Memory and Cache Coherence

Shared Memory Multiprocessors

Symmetric Multiprocessors (SMPs)

- Symmetric access to all of main memory from any processor

Dominate the server market

- Building blocks for larger systems; arriving to desktop

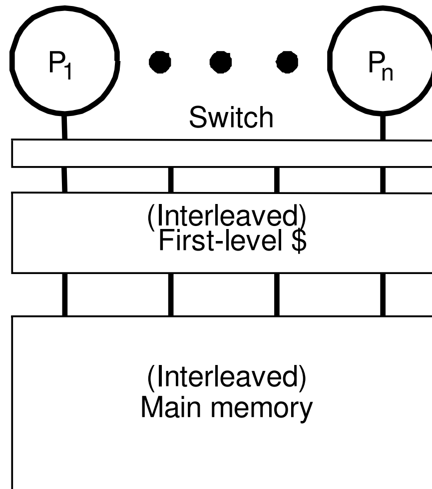
Attractive as throughput servers and for parallel programs

- Fine-grain resource sharing
- Uniform access via loads/stores
- Automatic data movement and coherent replication in caches
- Useful for operating system too

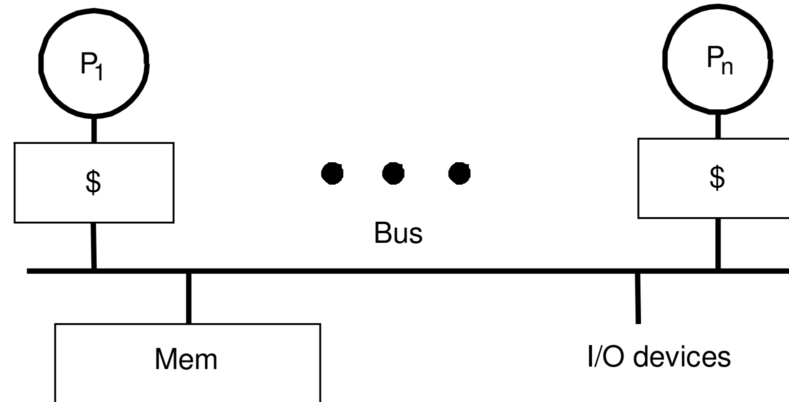
Normal uniprocessor mechanisms to access data (reads and writes)

- Key is extension of memory hierarchy to support multiple processors

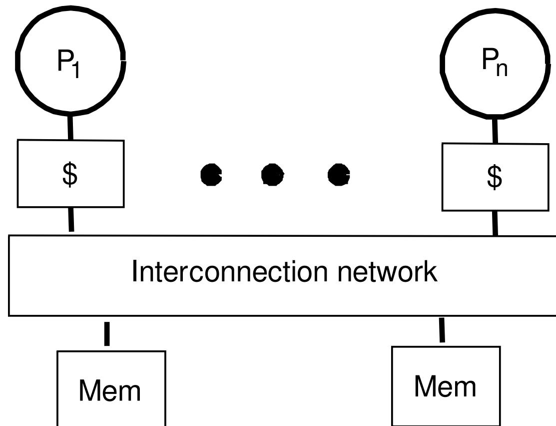
Natural Extensions of Memory System



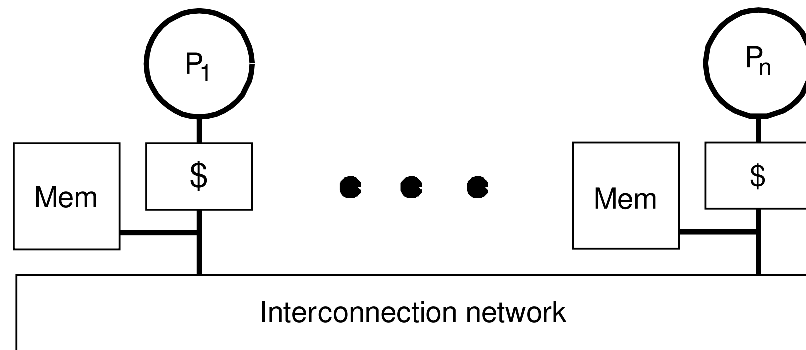
(a) Shared cache



(b) Bus-based shared memory



(c) Dancehall



(d) Distributed-memory

Caches and Cache Coherence

Caches play key role in all cases

- Reduce average data access time
- Reduce bandwidth demands placed on shared interconnect

But private processor caches create a problem

- Copies of a variable can be present in multiple caches
- A write by one processor may not become visible to others
  - They'll keep accessing stale value in their caches
- *Cache coherence* problem
- Need to take actions to ensure visibility

Focus: Bus-based, Centralized Memory

Shared cache

- Low-latency sharing and prefetching across processors
- Sharing of working sets
- No coherence problem (and hence no false sharing either)
- But high bandwidth needs and negative interference (e.g. conflicts)
- Hit and miss latency increased due to intervening switch and cache size
- Mid 80s: to connect couple of processors on a board (Encore, Sequent)
- Today: for multiprocessor on a chip (for small-scale systems or nodes)

Dancehall

- No longer popular: everything is uniformly *far* away

Distributed memory

- Most popular way to build scalable systems, discussed later

A Coherent Memory System: Intuition

Reading a location should return latest value written (by any process)

Easy in uniprocessors

- Except for I/O: coherence between I/O devices and processors
- But infrequent so software solutions work
  - uncacheable memory, uncacheable operations, flush pages, pass I/O data through caches

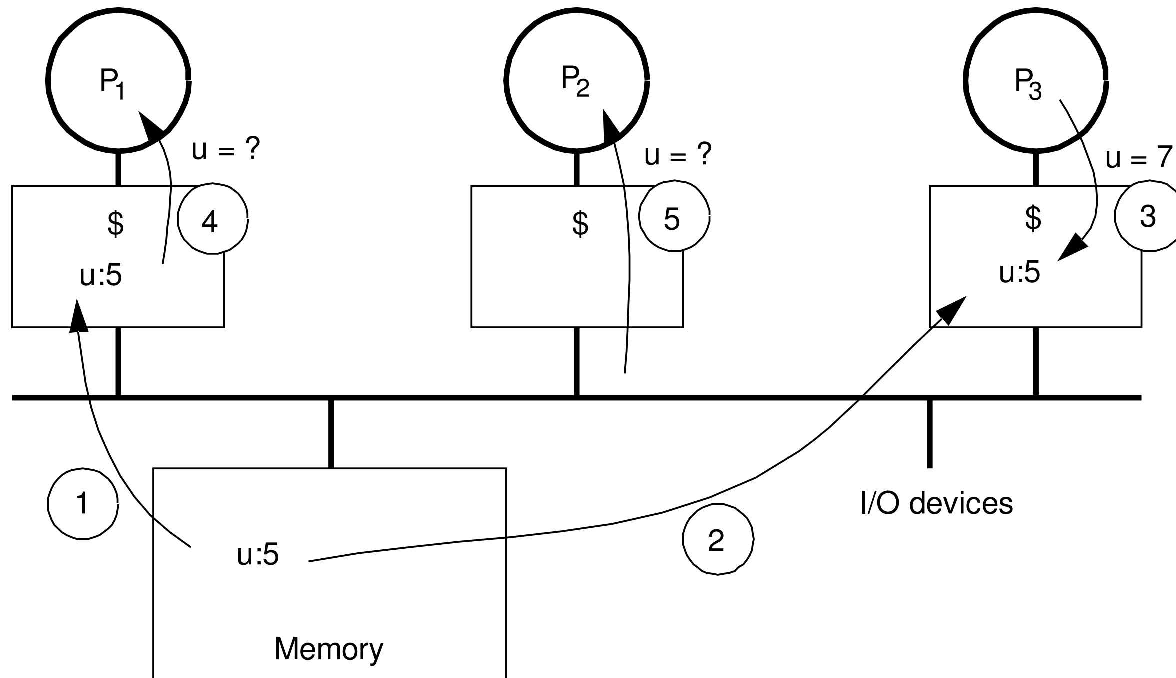
Would like same to hold when processes run on different processors

