Cache Coherence

15-740 FALL'21

NATHAN BECKMANN

Today: Cache coherence

What is it?

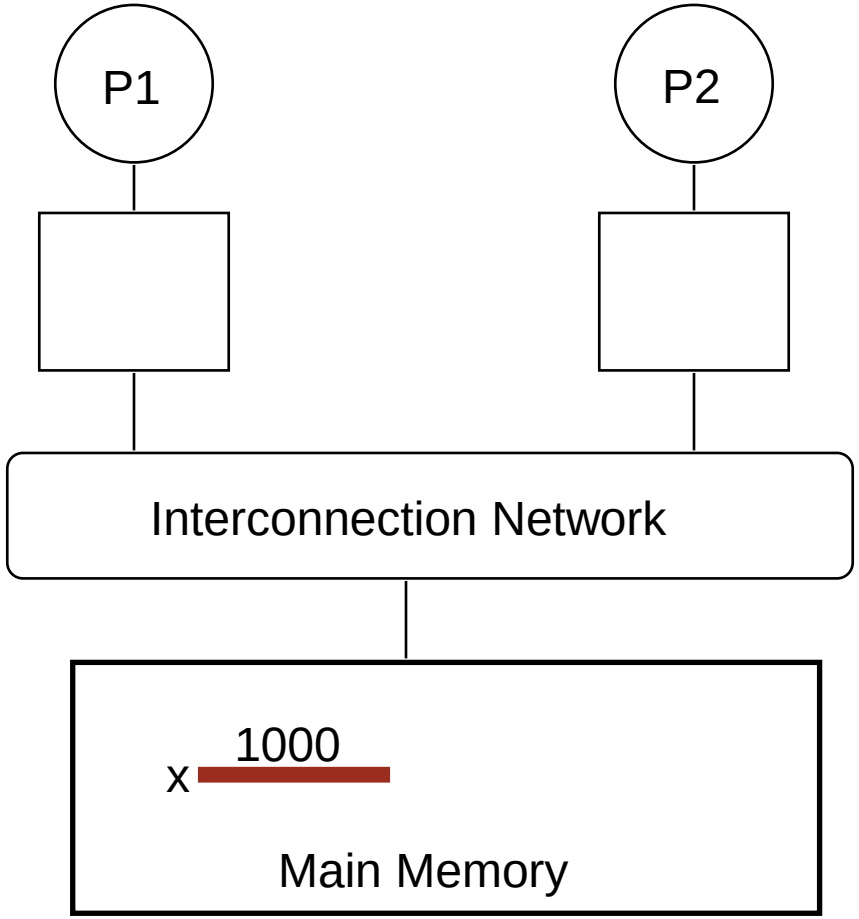
Protocol design

Snoopy cache coherence

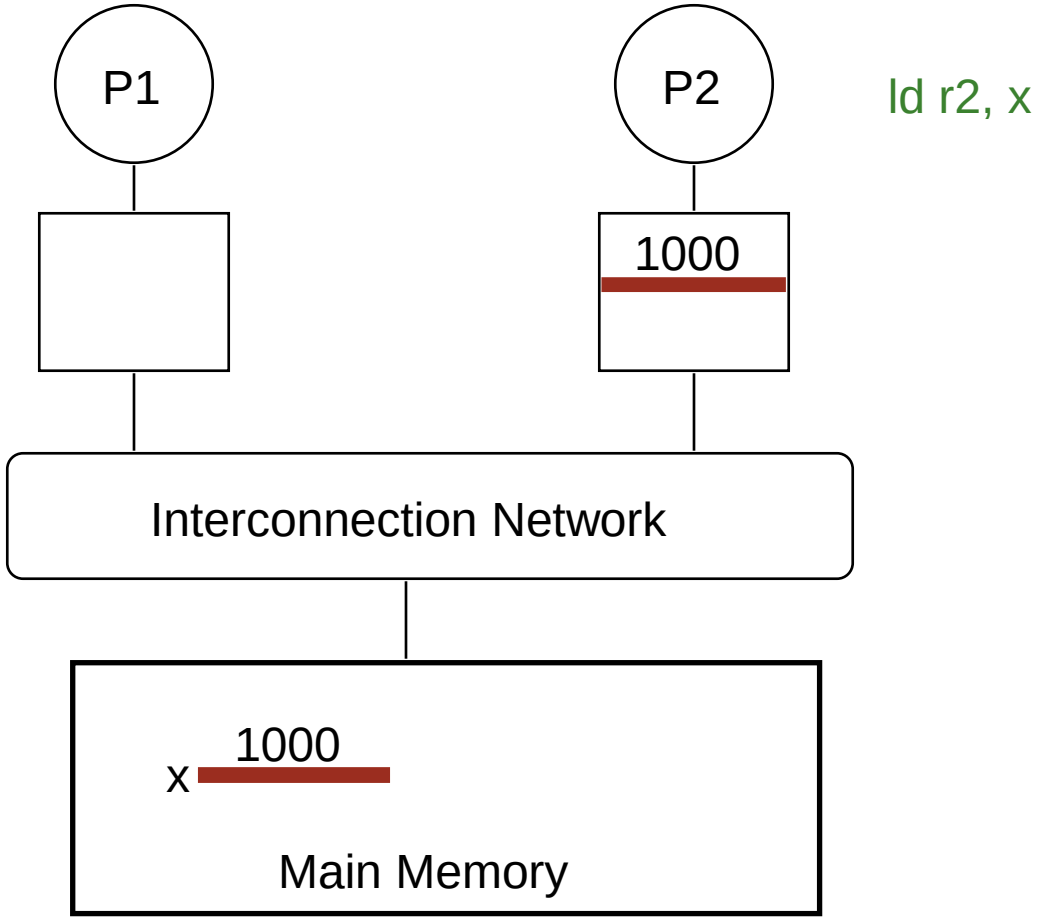
Directory cache coherence

Cache Coherence

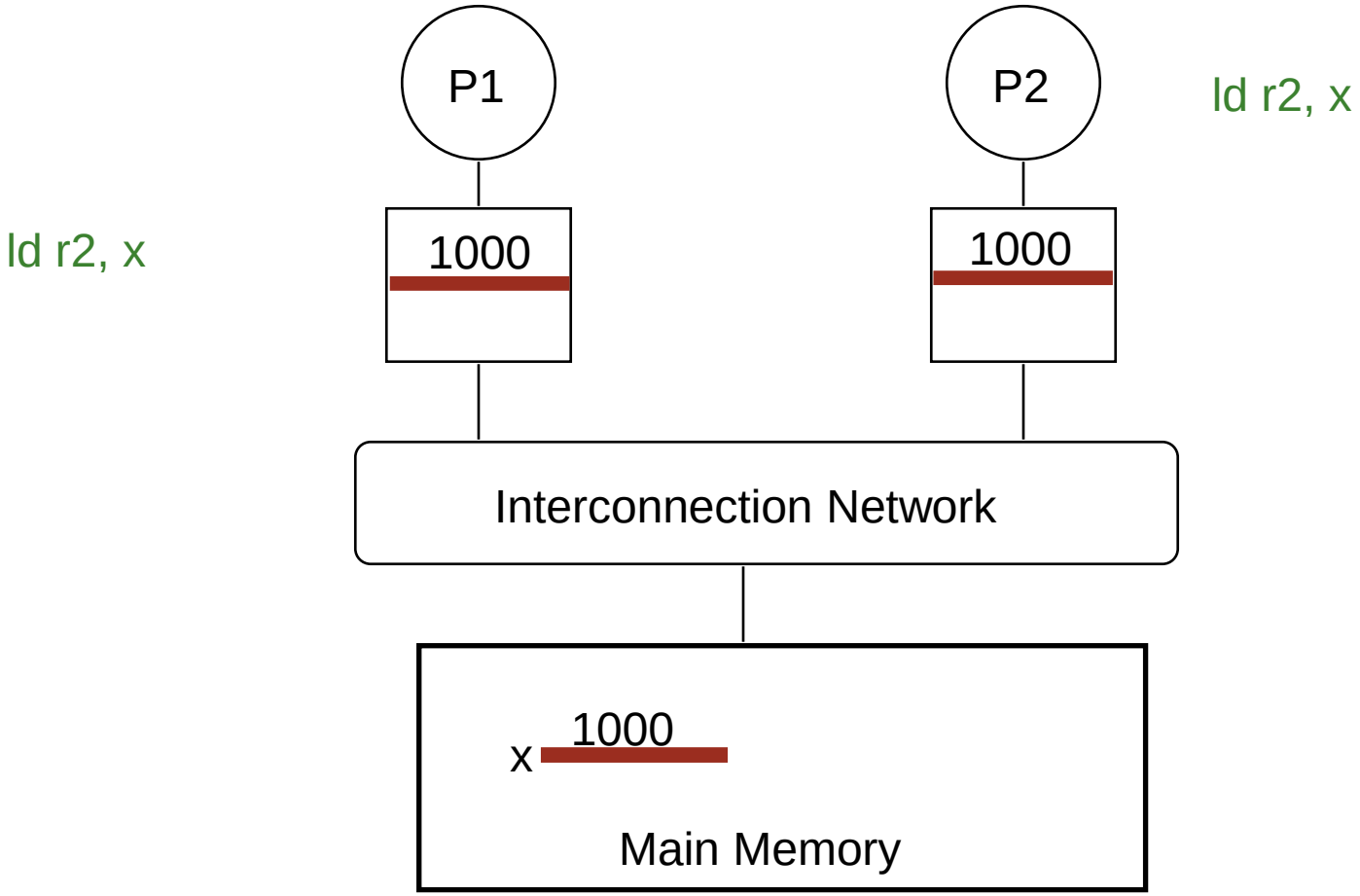
Basic question: If multiple processors cache the same block, how do they ensure they all see a consistent state?



