Computer Architecture

Lecture 20: Multiprocessors, Consistency, Cache Coherence (II)

Prof. Onur Mutlu
ETH Zürich

Fall 2017

30 November 2017

Summary of Yesterday

- Multiprocessor Basics
- Memory Ordering (Consistency)
- Cache Coherence

Today

- Cache Coherence Continued
- Review Session

Cache Coherence

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.

- Recommended
  - Censier and Feautrier, "A new solution to coherence problems in multicache systems," IEEE Trans. Computers, 1978.
  - Goodman, "Using cache memory to reduce processor-memory traffic," ISCA 1983.
  - Laudon and Lenoski, "The SGI Origin: a ccNUMA highly scalable server," ISCA 1997.
  - 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.

Review: 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 memory 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 which caches have each block
  - Directory coordinates invalidation and updates
    - E.g.: P1 asks directory for exclusive copy, directory asks P0 to invalidate, waits for ACK, then responds to P1

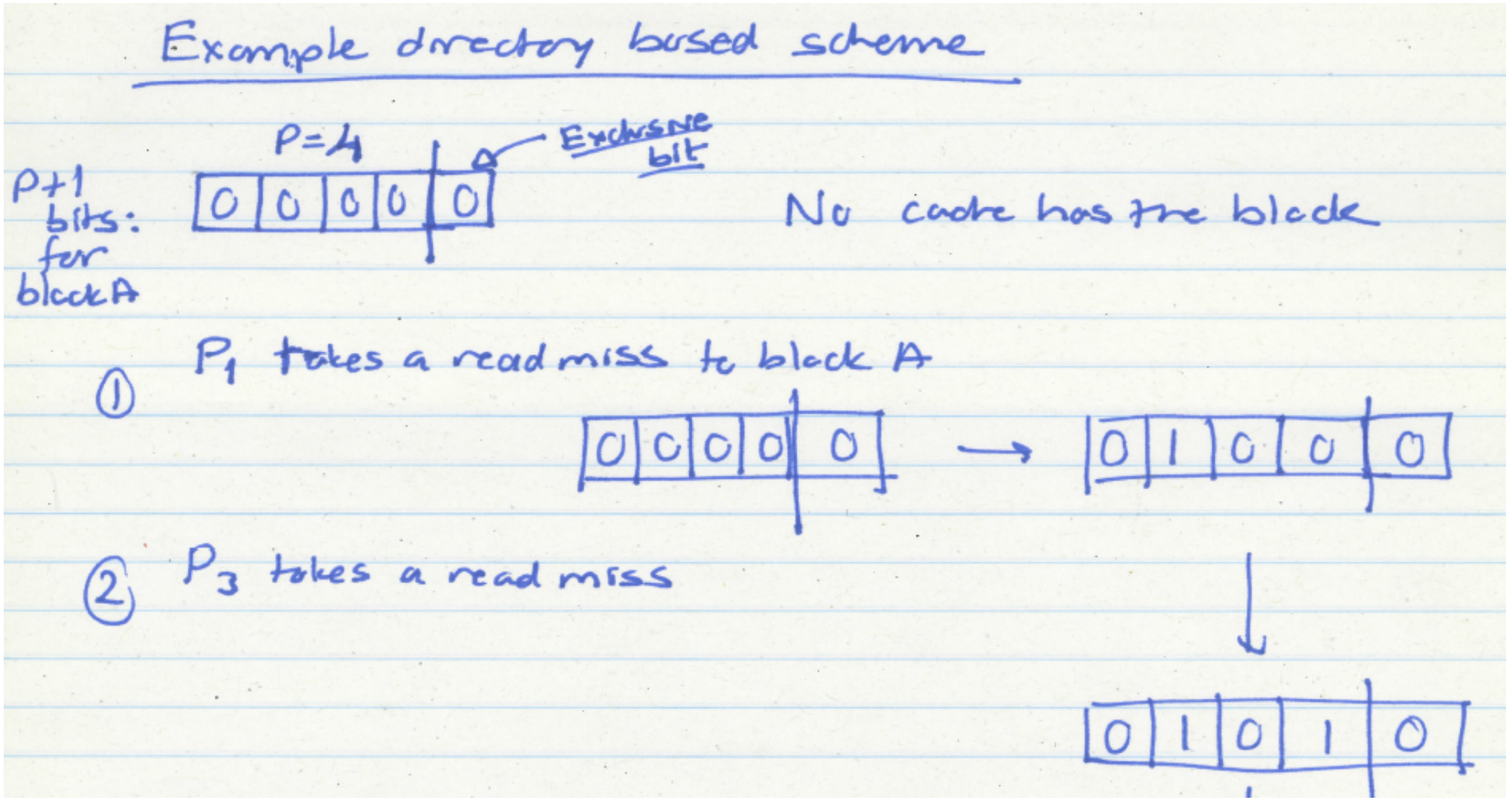
Directory Based Cache Coherence

Review: 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 (so that the cache can update the exclusive block silently)

Directory Based Coherence Example (I)



(3) P_2 takes a write miss

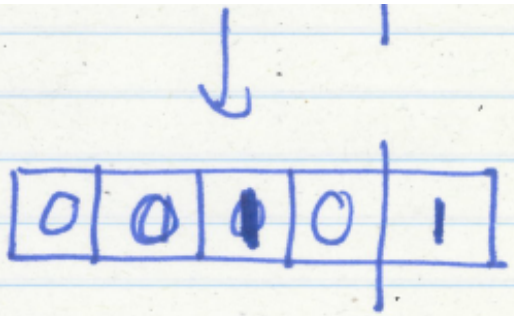
→ Invalidate P_1 & P_3's caches

→ 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

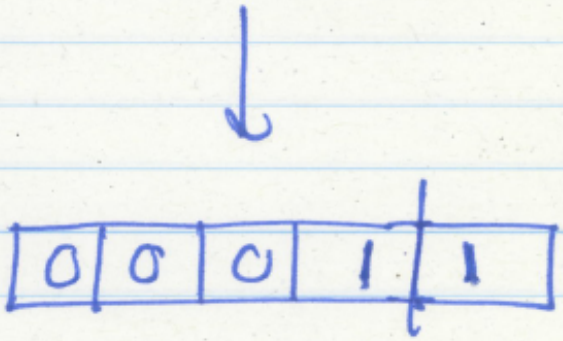


(4) P_3 takes a write miss

→ Mem ~~Cntrllr~~ Controller requests ~~true~~ block from P_2

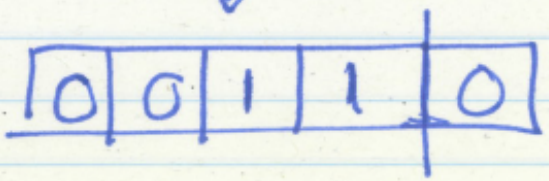
→ Mem Controller gives block to P_3

→ P2 invalidates its copy



(5) P_2 takes a read miss

→ P_3 supplies it



Directory Optimizations

- Directory is the coordinator for all actions to be performed on a given block by any processor
  - Guarantees correctness, ordering

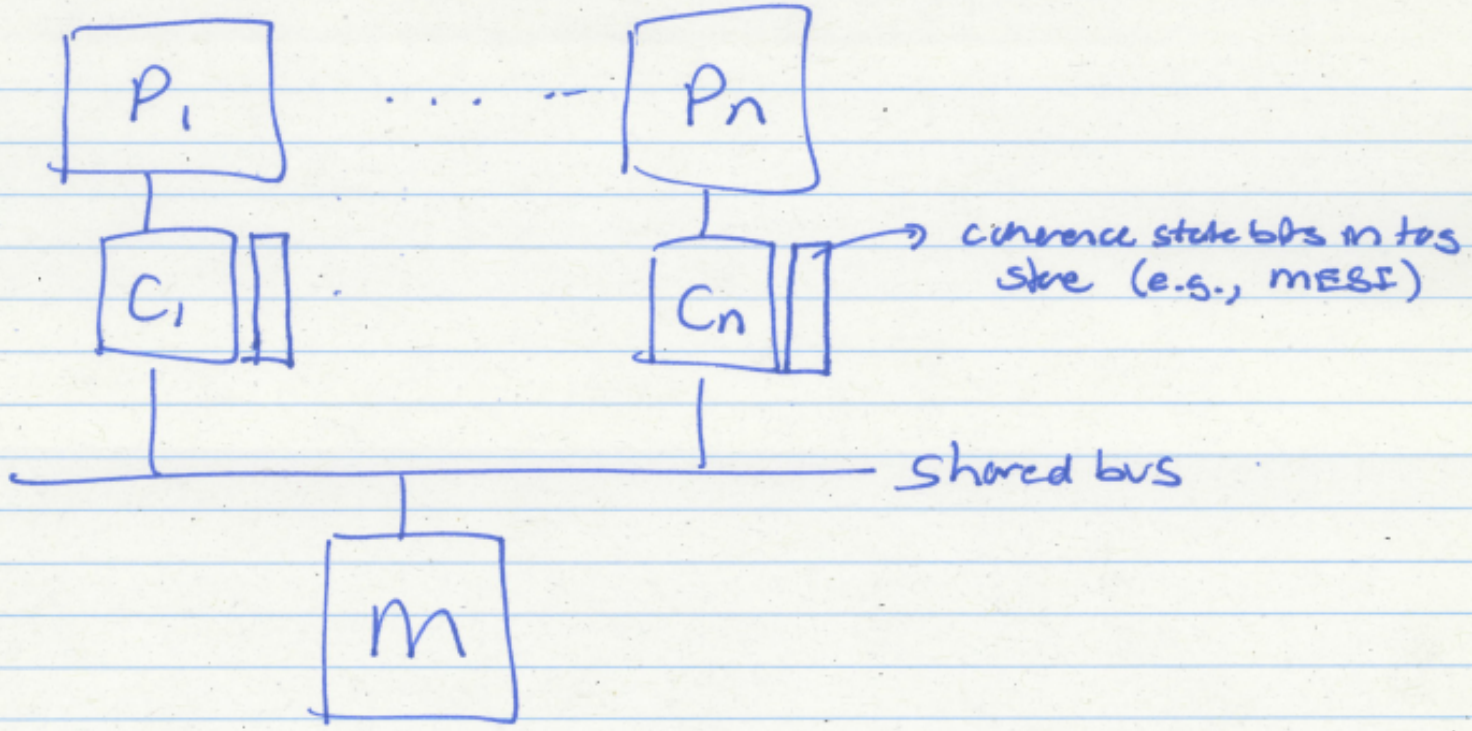
- Yet, there are many opportunities for optimization
  - Enabled by bypassing the directory and directly communicating between caches
  - We will see examples of these optimizations later

