Advanced Parallel Architecture

Annalisa Massini - 2016/2017

Cache Coherence

Computer Architecture - A Quantitative Approach, Fifth Edition

Hennessy Patterson

- Chapter 5 - Thread-Level Parallelism
  - **Section 5.2 - Centralized Shared Memory Architectures**
  - **Section 5.4 - Distributed Shared-Memory and Directory-Based Coherence**

Introduction

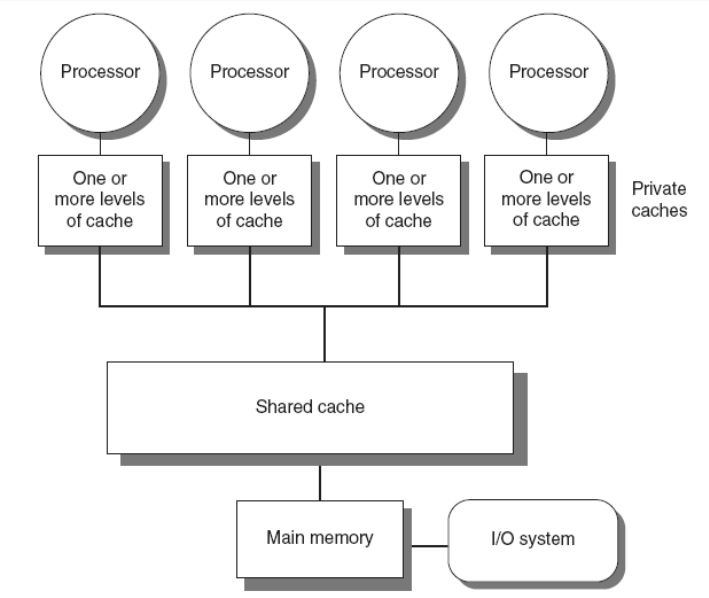
- Our focus is on ***multiprocessors***, which we define as **computers consisting of tightly coupled processors** whose coordination and usage are typically controlled by a single operating system and that **share memory through a shared address space**
- **Observe** sharing through memory implies a shared address space, it does **not** necessarily mean there is a **single physical memory**

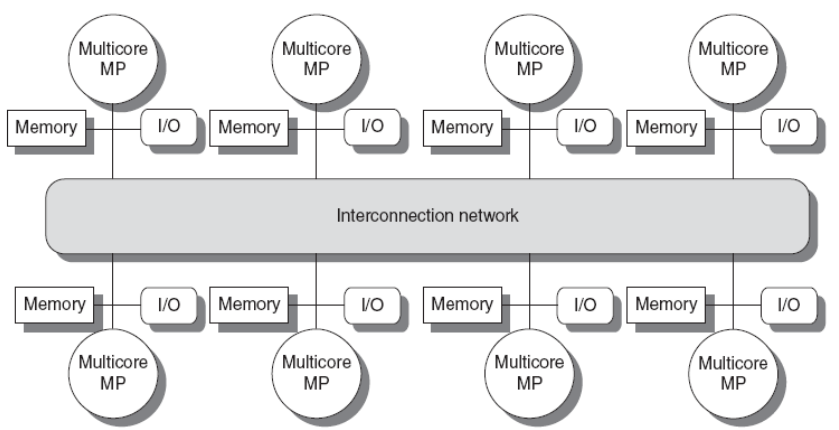
Introduction

- The **multiprocessors** we consider range in size **from a dual processor to dozens of processors**
- Multiprocessors include both **single-chip systems with multiple cores**, that is ***multicore***, and computers with **multiple chips**, each of which may be a multicore
- Thread-level parallelism is obtained through two different software models:
  - the execution of a tightly coupled set of threads collaborating on a single task, typically called ***parallel processing***
  - the execution of multiple, relatively independent processes that may originate from one or more users, which is a form of ***request level parallelism***

Shared-memory multiprocessors

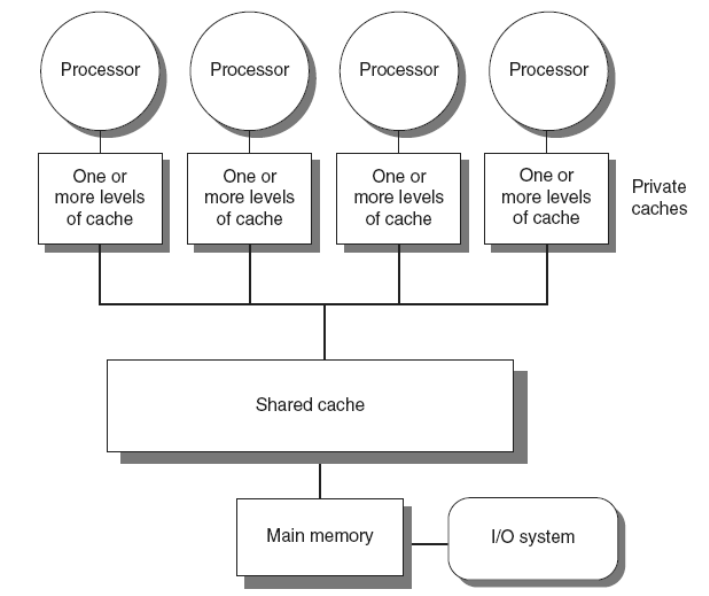
- Shared-memory multiprocessors fall into **two classes**, depending:
  - on the number of processors involved
  - a memory organization and interconnect strategy



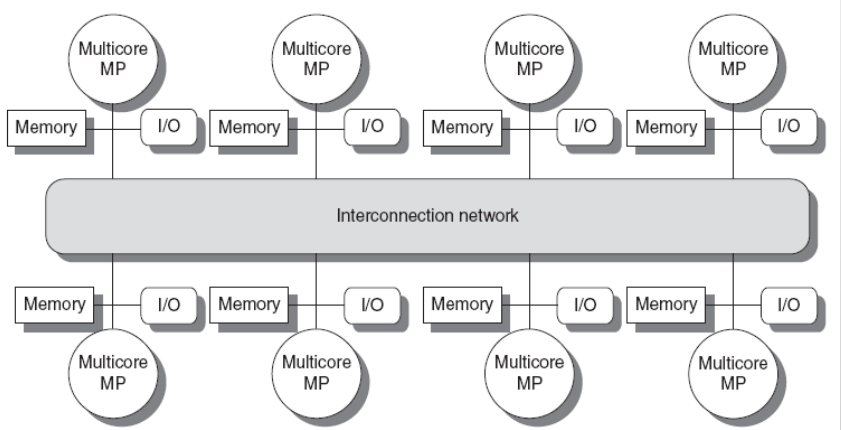


Shared-memory multiprocessors

- **Symmetric multiprocessors** (SMP)
  - Small number of cores
  - Have equal access to memory, hence the term *symmetric*
  - Share single memory with uniform memory latency

