Consistency & Coherence

15-740 SPRING'18

NATHAN BECKMANN (SLIDES BASED ON ONUR MULTLU'S)

Review: Multiprocessor Types

Loosely coupled multiprocessors

- No shared global memory address space
- Multicomputer network
  - Network-based multiprocessors
- Usually programmed via message passing
  - Explicit calls (send, receive) for communication

Tightly coupled multiprocessors

- Shared global memory address space
- Traditional multiprocessing: symmetric multiprocessing (SMP)
  - Existing multi-core processors, multithreaded processors
- Programming model similar to uniprocessors (i.e., multitasking uniprocessor) except
  - Operations on shared data require synchronization

Review: Main Issues in Tightly-Coupled MP

Shared memory synchronization

- Locks, atomic operations

Coherence: Caches keep values consistent

- More commonly called cache coherence

Consistency: Ordering of memory operations

- What should the programmer expect the hardware to provide?

Rule of thumb: Coherence is about one address, consistency about multiple addresses

- “Coherence” is a special case of consistency; term not used in other communities

Review: Caveats of Parallelism

Amdahl’s Law

- f: Parallelizable fraction of a program
- N: Number of processors

$$\text{Speedup} = \frac{1}{1 - f + \frac{f}{N}}$$

- Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” AFIPS 1967.

Maximum speedup limited by serial portion: Serial bottleneck

Parallel portion is usually not perfectly parallel

Bottlenecks in Parallel Portion

Synchronization: Operations manipulating shared data cannot be parallelized

- Locks, mutual exclusion, barrier synchronization
- Communication: Tasks may need values from each other

- Causes thread serialization when shared data is contended

Load Imbalance: Parallel tasks may have different lengths

- Due to imperfect parallelization or microarchitectural effects

- Reduces speedup in parallel portion

Resource Contention: Parallel tasks can share hardware resources, delaying each other

- Replicating all resources (e.g., memory) expensive

- Additional latency not present when each task runs alone

Difficulty in Parallel Programming

Little difficulty if parallelism is natural

- "Embarrassingly parallel" applications
- Multimedia, physical simulation, graphics
- Large web servers, databases?

Difficulty is in

- Getting parallel programs to work correctly
- Optimizing performance in the presence of bottlenecks

➔ Much of **parallel computer architecture** is about

- Designing machines that overcome the sequential and parallel bottlenecks to achieve higher performance and efficiency
- Making programmer's job easier in writing correct and high-performance parallel programs

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
- E.g., Load-store queues in out-of-order execution

Memory Ordering in a Single Processor

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 operations in program order

Advantages:

- Architectural state is precise within an execution.
- Architectural state is consistent across different runs of the program → Easier to debug programs

Disadvantage: Preserving order adds overhead, reduces performance, particularly with speculation

Memory Ordering in a Dataflow Processor

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

Memory Ordering in a MIMD Processor

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?

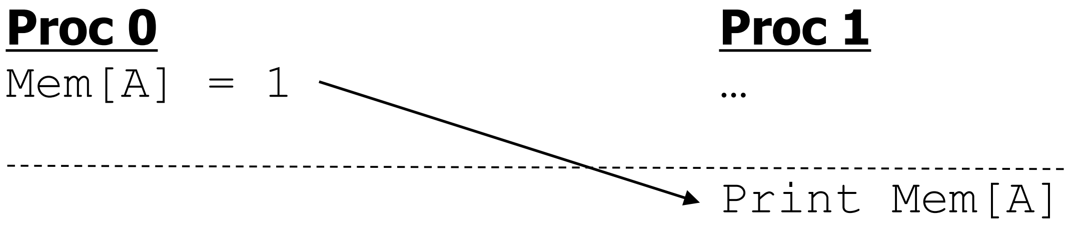
- In other words, what is the ordering of operations across different processors?

Memory Ordering in SAS 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



Proc 0

```
Mem[A] = 1
```

Proc 1

…

```
Print Mem[A]
```

Each read should receive the value last written by anyone

- This requires synchronization (what does “last written” mean?)

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

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)

Correctness

- Can we have incorrect execution if the order of memory operations is different from the point of view of different processors?

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* (locks, semaphores, condition variables)

Only one thread can execute a critical section at a given time

- Mutual exclusion principle

A multiprocessor should provide the synchronization primitives to enable the programmer to protect shared data

How Can We Solve The Problem?

Sequential consistency (SC)

- Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," IEEE Transactions on Computers, 1979

Idea: All processors see the same global ordering of memory operations

- Each thread's operations appear in sequential order within the global order

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 Orders

Potential correct global orders (all are correct):

- T1: A B
- T2: X Y

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

Corollaries

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?
  - E.g., total store order (TSO) memory model
- How about a enforcing a global order only at the boundaries of synchronization?
  - Relaxed memory models
  - Release consistency model

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
  - → Simplifies high-performance hardware implementation
  - → 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

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

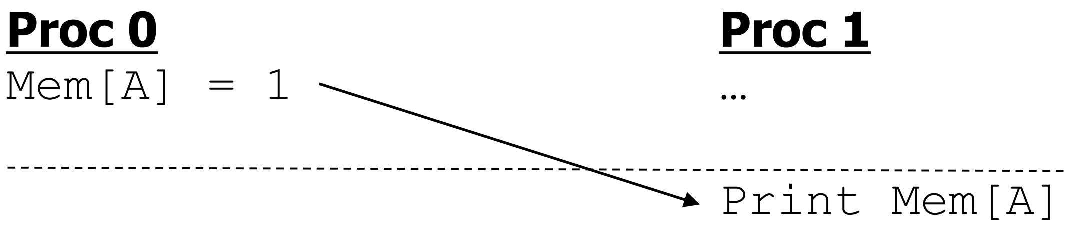
- A memory location is now present in multiple places
- Prevents the effect of a store to be seen by other processors
- ➔ Coherence vs consistency

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



Proc 0

```
Mem[A] = 1
```

Proc 1

…

```
Print Mem[A]
```

Each read should receive the value last written by anyone

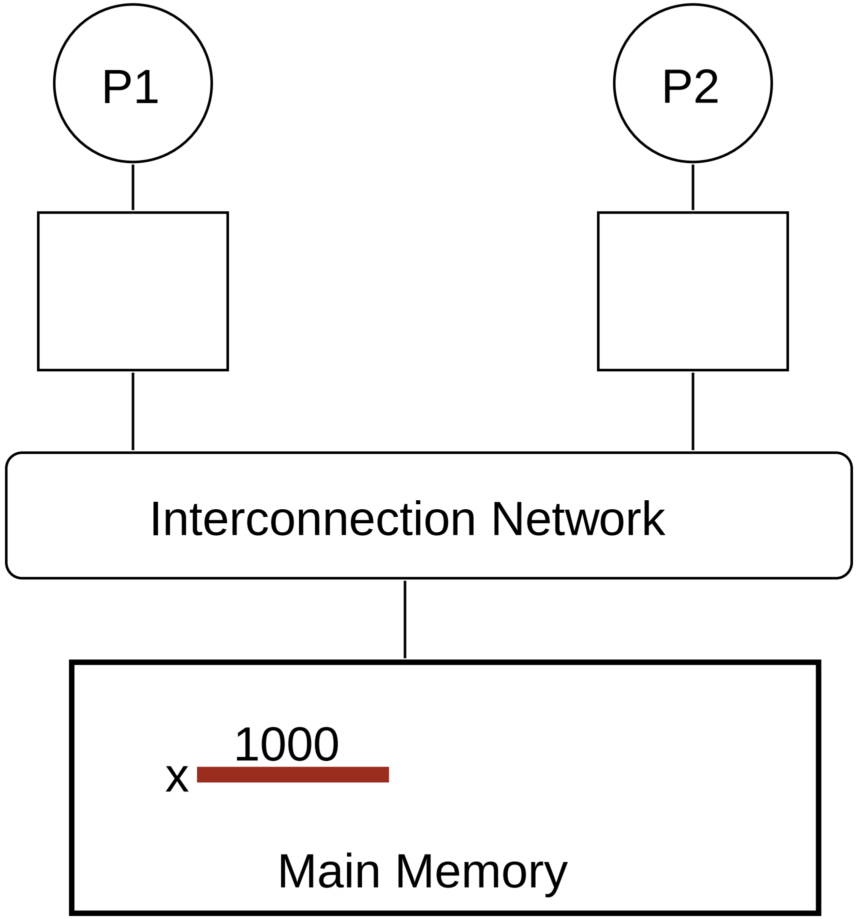
- This requires synchronization (what does last written mean?)

What if Mem[A] is cached (at either end)?