- E.g. as if the processes were interleaved on a uniprocessor

But coherence problem much more critical in multiprocessors

- Pervasive
- Performance-critical
- Must be treated as a basic hardware design issue

Example Cache Coherence Problem



- Processors see different values for u after event 3
- With write back caches, value written back to memory depends on happenstance of which cache flushes or writes back value when
  - Processes accessing main memory may see very stale value
- Unacceptable to programs, and frequent!

Problems with the Intuition

Recall: Value returned by read should be last value written

But "last" is not well-defined

Even in seq. case, last defined in terms of program order, not time

- Order of operations in the machine language presented to processor
- "Subsequent" defined in analogous way, and well defined

In parallel case, program order defined within a process, but need to make sense of orders across processes

Must define a meaningful semantics

Some Basic Definitions

Extend from definitions in uniprocessors to those in multiprocessors

Memory operation: a single read (load), write (store) or read-modify-write access to a memory location

- Assumed to execute atomically

Issue: a memory operation issues when it leaves processor's internal environment and is presented to memory system (cache, buffer …)

Perform: operation appears to have taken place, as far as processor can tell from other memory operations it issues

- A write performs w.r.t. the processor when a subsequent read by the processor returns the value of that write or a later write
- A read perform w.r.t the processor when subsequent writes issued by the processor cannot affect the value returned by the read

In multiprocessors, stay same but replace "the" by "a" processor

- Also, *complete*: perform with respect to all processors
- Still need to make sense of orderin operations from different processes

Sharpening the Intuition

Imagine a single shared memory and no caches

- Every read and write to a location accesses the same physical location
- Operation completes when it does so

Memory imposes a *serial* or *total order* on operations to the location

- Operations to the location from a given processor are in program order
- The order of operations to the location from different processors is some interleaving that preserves the individual program orders

"Last" now means most recent in a hypothetical serial order that maintains these properties

For the serial order to be consistent, all processors must see writes to the location in the same order (if they bother to look, i.e. to read)

Note that the total order is never really constructed in real systems

- Don't even want memory, or any hardware, to see all operations

But program should behave as if some serial order is enforced

- Order in which things appear to happen, not actually happen

Formal Definition of Coherence

Results of a program: values returned by its read operations

A memory system is *coherent* if the results of any execution of a program are such that each location, it is possible to construct a hypothetical serial order of all operations to the location that is consistent with the results of the execution and in which:

1. operations issued by any particular process occur in the order issued by that process, and

2. the value returned by a read is the value written by the last write to that location in the serial order

Two necessary features:

- *Write propagation*: value written must become visible to others
- *Write serialization*: writes to location seen in same order by all
  - if I see w1 after w2, you should not see w2 before w1
  - no need for analogous read serialization since reads not visible to others

Cache Coherence Using a Bus

Built on top of two fundamentals of uniprocessor systems

- Bus transactions
- State transition diagram in cache

Uniprocessor bus transaction:

- Three phases: arbitration, command/address, data transfer
- All devices observe addresses, one is responsible

Uniprocessor cache states:

- Effectively, every block is a finite state machine
- Write-through, write no-allocate has two states: valid, invalid
- Writeback caches have one more state: modified ("dirty")

Multiprocessors extend both these somewhat to implement coherence

Snooping-based Coherence

Basic Idea

Transactions on bus are visible to all processors

Processors or their representatives can snoop (monitor) bus and take action on relevant events (e.g. change state)

Implementing a Protocol

Cache controller now receives inputs from both sides:

- Requests from processor, bus requests/responses from snooper

In either case, takes zero or more actions

- Updates state, responds with data, generates new bus transactions

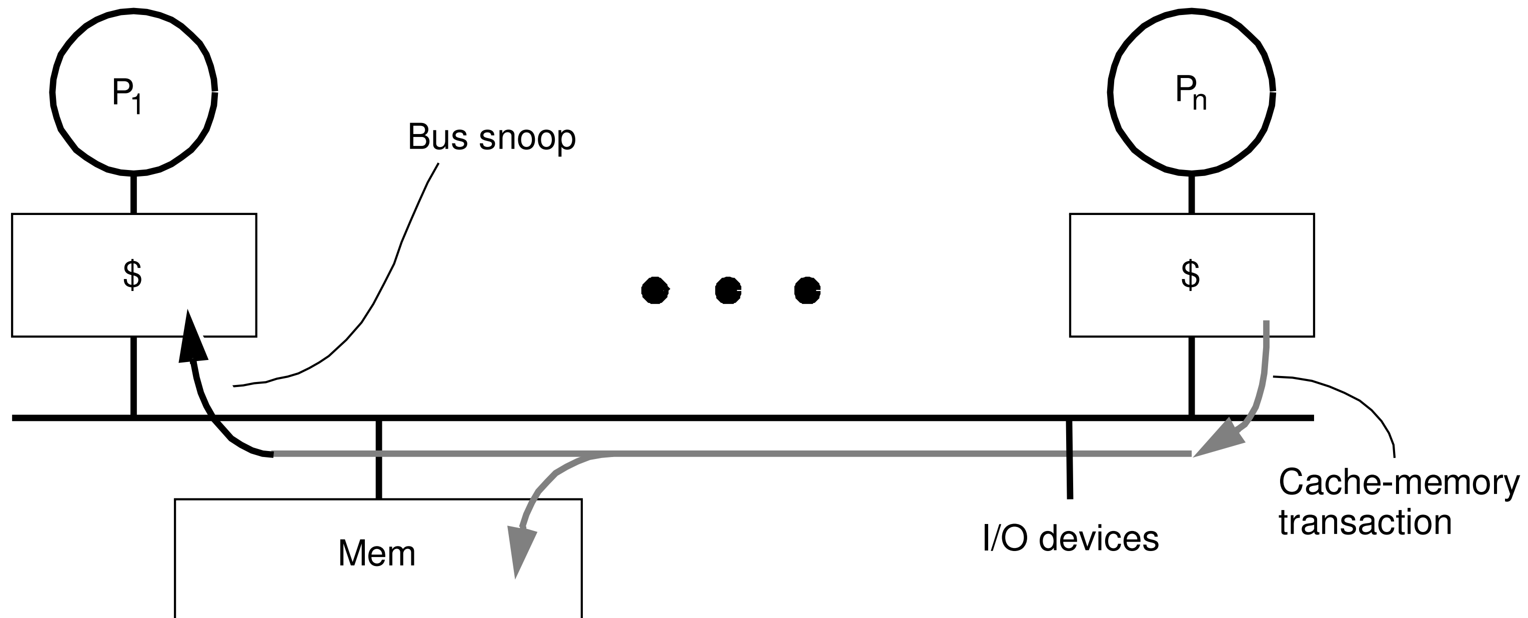
Protocol is distributed algorithm: cooperating state machines

