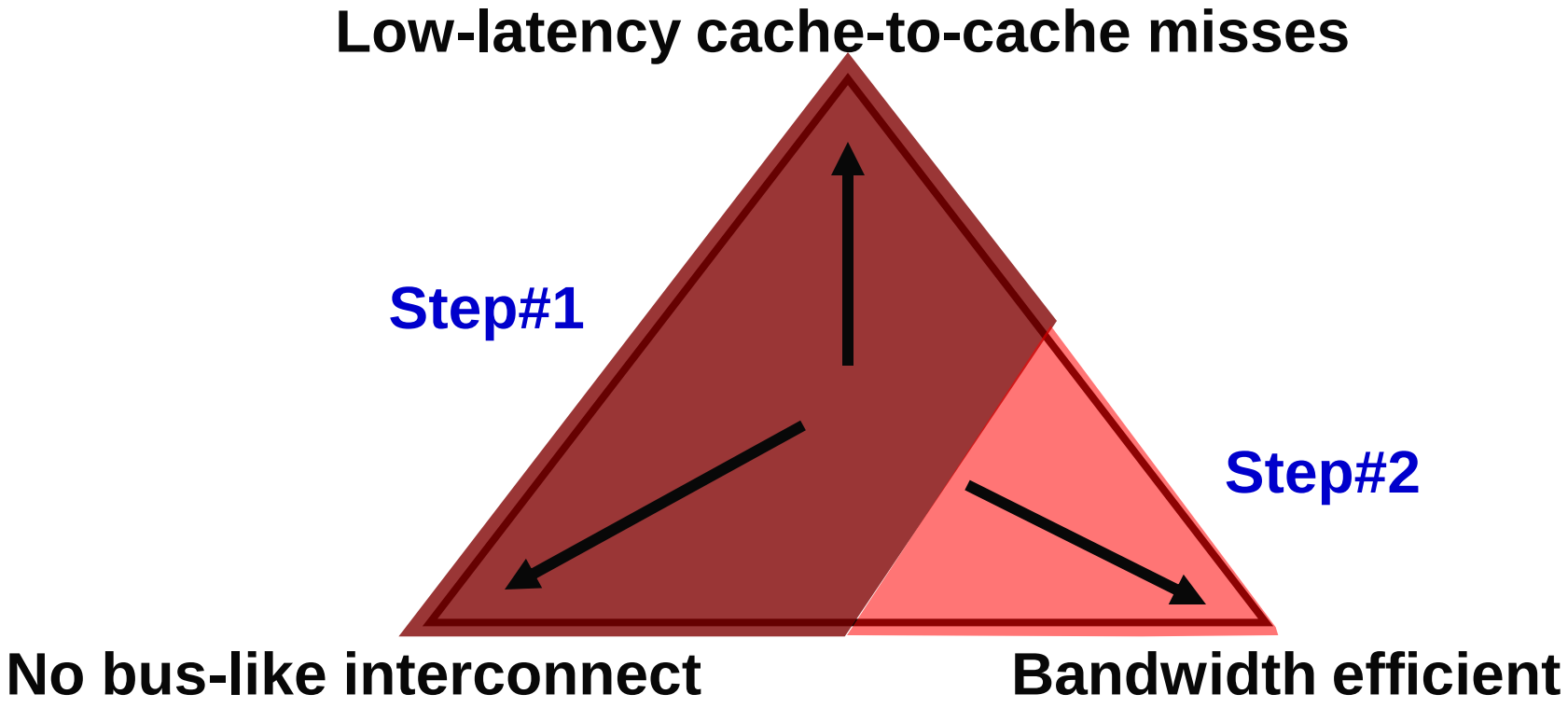
**No bus-like interconnect**

**(Yes: no ordering required)**

**Bandwidth efficient**

**(Yes: avoids broadcast)**

# Goal: All Three Attributes



# Token Coherence: Key Insight

- Goal of invalidation-based coherence
  - Invariant: **many readers -or- single writer**
  - Enforced by **globally** coordinated actions

Key insight

- Enforce this invariant directly using **tokens**
  - **Fixed number of tokens** per block
  - **One token to read, all tokens to write**

- Guarantees **safety** in all cases
  - Global invariant enforced with only **local** rules
  - Independent of races, request ordering, etc.

# Token Coherence: Contributions

1. **Token counting** rules for enforcing safety

2. **Persistent requests** for preventing starvation

3. **Decoupling correctness and performance** in cache coherence protocols
   - Correctness Substrate
   - Performance Policy

4. **Exploration of three performance policies**