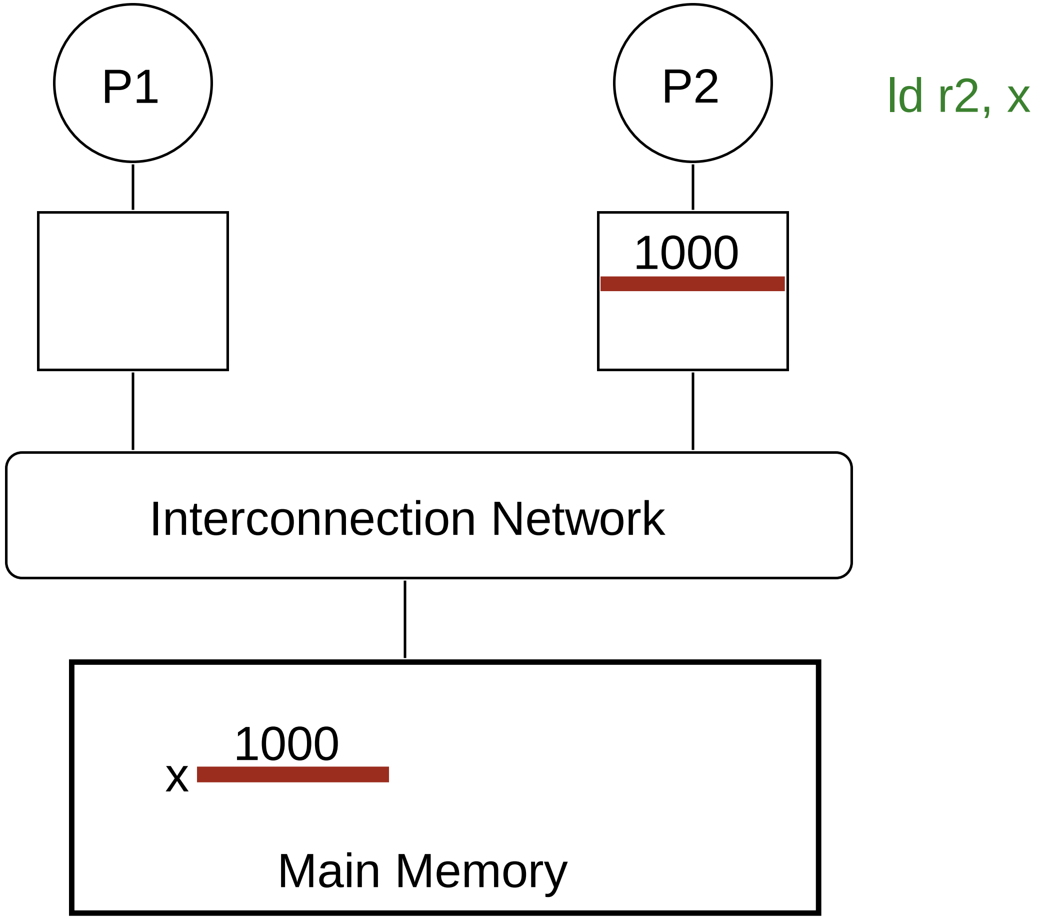
Cache Coherence

Cache Coherence

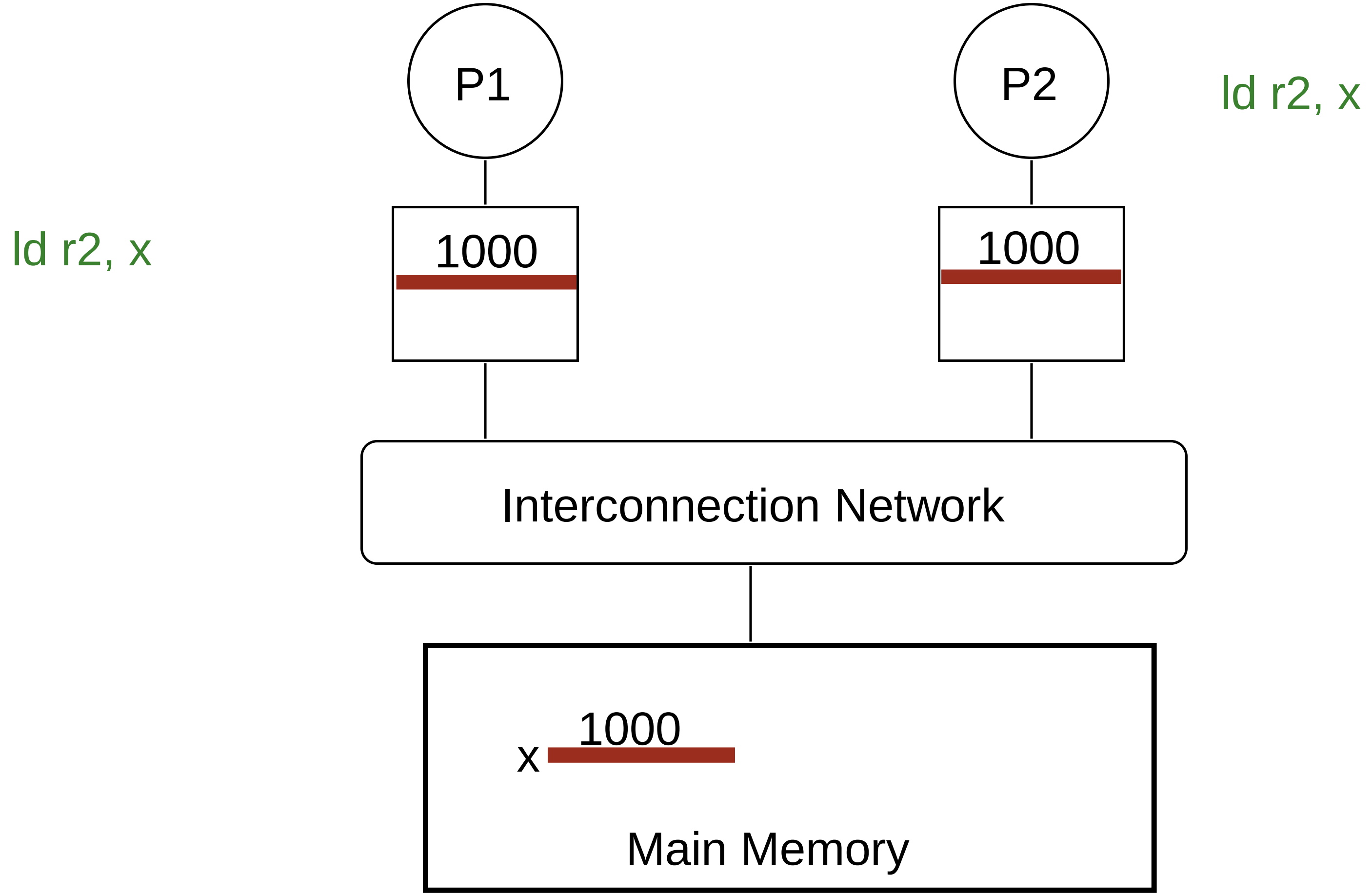
Basic question: If multiple processors cache the same block, how do they ensure they all see a consistent state?



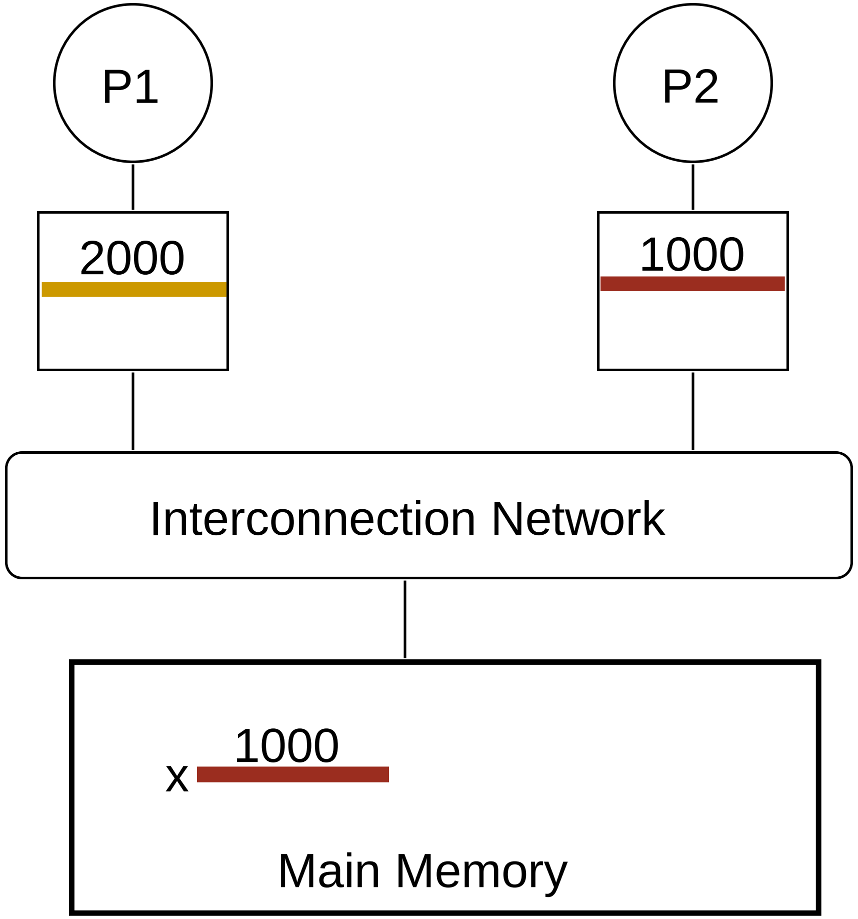
The Cache Coherence Problem



The Cache Coherence Problem



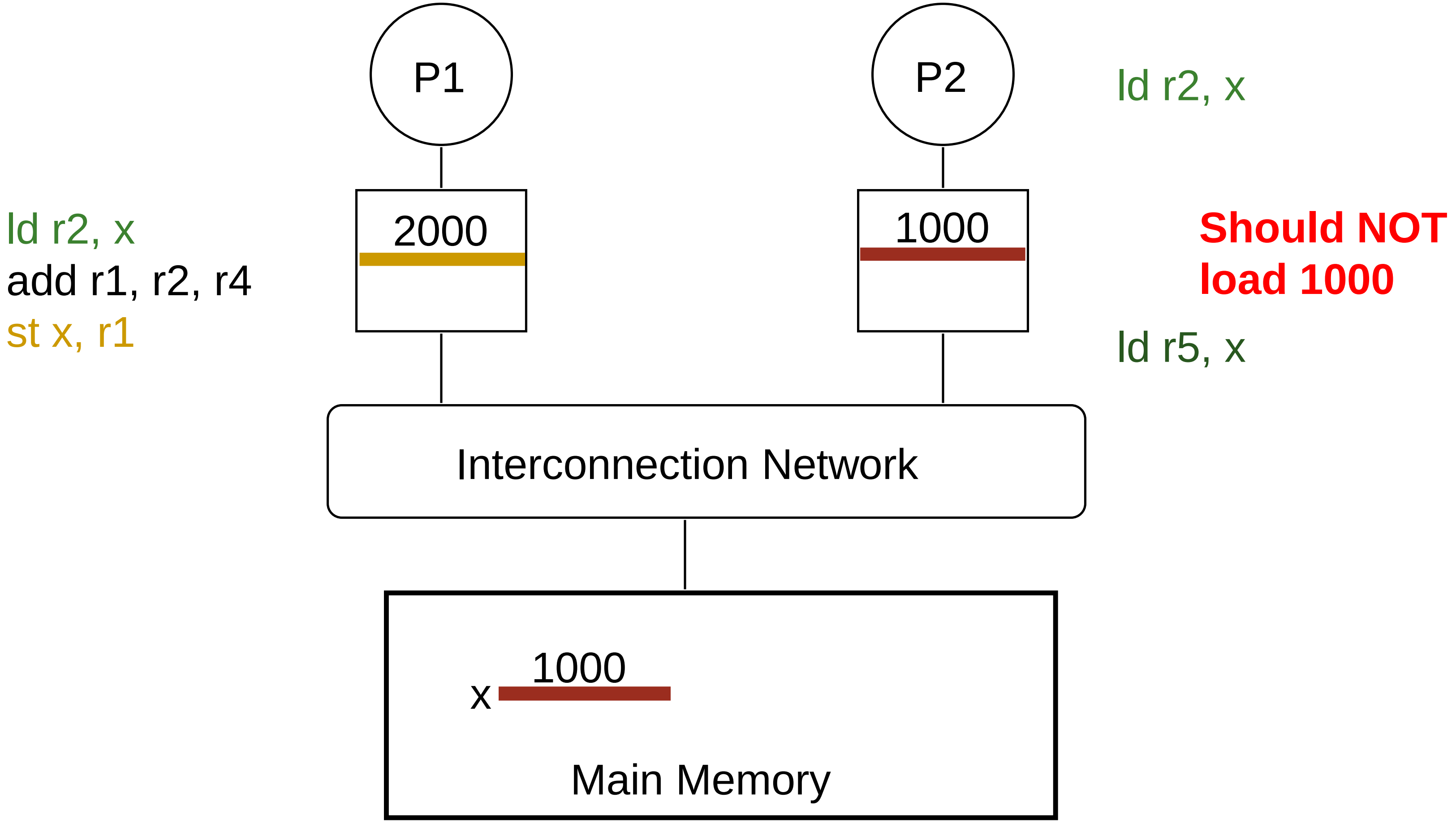
The Cache Coherence Problem



ld r2, x
add r1, r2, r4
st x, r1

ld r2, x

The Cache Coherence Problem



Cache Coherence: Possible Solutions?

Can software handle coherence?

- 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.
- DeNovo [Choi et al., PACT’11] does this via “disciplined parallelism”
- ➔ Software coherence is possible but impractical in most settings (programmability, overheads)

Do we need coherence at all really?

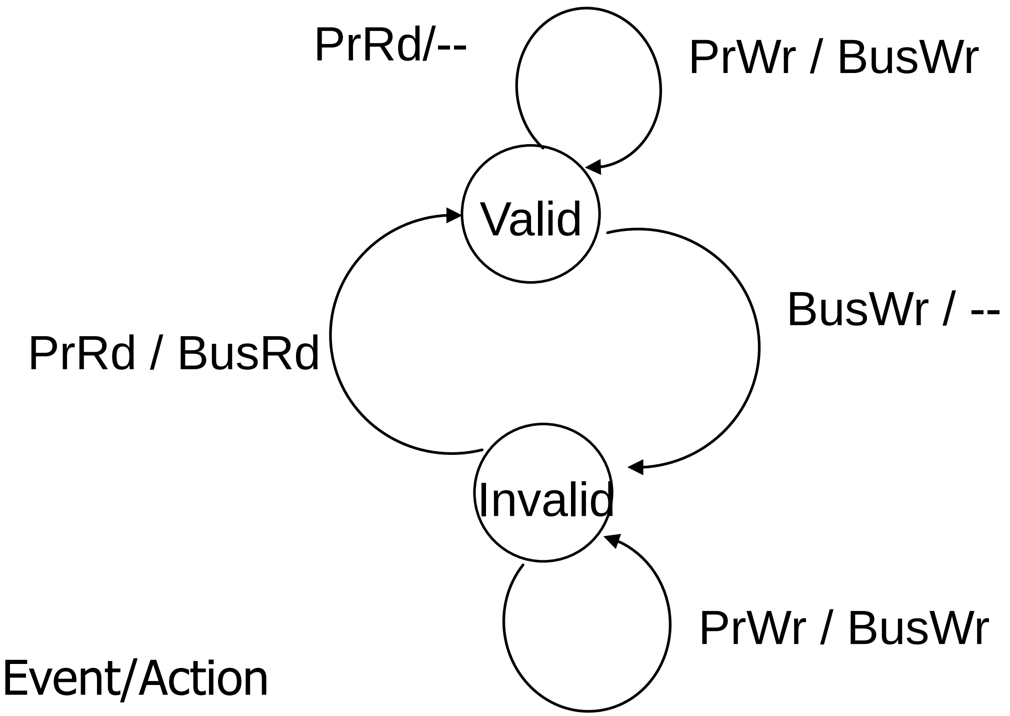
- Make all caches shared …but its hard/impossible to design a scalable shared cache
- At a minimum, L1s need to be private ➔ coherence still required