- Set of states, state transition diagram, actions

Granularity of coherence is typically cache block

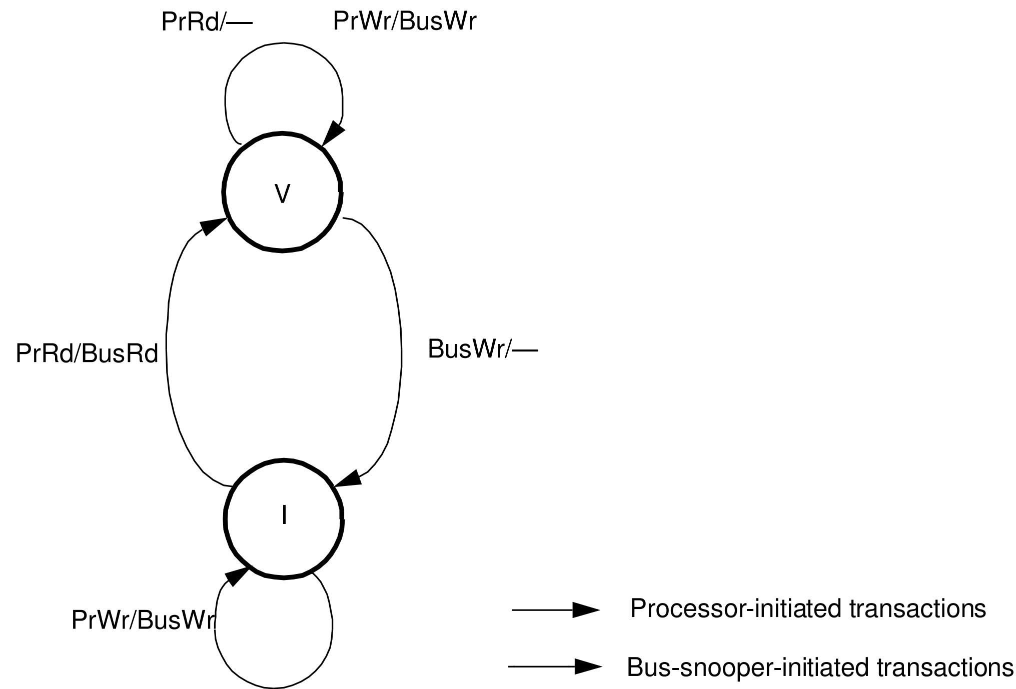
- Like that of allocation in cache and transfer to/from cache

Coherence with Write-through Caches



- Key extensions to uniprocessor:snooping, invalidating/updating caches
  - no new states or bus transactions in this case
  - invalidation- versus update-based protocols
- Write propagation: even in invalidate case, laterreads will see new value
  - invalidate causes miss on later access, and memory up-to-date via write-through

Write-through State Transition Diagram



- Two states per block in each cache, as in uniprocessor
  - state of a block can be seen as *p*-vector
- Hardware state bits associated with only blocks that are in the cache
  - other blocks can be seen as being in invalid (not-present) state in that cache
- Write will invalidate all other caches (no local change of state)
  - can have multiple simultaneous readers of block,but write invalidates them

Is it Coherent?

Construct total order that satisfies program order, write serialization?

Assume atomic bus transactions and memory operations for now

- all phases of one bus transaction complete before next one starts
- processor waits for memory operation to complete before issuing next

All writes go to bus + atomicity

- Writes serialized by order in which they appear on bus (*bus order*)
- Per above assumptions, invalidations applied to caches in bus order

How to insert reads in this order?

- Important since processors see writes through reads, so determines whether write serialization is satisfied
- But read hits may happen independently and do not appear on bus or enter directly in bus order

Problem with Write-Through

High bandwidth requirements

- Every write from every processor goes to shared bus and memory
- Consider 200MHz, 1CPI processor, and 15% instrs. are 8-byte stores
- Each processor generates 30M stores or 240MB data per second
- 1GB/s bus can support only about 4 processors without saturating
- Write-through especially unpopular for SMPs

Write-back caches absorb most writes as cache hits

- Write hits don't go on bus
- But now how do we ensure write propagation and serialization?
- Need more sophisticated protocols: large design space

Design Space for Snooping Protocols

No need to change processor, main memory, cache …

- Extend cache controller andexploit bus (provides serialization)

Focus on protocols for write-back caches

Dirty state now also indicates exclusive ownership

- Exclusive: only cache with a valid copy (main memory may be too)
- Owner: responsible for supplying bbck upon a request for it

Design space

- Invalidation versus Update-based protocols
- Set of states

Invalidation-based Protocols

Exclusive means can modify without notifying anyone else

- i.e. without bus transaction
- Must first get block in exclusive state before writing into it
- Even if already in valid state, need transaction, so called a write miss

Store to non-dirty data generates a *read-exclusive* bus transaction

- Tells others about impending write, obtains exclusive ownership
  - makes the write visible, i.e. write is performed
  - may be actually observed (by a read miss) only later
  - write hit made visible (performed) when block updated in writer's cache
- Only one RdX can succeed at a time for a block: serialized by bus

Read and Read-exclusive bus transactions drive coherence actions

- Writeback transactions also, but not caused by memory operation and quite incidental to coherence protocol
  - note: replaced block that is not in modified state can be dropped

Update-based Protocols

A write operation updates values in other caches

- New, update bus transaction

Advantages

- Other processors don't miss on next access: reduced latency
  - In invalidation protocols, they would miss and cause more transactions
- Single bus transaction to update severalcaches can save bandwidth
  - Also, only the word written is transferred, not whole block

Disadvantages

- Multiple writes by same processor cause multiple update transactions
  - In invalidation, first write gets exclusive ownership, others local

Detailed tradeoffs more complex

Invalidate versus Update

Basic question of program behavior

- Is a block written by one processor read by others before it is rewritten?