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

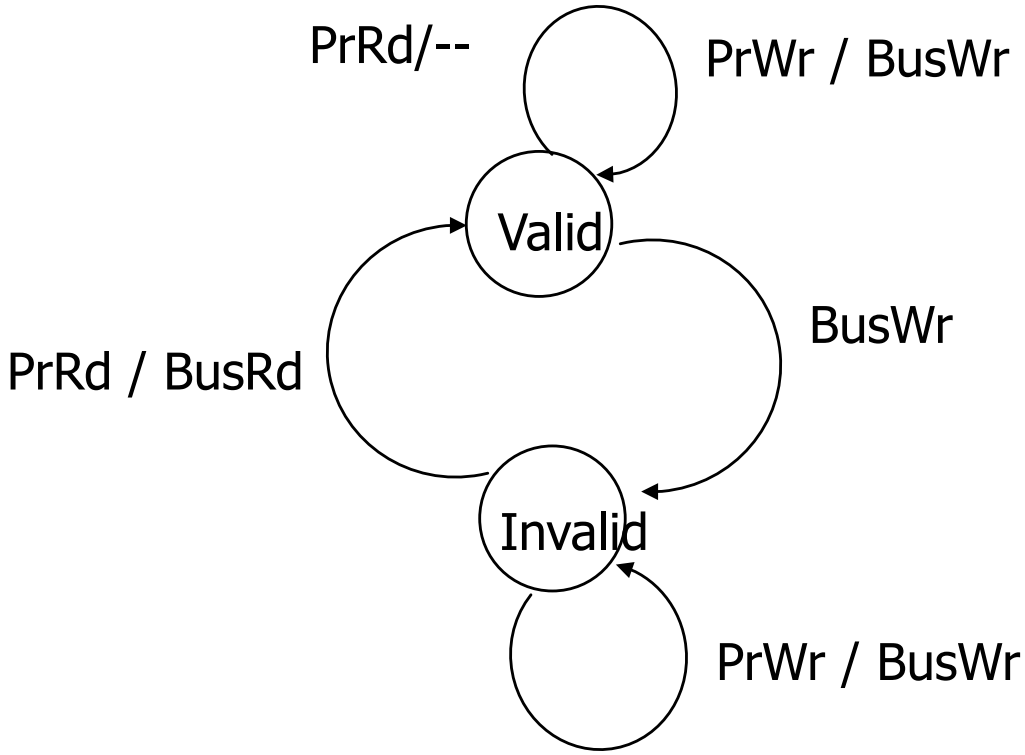
Each Cache observes its own processor & the bus
- Changes the state of the cached block based on observed actions by processor & the bus

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

Bus actions to a block (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 others' write/read operations
- A simple protocol (VI protocol):



- **Write-through**, no-write-allocate cache
- Actions of the local processor on the cache block: PrRd, PrWr,
- Actions that are broadcast on the bus for the block: BusRd, BusWr

Extending the Protocol

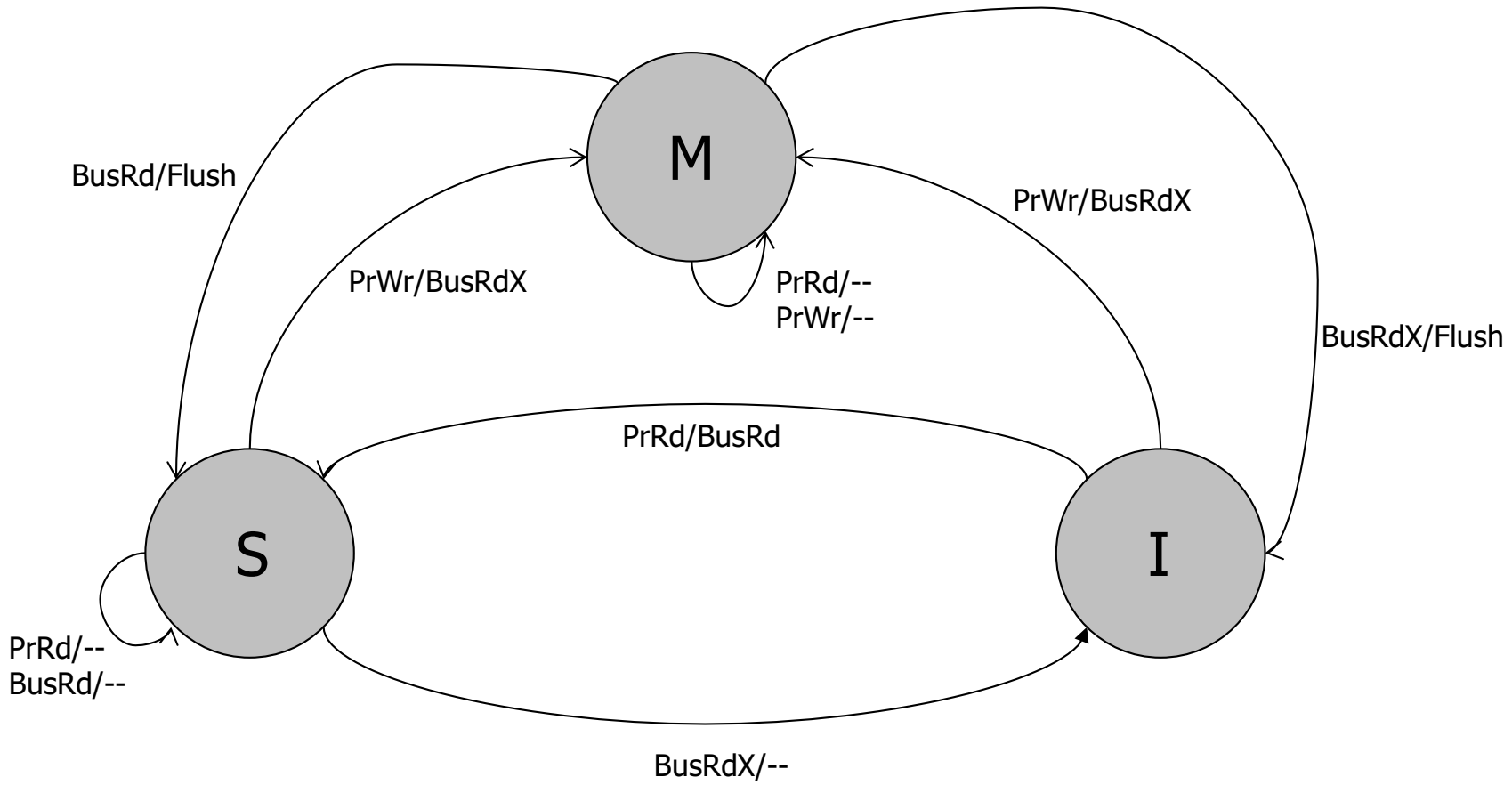
- What if you want write-back caches?
  - We want a “modified” state

A More Sophisticated Protocol: MSI

- Extend metadata per block to encode three states:
  - **M**(odified): cache line is the only cached copy and is dirty
  - **S**(hared): cache line is potentially one of several cached copies and it is clean
  - **I**(nvalid): cache line is not present in this cache

- 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 reads 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