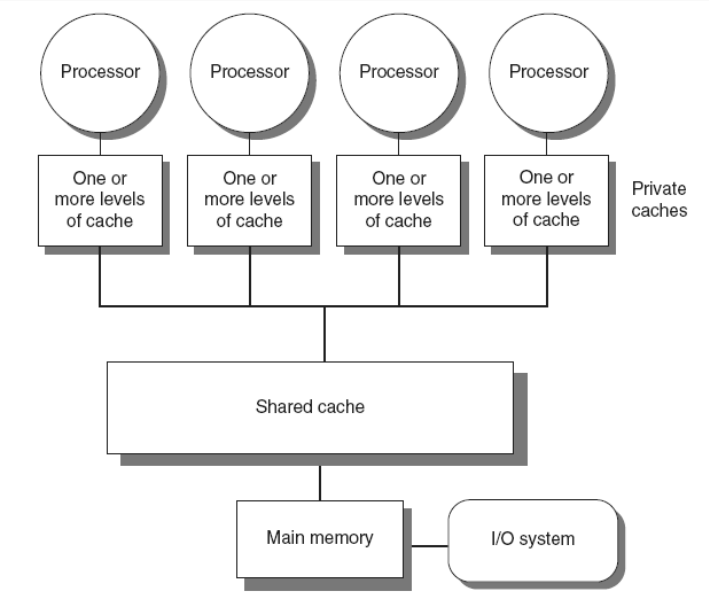


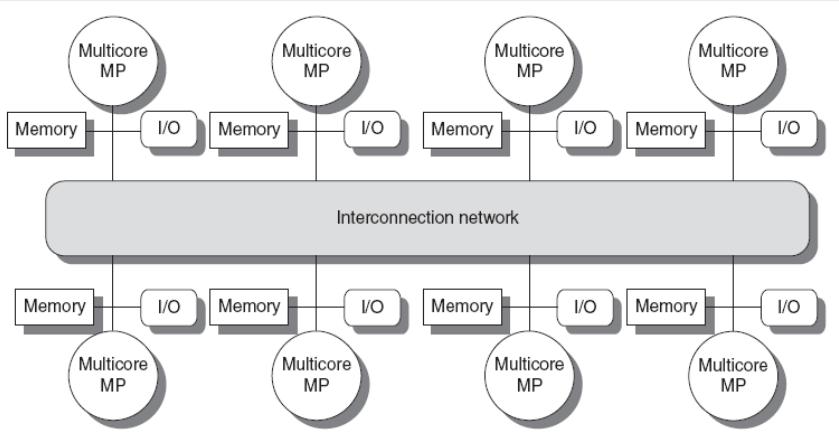
- **Distributed shared memory** (DSM)
  - Memory distributed among processors
  - Non-uniform memory access/latency (NUMA)
  - Processors connected via direct (switched) and non-direct (multi-hop) interconnection networks



Shared-memory multiprocessors

- In both SMP and DSM architectures, communication among threads occurs through a **shared address space**
- A memory reference can be made by any processor to any memory location, assuming it has the correct access rights
- The term *shared memory* associated with both SMP and DSM refers to the fact that the *address space* is shared





Centralized Shared-Memory Architectures

- **Symmetric shared-memory** machines usually support the caching of both **shared** and **private** data
- ***Private data*** are used by a single processor, whereas ***shared data*** are used by multiple processors, providing communication among the processors through reads and writes of the shared data
- When a **private** item is cached, its location is put in cache
- When **shared** data are cached, the shared value may be replicated in **multiple caches**
- **Both** reduce average access time and memory bandwidth
- Caching of shared data → **cache coherence problem**

Cache Coherence

- The view of memory by two different processors:
  - is through their individual caches
  - without any additional precautions, could see different values

Time	Event	Cache contents for processor A	Cache contents for processor B	Memory contents for location X
0				1
1	Processor A reads X	1		1
2	Processor B reads X	1	1	1
3	Processor A stores 0 into X	0	1	0

We assume a **write-through** cache. After the value of X has been written by A, A's cache and the memory both contain the new value, but B's cache does not, and if B reads the value of X it will receive 1!

Cache Coherence

- The view of memory by two different processors:
  - is through their individual caches
  - without any addit… …ent values

Time	Event	…emory …tents for …cation X
0		1
1	Processor A reac…	1
2	Processor B reac…	1
3	Processor A stor… into X	0

Write-through

- **All writes go to main memory as well as cache**
- Lots of traffic
- Slows down writes

Write-back

- **Updates initially made in cache only and *update bit* is set**
- If block is to be replaced, write to main memory only if update bit is set

We assume a **write-through** cache. After the value of X has been written by A, A's cache and the memory both contain the new value, but B's cache does not, and if B reads the value of X it will receive 1!

Cache Coherence

A memory system is **coherent** if

1. Processor P writes to location X, then P reads location X, with no writes of X by another processor occurring between the write and the read by P

   → **the read by P always returns the value written by P**

This property *preserves program order* (true in uniprocessors too)

2. Processor P' writes to location X, then P reads location X, with no writes to X occur between the two accesses

   → **the read by P returns the written value (*if the read and write are sufficiently separated in time*)**

This property defines the notion of *coherent view of memory*

Cache Coherence

A memory system is **coherent** if

3. Writes to the same location are **serialized**; that is, two writes to the same location by any two processors are seen in the same order by all processors
   - For example, if processors P1 and P2 are write to location X, serializing the writes ensures that every processor will see writes in the same order

Coherence and Consistency

- The three properties are sufficient to ensure coherence
- The question of **when a written value will be seen** is also important
- We cannot require that a read of X instantaneously see the value written for X by some other processor
- **Example**
  - a **write** of X on one processor **precedes** a **read** of X on another processor by a **very small time**
  - It is impossible to ensure that the read returns the value written, since the written data may not even have left the processor
- The issue of exactly *when* a written value must be seen by a reader is defined by a ***memory consistency model***