Invalidation:

- Yes => readers will take a miss
- No => multiple writes without additional traffic
  - and clears out copies that won't be used again

Update:

- Yes => readers will not miss if they had a copy previously
  - single bus transaction to update all copies
- No => multiple useless updates, even to dead copies

Need to look at program behavior and hardware complexity

Invalidation protocols much more popular (more later)

- Some systems provide both, oreven hybrid

Basic MSI Writeback Inval Protocol

States

- Invalid (I)
- Shared (S): one or more
- Dirty or Modified (M): one only

Processor Events:

- PrRd (read)
- PrWr (write)

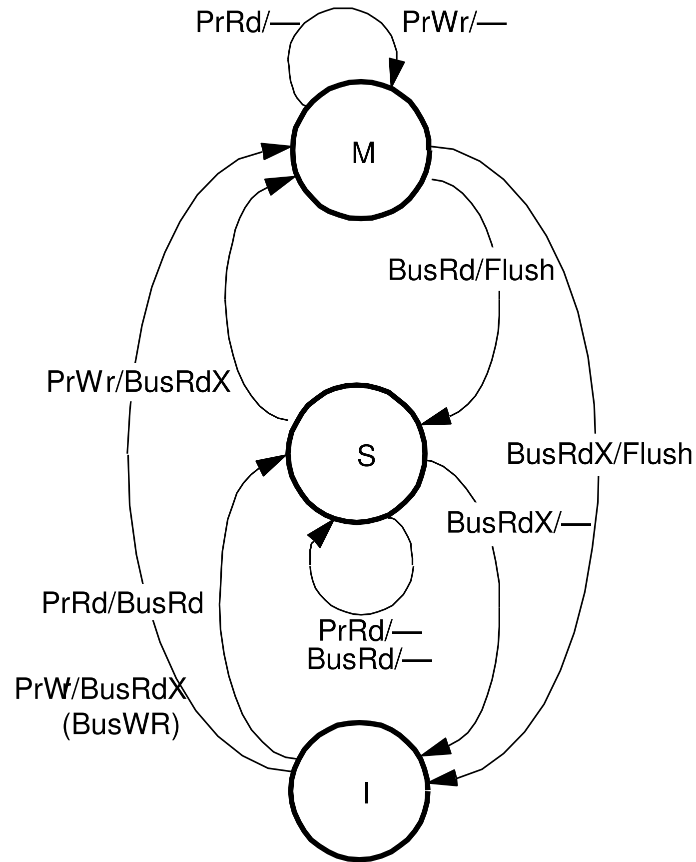
Bus Transactions

- BusRd: asks for copy with no intent to modify
- BusRdX: asks for copy with intent to modify
- BusWR: updates memory

Actions

- Update state, perform bus transaction, flush value onto bus

State Transition Diagram



MESI (4-state) Invalidation Protocol

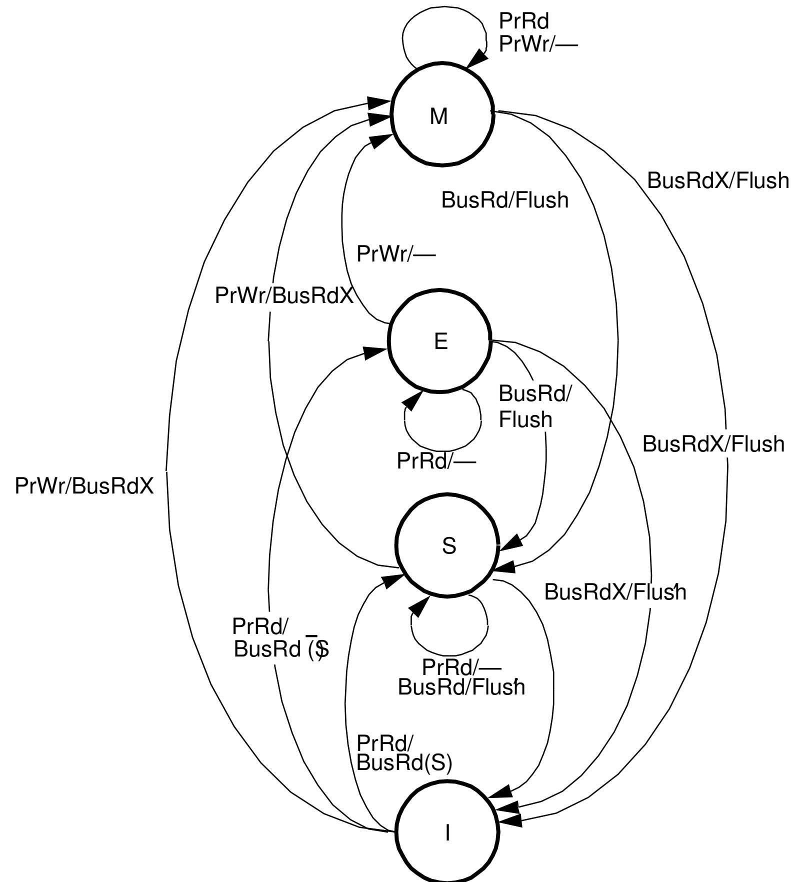
Problem with MSI protocol

- Reading and modifying data is 2 bus Xactions, even if no one sharing
  - e.g. even in sequential program
  - BusRd (I->S) followed by BusRdX or BusUpgr (S->M)

Add *exclusive* state: write locally without Xaction, but not modified

- Main memory is up to date, so cache not necessarily owner
- States
  - invalid
  - exclusive or *exclusive-clean* (only this cache has copy, but not modified)
  - shared (two or more caches may have copies)
  - modified (dirty)
- I -> E on PrRd if no one else has copy
  - needs "shared" signal on bus: wired-or line asserted in response to BusRd

MESI State Transition Diagram



- BusRd(S) means shared line asserted on BusRd transaction
- Flush': if cache-to-cache sharing, only one cache flushes data
- MOESI protocol: Owned state: exclusive but memory not valid

Lower-level Protocol Choices

Who supplies data on miss when not in M state: memory or cache

Original, *Illinois* MESI: cache, since assumed faster than memory

- *Cache-to-cache sharing*

Not true in modern systems

- Intervening in another cache more expensive than getting from memory

Cache-to-cache sharing also adds complexity

- How does memory know it should supply data (must wait for caches)
- Selection algorithm if multiple caches have valid data

But valuable for cache-coherent machines with distributed memory

- May be cheaper to obtain from nearby cache than distant memory
- Especially when constructed out of SMP nodes (Stanford DASH)