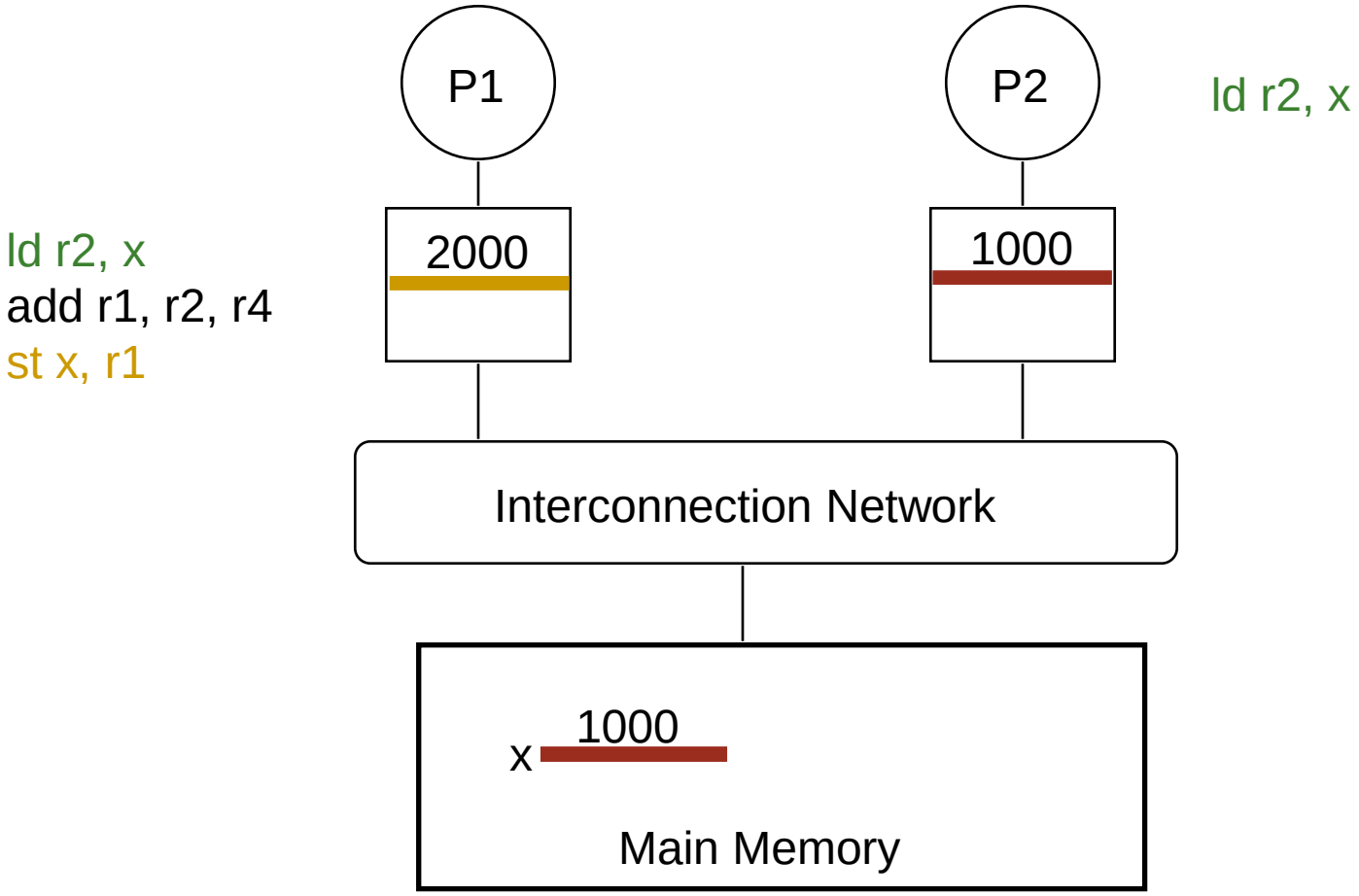
The Cache Coherence Problem



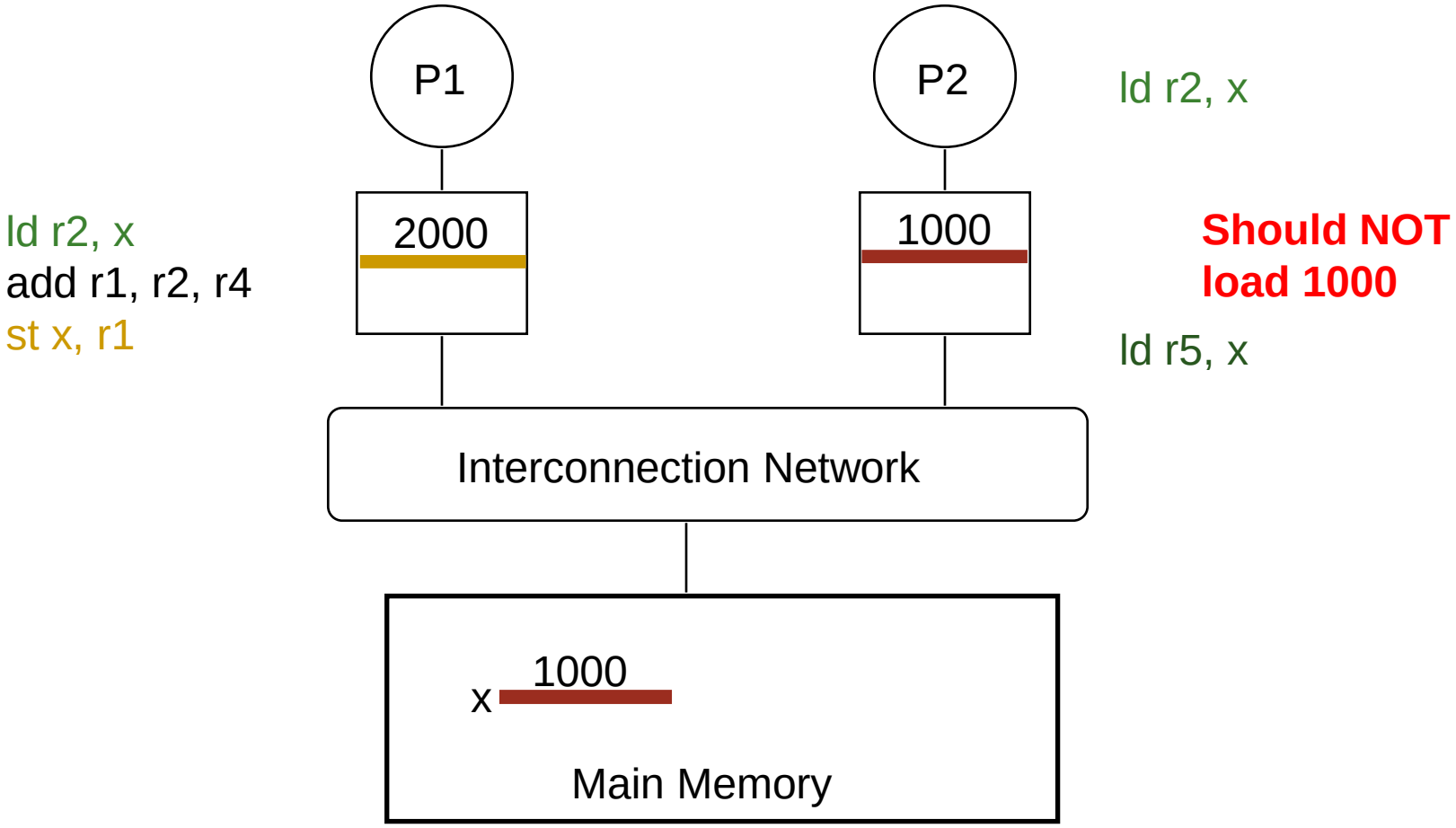
The Cache Coherence Problem



The Cache Coherence Problem



The Cache Coherence Problem



(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

– Overhead in ensuring coherence in software

  Extra cache-flush instructions must be inserted

  DeNovo [Choi, PACT'11] does this via "disciplined parallelism" (restrictive programming model)

  [Wang Ta Cheng Batten, ISCA'20] does something similar for work-stealing task-parallel programs

  Nevertheless, where do you commonly see software coherence in real systems?

All caches are shared between all processors

+ No need for coherence

– Shared cache becomes the bandwidth bottleneck

– Getting low latency + scalable cache is very hard/impossible

– At a minimum, L1s should be *private* ➔ still need coherence

What is coherence?

A memory system is **coherent** if, for each memory location, it is possible to construct a global serial order of memory operations to the location that is consistent with the results of execution and that obeys two invariants:

1. Operations issued by each core occur in the order in which they were issued to the memory system by that core.

2. The value returned by each read operation is the last value written to that location in the global serial order.

D. Culler, J. Singh, and A. Gupta, Parallel computer architecture: a hardware/software approach. Morgan Kaufmann, 1999

Maintaining Coherence

In other words…

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:

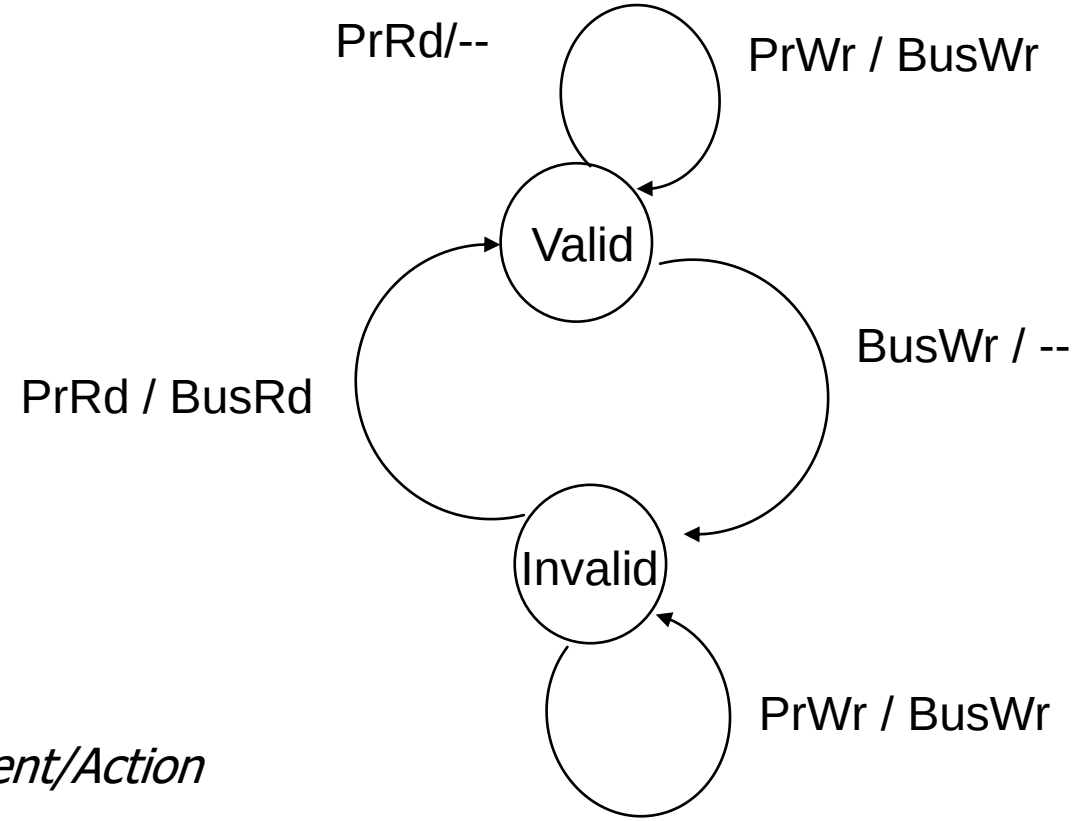
- **Write propagation**: guarantee that updates will propagate
- **Write serialization**: provide a consistent *global order* seen by all processors

Need a **global point of serialization** for this store ordering

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:



Legend: *Event/Action*

- Write-through, no-write-allocate cache
- Events & actions:
  PrRd – processor read
  PrWr – processor write
  BusRd – bus read
  BusWr – bus write

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)

Which is better?