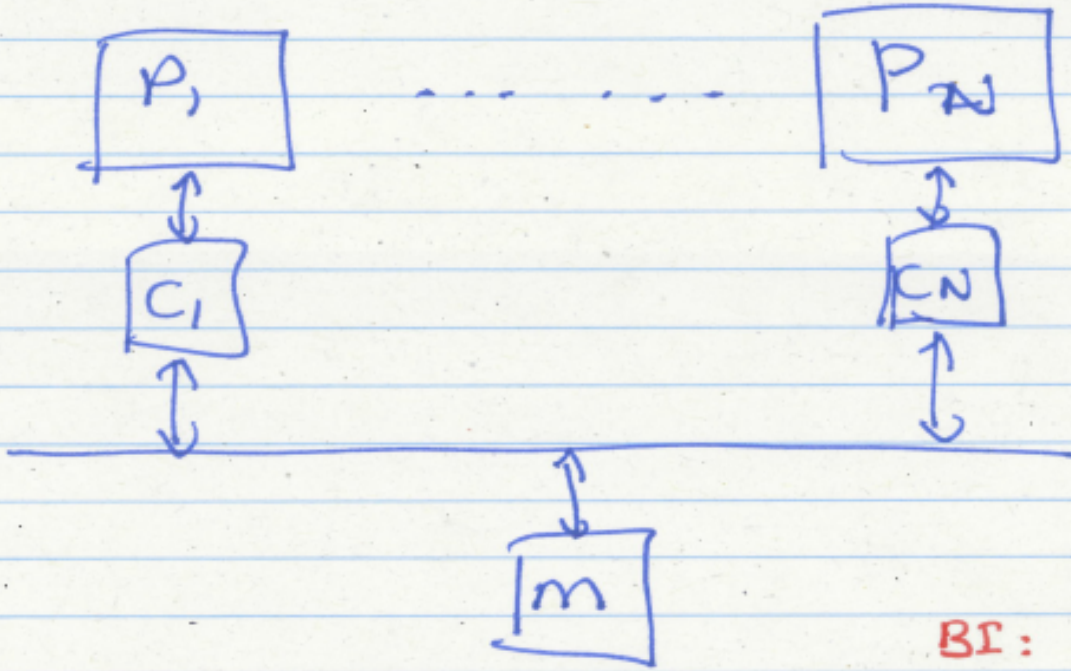
- Block is placed into the *exclusive* state if, during *BusRd*, no other cache had it
  - Wired-OR "shared" signal on bus can determine this: snooping caches assert the signal if they also have a copy

- 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



↓ PR, PW

↑ BR

BW → BI

BW → BRI

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

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

4 States

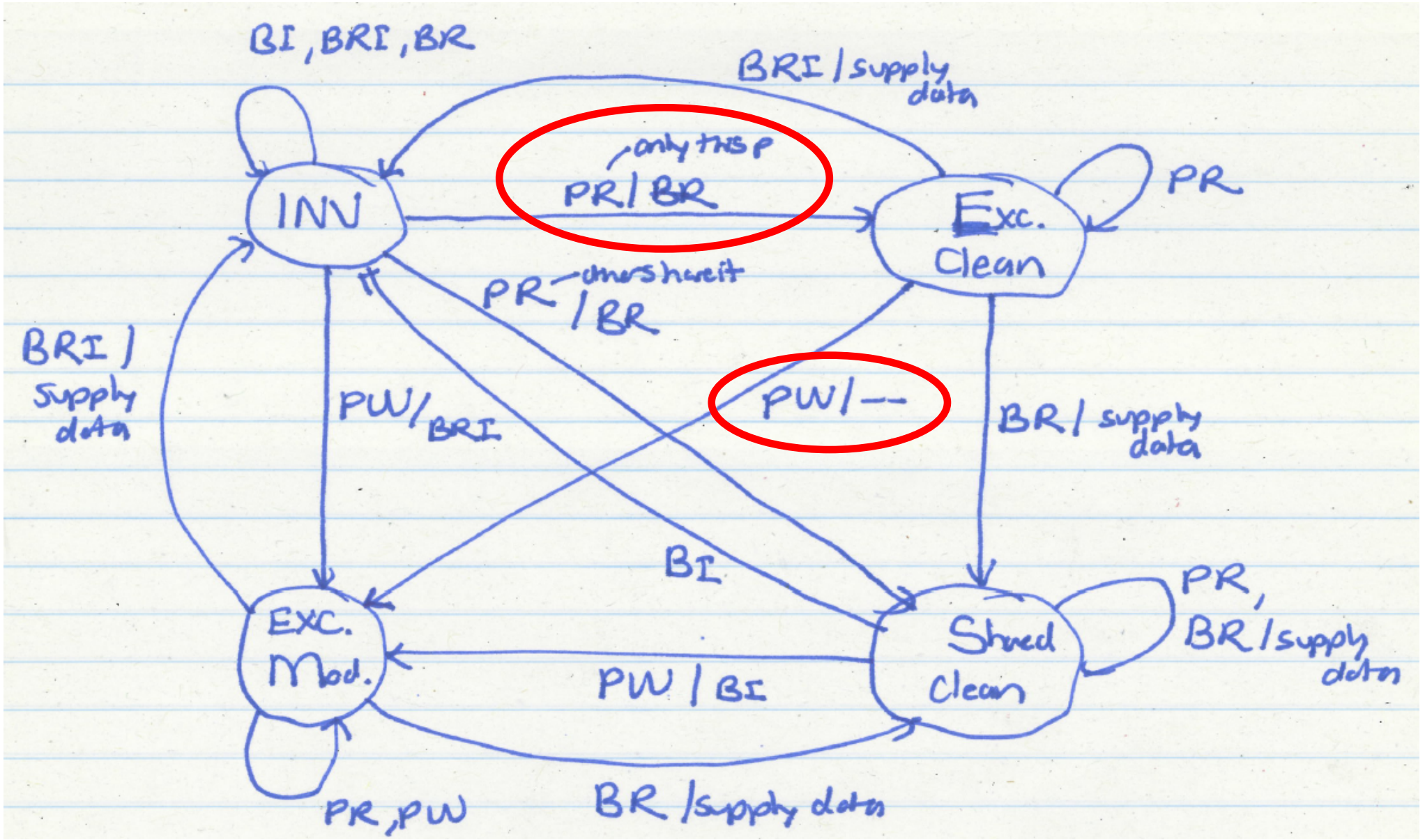
M: Modified (Exclusive copy, modified)

E: Exclusive (" ", clean)

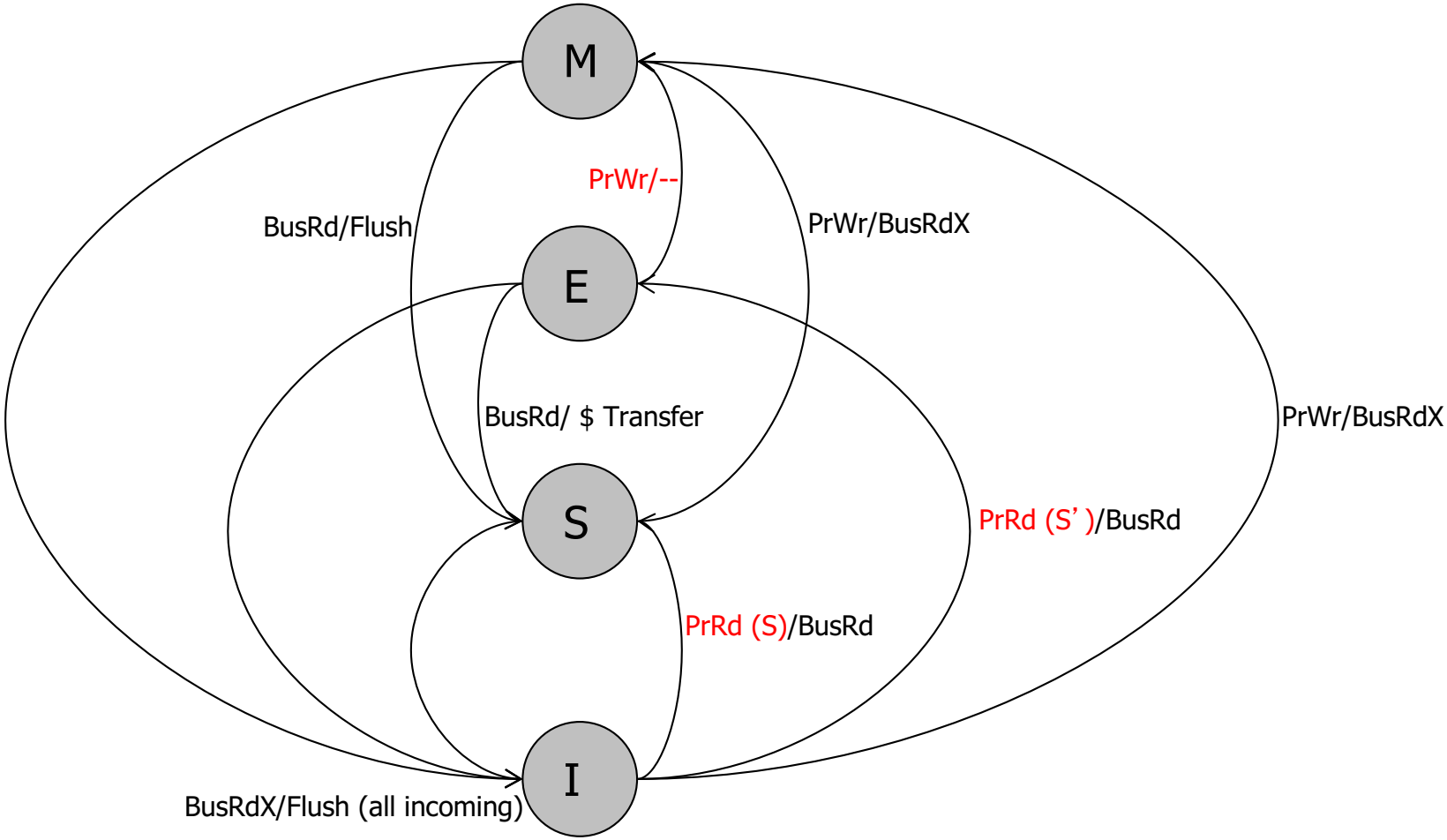
S: Shared (Shared copy, clean)