➔ Hardware-based cache coherence best solution in general

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:



ObservedEvent/Action

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

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

Basic Idea of Hardware Cache Coherence

1. Processor/cache broadcasts its write/update to a memory location to all other processors

2. Other caches either update or invalidate their local copies

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

Coherence: Update vs. Invalidate (I)

On a Read:

- If local copy isn’t valid, put out request
- If another node has a copy, it responds
- Otherwise, memory does

No difference between update/invalidate for reads

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-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 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

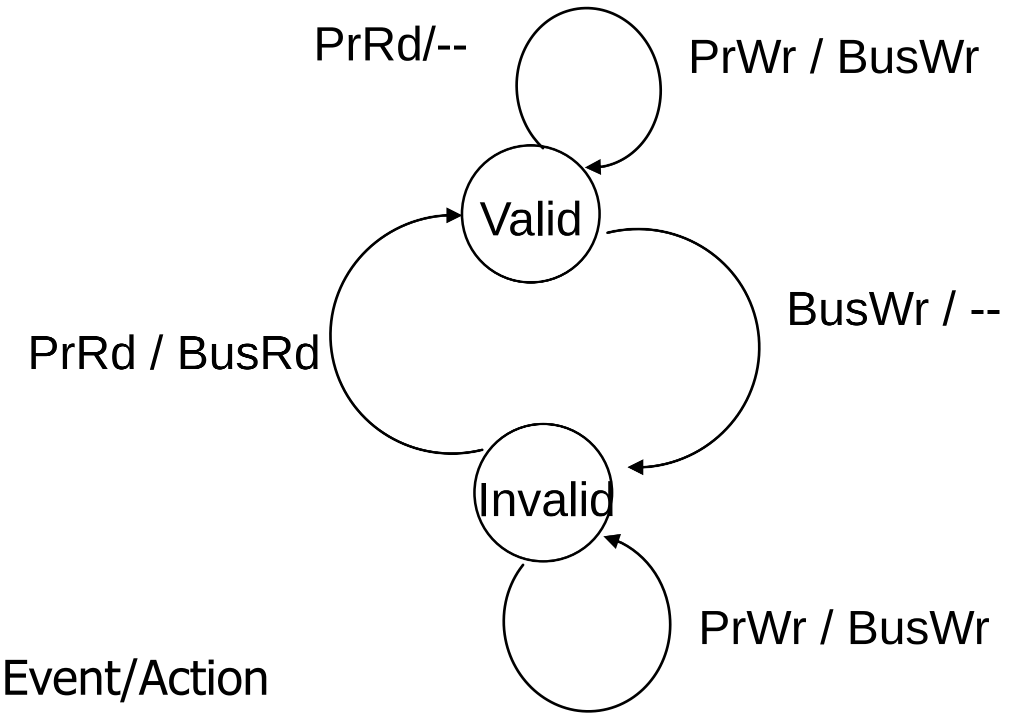
Easy to implement if all caches share a common bus

- 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?

A Very Snoopy 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:



ObservedEvent/Action

- 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

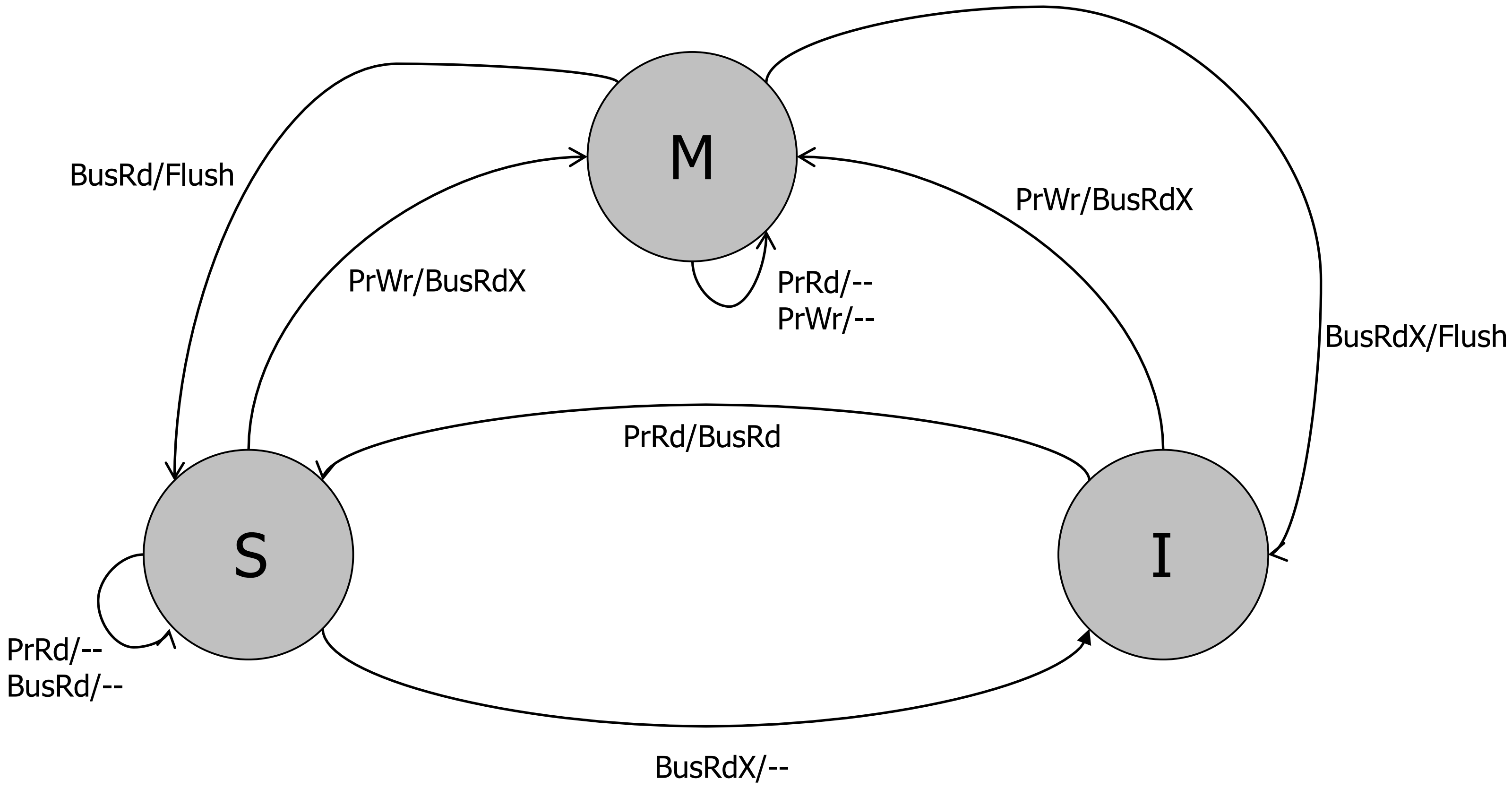
Read miss makes a *Read* request on bus, transitions to **S**

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.

MESI State Machine

Modified:

- 1 owner
- dirty data
- R/W access

Exclusive:

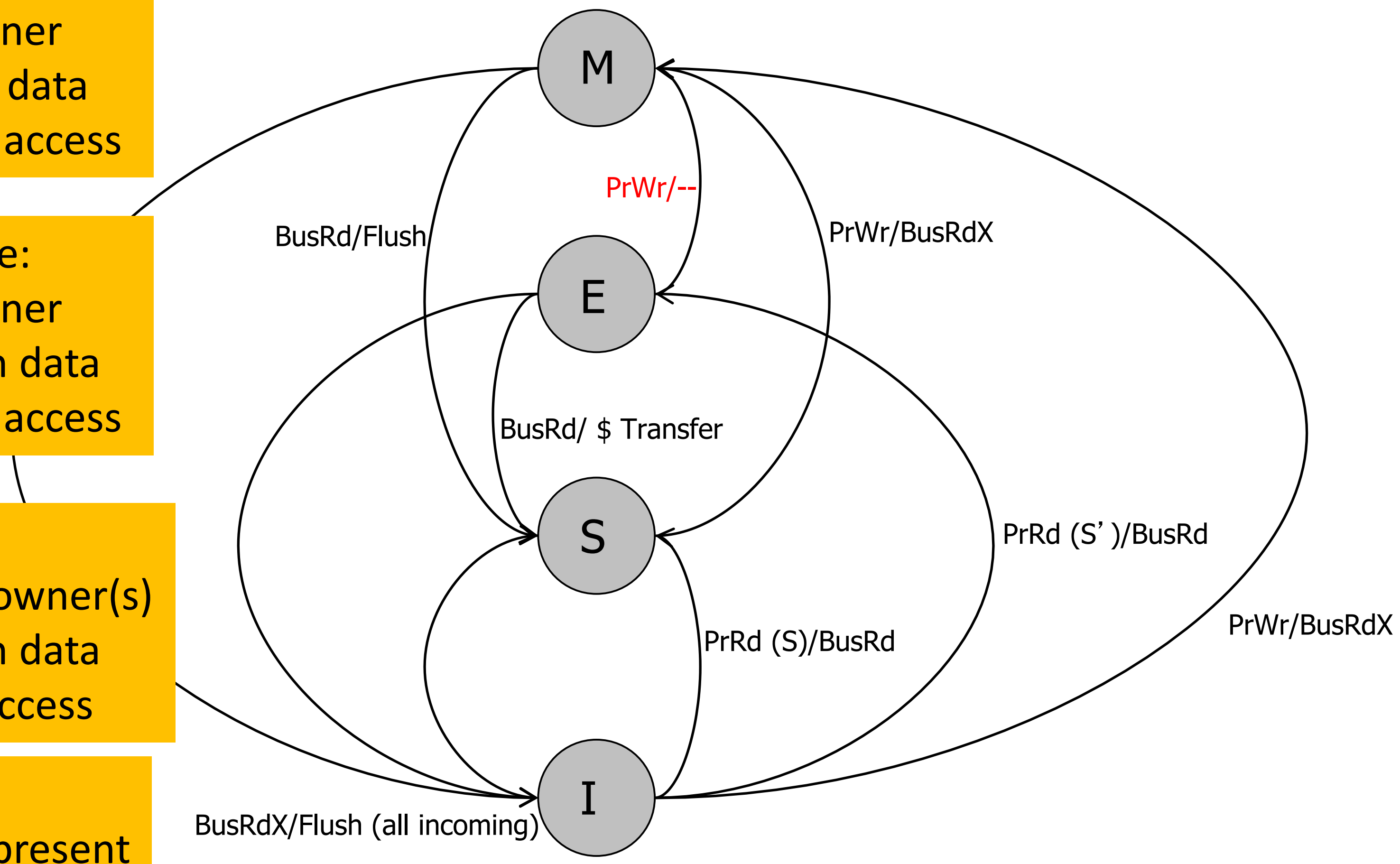
- 1 owner
- clean data
- R/W access

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
  - Gets data faster
- 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

Further Coherence Protocol Optimization

Protocol can be extended with more states & prediction mechanisms
+ Reduce unnecessary invalidates and transfers

- "Exploiting Commutativity to Reduce the Cost of Updates to Shared Data in Cache-Coherent Systems" by Zhang et al. in MICRO'15

However more states and optimizations
-- Are difficult to verify
-- Provide diminishing returns

We haven't shown all the transient states in these protocols!