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 is the 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]

- **Each block has a unique point of serialization**
- 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 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

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

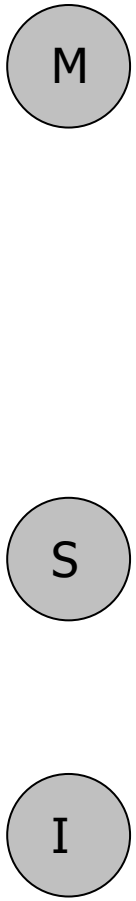
Read miss makes a *Read* request on bus, transitions to **S**

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)

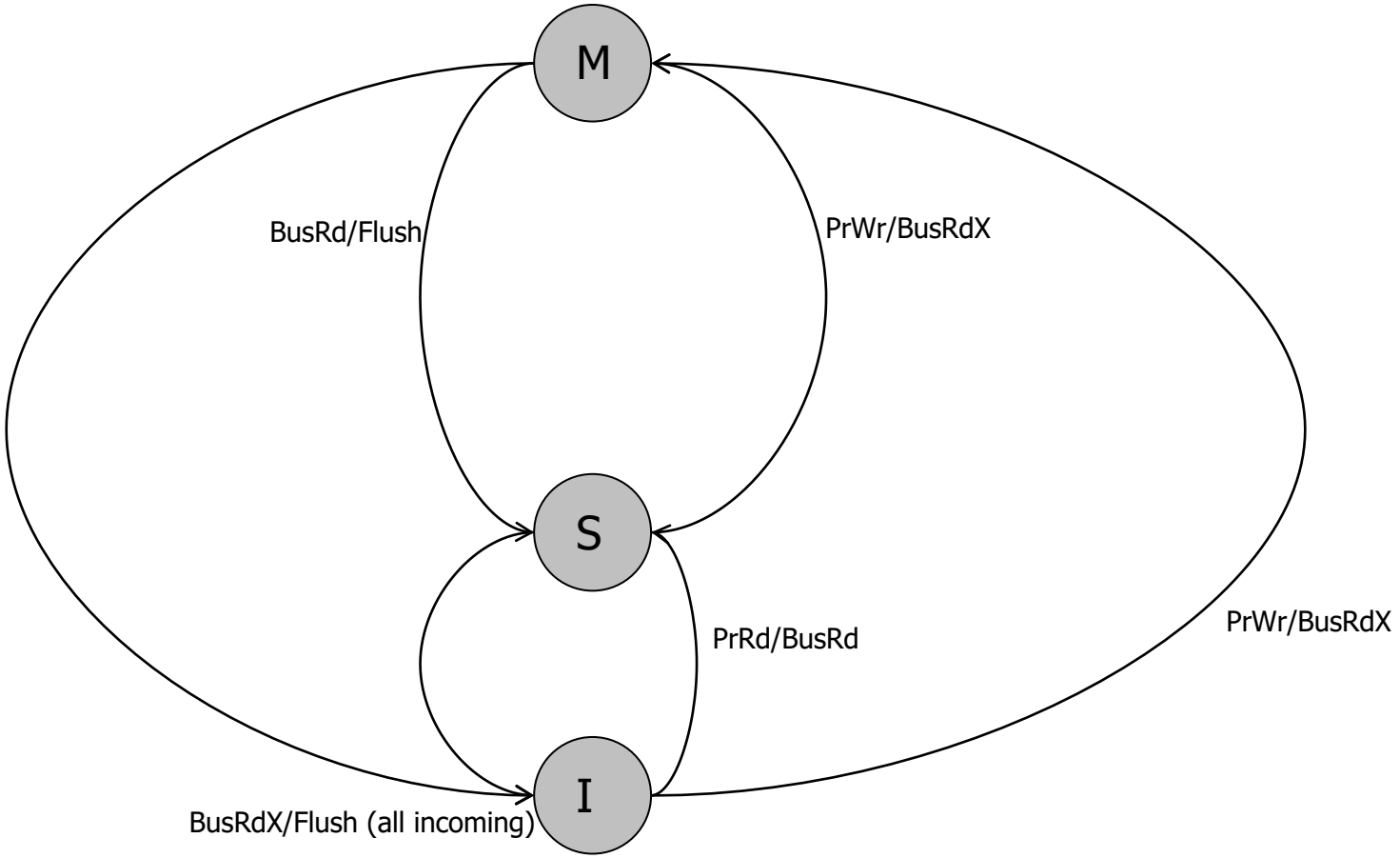
S→M upgrade can be made without re-reading data from memory (via *Invalidations)*

MESI State Machine



[Culler/Singh96]

MESI State Machine



[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* (E) state

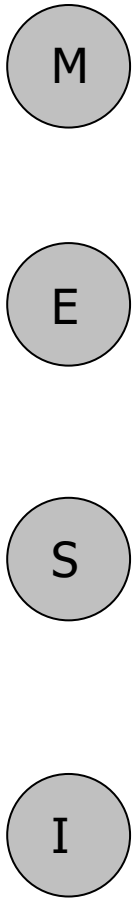
Block is placed into E state if, during *BusRd*, no other cache had it

- Implementation: Wired-OR "shared" signal on bus can determine this—snooping caches assert the signal if they also have a copy

Silent transition E→M is possible on write

Papamarcos and Patel, "A low-overhead coherence solution for multiprocessors with private cache memories," ISCA 1984.

MESI State Machine



[Culler/Singh96]

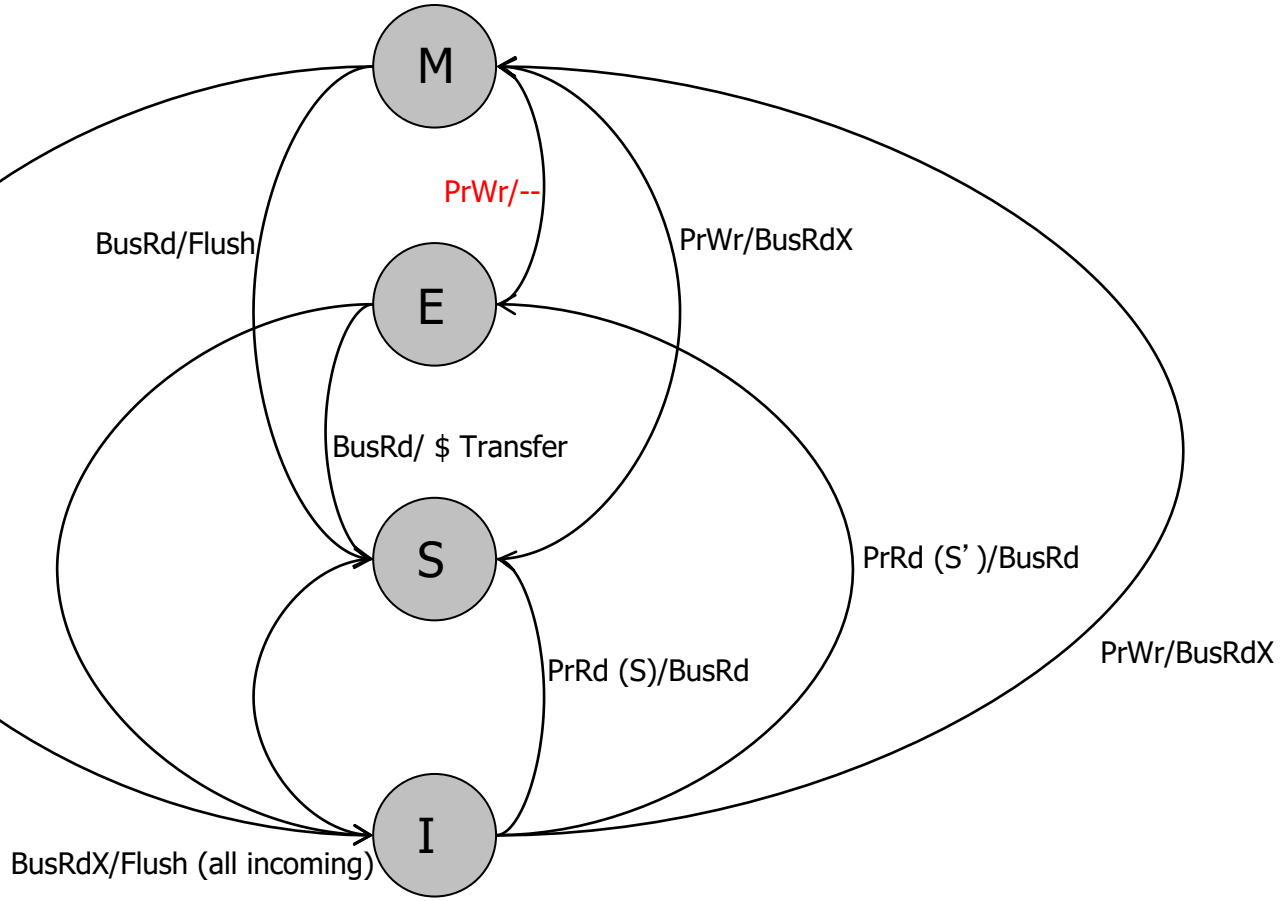
MESI State Machine

Modified:
- 1 owner
- dirty data
- R/W access

Exclusive:
- 1 owner
- clean data
- R/W access

Shared:
- >=1 owner(s)
- clean data
- RO access

Invalid:
- Not present
- No data
- No access



[Culler/Singh96]

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)
- *How would you know?*

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

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 Coherence Protocols

The protocol can be further optimized with more states & prediction mechanisms

- Eliminate more unnecessary invalidates and transfers of blocks

But states are not free

- Difficult to design and verify (many cases and possible race conditions)
- Provide rapidly diminishing returns

We are showing simple cartoons, but actual implementations need many transient states (>20) & are extremely hard to verify

➔ Industry is quite reluctant to change a working coherence protocol

Directory-Based Cache Coherence

Directory-Based Protocols

Buses are simple but don't scale

- Single, shared communication channel is bottleneck
- *What does snoopy coherence look like with 100 cores?*

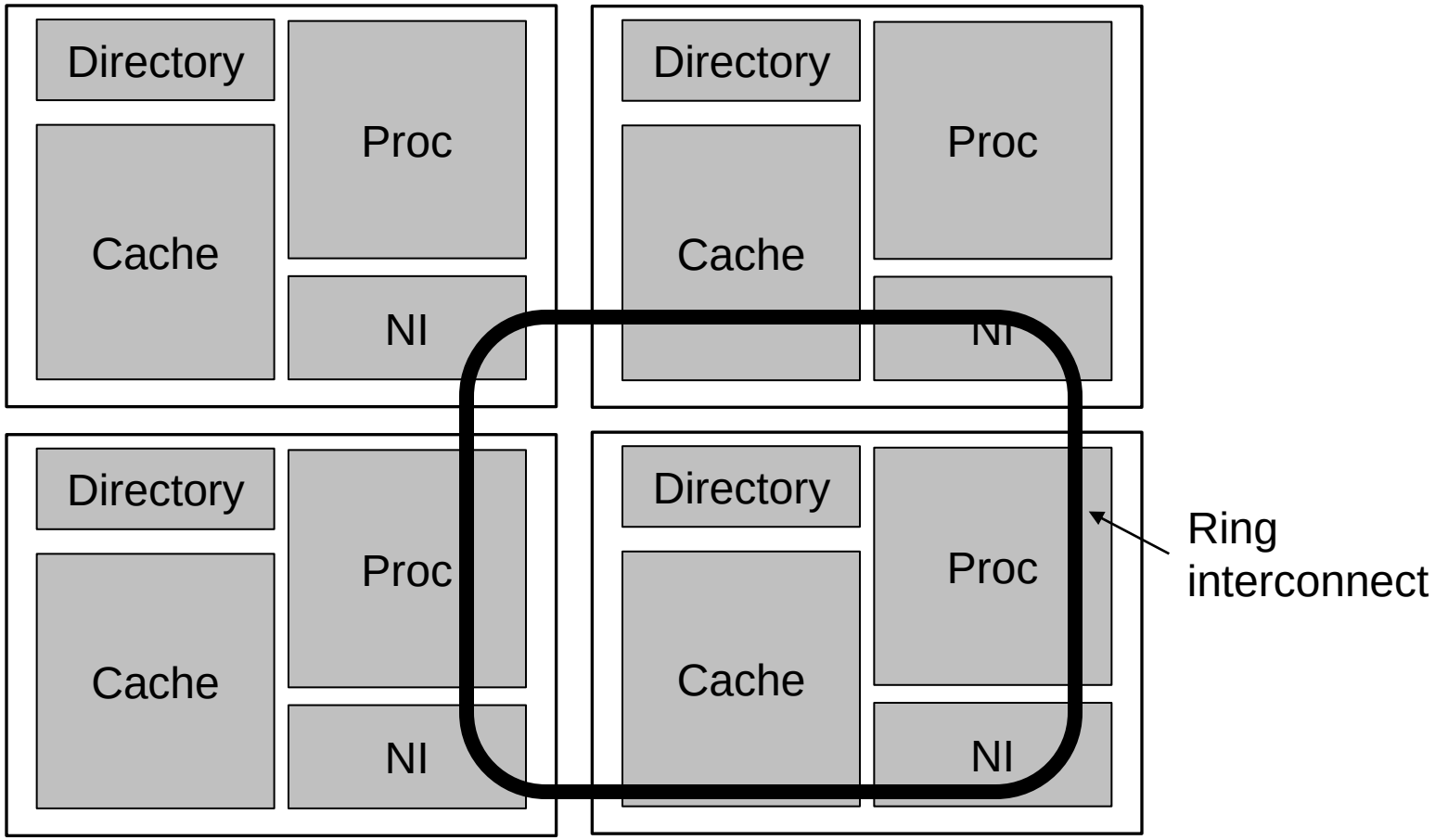
Solution: distributed coherence via *directories*