Coherence and Consistency

- **Coherence** defines the behavior of reads and writes to the same memory location
- **Consistency** defines the behavior of reads and writes with respect to accesses to other memory locations
- Assumptions:
  - a write does not complete (and allow the next write to occur) until all processors have seen the effect of that write
  - the processor does not change the order of any write with respect to any other memory access
- These restrictions **allow** the processor **to reorder reads**, but **forces** the processor to finish a **write in program order**

Enforcing Coherence

- Program running on multiple processors will normally have copies of the same data in several caches
- In a **coherent** multiprocessor, the caches provide:
  - *Migration*: movement of data
  - *Replication*: multiple copies of data

- Cache coherence protocols track the state of any sharing of a data block
  - Directory based
    - Sharing status of each block kept in one location
  - Snooping
    - Each core tracks sharing status of each block

Coherence Protocols

- **Snooping protocols** became popular with multiprocessors using microprocessors (single-core) and caches attached to a single shared memory by a bus
- The **bus** provided a convenient broadcast medium to implement the **snooping protocols**
- Multicore architectures changed the picture significantly, since all multicores share some level of cache on the chip
- Thus, some designs switched to using **directory protocols**, since the overhead was small

Snooping Coherence Protocols

- One method to maintain the coherence requirement is to ensure that a processor has **exclusive access** to a data item before it writes that item
- The other copies of the block are invalidated on a write and the protocol is called a ***write invalidate***
- Exclusive access ensures that no other readable or writable copies of an item exist when the write occurs: **all other cached copies of the item are invalidated**

- It is the most common protocol

Snooping Coherence Protocols

Example of an invalidation protocol with write-back caches in action

Processor activity	Bus activity	Contents of processor A's cache	Contents of processor B's cache	Contents of memory location X
				0
Processor A reads X	Cache miss for X	0		0
Processor B reads X	Cache miss for X	0	0	0
Processor A writes a 1 to X	Invalidation for X	1		0
Processor B reads X	Cache miss for X	1	1	1

Consider:

- a **write followed by a read** by another processor
- since the write requires exclusive access, **any copy** held by the reading processor **must be invalidated**
- when the read occurs, it **misses** and must **fetch a new copy**

Snooping Coherence Protocols

Example of an invalidation protocol with write-back caches in action

Processor activity	Bus activity	Contents of processor A's cache	Contents of processor B's cache	Contents of memory location X
				0
Processor A reads X	Cache miss for X	0		0
Processor B reads X	Cache miss for X	0	0	0
Processor A writes a 1 to X	Invalidation for X	1		0
Processor B reads X	Cache miss for X	1	1	1

Observe:

- when the second **miss by B** occurs, **processor A responds** with the value canceling the response from memory
- in addition, **both** the contents of **B's cache** and the **memory** contents of X are **updated**

Snooping Coherence Protocols

Example of an invalidation protocol with write-back caches in action

Processor activity	Bus activity	Contents of processor A's cache	Contents of processor B's cache	Contents of memory location X
				0
Processor A reads X	Cache miss for X	0		0
Processor B reads X	Cache miss for X	0	0	0
Processor A writes a 1 to X	Invalidation for X	1		0
Processor B reads X	Cache miss for X	1	1	1

Observe:

- introduction of additional state, **owner**
- It indicates that a block may be shared, but the **owning processor is responsible for updating** any other processors and memory when it changes the block or replaces it

Snooping Coherence Protocols

- If two processors do attempt to write the same data simultaneously:
  - one of them wins the race
  - the other processor's copy to be invalidated
- For the other processor to complete its write, it must obtain a new copy of the data, which must now contain the updated value
- This protocol enforces **write serialization**

Snooping Coherence Protocols

- The alternative to an invalidate protocol is to **update all the cached copies** of a data item when that item is written
- This protocol is called a ***write update*** or ***write broadcast protocol***
- Because a **write update** protocol must **broadcast** all writes to shared cache lines, it consumes considerably **more bandwidth**

- Recent multiprocessors have opted to implement a write invalidate protocol

Basic Implementation Techniques

- An **invalidate protocol** in a multicore is based on the use of the **bus**, or another **broadcast medium**, to perform invalidates
- In older multiple-chip **multiprocessors**, the bus used for coherence is the **shared-memory access bus**
- In a **multicore**, the bus can be the **connection between the private caches and the shared outer cache** (in the Intel Core i7, L1 and L2 are private and L3 is shared)
- To perform an **invalidate**, the processor simply **acquires bus access and broadcasts the address** to be invalidated on the bus

Basic Implementation Techniques

- Actions
  - All processors continuously **snoop** on the bus, **watching the addresses**
  - The processors **check** whether the **address** on the bus is **in their cache**
  - If so, the corresponding data in the cache are **invalidated**

Basic Implementation Techniques

- We saw that:
  - when a write to a block that is shared occurs, the writing processor must acquire bus access to broadcast its invalidation
- If **two processors** attempt to **write shared blocks** at the same time, their attempts to broadcast an invalidate operation will be **serialized when they arbitrate** for the bus
- The first processor:
  - obtains bus access
  - causes other copies of the block it is writing to be invalidated
- If the processors were attempting to **write the same block**, the serialization enforced by the bus also serializes their writes

Basic Implementation Techniques

- In addition to invalidating outstanding copies of a cache block that is being written into, we also need to locate a data item when a **cache miss** occurs
- In a **write-through cache**, it is **easy** to find the recent value of a data item:
  - all written data are always sent to the memory, from which the most recent value of a data item can always be fetched
- For a **write-back cache**, the problem of finding the most recent data value is **harder**:
  - the most recent value of a data item can be in a private cache rather than in the shared cache or memory

Basic Implementation Techniques

- **Write-back caches** can use the same snooping scheme **both for cache misses and for writes**:
  - Each processor snoops every address placed on the shared bus
  - If a processor finds that it has a dirty copy of the requested cache block, it provides that cache block in response to the read request and causes the memory (or L3) access to be aborted
- Retrieving the cache block from another processor's private cache (L1-L2) takes longer than retrieving it from L3
- **Write-back caches**
  - generate lower requirements for memory bandwidth
  - can support larger numbers of faster processors
  - then multicore processors use write-back at the outermost levels of the cache