- Actual implementations need ≈20 states and are difficult to verify
- ➔ Industry is very reluctant to change protocol in any way

Directory-Based Cache Coherence

Directory-Based Protocols

Buses are simple but don’t scale

- Single, shared communication channel is bottleneck

Solution: distributed coherence via *directories*

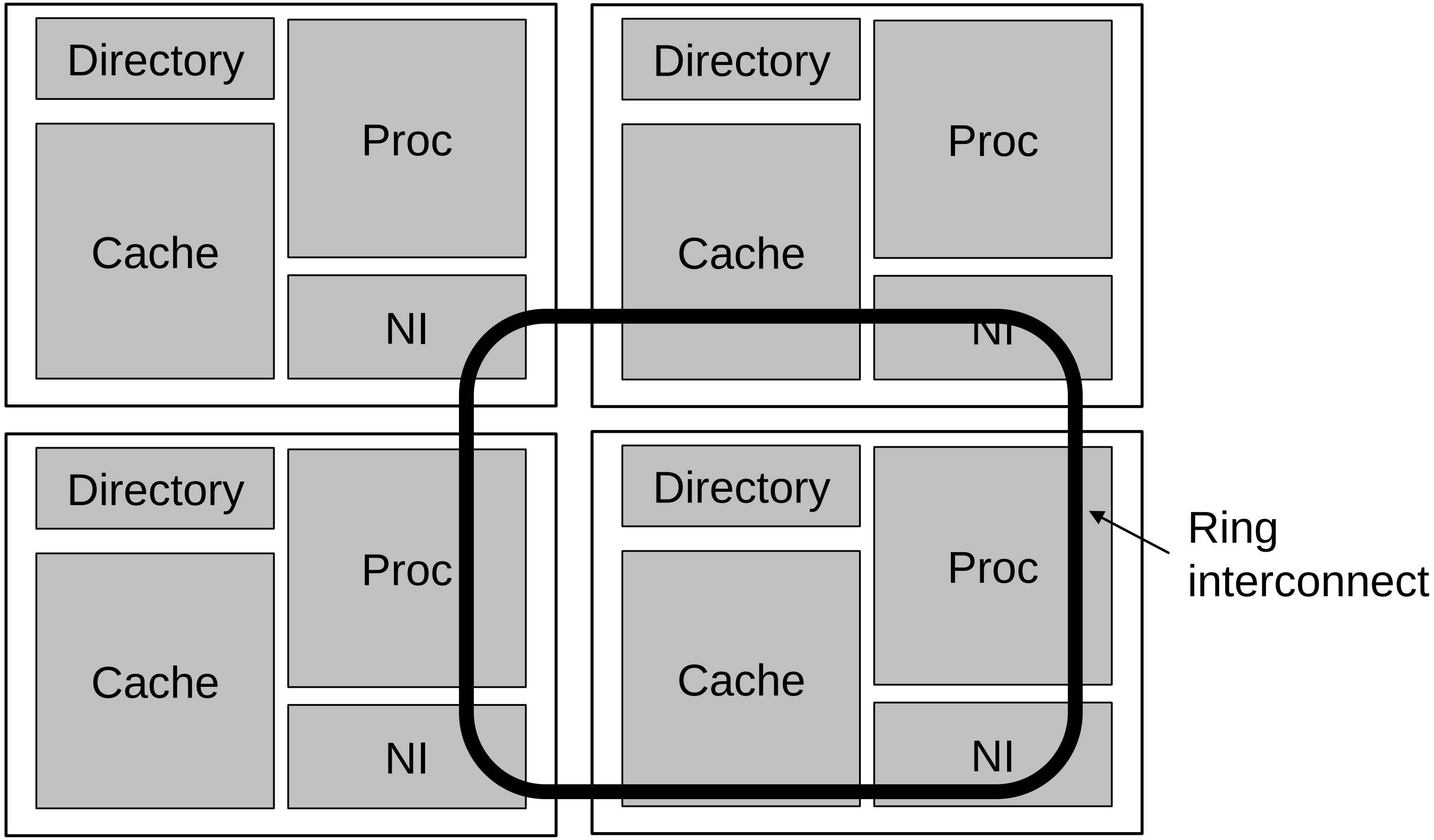
- Coherence still requires single point of serialization (for write serialization)
- Serialization location can be different for every block (e.g., striped across nodes)

We can reason about the protocol for a single block:

- One *server* (directory node), many *clients* (private caches)

Distributed Directories

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 with explicit request, reply messages to/from directory

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

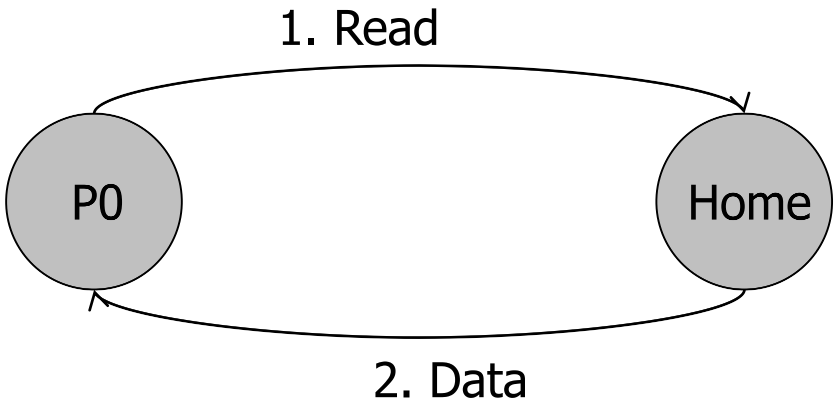
Protocol design is flexible (VI, MSI, MESI, MOESI, etc)

MESI Directory Transaction: Read

P0 acquires an address for reading:

No other sharers

➔ Grant E (in MESI) or S (in MSI)

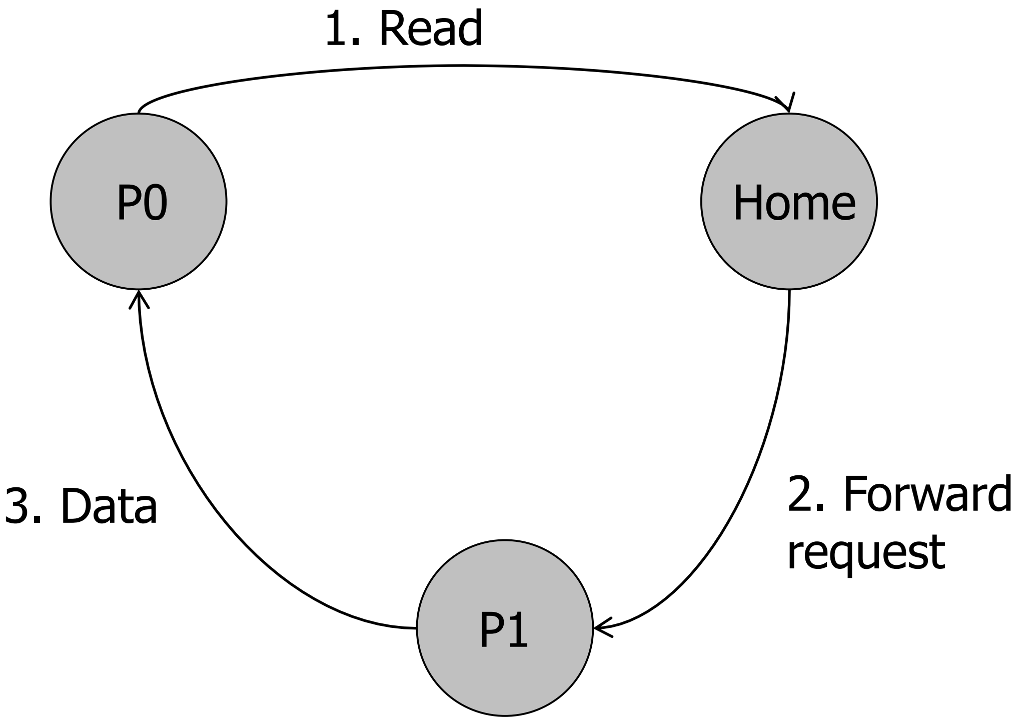


Culler/Singh Fig. 8.16

MESI Directory Transaction: Read

P0 acquires an address for reading:

Other sharers in S
➔ Grant S



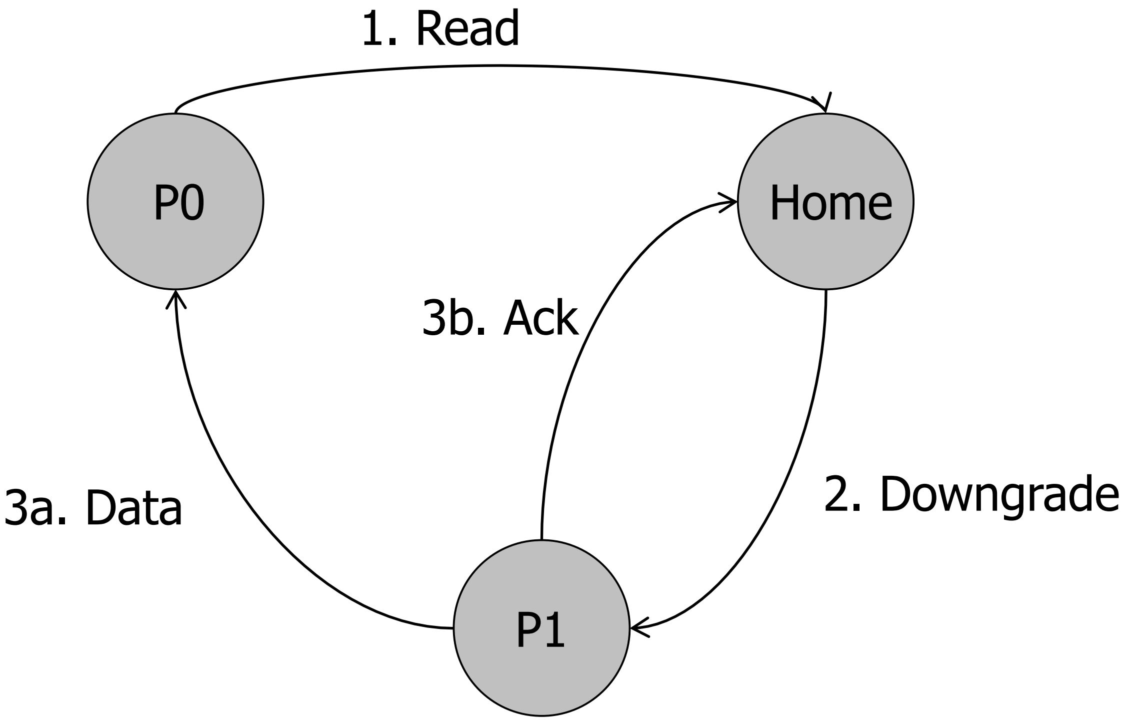
Culler/Singh Fig. 8.16

MESI Directory Transaction: Read

P0 acquires an address for reading:

Other sharers in E
➔ Grant S and downgrade

Need ACK to update directory
(why?)