- Coherence still requires single point of serialization (for write serialization)
- But, 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)

Distributed Directories (more detail later)

Example: 4-core multicore



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: Basic Operations

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

Directory:

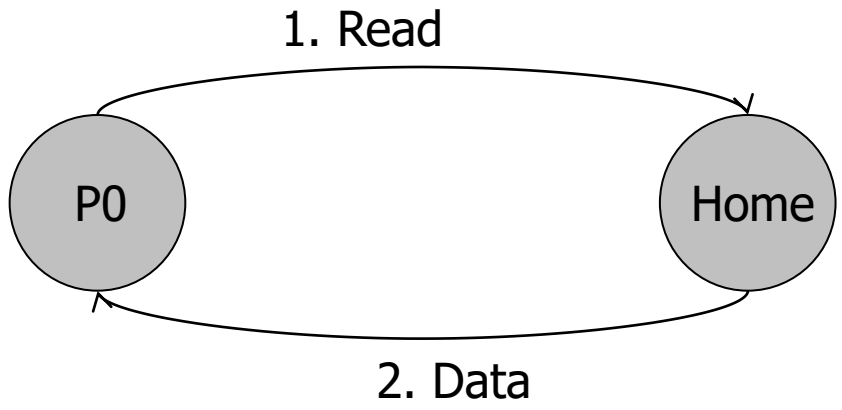
- Receives *Read, ReadEx, Upgrade* requests from nodes
- Sends *Inval/Downgrade* messages to sharers if needed
- Forwards request to memory if needed
- Replies to requestor and updates sharing state

Protocol design is flexible (VI, MSI, MESI, MOESI, etc)

MESI Directory Transaction: Read

P0 acquires an address for reading:

No other sharers
➔ Grant E (in MESI) or S (in MSI)

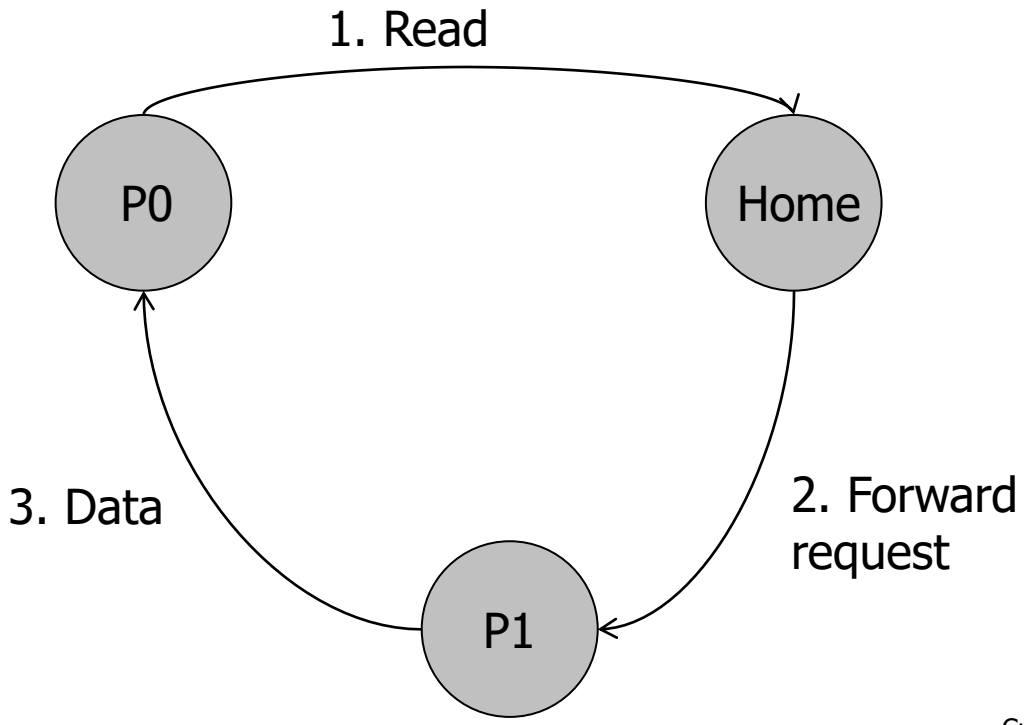


Culler/Singh Fig. 8.16

MESI Directory Transaction: Read

P0 acquires an address for reading:

Other sharers in S
➔ Grant S



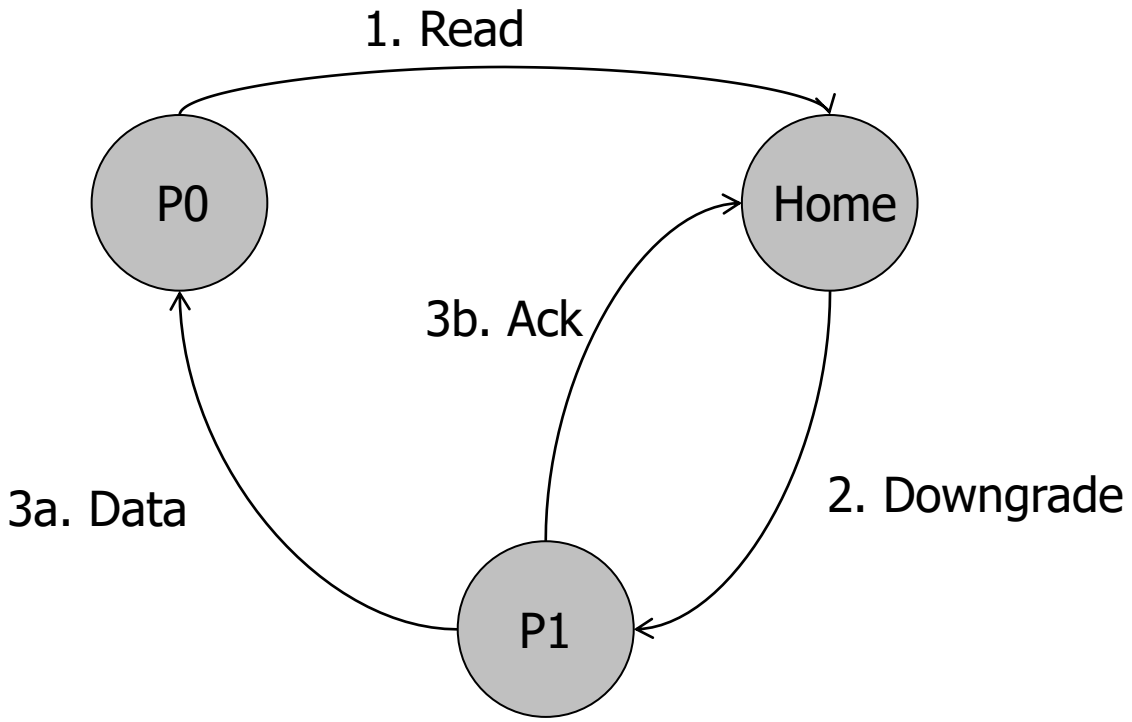
Culler/Singh Fig. 8.16

MESI Directory Transaction: Read

P0 acquires an address for reading:

Other sharers in E
➔ Grant S and downgrade

Need ACK to update directory (why?)



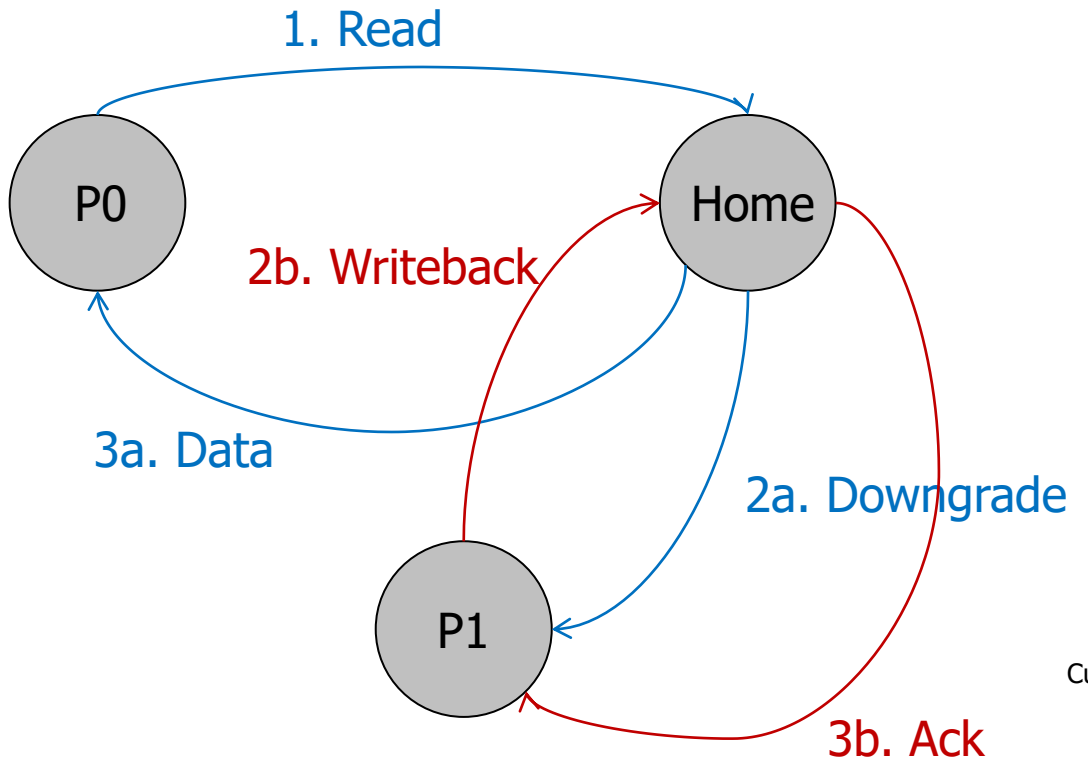
Culler/Singh Fig. 8.16

MESI Directory Transaction: Read

P0 acquires an address for reading:

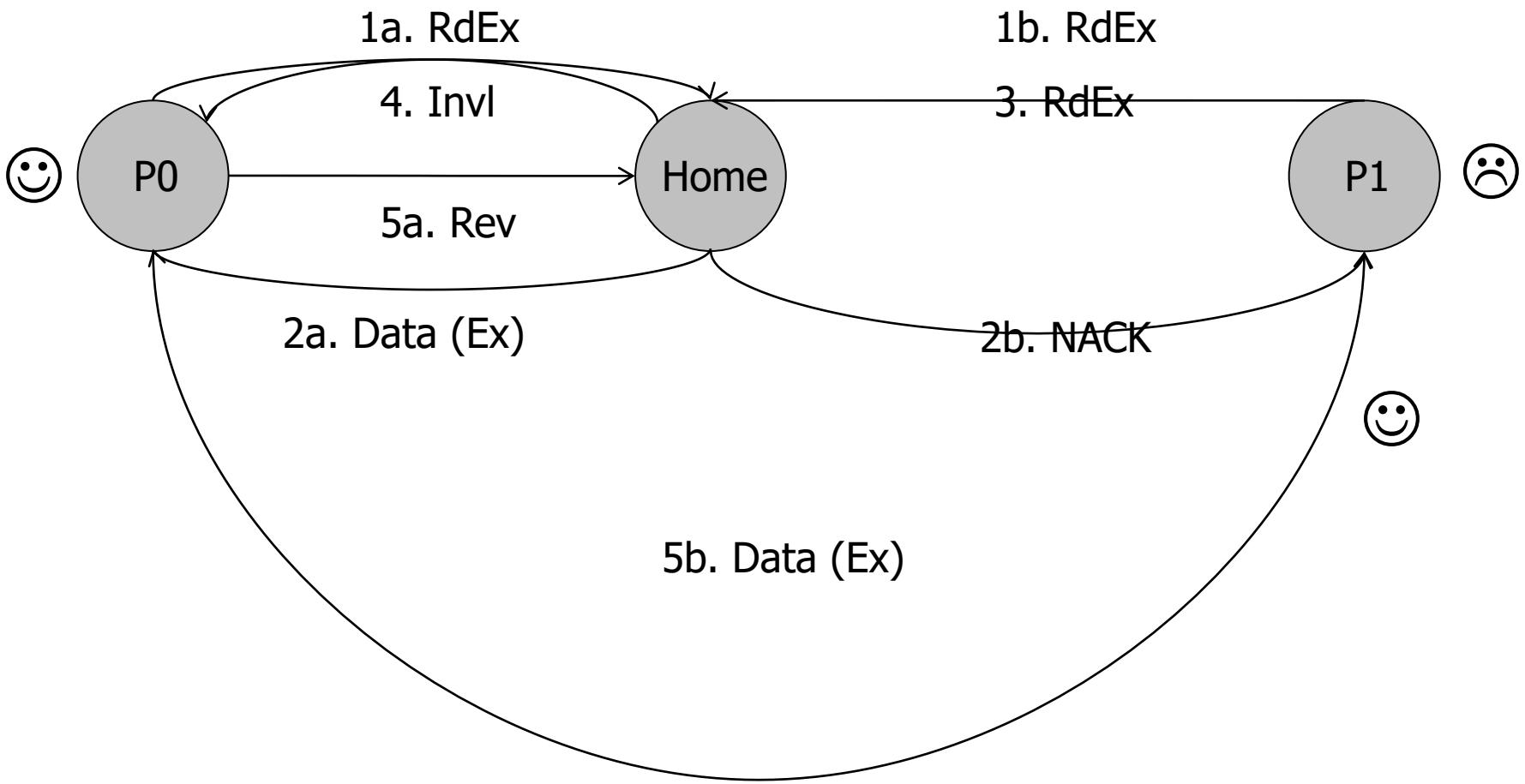
Other sharers in E
➔ Grant S and downgrade

Need ACK to update directory (why?)



Culler/Singh Fig. 8.16

Contention Resolution (for Write)



Issues with Contention Resolution

Need to escape race conditions by:

- NACKing requests to busy (pending invalidate) entries
- OR, queuing requests and granting in sequence
- (Or some combination thereof)

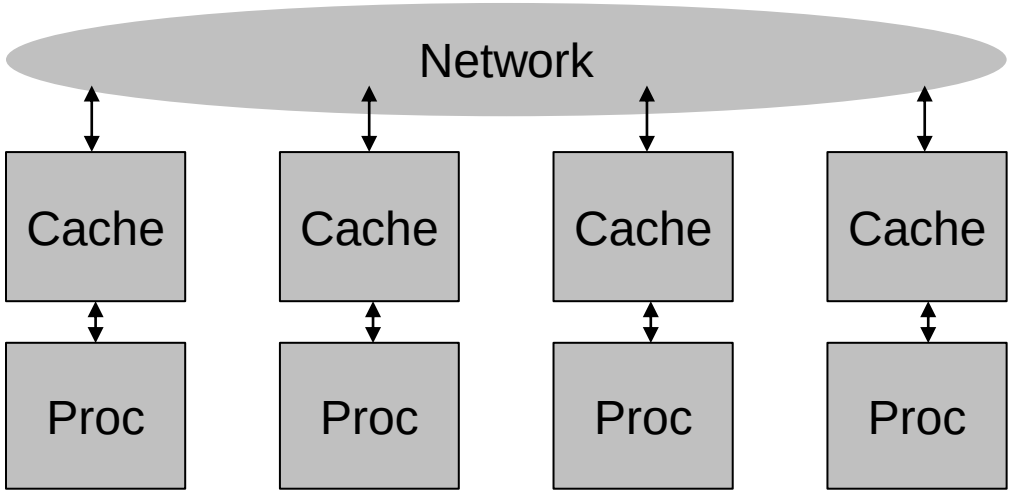
Fairness

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

Implementing directories

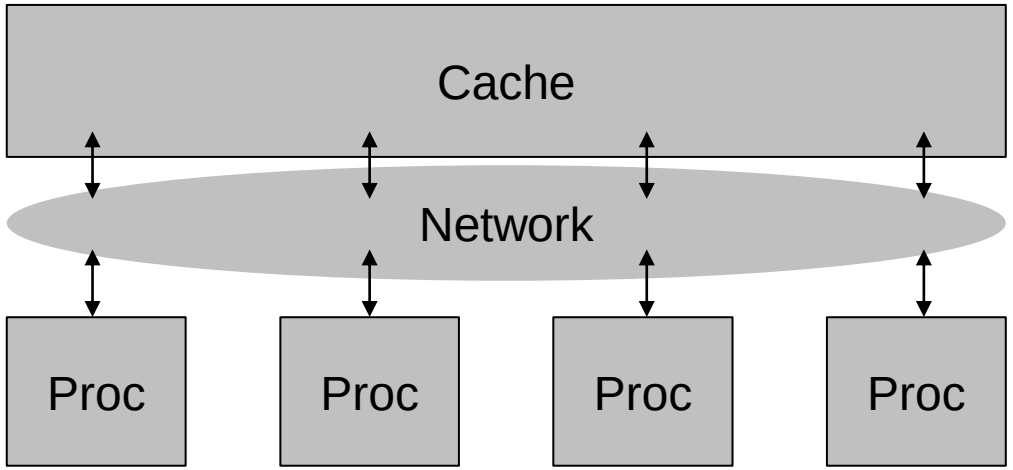
Shared vs private caches

Private caches



+ Data is nearby (low latency, high bw)
– Limited cache capacity
– Need coherence

Shared cache

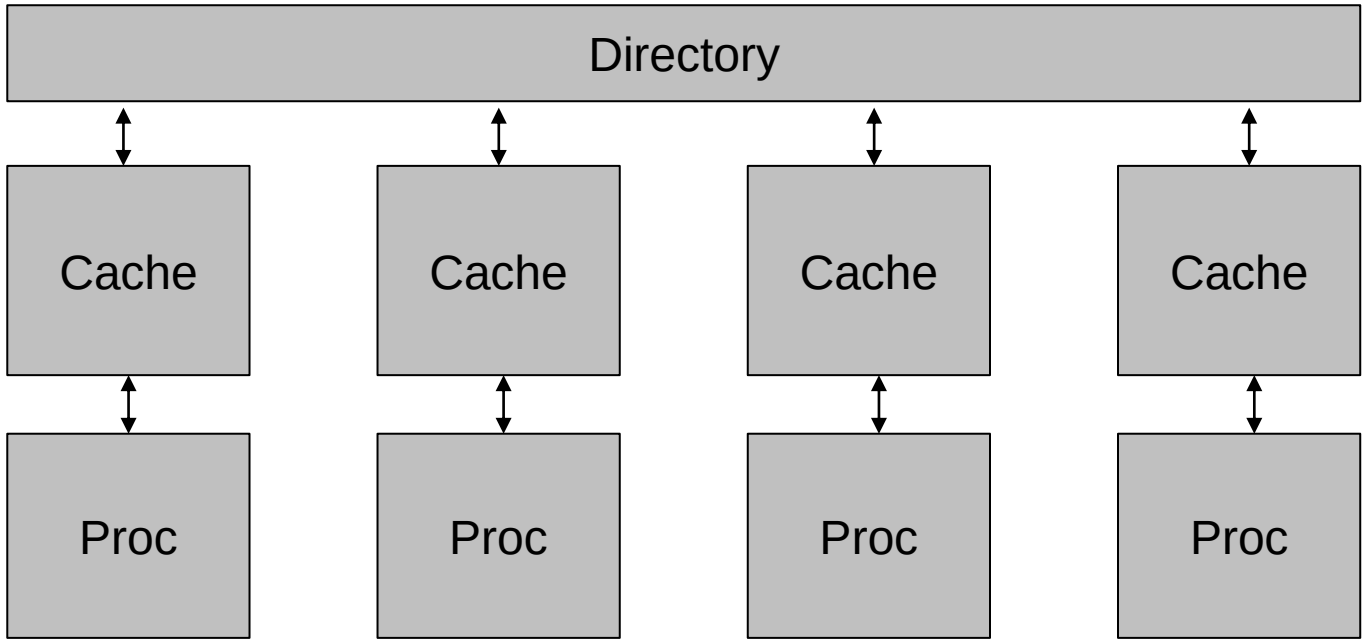


+ Lots of capacity
– Data is far away (high latency, low bw)
+ Don’t need coherence?