Basic Implementation Techniques

- The normal cache tags can be used to implement the process of snooping
- The valid bit for each block makes invalidation easy to implement
- **Read** misses, whether generated by an invalidation or by some other event, are also straightforward since they simply rely on the snooping capability
- For **writes** if there are no other cached copies of the block
  - then the write need not be placed on the bus in a write-back cache
  - both the time to write and the required bandwidth are reduces

Basic Implementation Techniques

- To track whether or not a **cache block is shared**, we can add an **extra state bit** associated with each cache block, just as we have a valid bit and a dirty bit
- By adding a bit indicating whether the block is shared, we can decide whether a write must generate an invalidate
- When a write to a block in the shared state occurs, the cache generates an invalidation on the bus and marks the block as **exclusive**
- No further invalidations will be sent by that core for that block

Basic Implementation Techniques

- The core with the sole copy of a cache block is called the ***owner*** of the cache block
- When an invalidation is sent, the state of the owner's cache block is **changed from shared to unshared** (or **exclusive**)
- If another processor later requests this cache block, *the state must be made shared again*
- Since our snooping cache also sees any misses, it knows when the exclusive cache block has been requested by another processor and the state should be made shared

Snooping Coherence Protocols

- Locating an item when a read miss occurs
  - In write-back cache, the updated value must be sent to the requesting processor

- Cache lines marked as shared or exclusive/modified
  - Only writes to shared lines need an invalidate broadcast
    - After this, the line is marked as exclusive

Snooping Coherence Protocols

- A snooping coherence protocol is usually implemented by incorporating a **finite state controller** in each core
- This controller:
  - responds to requests **from the processor** in the core and **from the bus** (or other broadcast medium)
  - **changes the state of the selected cache block**
  - uses the bus to **access data** or to **invalidate it**

Snooping Coherence Protocols

- Consider a simple protocol with three states:
  - **invalid** indicates that the block has been updated somewhere
  - **shared** indicates that the block in the private cache is potentially shared
  - **modified/*exclusive*** indicates that the block has been updated in the private cache

Snooping Coherence Protocols

- Table shows the requests generated **by a core** for a write-back cache

Request	Source	State of addressed cache block	Type of cache action	Function and explanation
Read hit	Processor	Shared or modified	Normal hit	Read data in local cache.
Read miss	Processor	Invalid	Normal miss	Place read miss on bus.
Read miss	Processor	Shared	Replacement	Address conflict miss: place read miss on bus.
Read miss	Processor	Modified	Replacement	Address conflict miss: write-back block, then place read miss on bus.
Write hit	Processor	Modified	Normal hit	Write data in local cache.
Write hit	Processor	Shared	Coherence	Place invalidate on bus. These operations are often called upgrade or *ownership* misses, since they do not fetch the data but only change the state.
Write miss	Processor	Invalid	Normal miss	Place write miss on bus.
Write miss	Processor	Shared	Replacement	Address conflict miss: place write miss on bus.
Write miss	Processor	Modified	Replacement	Address conflict miss: write-back block, then place write miss on bus.

Snooping Coherence Protocols

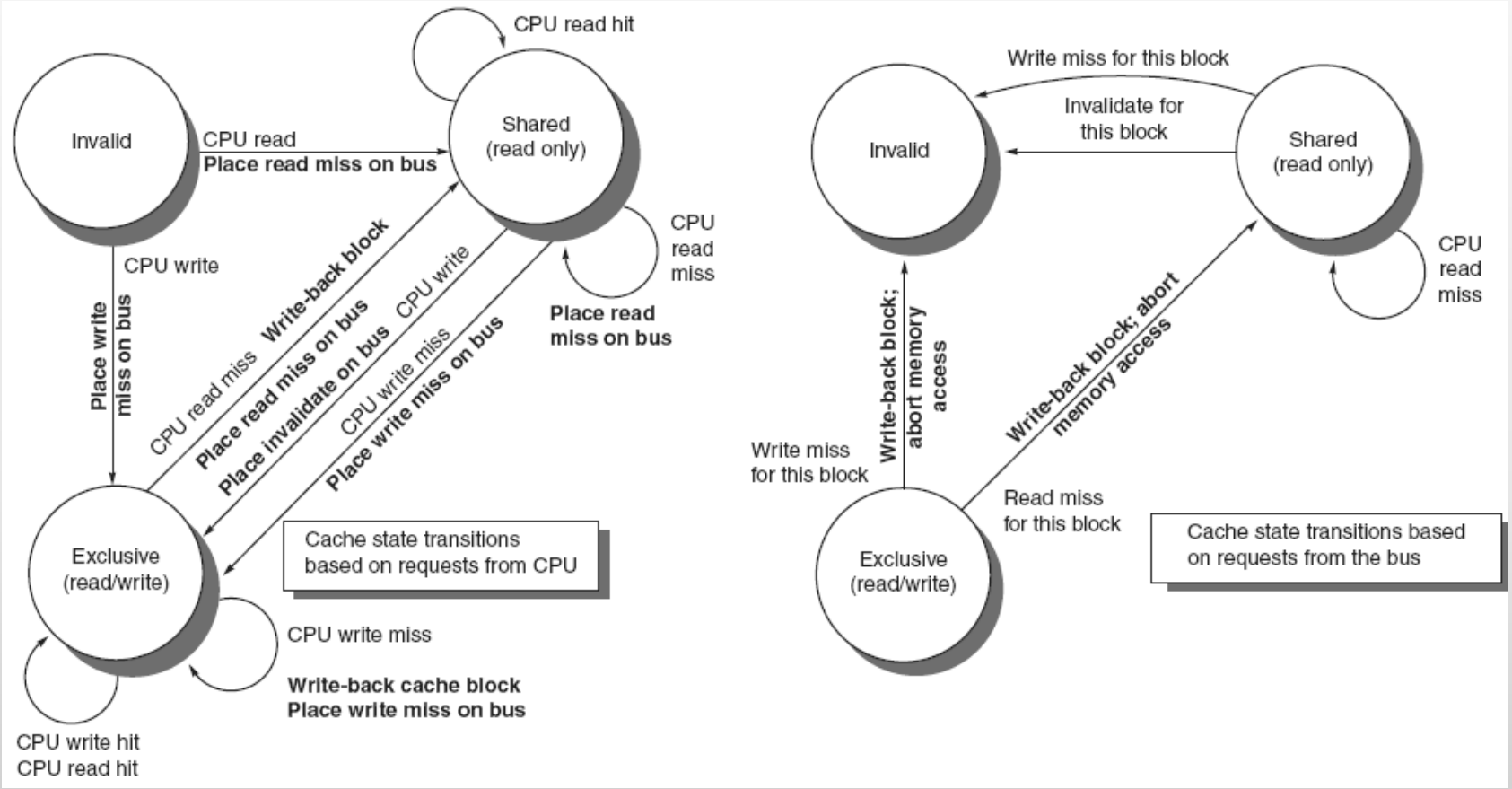
- Table shows the requests generated **by the bus** for a write-back cache