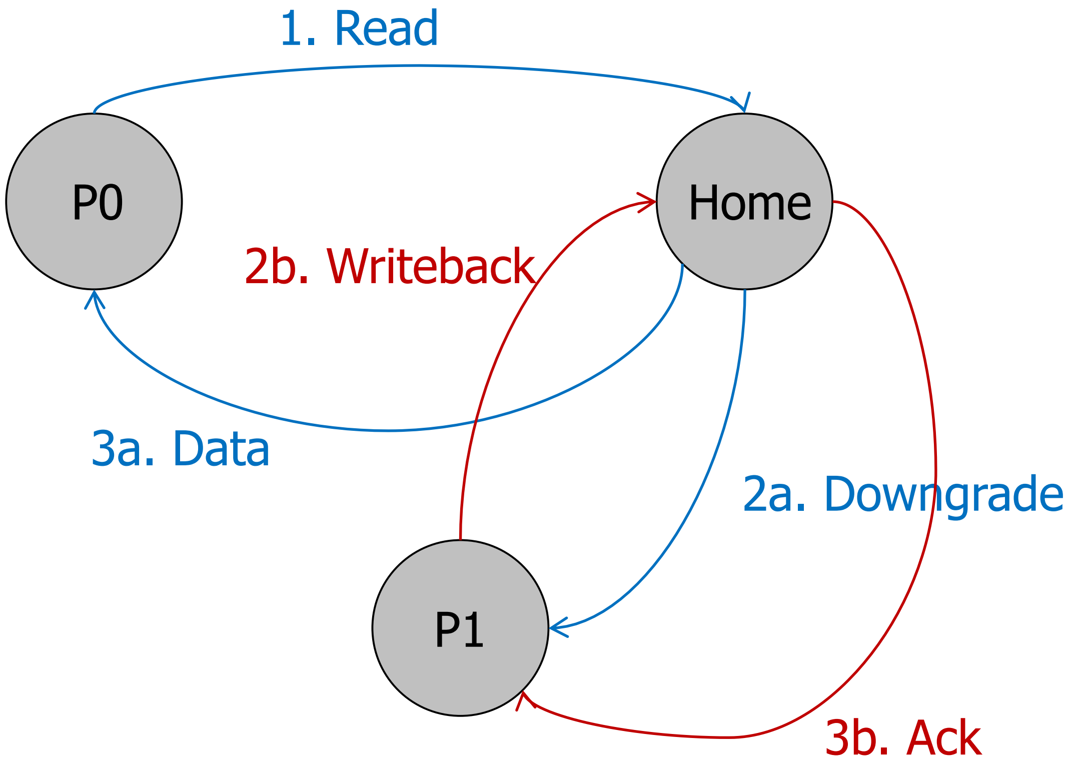
Culler/Singh Fig. 8.16

MESI Directory Transaction: Read

P0 acquires an address for reading:

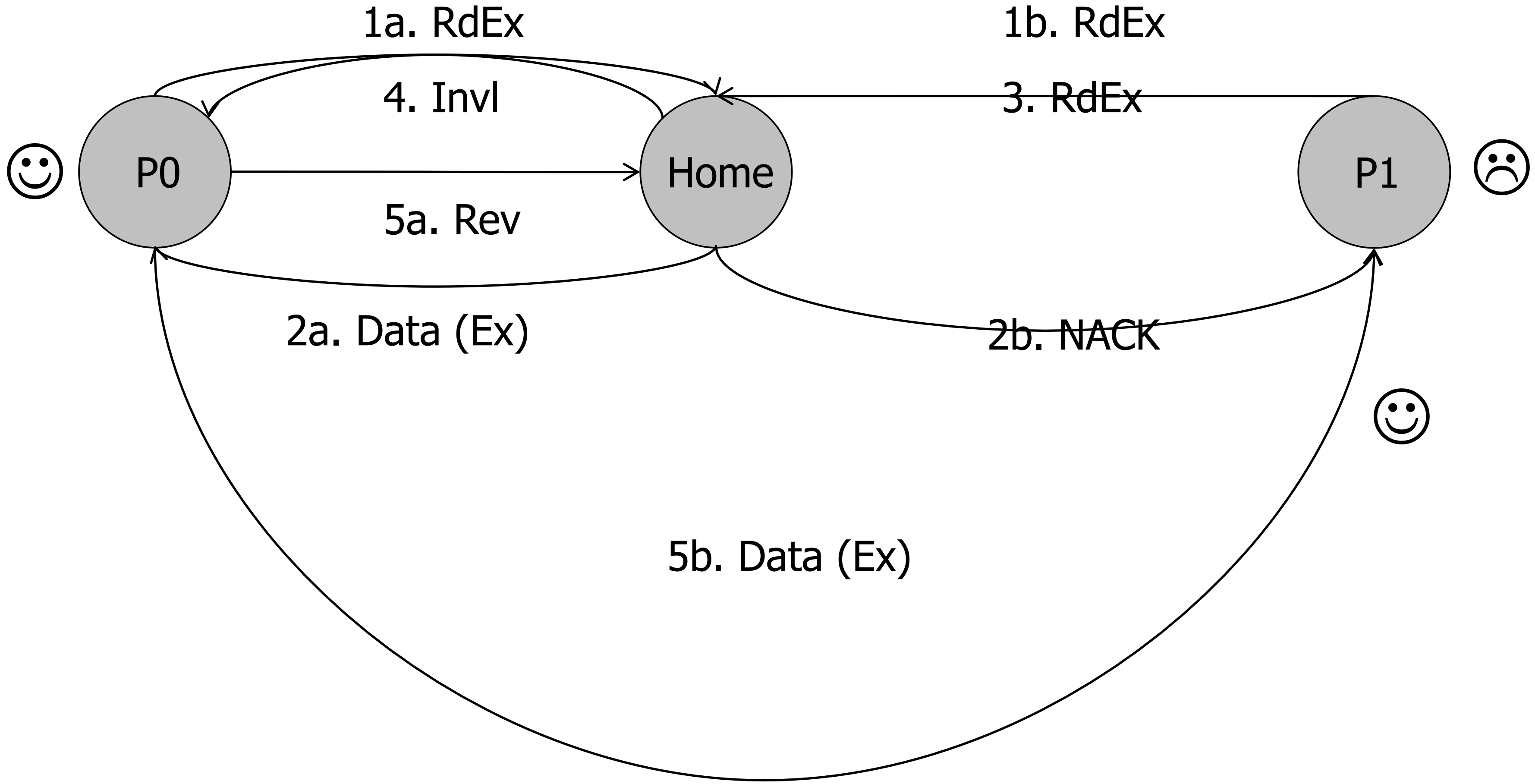
Other sharers in E
➔ Grant S and downgrade

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
  - Original requestor retries
- OR, queuing requests and granting in sequence
- (Or some combination thereof)

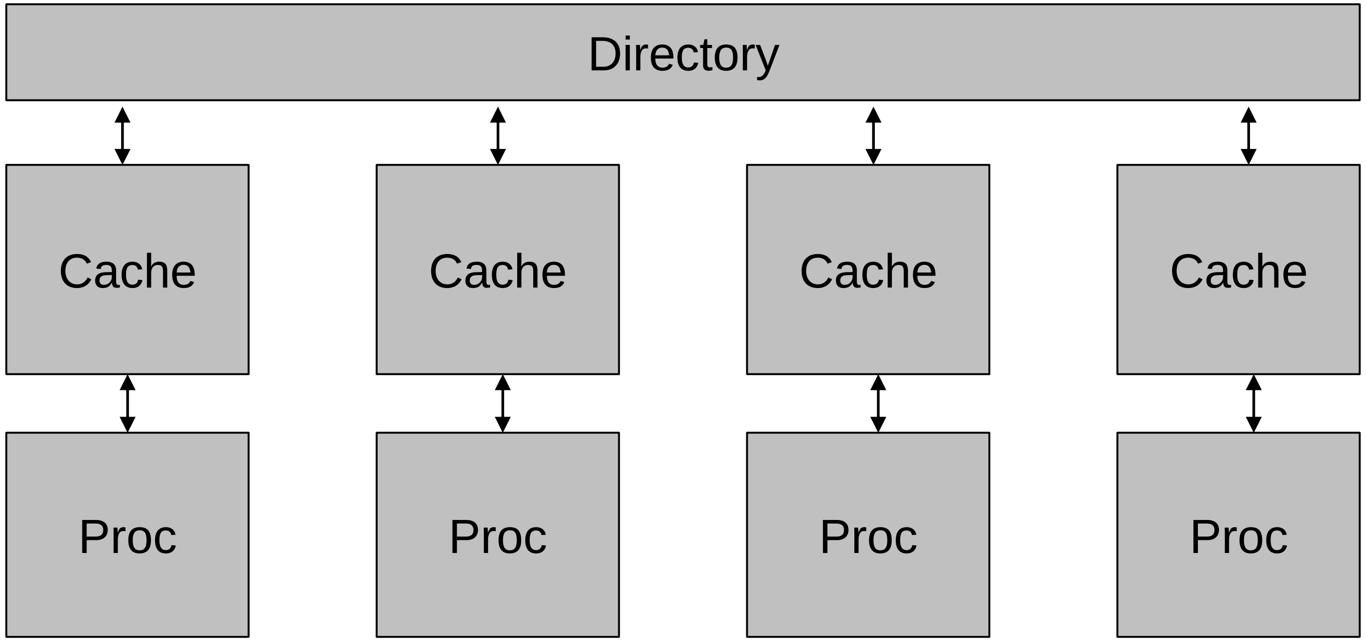
Fairness

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

Shared vs private caches

Private caches – logical view

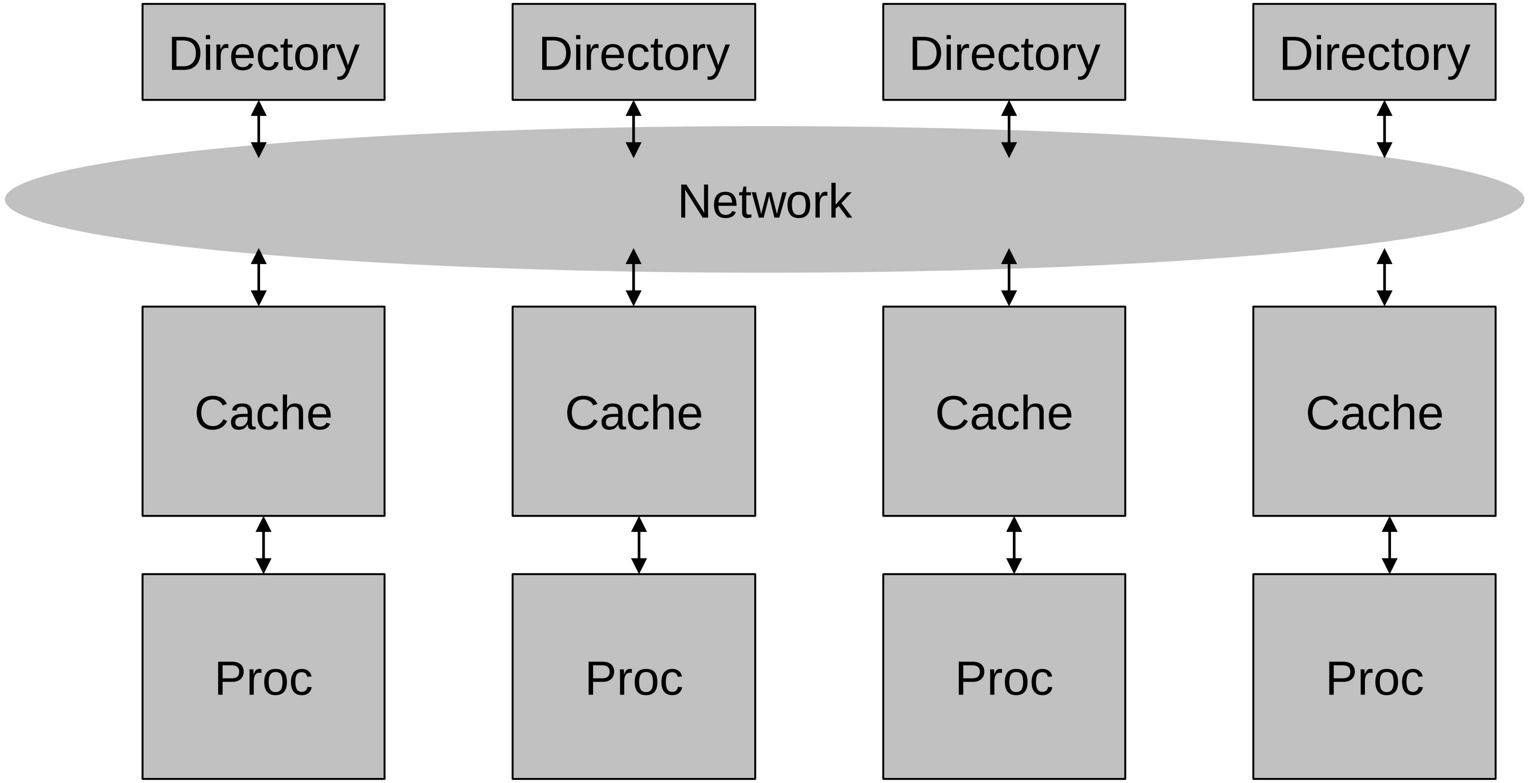
Private caches keep data local near processor