Private caches – logical view

Private caches keep data local near processor

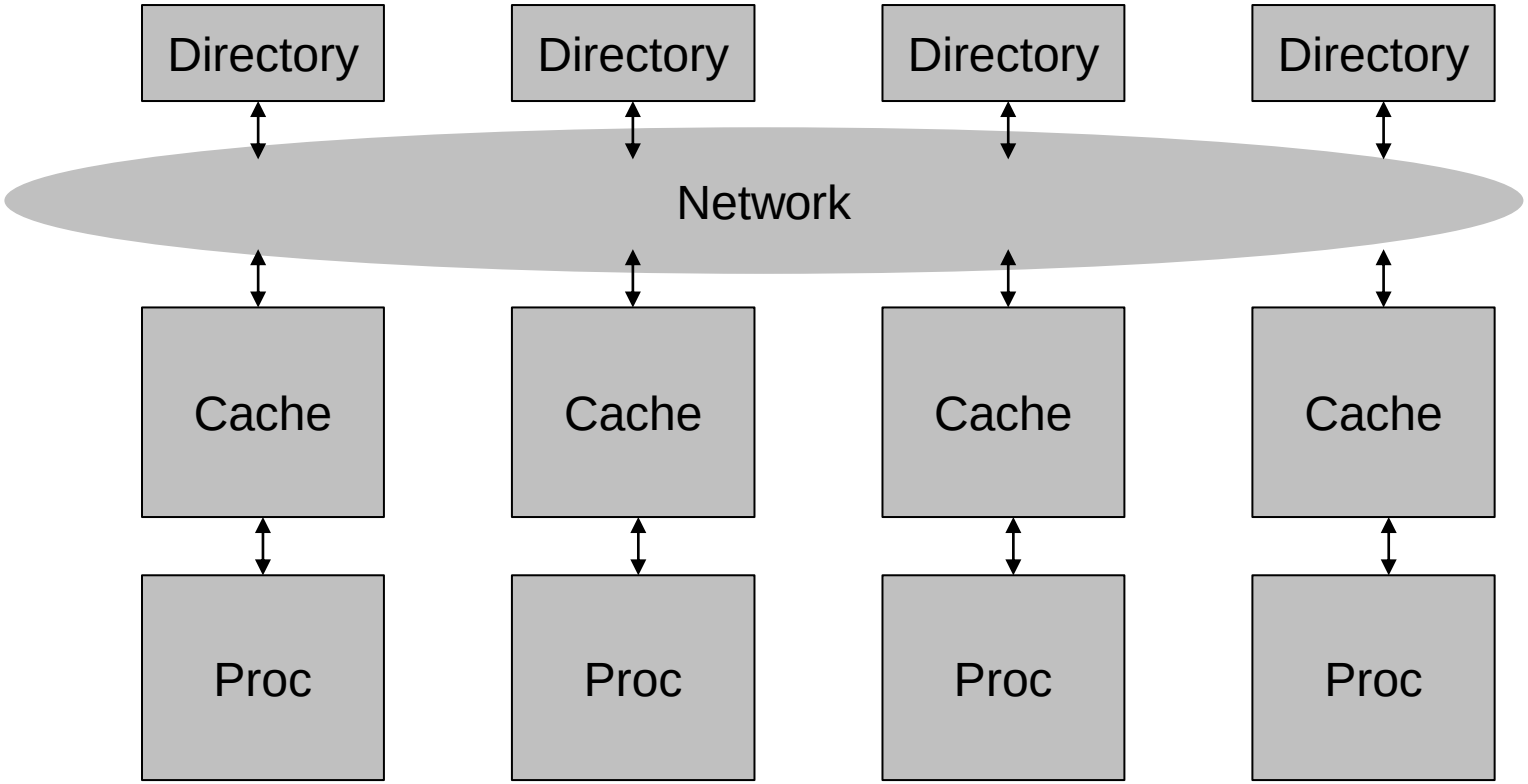
Directory arbitrates accesses + keeps coherence



Private caches – implementation

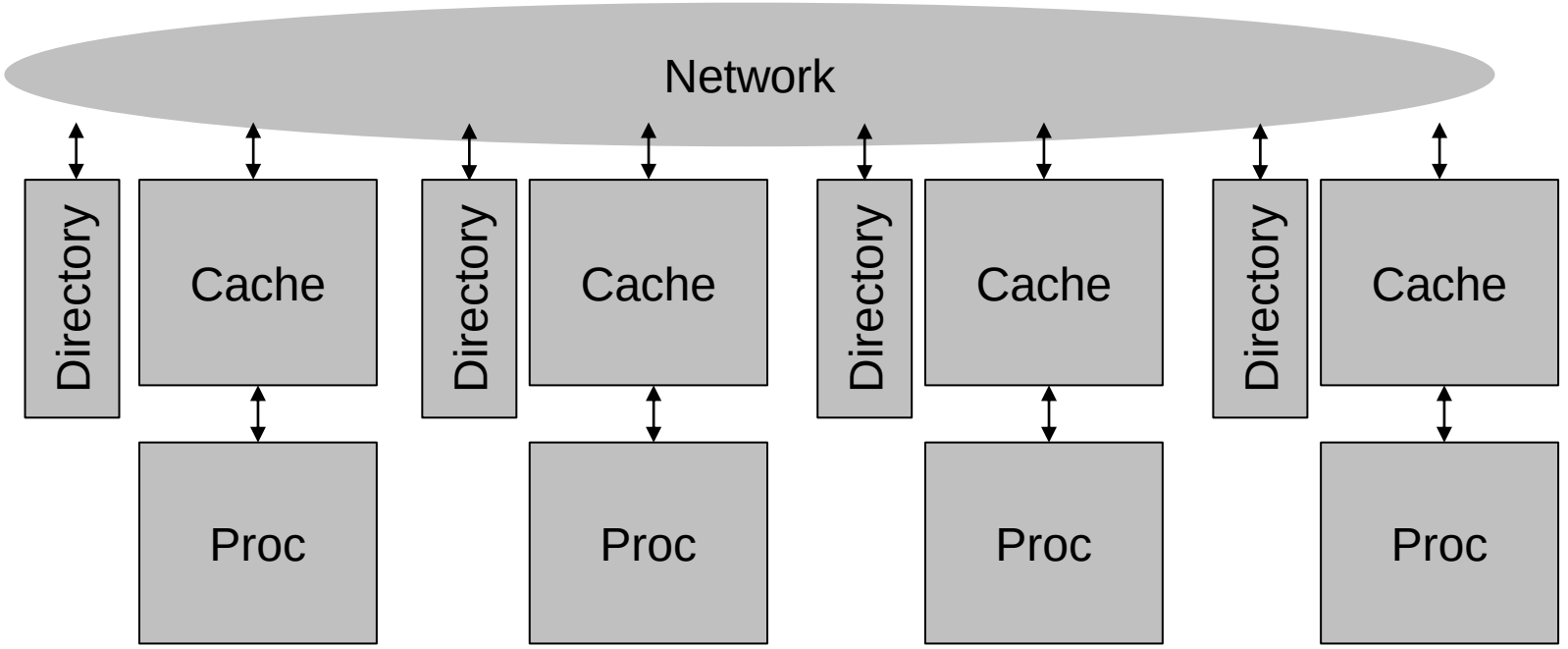
To increase bandwidth, directory is **banked**

- Increasing # ports is very expensive; better to have many, single-ported structures
- Each bank responsible for static region of address space