I: Invalid

MESI State Machine

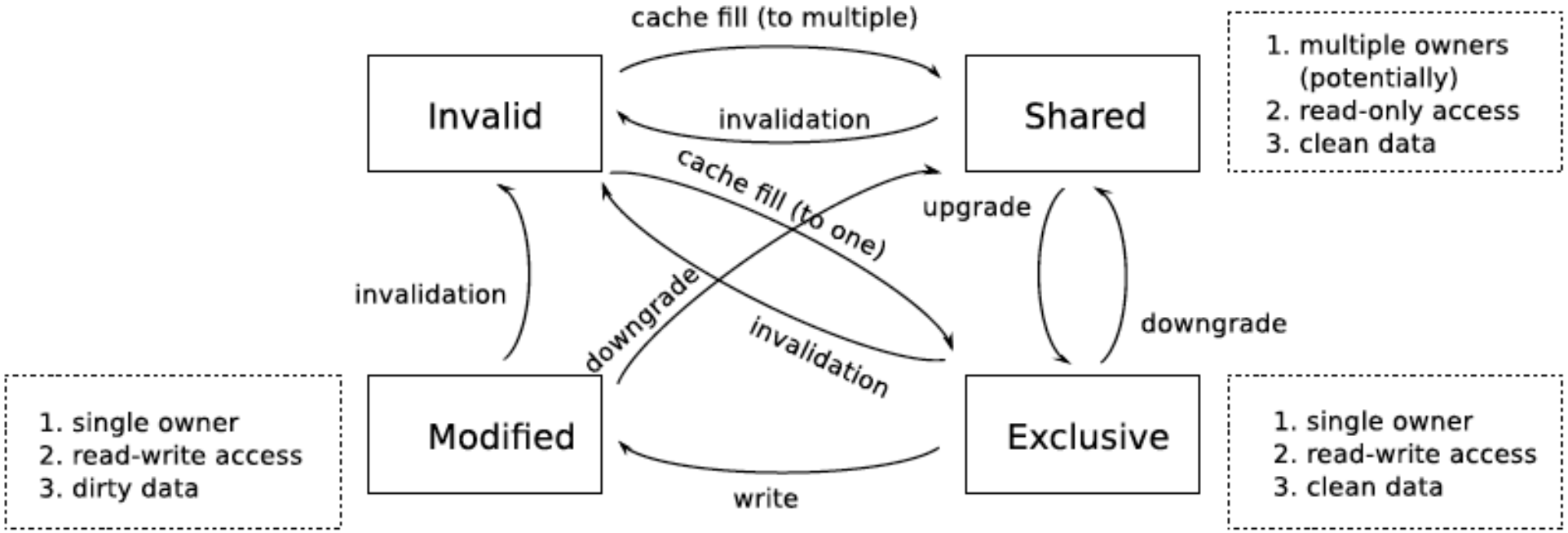


MESI State Machine



[Culler/Singh96]

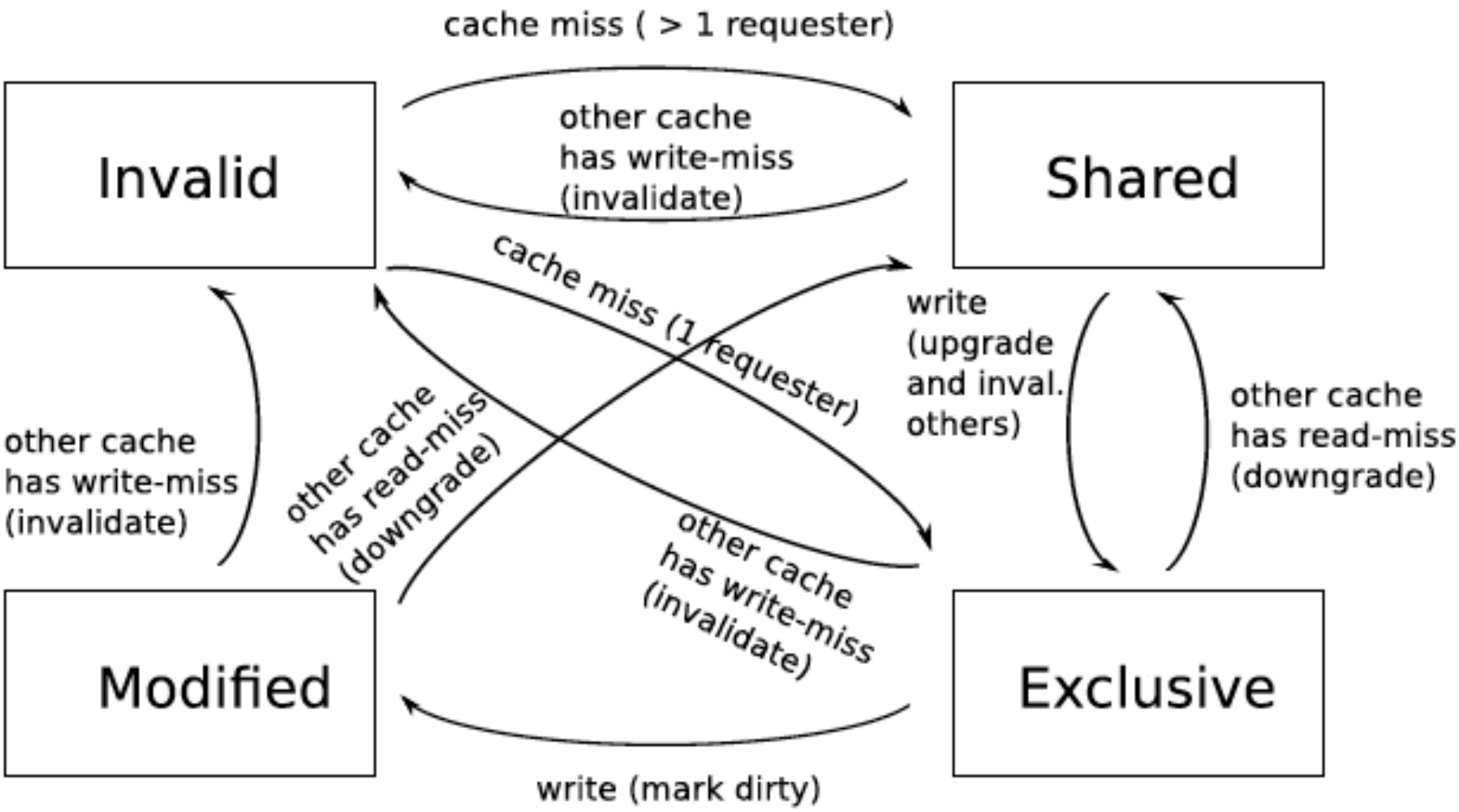
MESI State Machine from Optional Lab 5



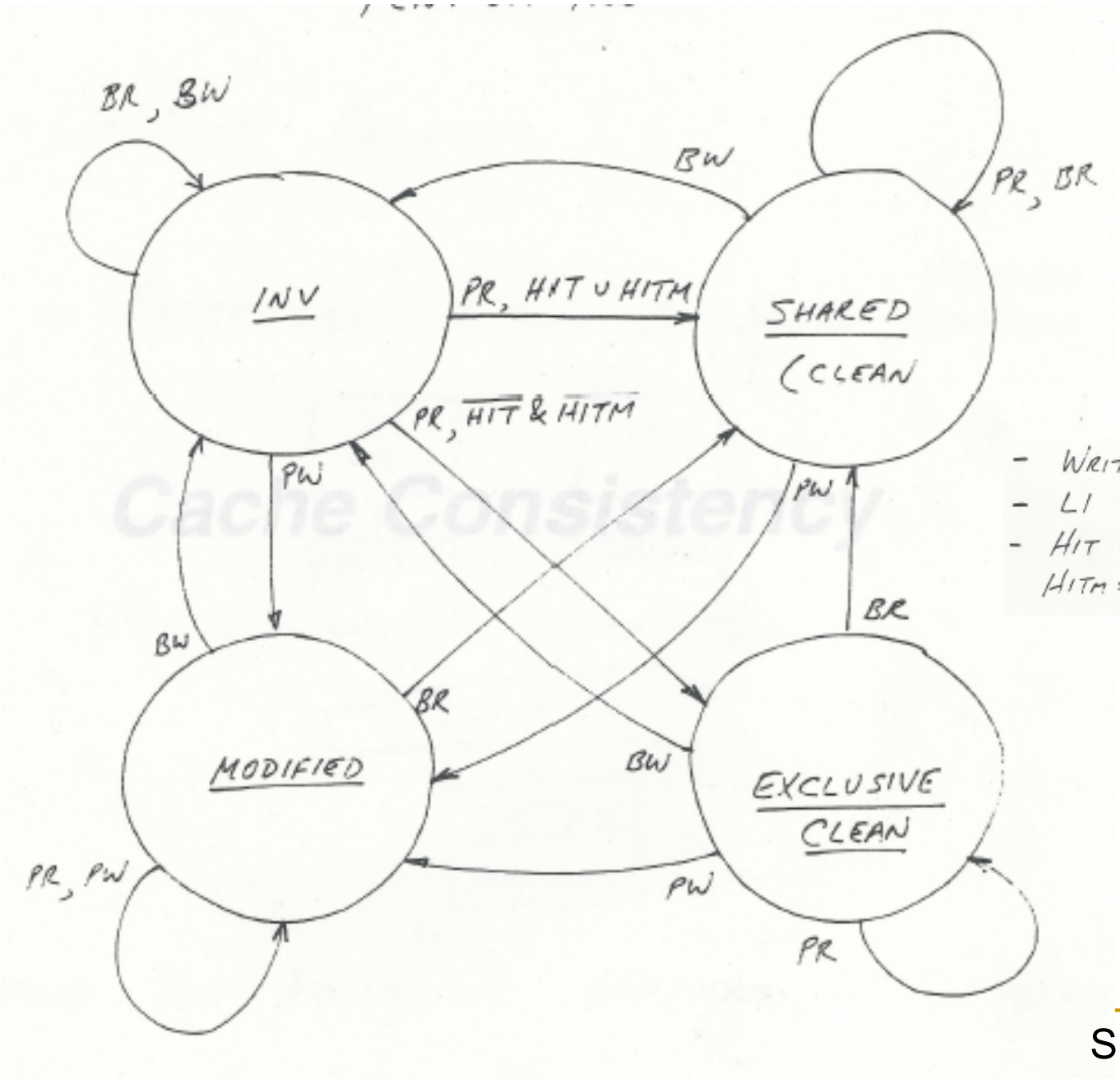
A transition from a single-owner state (Exclusive or Modified) to Shared is called a downgrade, because the transition takes away the owner's right to modify the data