Private caches – implementation

To increase bandwidth, directory is **banked**

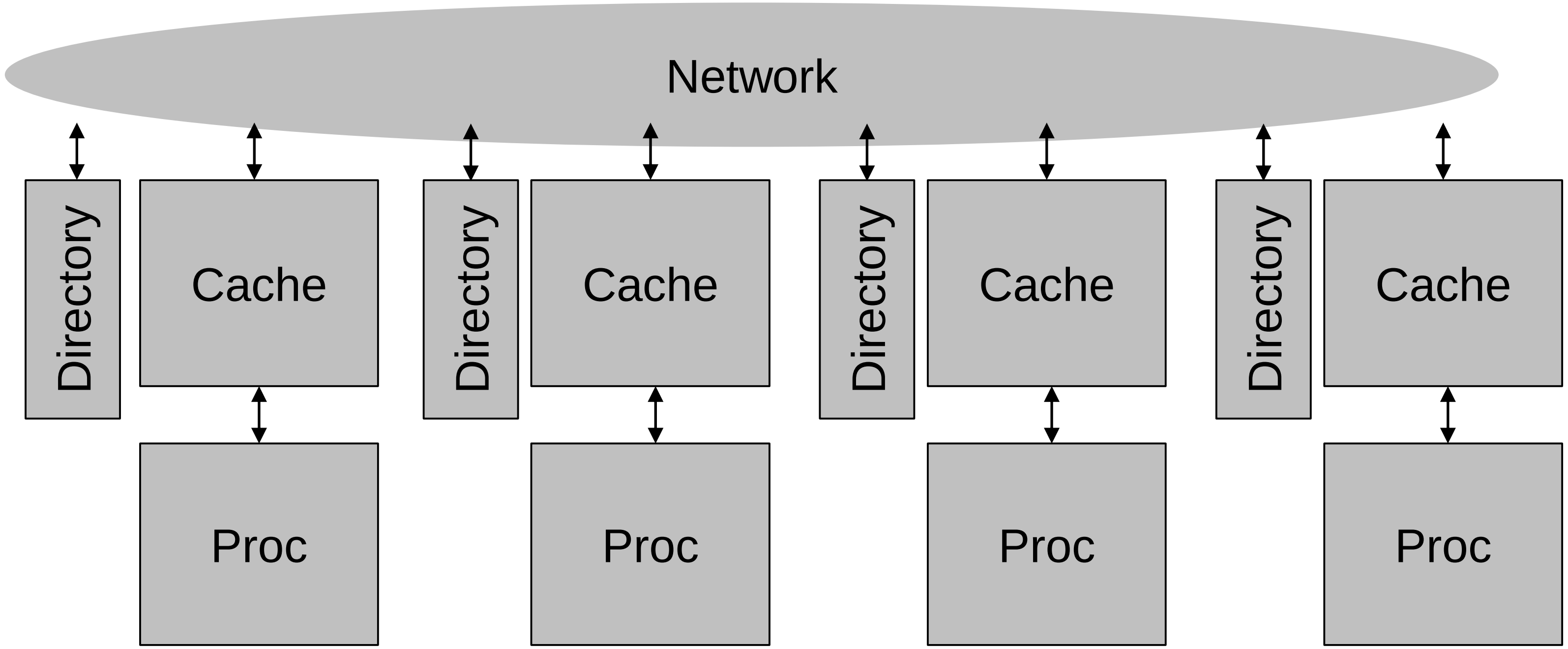
- Each bank responsible for static region of address space



Private caches – implementation

To increase bandwidth, directory is **banked**

And we can put the directory on each node

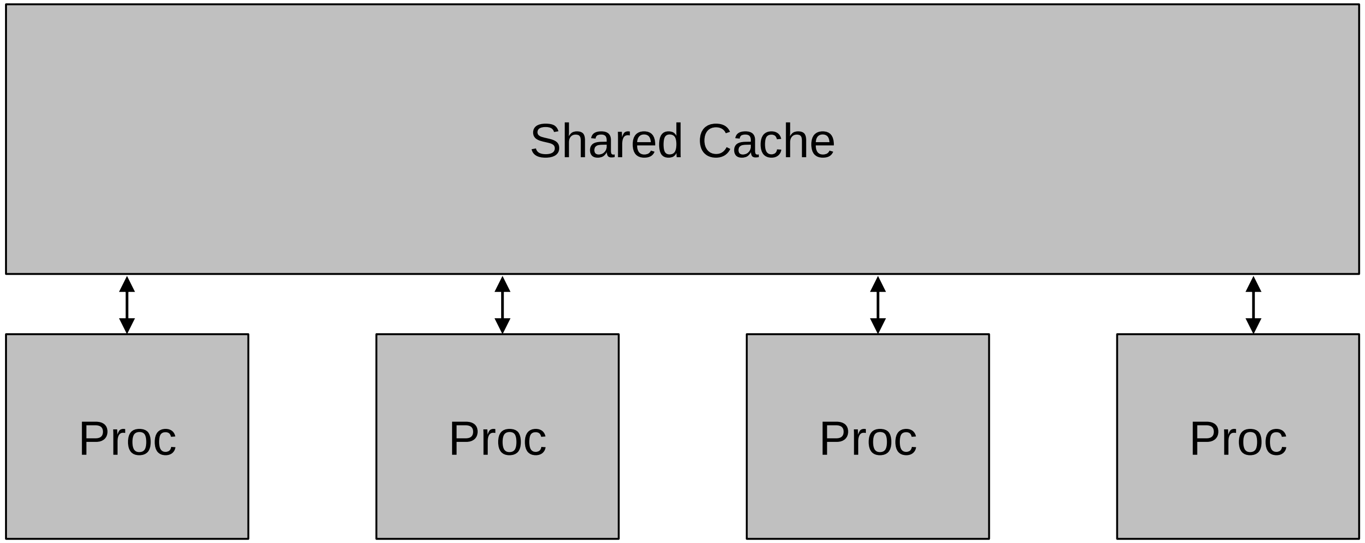


Shared caches – logical view

Shared caches use full cache capacity

Do we still need directory? coherence?

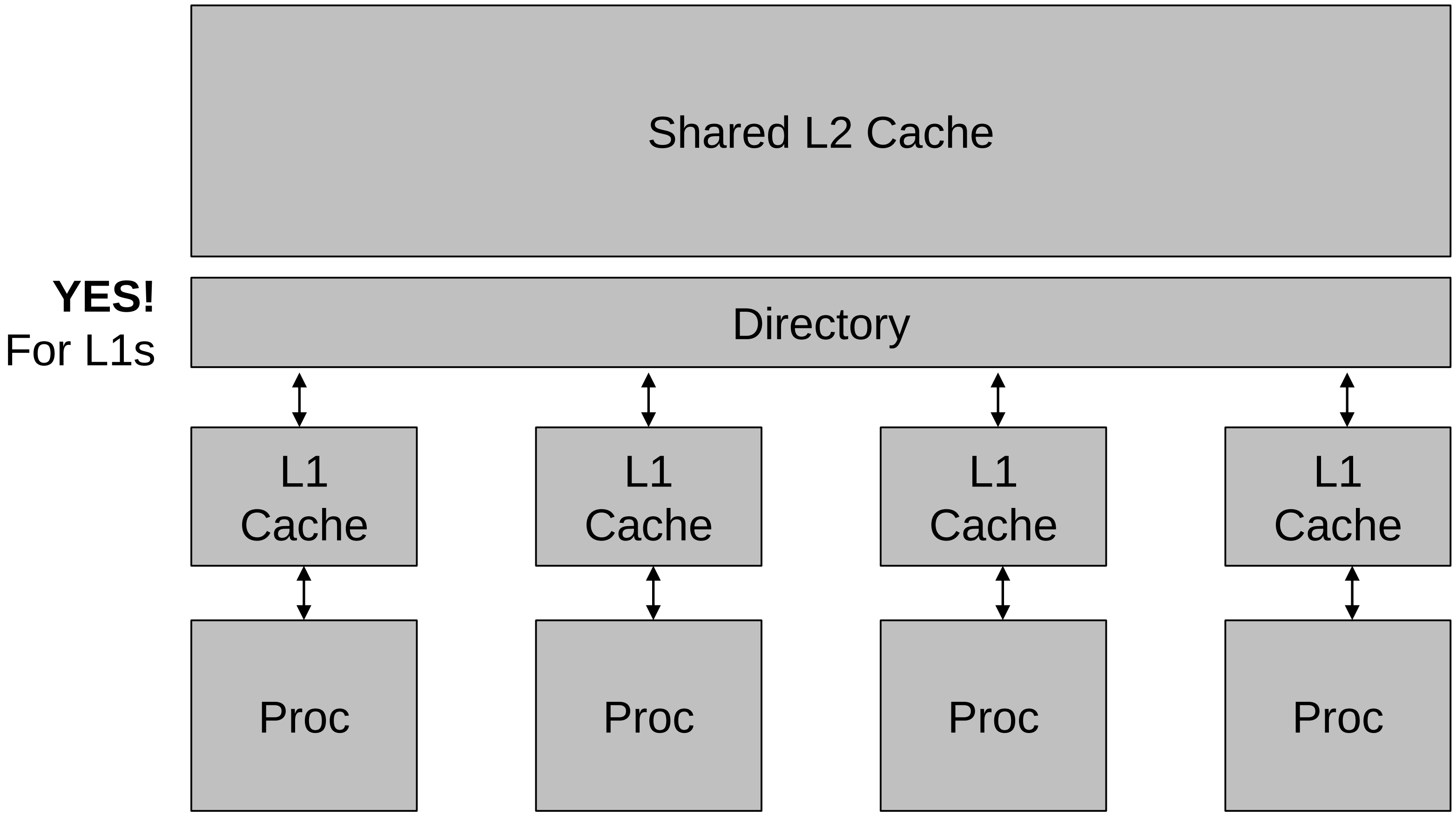
NO! Only one copy of data



Shared caches – logical view

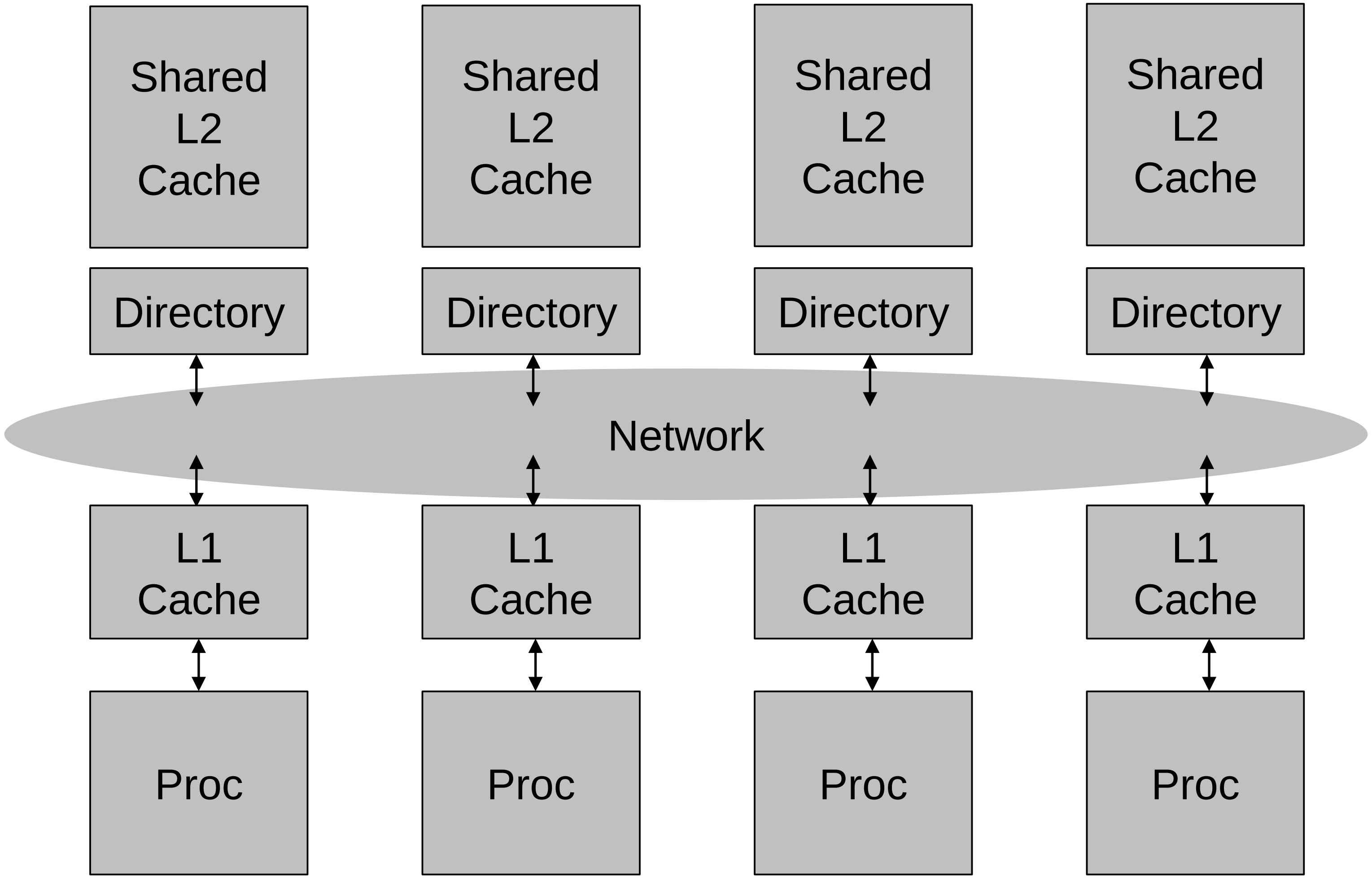
Shared caches use full cache capacity

Do we still need directory? coherence?