Design of a snooping cache for the base machine

Each processor has a single-level write-back cache

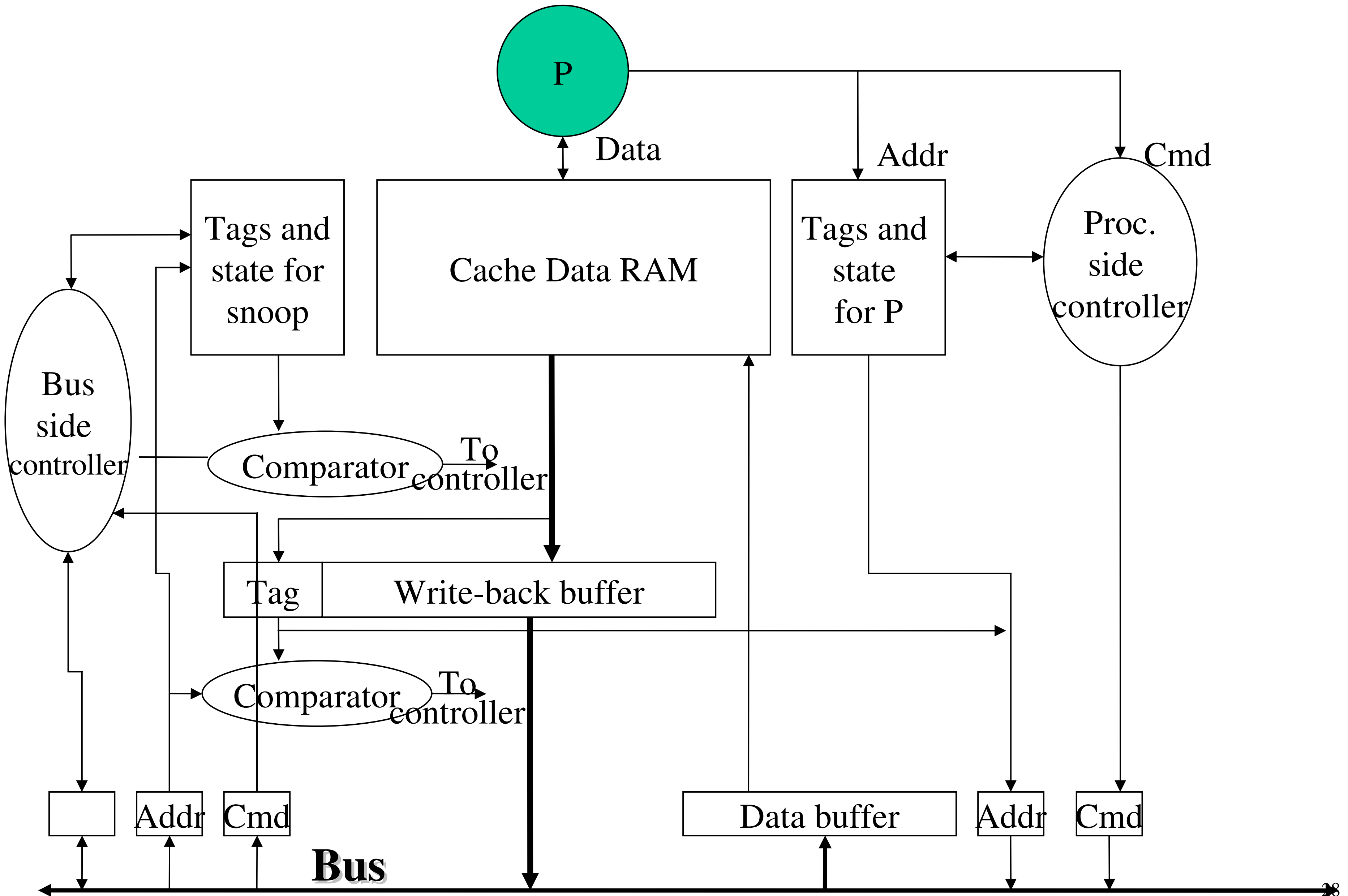
An invalidation protocol is used

Processor has only one memory request outstanding

System bus is atomic

Bus arbitration and logic is not shown

Design of a snooping cache



Directory Based Protocols

For some interconnection networks, updating or invalidating on a snoop basis is impractical

Coherence commands need to be sent to only caches that might be affected by an update

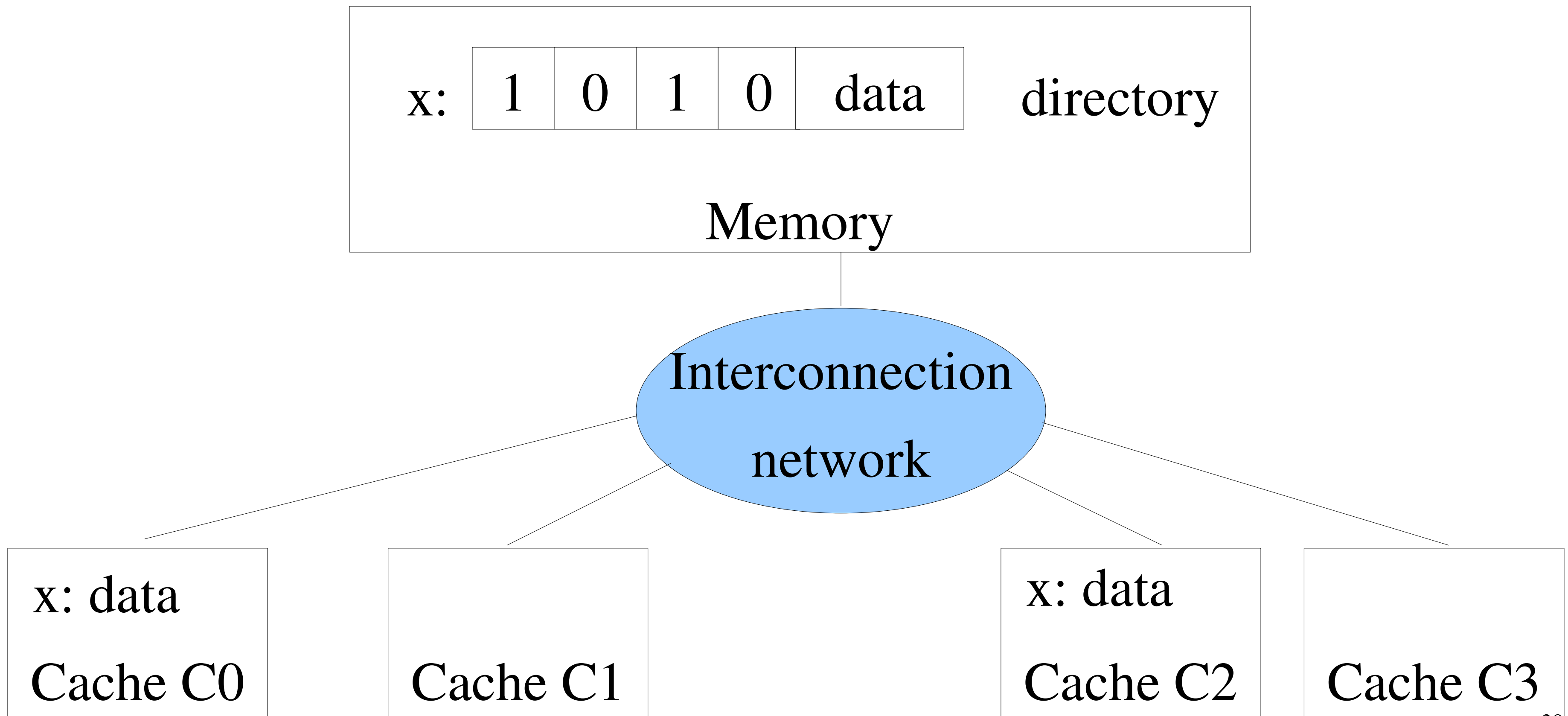
Directory Based Protocols store info on where copies of blocks lies