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 Optional Lab 5



Intel Pentium Pro



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 another cache has the 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

- Observation: 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

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 memory 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 which caches have each block
  - Directory coordinates invalidation and updates
    - 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: request → bus transaction to mem.
  - \+ Global serialization is easy: bus provides this already (arbitration)
  - \+ Simple: can 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 for correctness)
  - – 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)

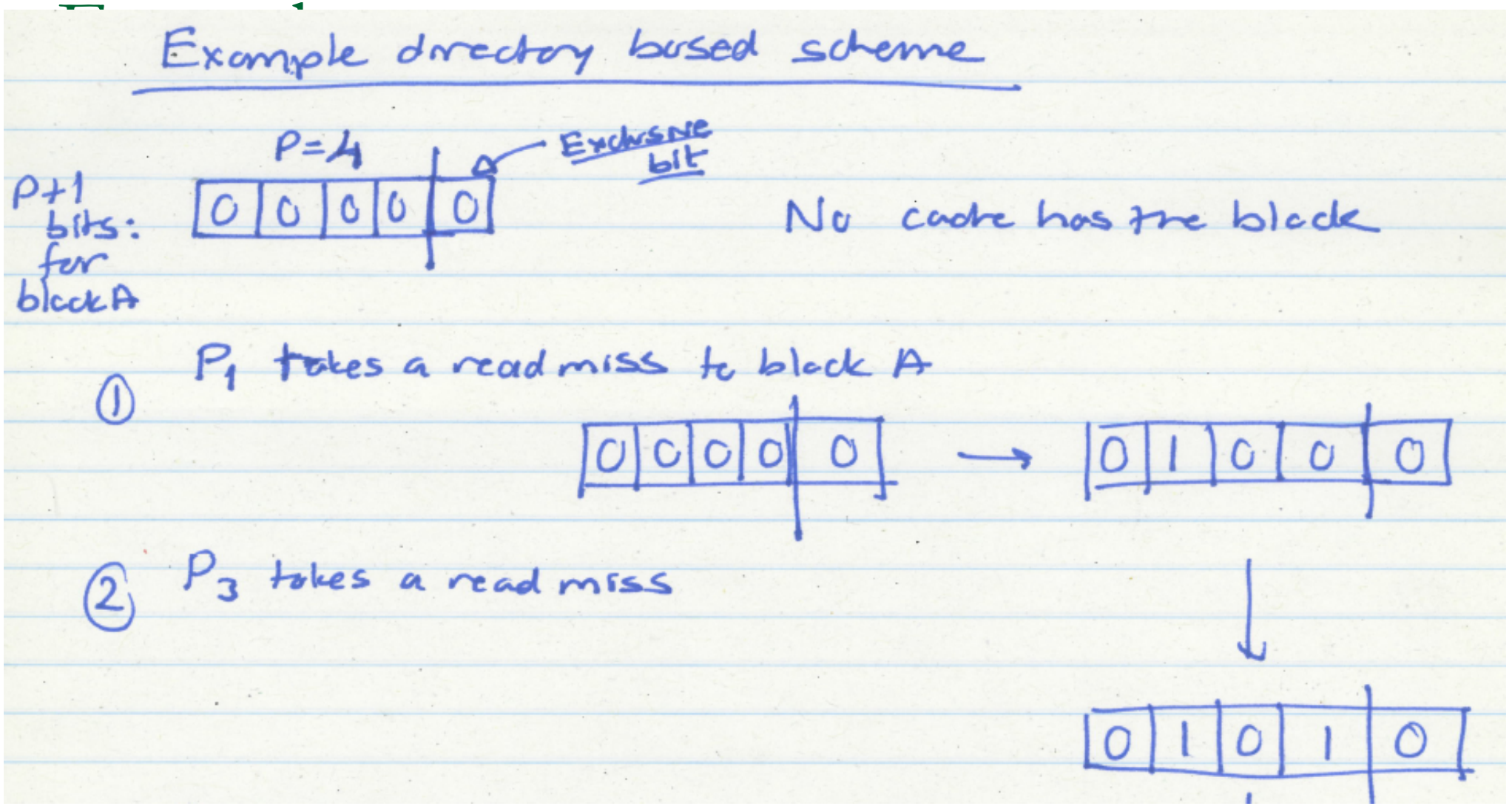
Revisiting Directory-Based Cache Coherence

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

- Required when scaling past the capacity of a single bus
- Distributed*:*
  - Coherence still requires single point of serialization (for write serialization)
  - Serialization location can be different for every block (striped across nodes/memory-controllers)

- 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	---
…	---

- Required to support invalidation and cache block requests
- 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

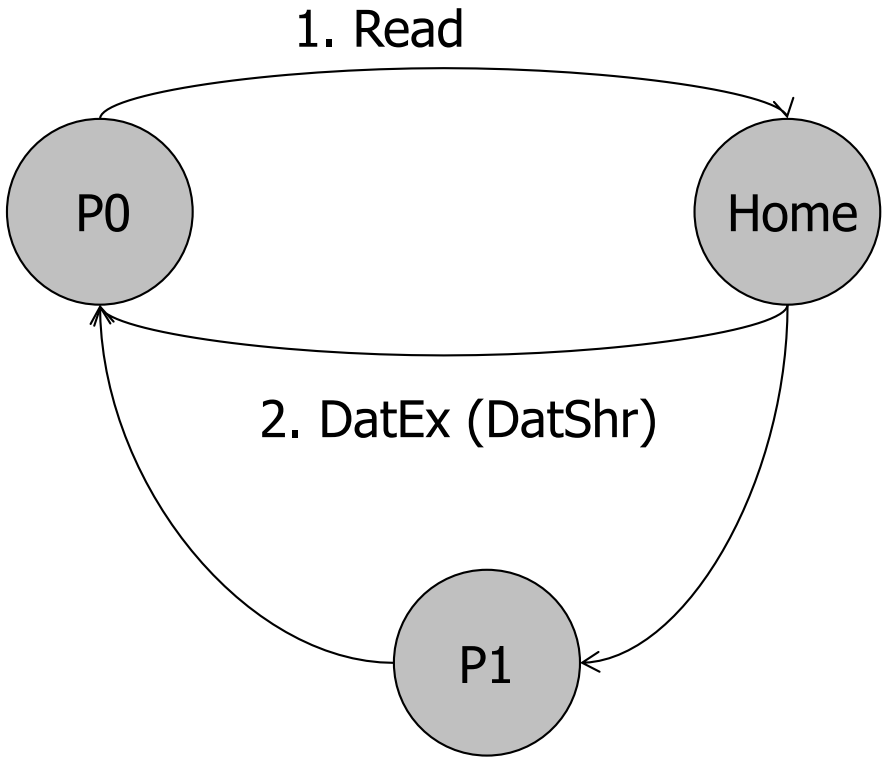
- Follow *semantics* of snoop-based system
  - but with explicit request, reply messages

- 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?

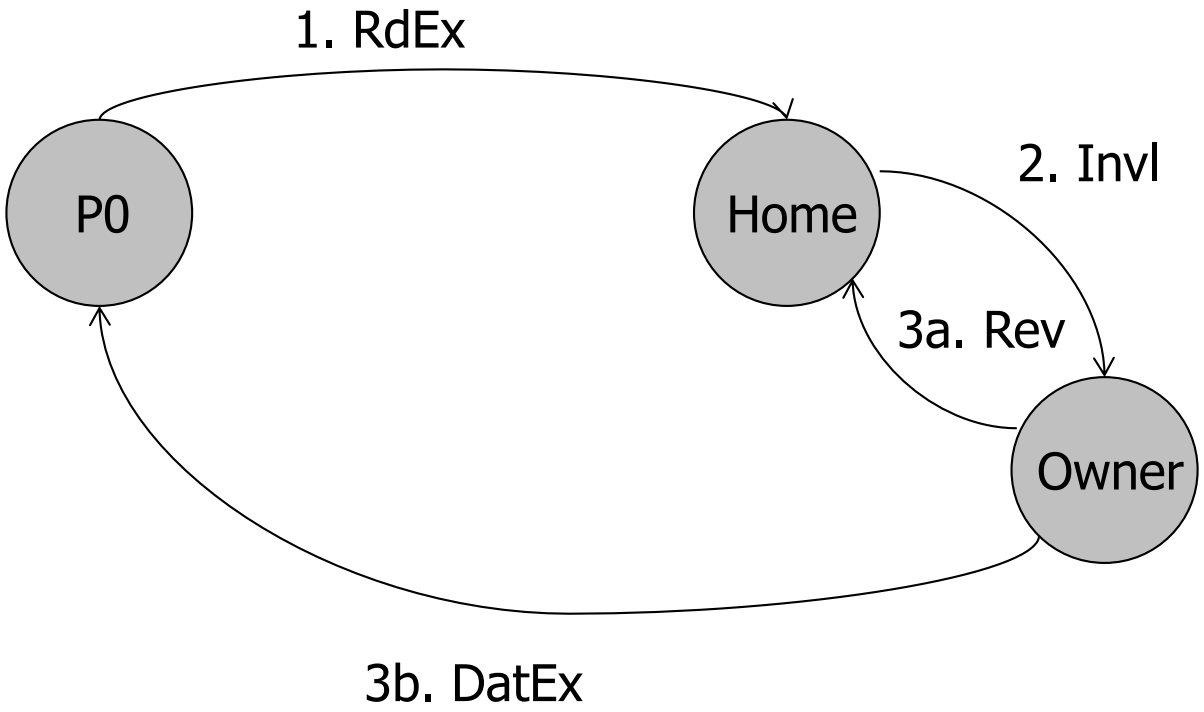
MESI Directory Transaction: Read

P0 acquires an address for reading:

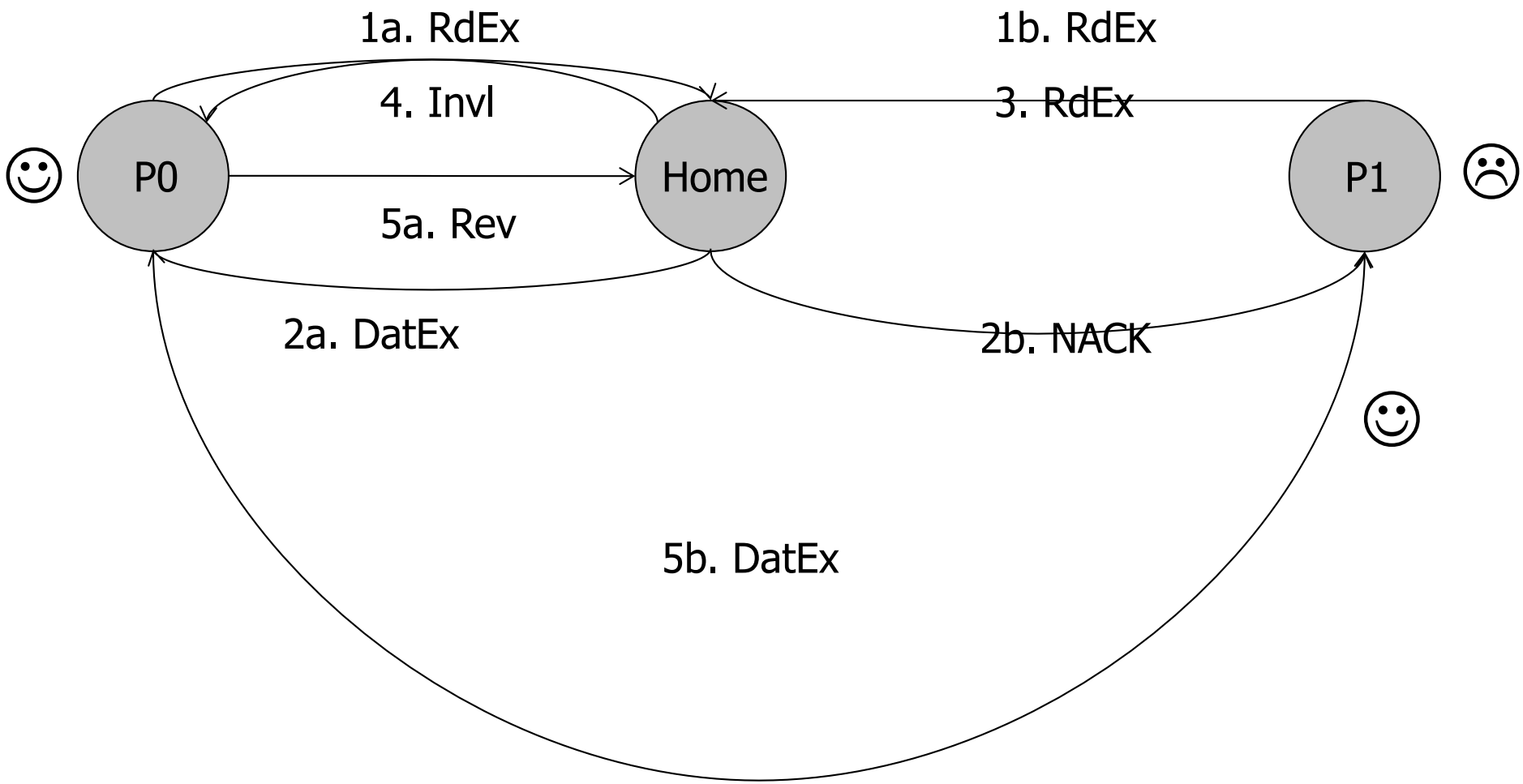


Culler/Singh Fig. 8.16

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 can be reduced w/ protocol optimizations OR better higher-level synchronization
  - 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]

An Example Question (I)

(f) **Directory** **[11 points]**

Assume we have a processor that implements the directory based cache coherence protocol we discussed in class. The physical address space of the processor is 32GB (2^{35} bytes) and a cache block is 128 bytes. The directory is equally distributed across randomly selected 32 nodes in the system.

You find out that the directory size in each of the 32 nodes is a total of 200 MB.

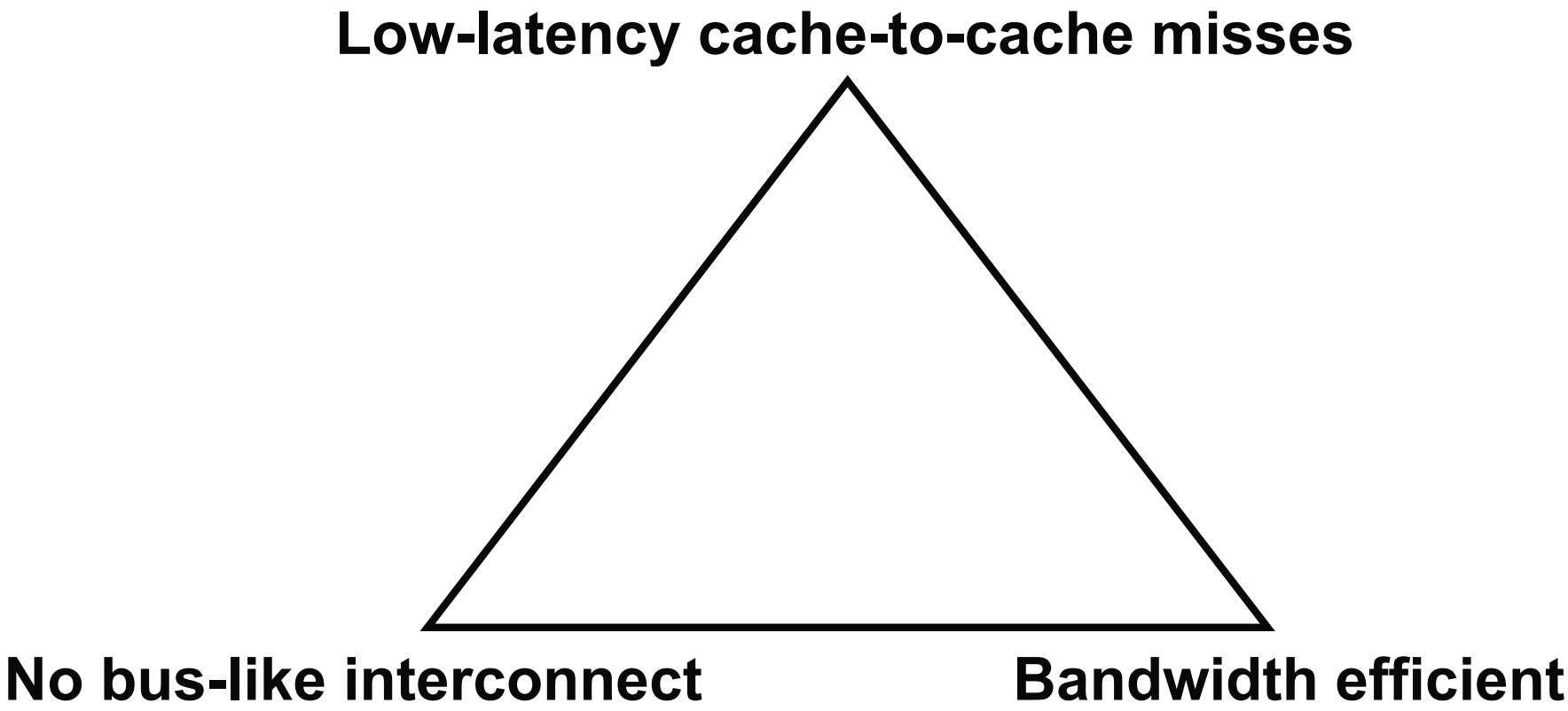
How many total processors are there in this system? Show your work.