Request	Source	State of addressed cache block	Type of cache action	Function and explanation
Read miss	Bus	Shared	No action	Allow shared cache or memory to service read miss.
Read miss	Bus	Modified	Coherence	Attempt to share data: place cache block on bus and change state to shared.
Invalidate	Bus	Shared	Coherence	Attempt to write shared block; invalidate the block.
Write miss	Bus	Shared	Coherence	Attempt to write shared block; invalidate the cache block.
Write miss	Bus	Modified	Coherence	Attempt to write block that is exclusive elsewhere; write-back the cache block and make its state invalid in the local cache.

- When an invalidate or a write miss is placed on the bus, any cores whose private caches have copies of the cache block invalidate it

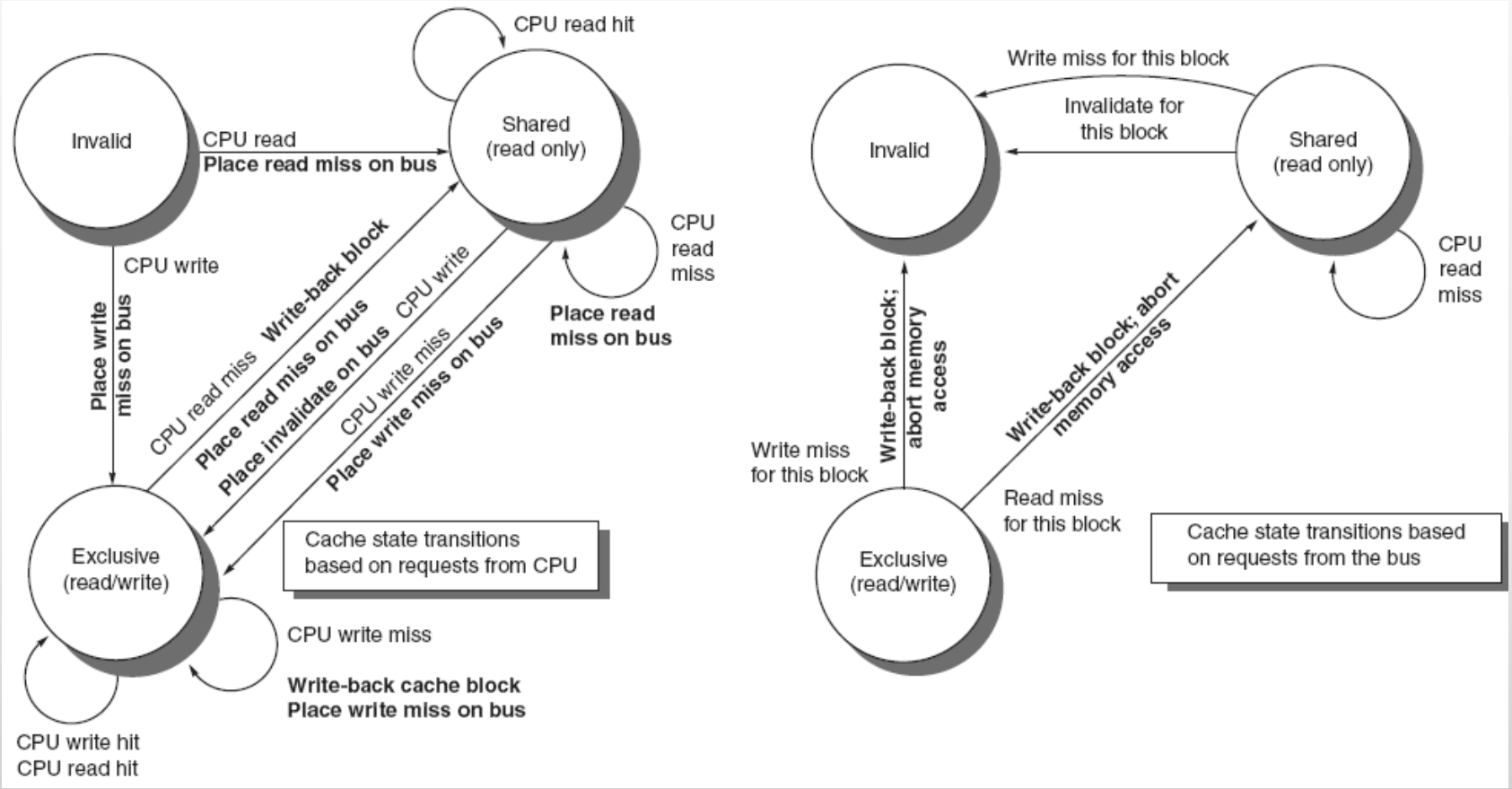
Snooping Coherence Protocols

- Figure shows a finite-state transition diagram for a single private cache block using a write invalidation protocol and a write-back cache