Directories can be centralized or distributed

Directory Based Protocols

Full-map Directories

A home node (main directory) directory contains N pointers, N=#Processors of the system

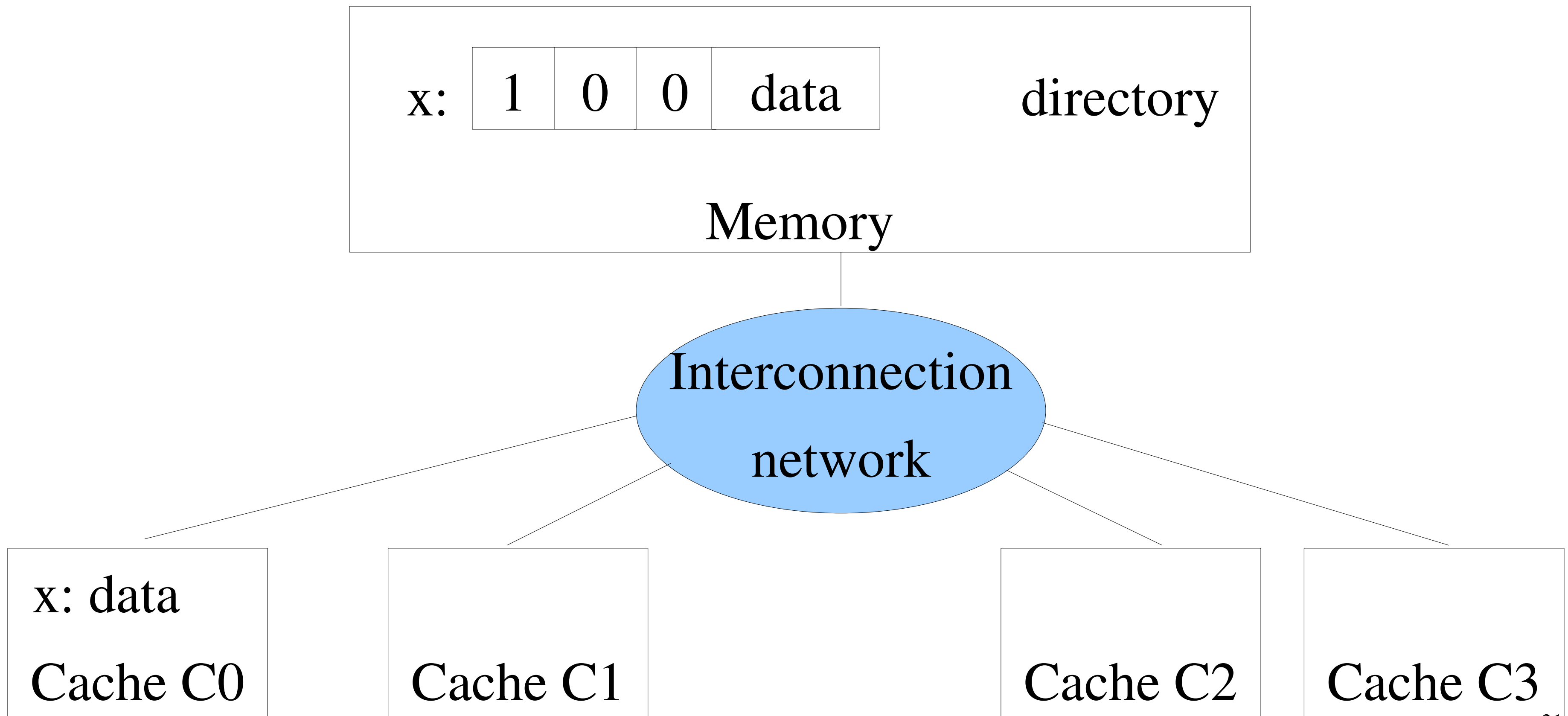


Directory Based Protocols

Limited Directories

Size of the presence vectoris limited to M (M<N)

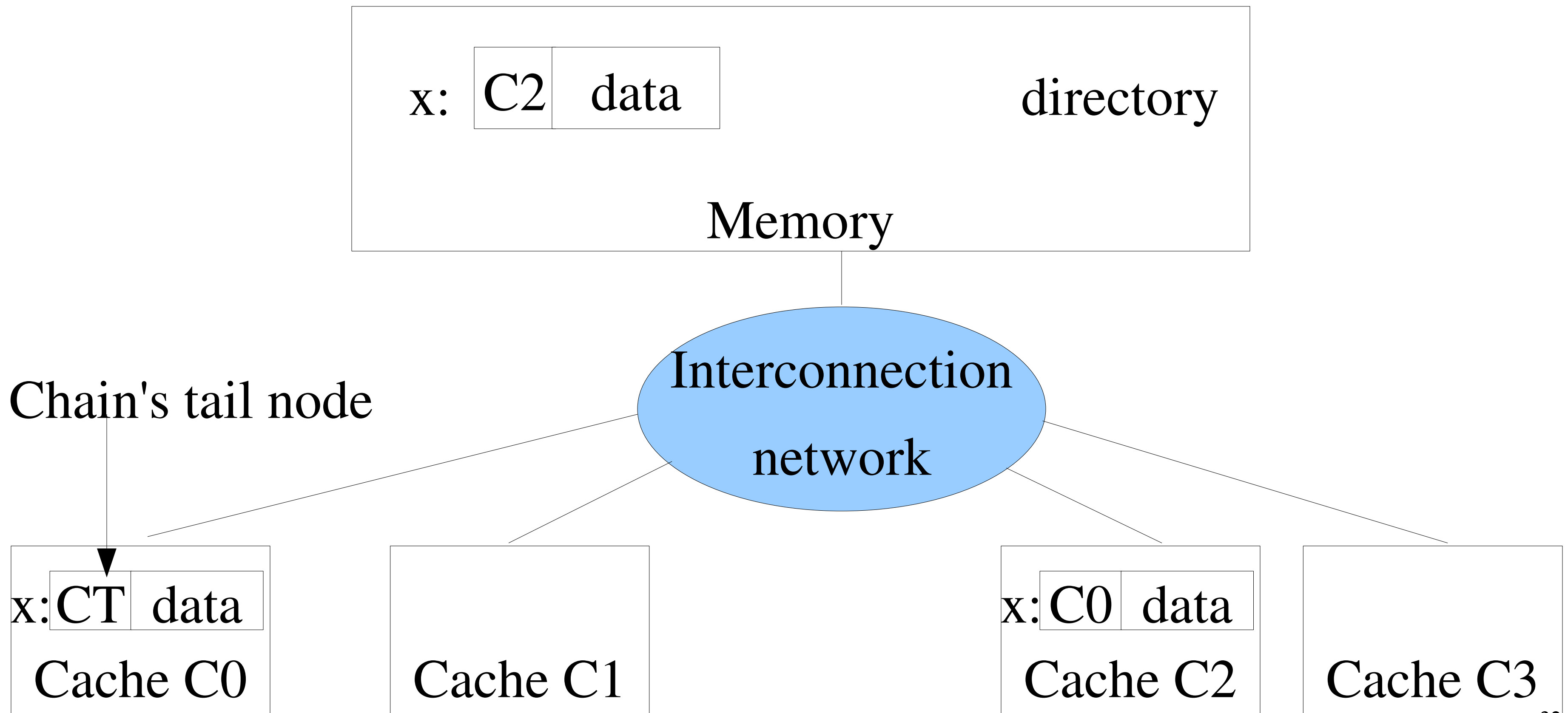
Only M processors can simultaneously share the same cache block