Shared caches – implementation

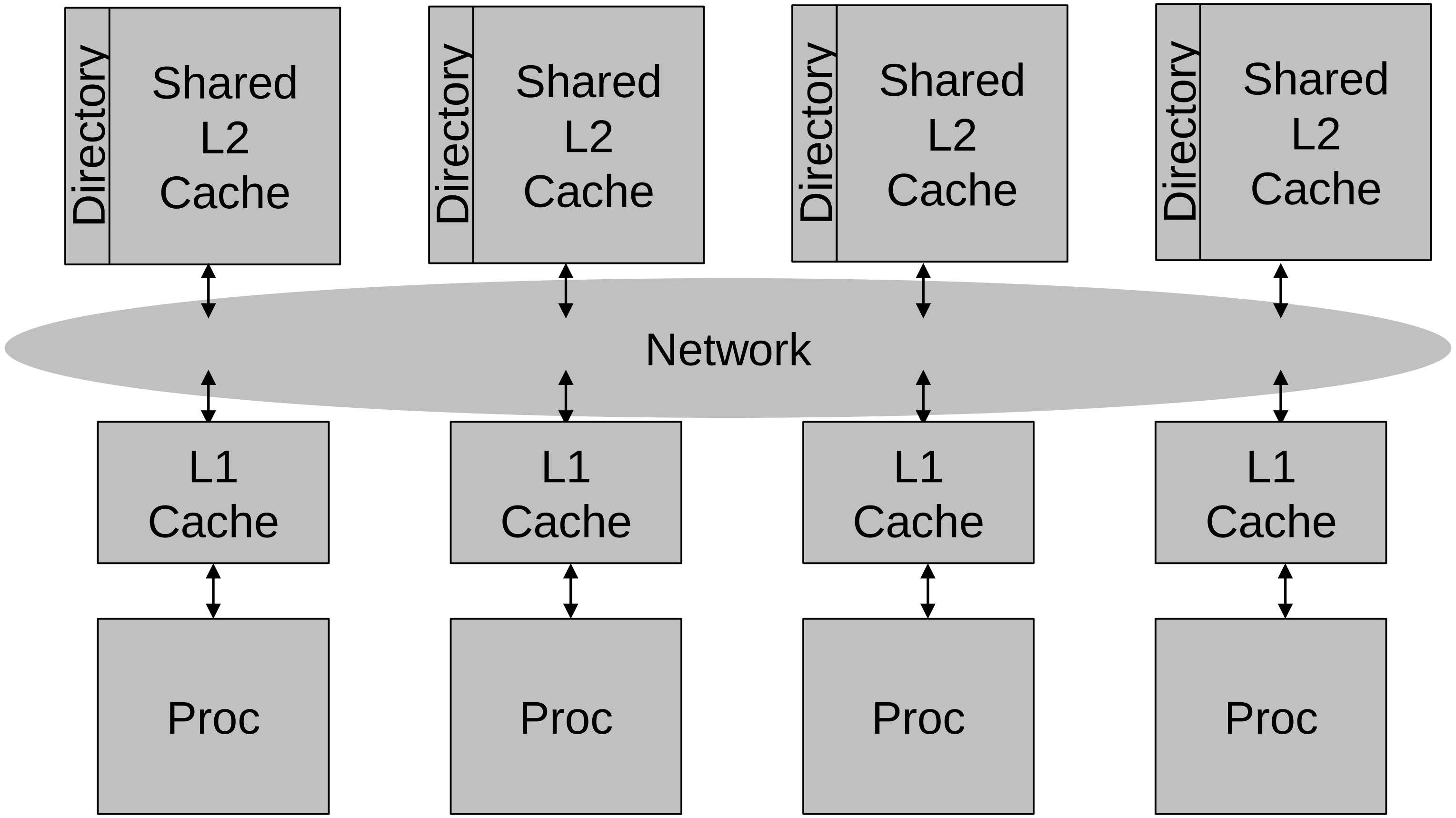
To increase bandwidth, shared cache is **banked**