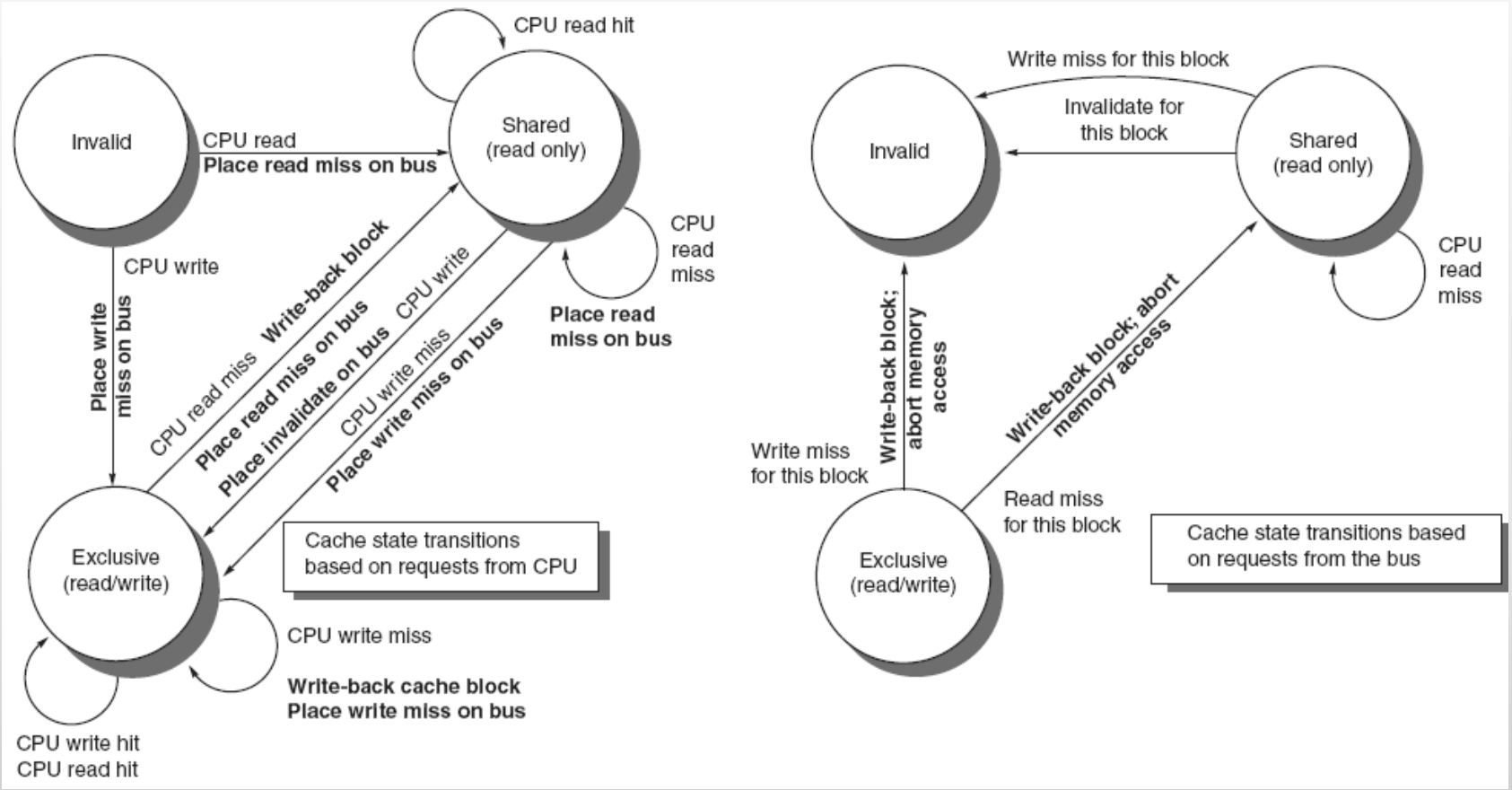
Snooping Coherence Protocols

- The three states of the protocol are duplicated to represent transitions based on processor requests (left), and bus requests (right)