Directory Based Protocols

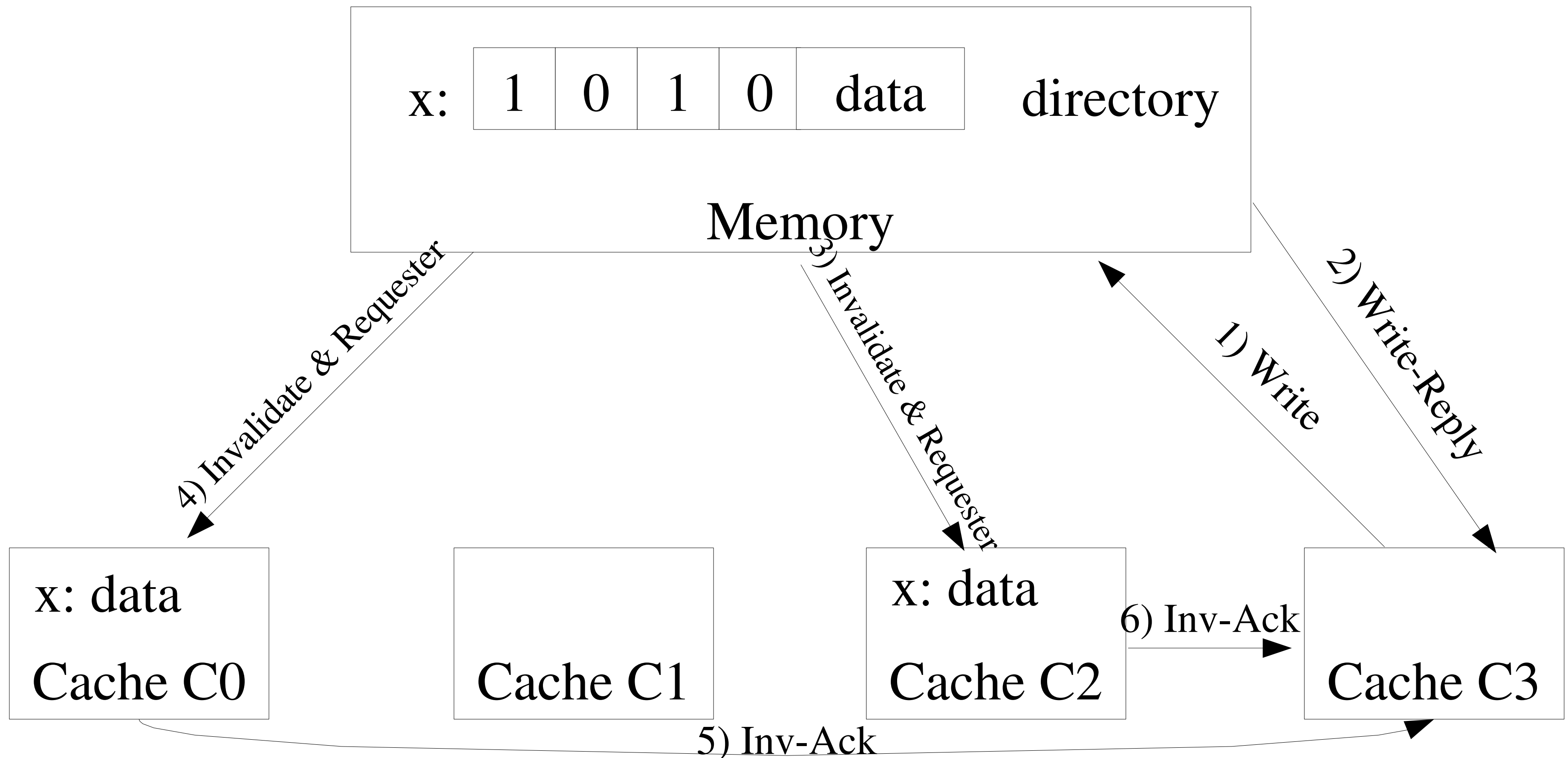
Chained directories

- Directories are distributed along the system
- A directory pointers to another cache that sharesthe cache block



Directory Based Protocols

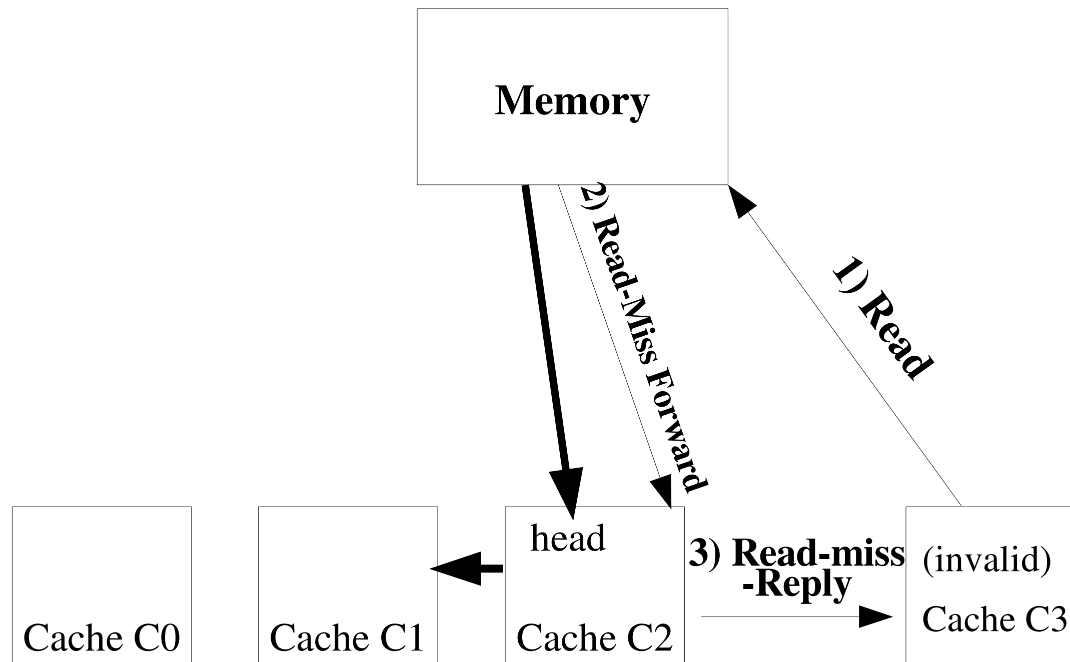
Invalidate Strategies in DBP: Centralized Directory Invalidate