An Example Answer

- Blocks per node
  - (32GB address space / 128 bytes per block) / 32 nodes
  - 2^(35-7-5) = 2^23
- Directory storage per node
  - **200 MB** = 25 * 2^23 bytes = 25 * 2^26 bits
- Directory storage per block
  - 25 * 2^26 bits / 2^23 blocks = 200 bits per block
- Each directory entry has P+1 bits
  - P+1 = 200 => **P = 199**

Advancing Coherence

Motivation: Three Desirable Attributes

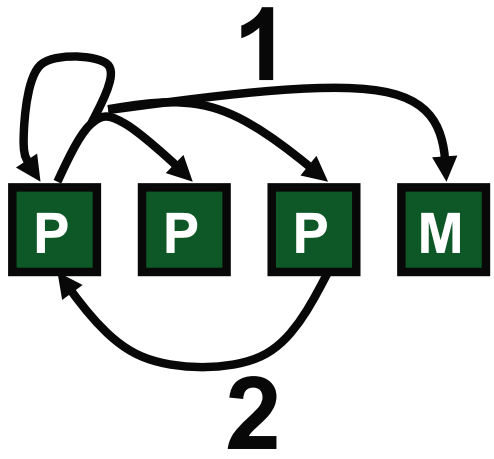


Dictated by workload and technology trends

Martin+, "Token Coherence: Decoupling Performance and Correctness," ISCA 2003.

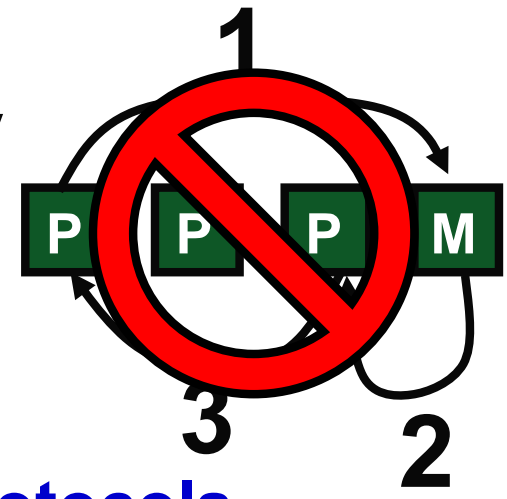
Workload Trends

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



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

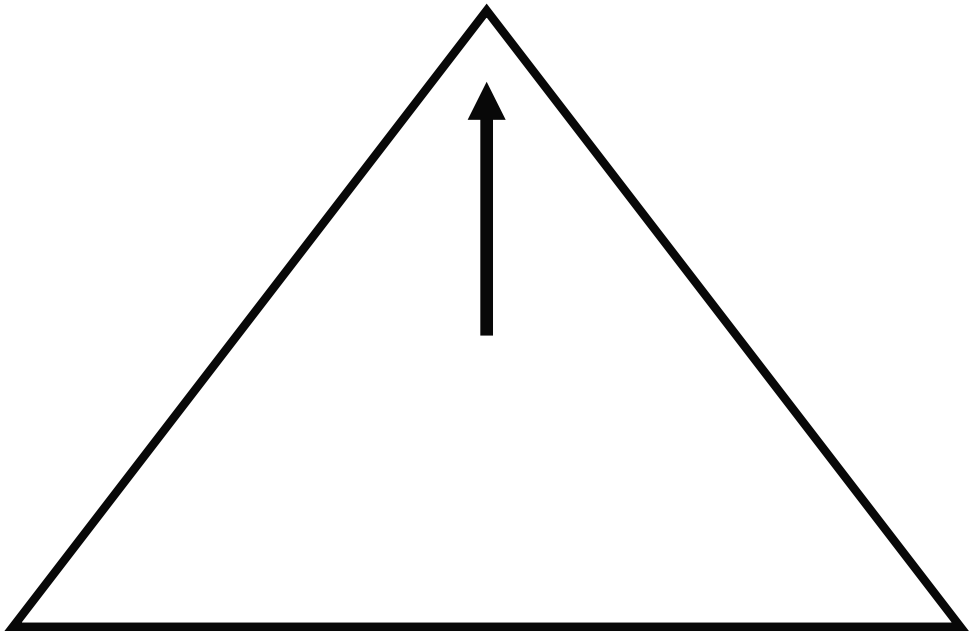
Directory Protocol



Workload trends → snooping protocols

Workload Trends

Low-latency cache-to-cache misses

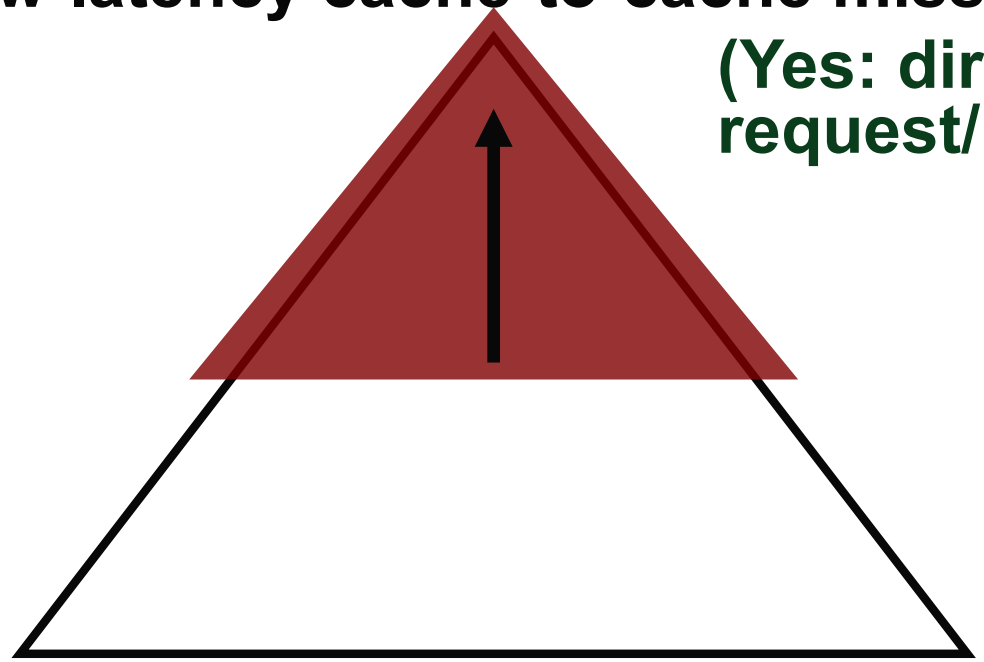


No bus-like interconnect **Bandwidth efficient**

Workload Trends ⇒ Snooping Protocols

Low-latency cache-to-cache misses

(Yes: direct request/response)



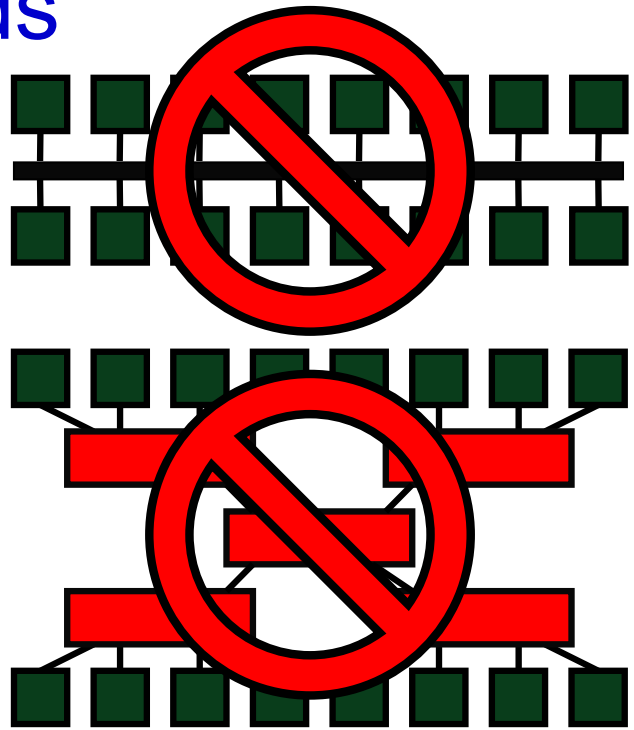
No bus-like interconnect

(No: requires a "virtual bus")

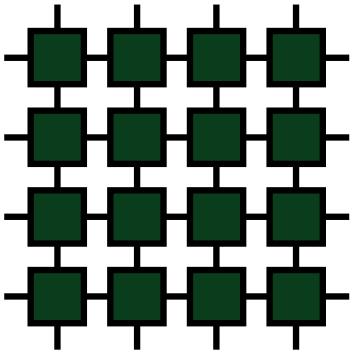
Bandwidth efficient

(No: broadcast always)