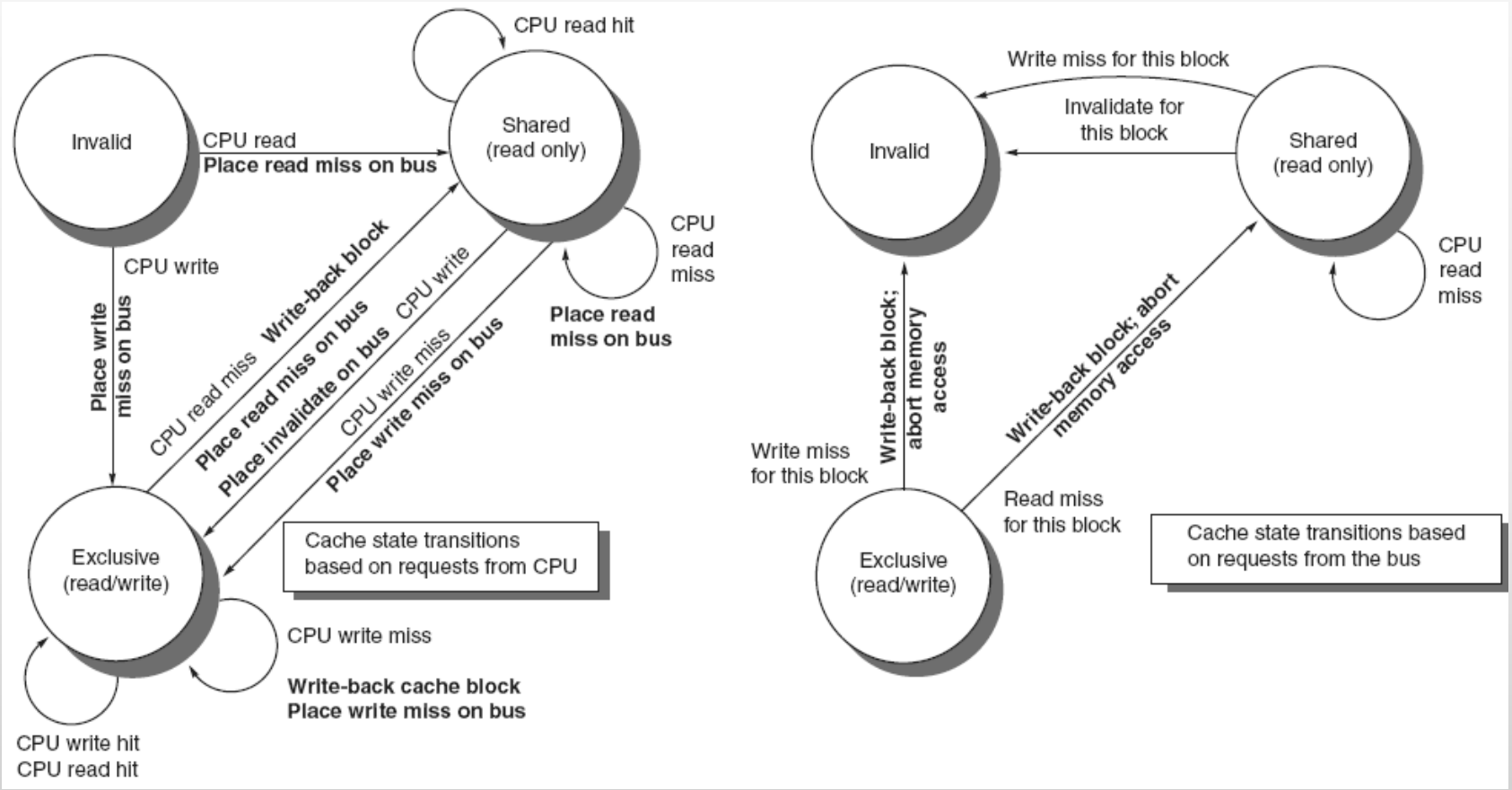
Snooping Coherence Protocols

- There is **only one finite-state machine per cache**, with stimuli coming either from the attached processor or from the bus



Snooping Coherence Protocols

- The stimulus causing a state change is shown on the transition arcs in regular type, and any **bus actions** generated as part of the state transition are shown on the transition arc in **bold**



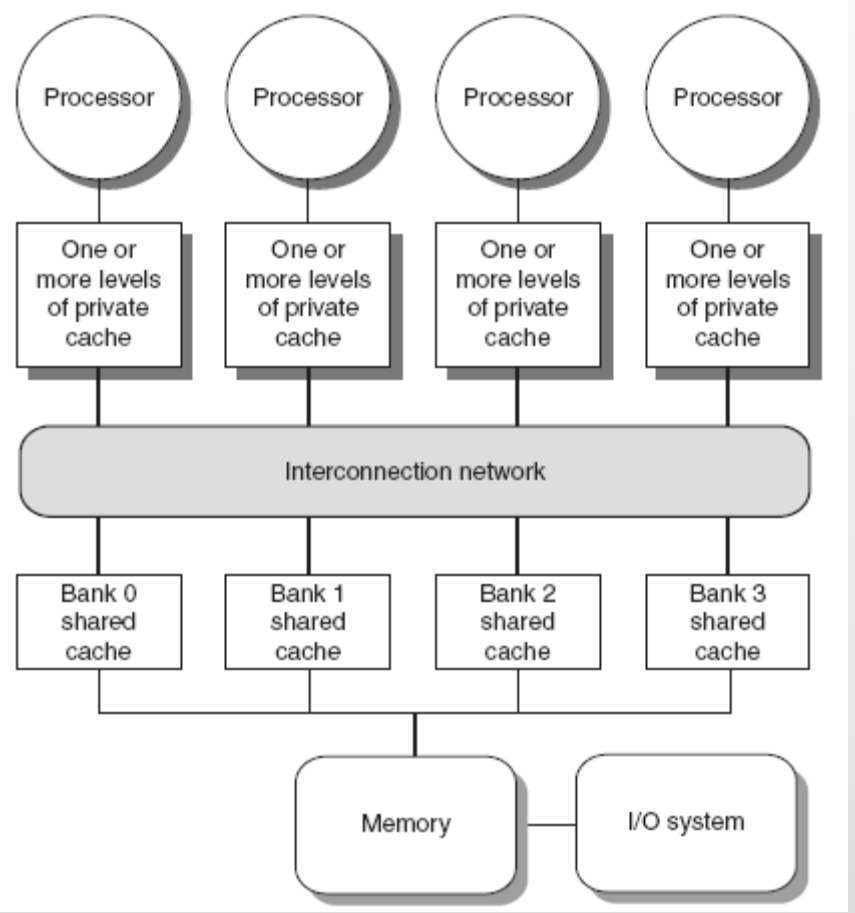
Snooping Coherence Protocols

- The simple cache protocol is referred to by the first letter of the states, making it a **MSI** (Modified, Shared, Invalid)
- It is correct, but omits a number of complications:
  - **Operations are not atomic**
    - E.g. detect miss, acquire bus, receive a response is **not atomic**
    - Non atomic actions creates possibility of **deadlock**
    - One solution: processor that sends invalidate can hold bus until other processors receive the invalidate
- Extensions:
  - Add **exclusive state** to indicate clean block in only one cache (**MESI** protocol)
  - Add **owned state** to indicate that the associated block is owned by that cache and out-of-date in memory

Coherence Protocols

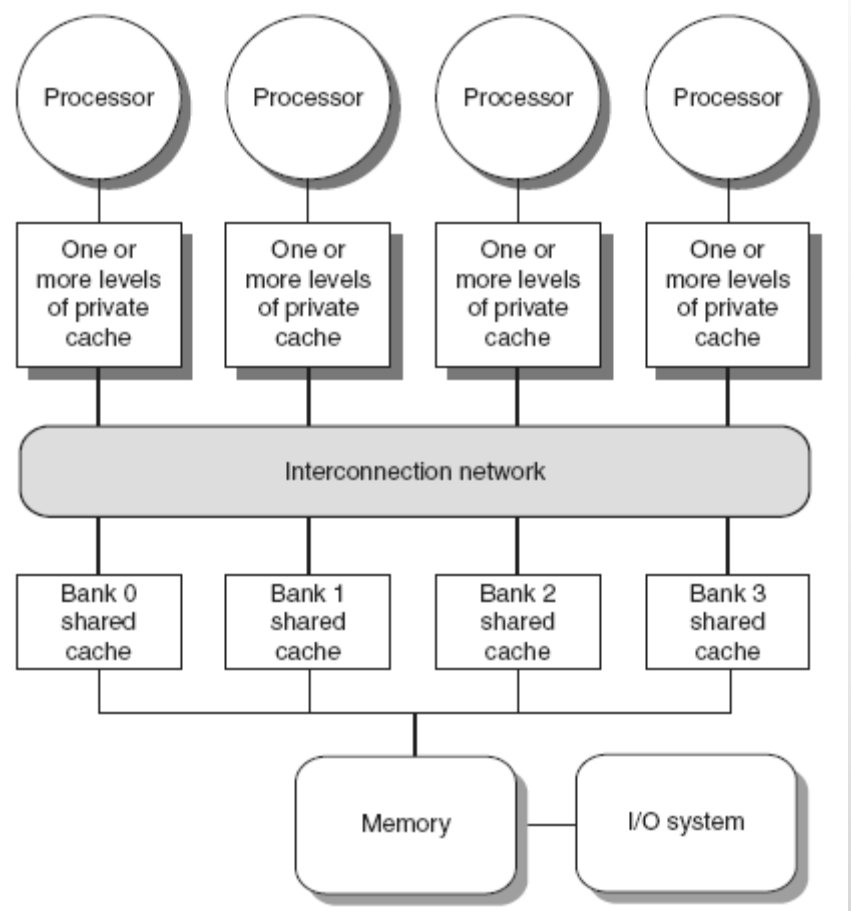
- Any centralized resource in the system can become a bottleneck:
  - As the number of processors in a multiprocessor grows
  - or as the memory demands of each processor grow,