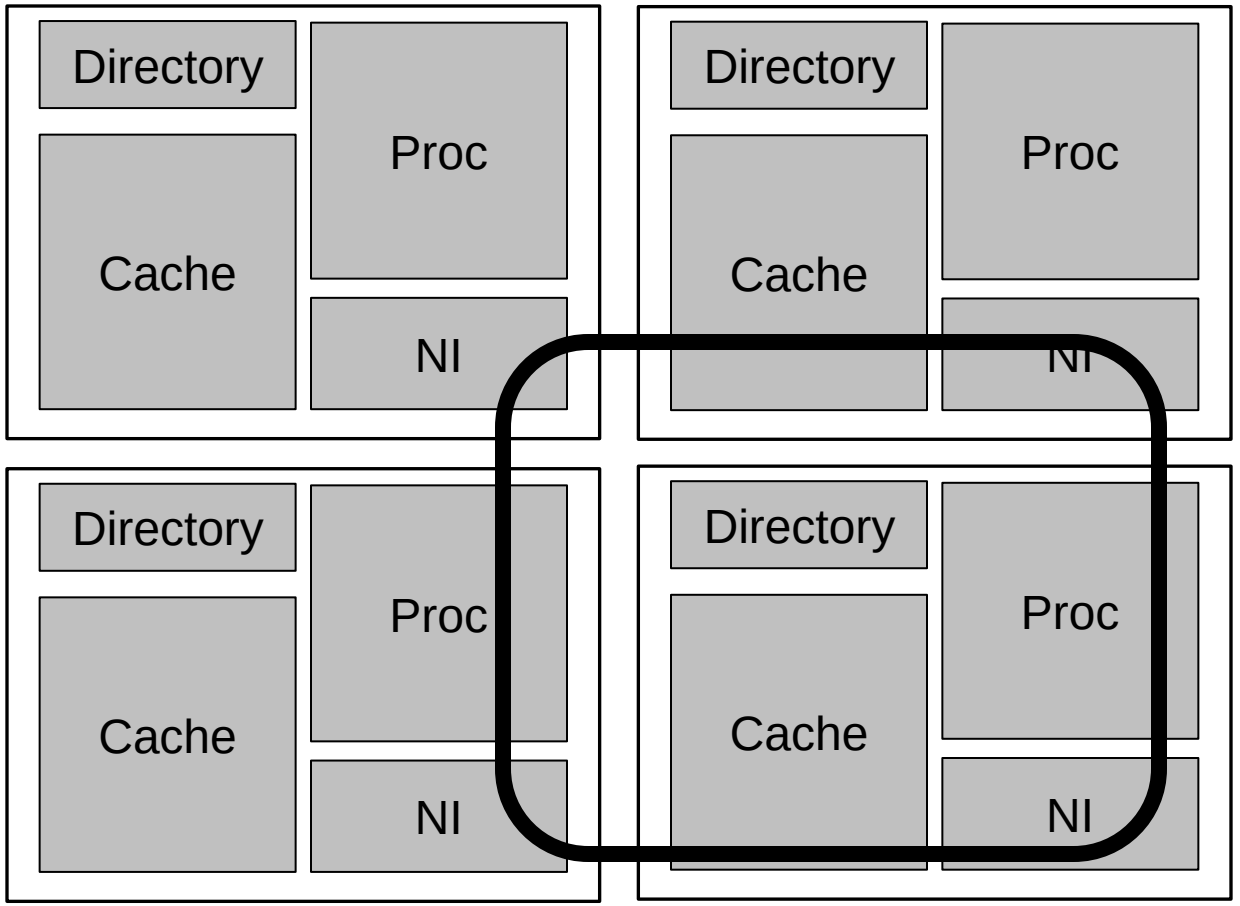
Private caches – implementation

We can put the directory on each node



Distributed caches today

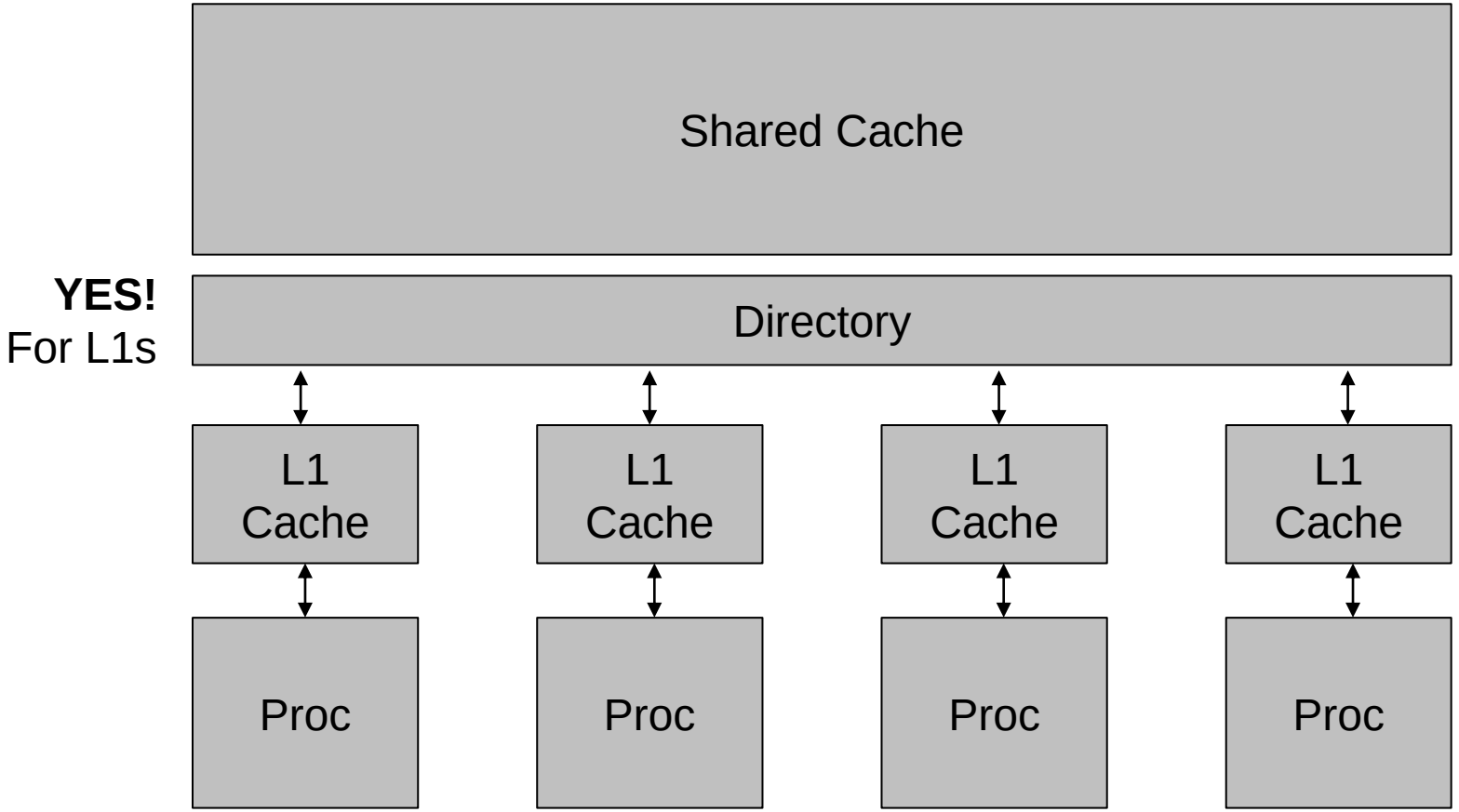
4-core system



Shared caches – logical view

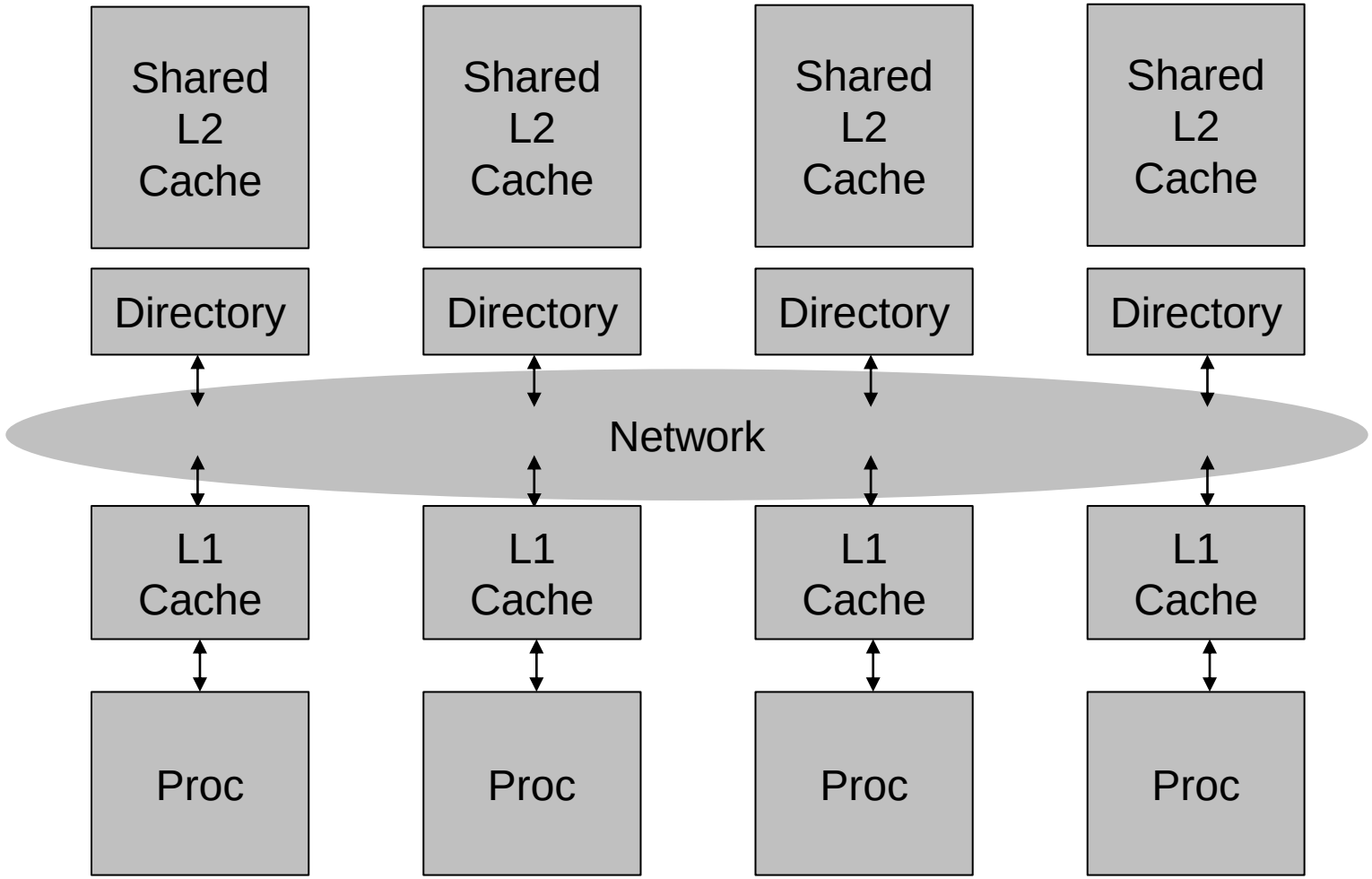
Shared caches use full cache capacity

Do we still need directory? coherence?



Shared caches – implementation

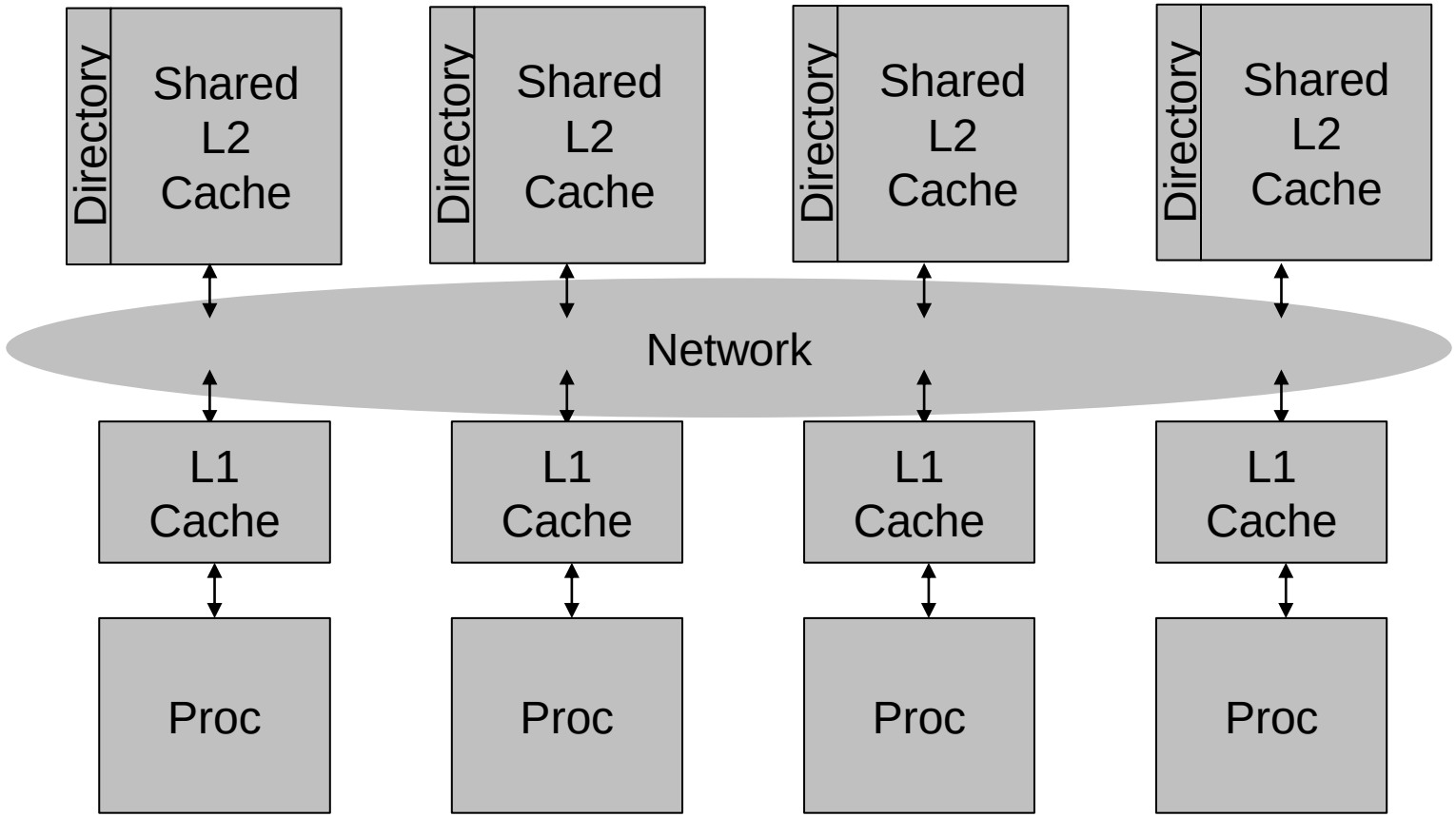
To increase bandwidth, shared cache is **banked**



Shared caches – implementation

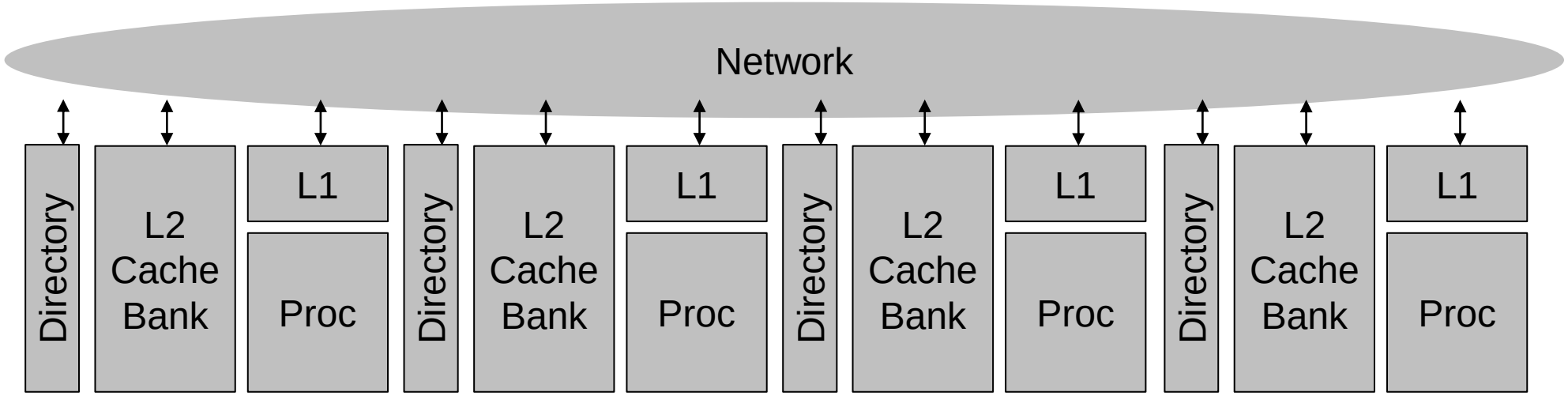
L2 + directory banks track **same addresses**