Technology Trends

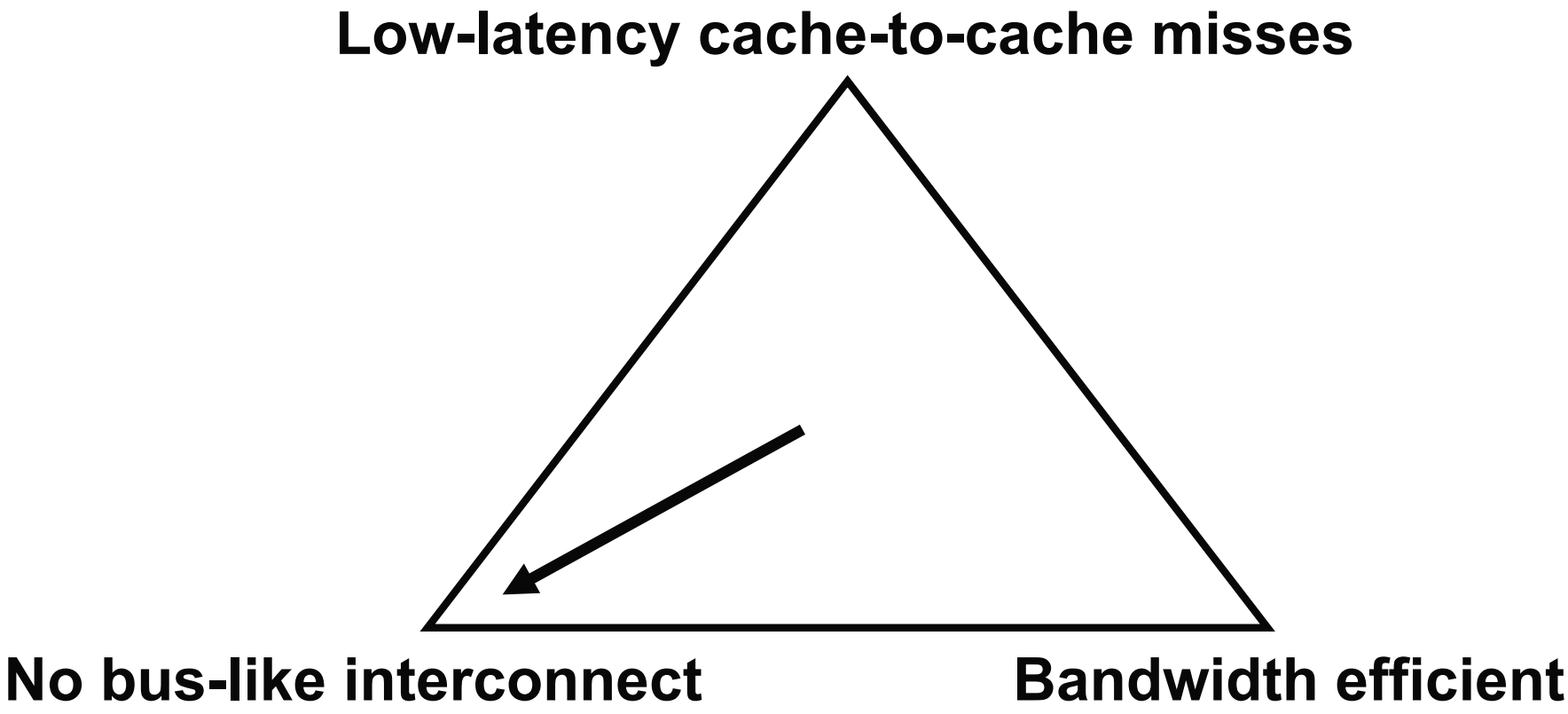


- High-speed point-to-point links
  - No (multi-drop) busses
- Increasing design integration
  - "Glueless" multiprocessors
  - Improve cost & latency
- Desire: low-latency interconnect
  - **Avoid "virtual bus" ordering**
  - **Enabled by directory protocols**



Technology trends → unordered interconnects

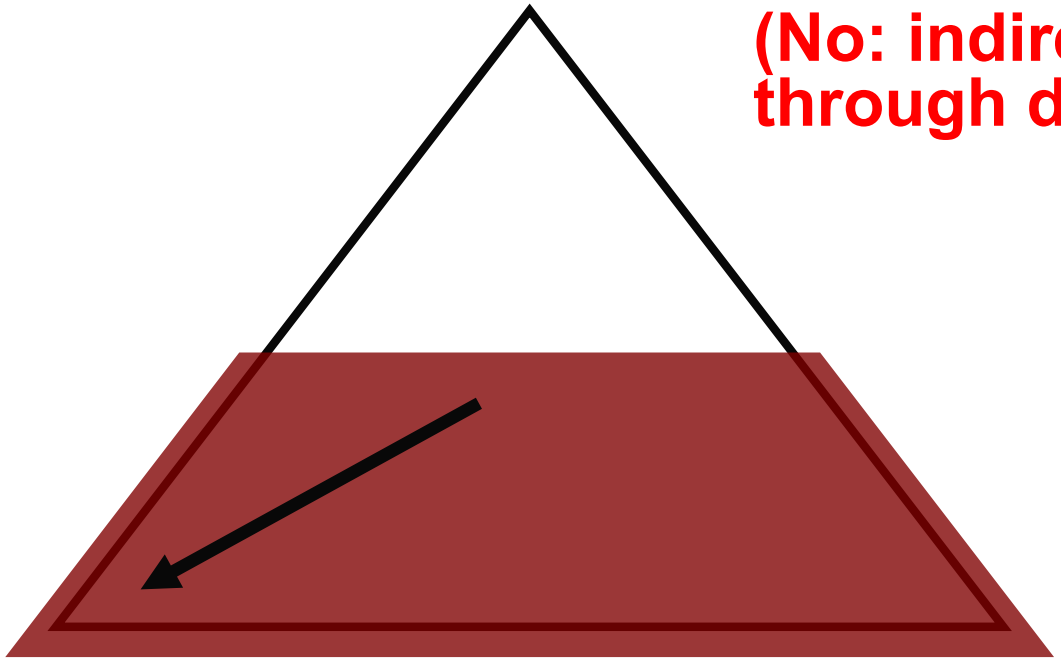
Technology Trends



Technology Trends ⇒ Directory Protocols

Low-latency cache-to-cache misses

(No: indirection through directory)



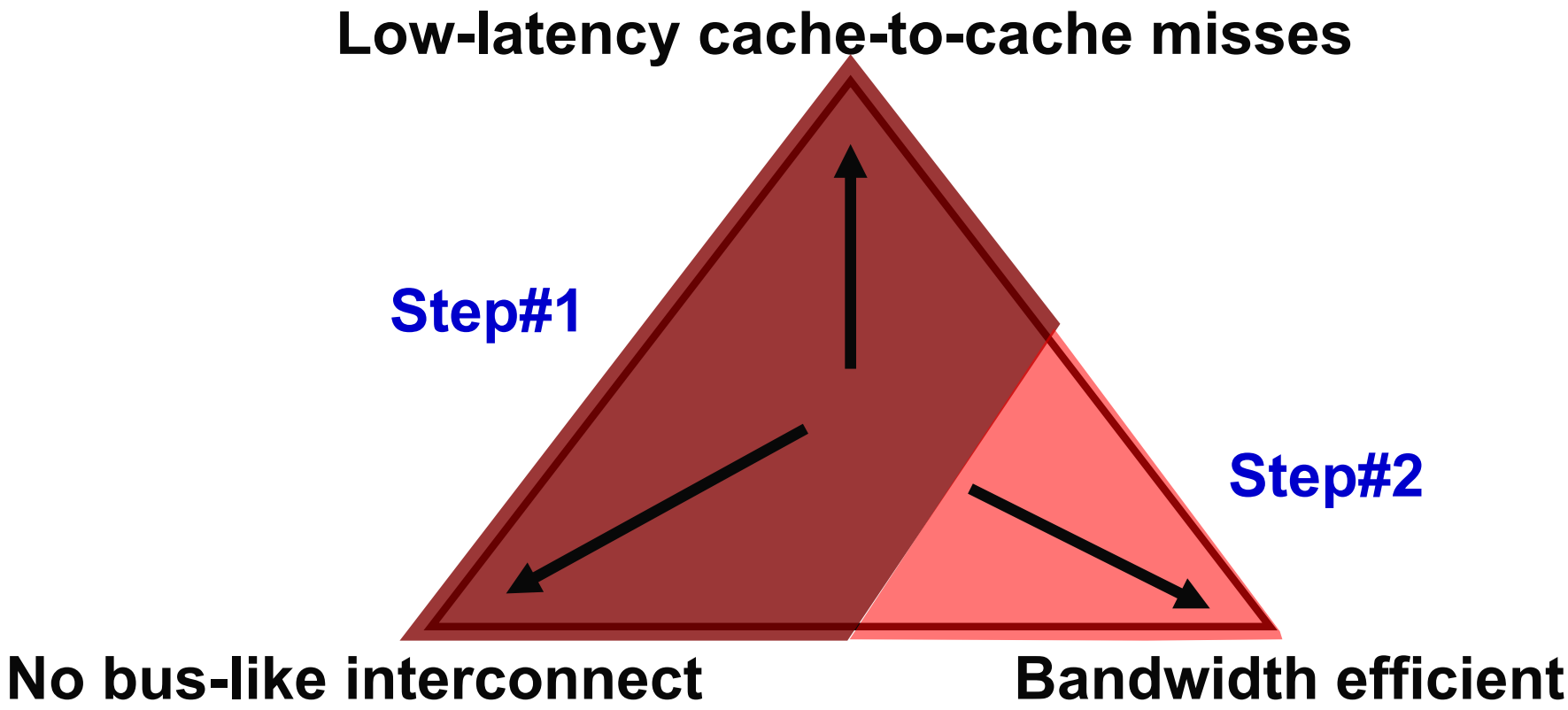
No bus-like interconnect

(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.

Martin+, "Token Coherence: Decoupling Performance and Correctness," ISCA 2003.

Next Topic:
Interconnects

Computer Architecture

Lecture 20: Multiprocessors, Consistency, Cache Coherence (II)

Prof. Onur Mutlu
ETH Zürich

Fall 2017

30 November 2017