Directory Based Protocols

Invalidate Strategies in DBP: Stanford Distributed Directory

A Read from a new cache (a cache that there's not in the chain yet)

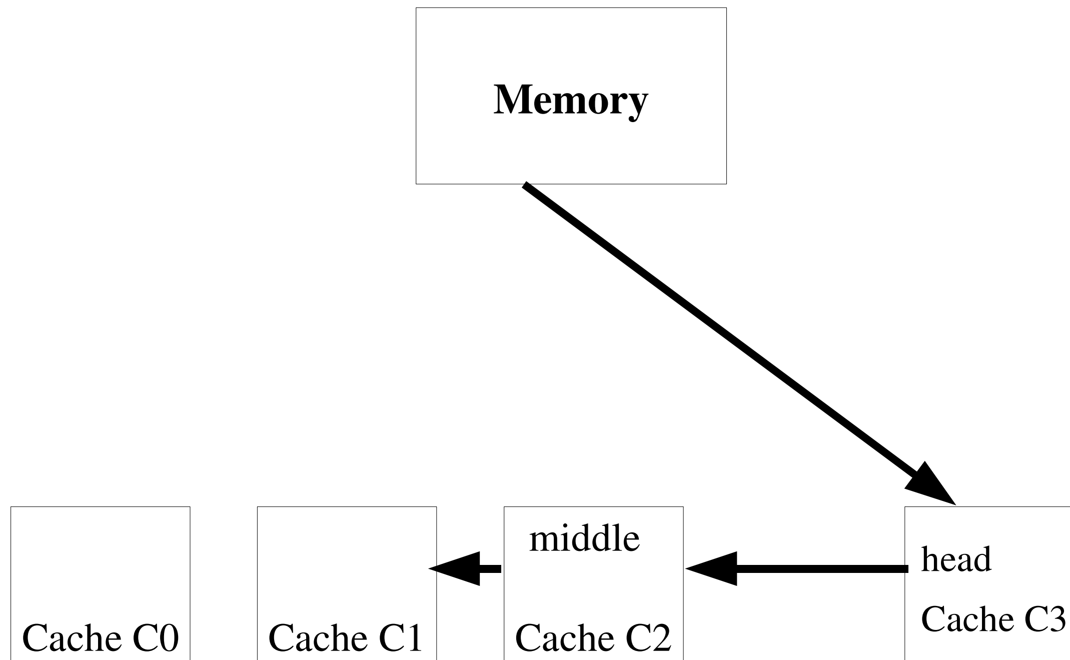


Directory Based Protocols

Invalidate Strategies in DBP: Stanford Distributed Directory

Result of the Read

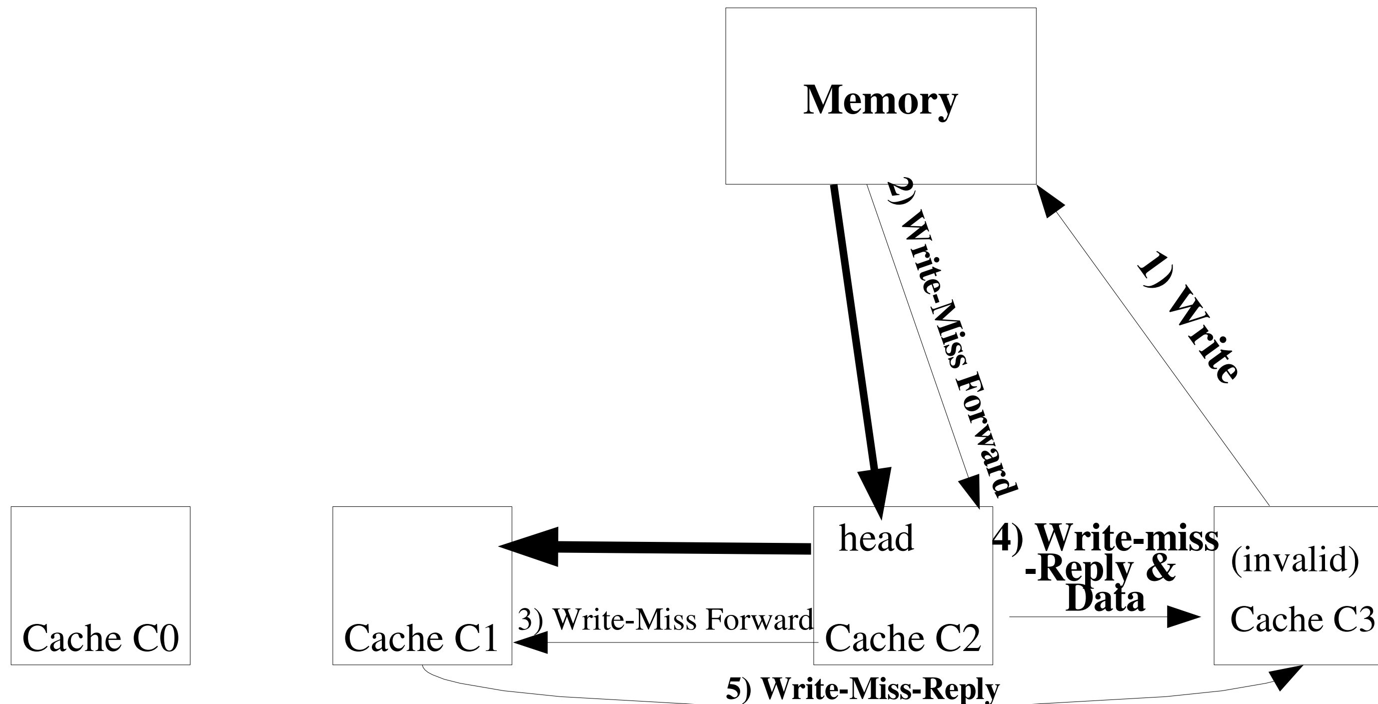
Observe that a chain has changed



Directory Based Protocols

Invalidate Strategies in DBP: Stanford Distributed Directory

A Write from a new cache (a cache that there's not in the chain yet)



Directory Based Protocols

Invalidate Strategies in DBP: Stanford Distributed Directory

Result of the Write

Observe that a chain has changed