➔ Directory can be implemented in L2 tags



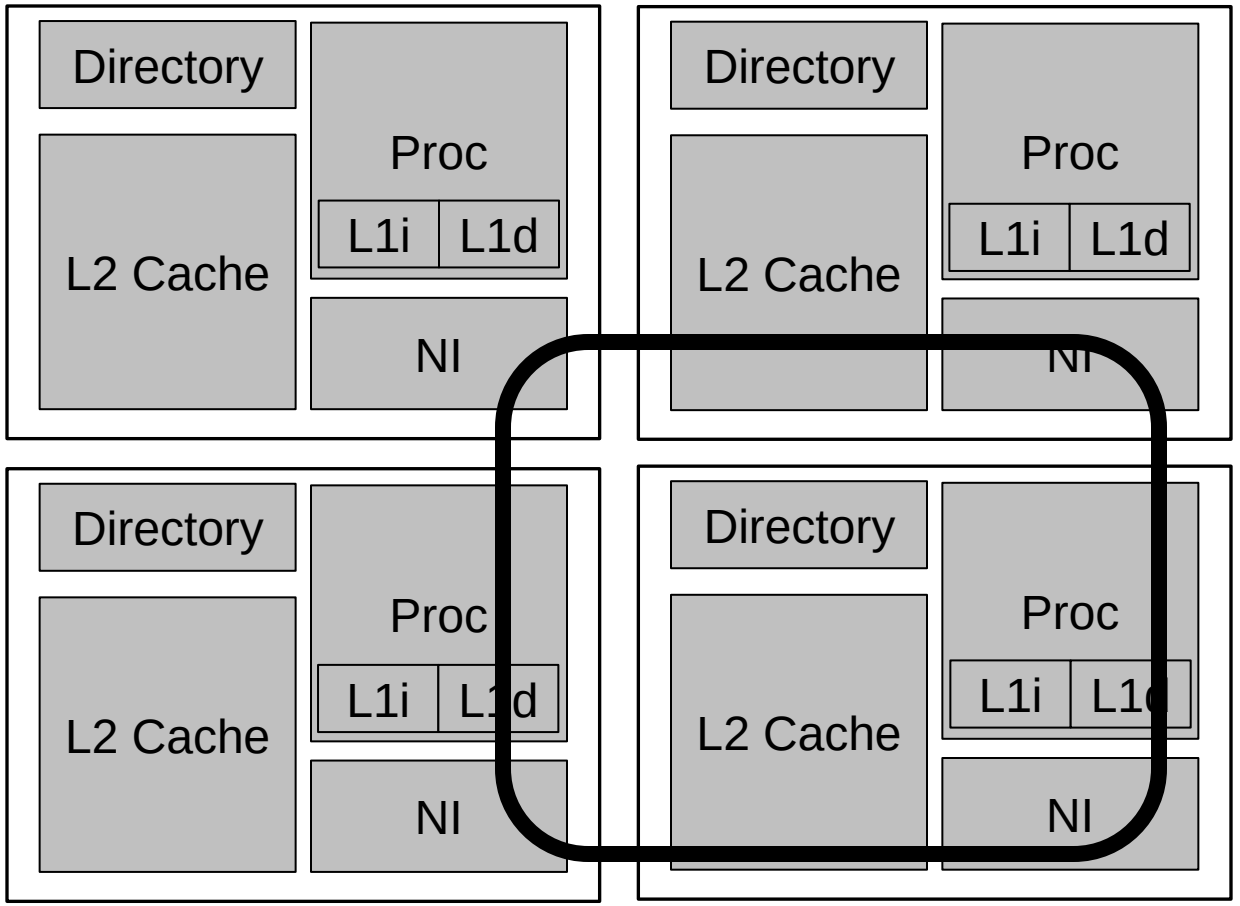
Shared caches – implementation

We can put the directory & L2 bank on each node



Distributed caches today

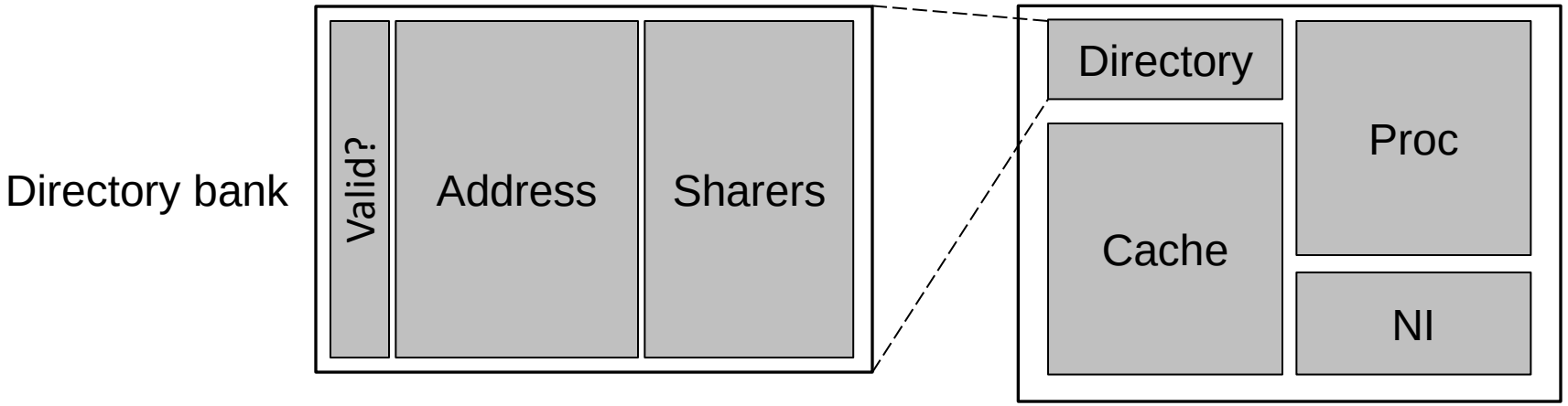
4-core system



Implementing directories

Directories are either

- Part of tags (“in-cache directories”)
- Separate cache banks that hold **metadata**, not data



What happens on a directory (not cache) eviction?

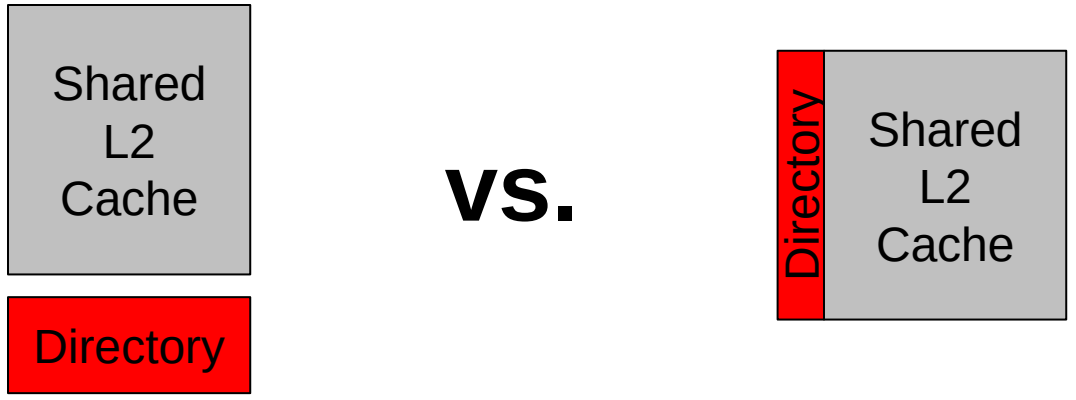
- Must invalidate all sharers, or lose coherence
- ➔ Directories tend to be intentionally overprovisioned

Directory implementation tradeoffs

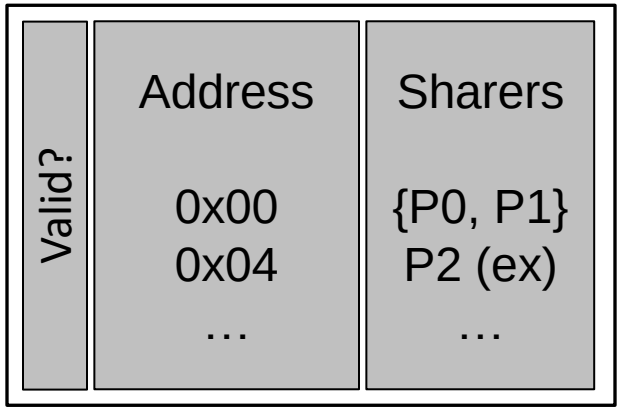
With shared caches, directory is keeping coherence **for the L1s** not the L2

➔ Many fewer lines than L2 to track

Tradeoff: do separate directory banks or in-cache directories take more area?



Scaling problems with directories



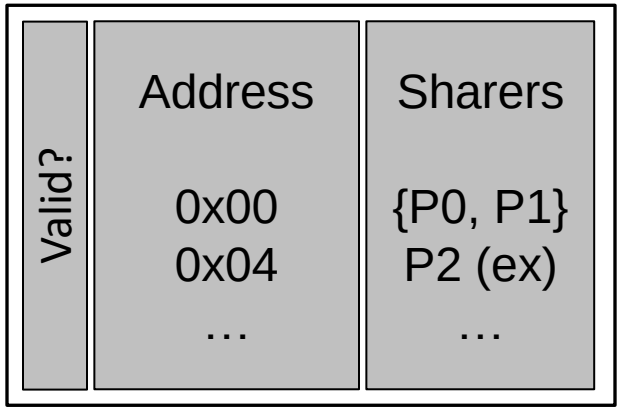
Idealized model: track all possible sharers for each line

How many bits does it take to implement a *full-mapped* directory with P processors?

- Each directory entry needs P bits
  ➔ Each directory bank is $\propto N \times P$ bits ($N =$ # lines)

- With P directory banks, overhead is $\propto N \times P^2$
  ➔ Oops!

Scalable directory implementions



Key operation is *set-inclusion* (eg, “is P3 a sharer?”)

- False positives are OK for correctness
- False positive rate determines performance

Key tradeoff: area/complexity vs runtime

- More area/complexity ➔ lower false positives

Scalable directory implementations

(A) TOLERATING FALSE POSITIVES

Limited directories

- Observation: most lines shared by few sharers
- Idea: Support ~4 sharers, then broadcast

Bloom filters

- Space-efficient approximate tracking of sharers
- Problem: How to remove a sharer?

(B) TOLERATING COMPLEXITY

Observation: There can be **at most** $N \times P$ sharers across all lines!

➔ This should scale; $N \times P^2$ is overkill

Lists of sharers

- Distributed doubly-linked list maintained at each sharer

More efficient encoding

- Vary directory bits per line based on # sharers
- [Sanchez+, HPCA'12]

Cache Coherence Summary

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*

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)

Self-check questions

Give an example program where an update protocol would outperform an invalidate protocol, and vice versa. Why is invalidate generally preferred?

Does directory-based coherence offer any advantages over snoopy coherence in a bus-based system?

Discuss the advantage, from a programming perspective, of cache coherence in easing the transition from writing sequential to parallel code.

Do other models of parallel computation (e.g., dataflow, data parallel) require coherence? Why or why not?