Shared caches – implementation

L2 + directory banks track **same addresses**

➔ Directory can be implemented in L2 tags

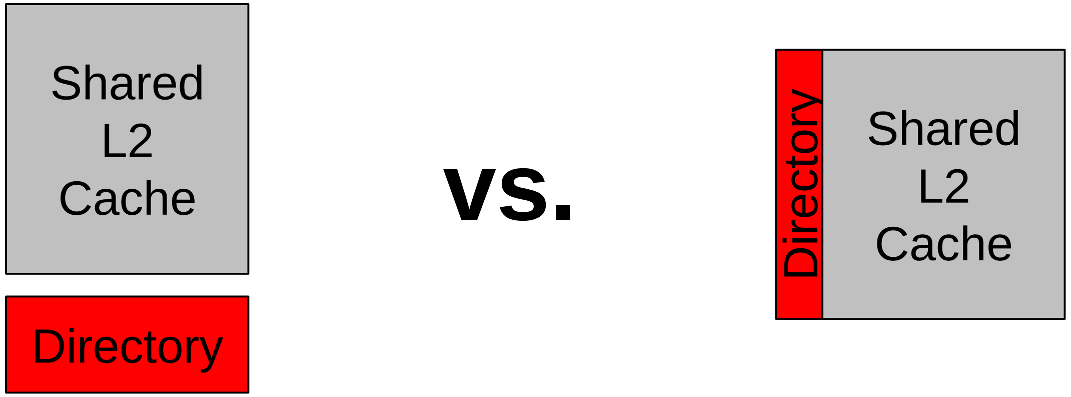


Directory implementation tradeoffs

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

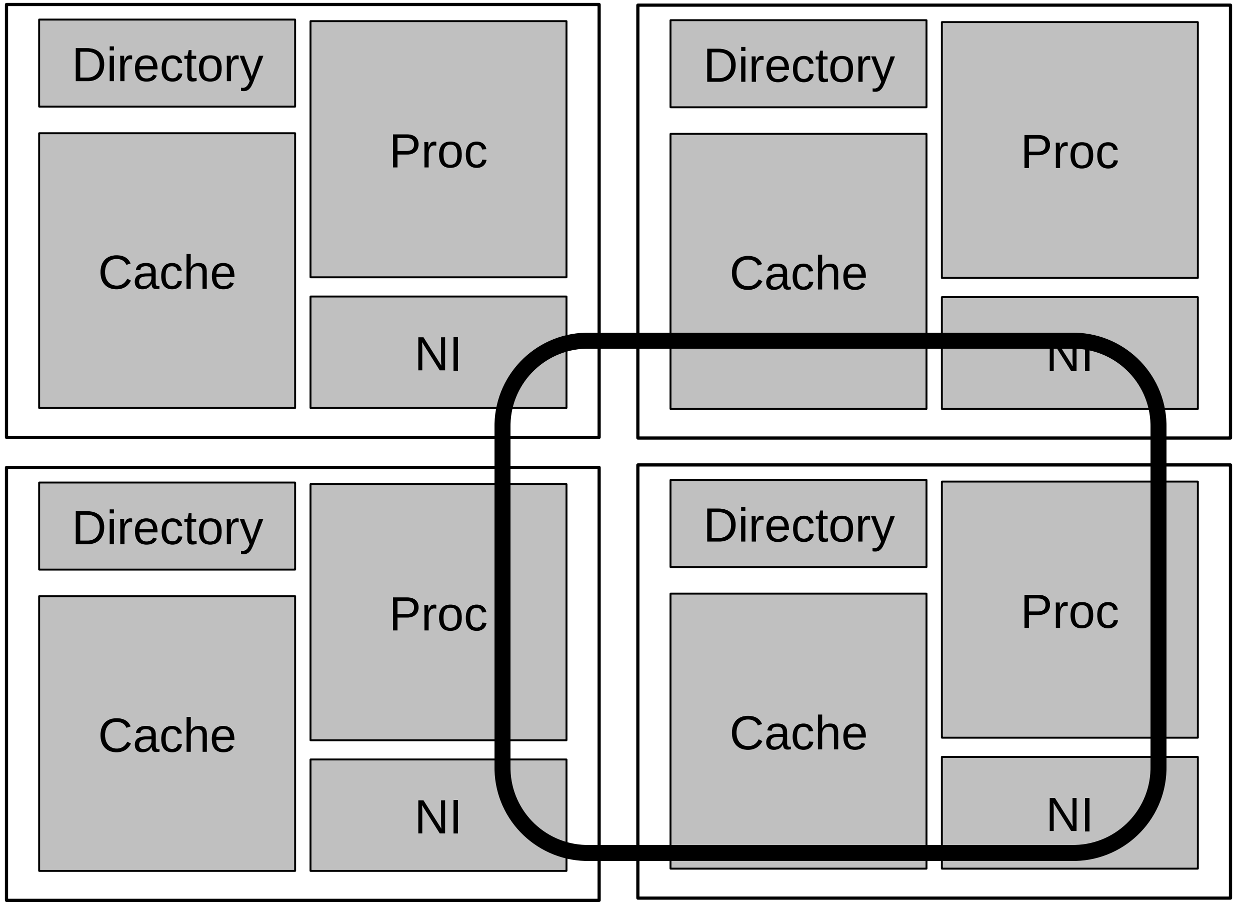
➔ Many fewer lines than L2 to track

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



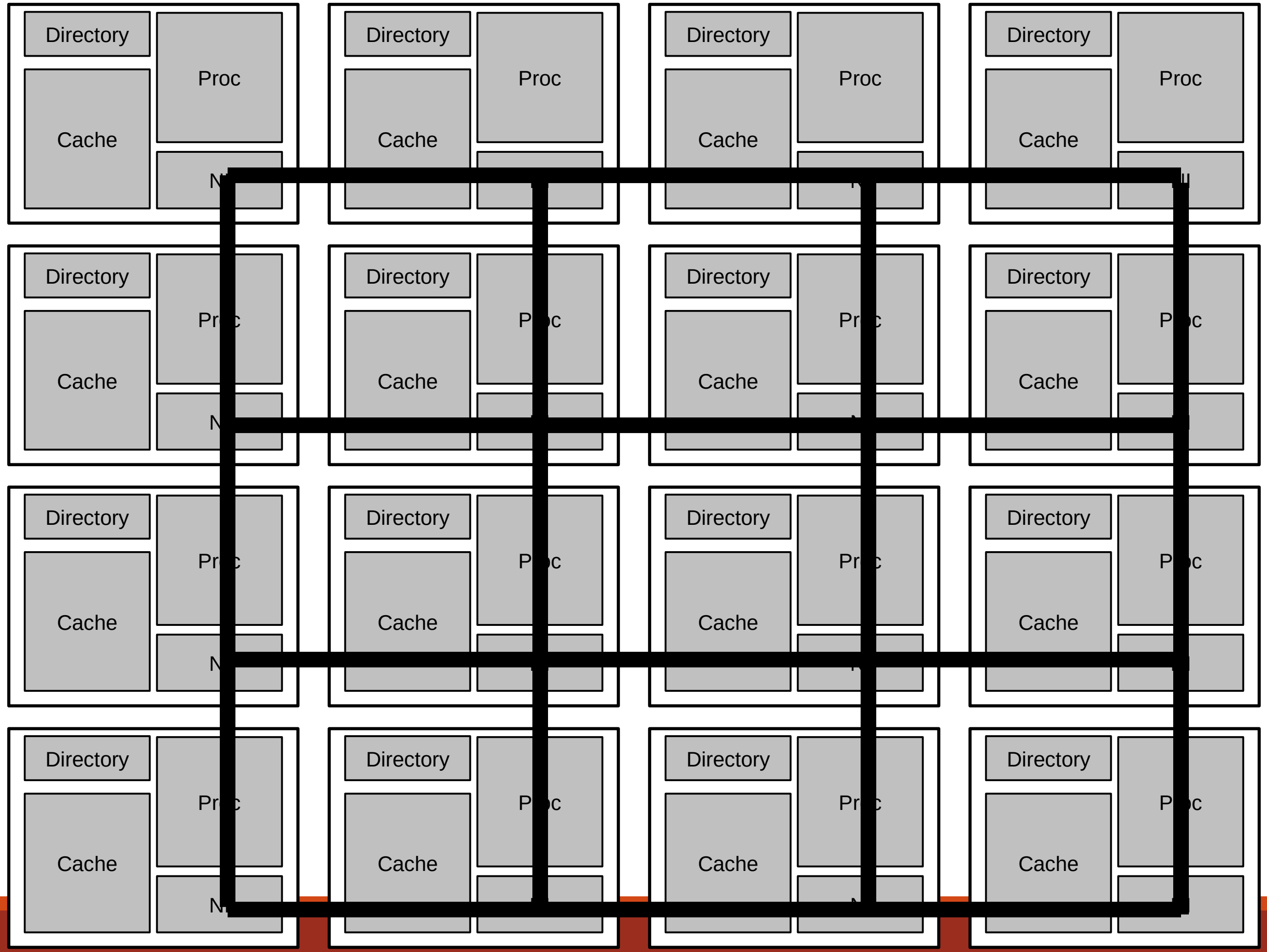
Distributed caches today

4-core system



Distributed caches today

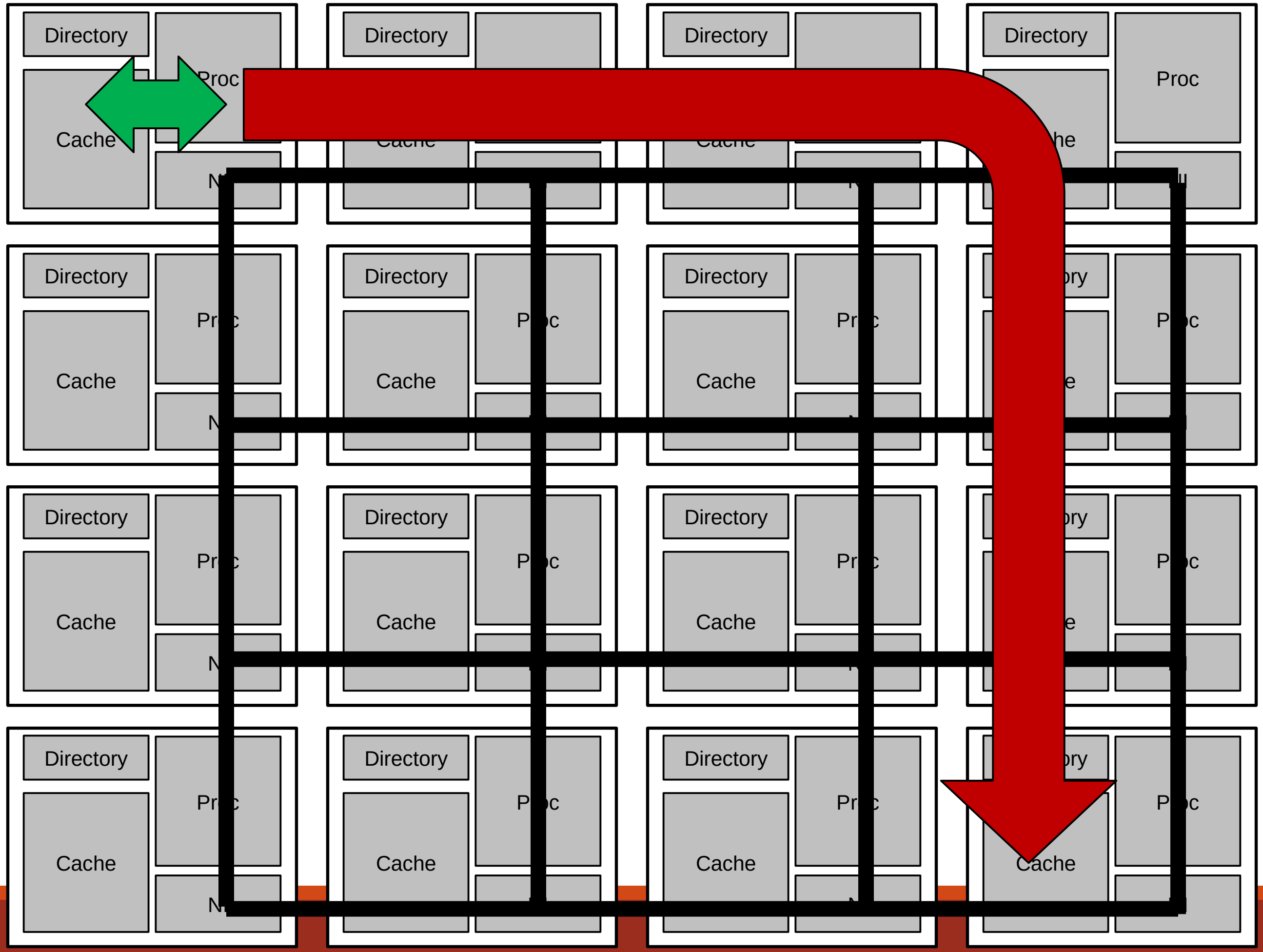
16-core system w/ mesh interconnect



Distributed caches today

Non-uniform cache access (NUCA): 7 – 40 cycles for LLC

Problem worsening with larger systems



Data placement in LLC is critical

Hybrid private/shared organization

Check local ➔ then global home node

What are the tradeoffs here?

- Hit time
- Miss ratio
- Miss penalty

Capacity contention between private/shared data

Multiple lookups to find data

Can we do better?

Shared vs private cache summary

Tension between hit latency and miss ratio

- Private caches keep data local
- Shared caches maximize on capacity

Banking is a necessary optimization for bandwidth

- Introduces NUCA
- Data placement becomes a problem

Simple hardware mechanisms can let software place data

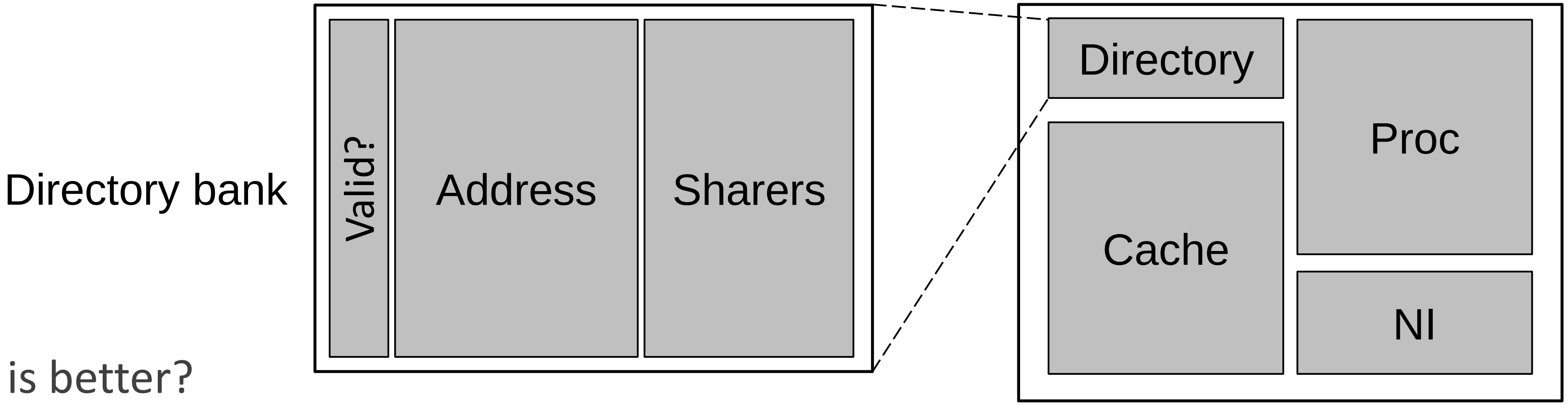
- Hard optimization problem – different programs access memory very differently
- “Jigsaw: Scalable, software-defined caches”, Beckmann and Sanchez, PACT’13

Implementing directories

Implementing directories

Directories are either

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



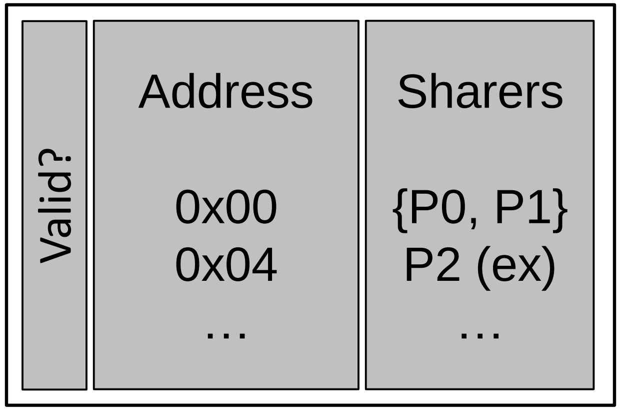
Which is better?

- Depends on amount of data being tracked for coherence

What happens on an eviction from directory?

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

Scaling problems with directories



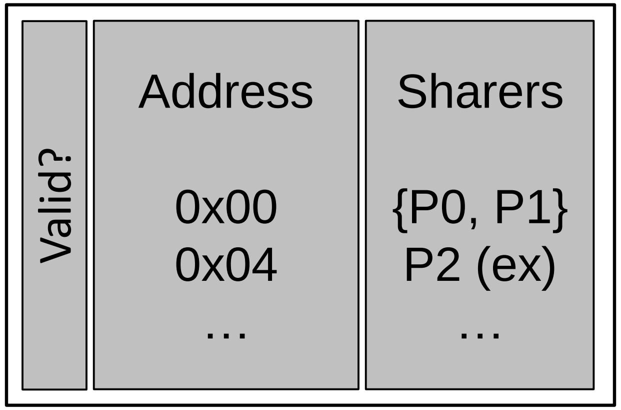
Valid?	Address	Sharers
	0x00	{P0, P1}
	0x04	P2 (ex)
	…	…

“Full-mapped directory” (Idealized model): track all possible sharers for each line

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

- Each processor has N lines ➔ $\propto N \times P$ bytes on-chip
- Each directory entry needs P bits
  ➔ each directory bank is $\propto N \times P$ bits
- With P directory banks, need bits $\propto N \times P^2$
  ➔ Oops! Directories take an increasing $O(P)$ fraction of on-chip capacity

Scalable directory implementions



Valid?	Address	Sharers
	0x00 0x04 …	{P0, P1} P2 (ex) …

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

Tolerating false positives

- Limited directories
  - Observation: Most lines shared by few sharers
  - Idea: Support up to ~4 sharers, then broadcast
- Bloom filters
  - Space-efficient approximate tracking of sharers (no false negatives, some false positives)

Observation: There can be **at most** $N \times P$ sharers

- This should scale! Perfect tracking ➔ $O(1)$ overhead
- Full-mapped directories are way over-provisioned

Tolerating complexity

- Distributed doubly linked-list across sharers (pros? cons?)
- More efficient sharer encodings
  - "SCD: A scalable coherence directory with flexible sharer set encoding", Sanchez and Kozyrakis, HPCA'12

Scalable directory implementations

Hierarchical directories

+ Less state at each directory: e.g., 2 levels ➔ $\sqrt{P}$ bits per entry ➔ $O(\sqrt{P})$ overhead

− Increased latency: e.g., $2 \times$ for 2 levels

− **Design & verification complexity**

Distributed doubly-linked list

+ Minimal state at each directory: $O(1)$ overhead

− **Increased latency:** $O(P)$ hops to invalidate

More efficient sharer encodings

- "SCD: A scalable coherence directory with flexible sharer set encoding", Sanchez and Kozyrakis, HPCA'12
- $O(1)$ overhead, no increased latency, but more complex

Cache Coherence Summary

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 better than bus)