  shared memory bus and **snooping bandwidth** is **bottleneck** for scaling **symmetric multiprocessors**



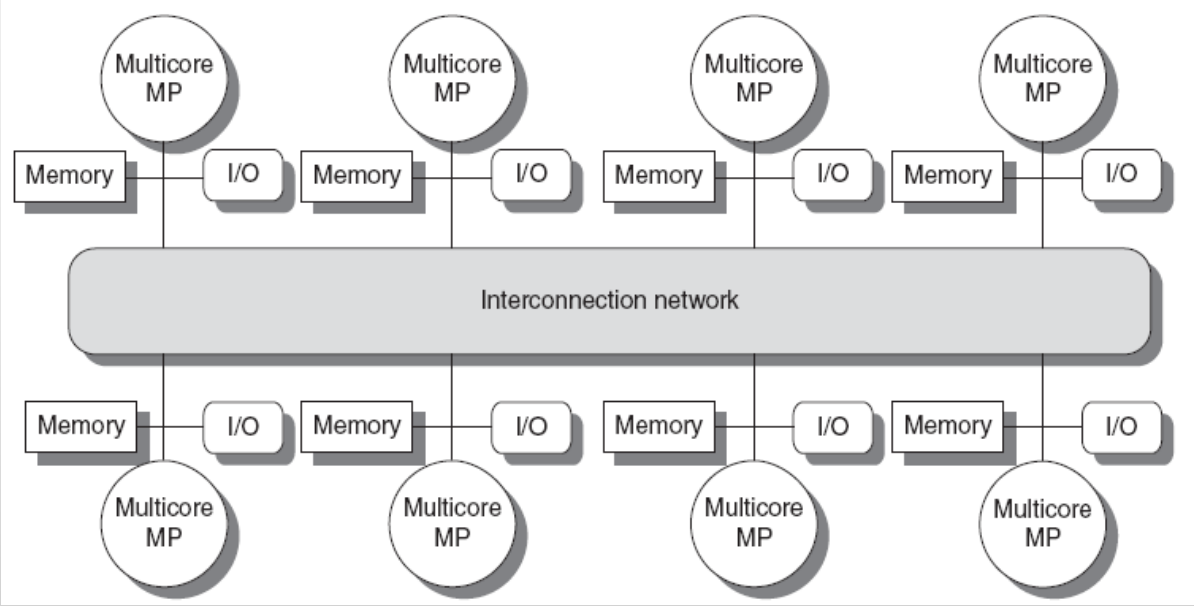
Coherence Protocols: Extensions

- Solutions to bus and snooping bottlenecks:
  - **Duplicating tags** to have direct snoop access without interfering with processor cache accesses
  - Place **directory in outermost cache** → the directory indicates whether a given block is shared and possibly which cores have copies
  - Interconnection networks (crossbars or small point-to-point networks) or multiple buses with banked memory



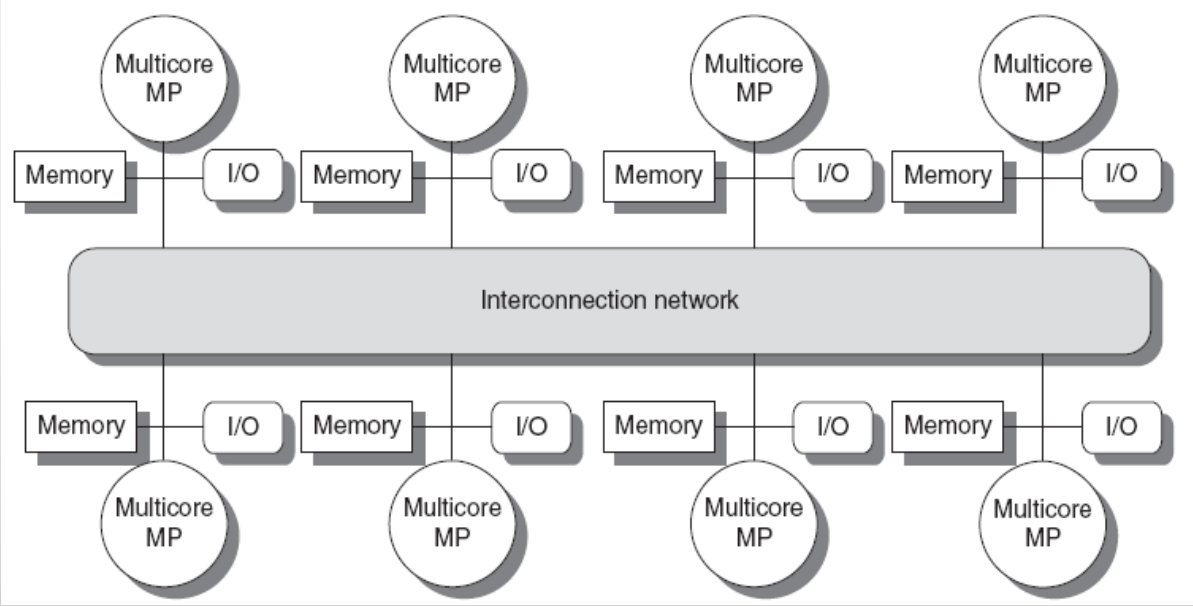
Distributed Shared-Memory

- The development of multiprocessors composed of multicores forced designers to some form of distributed memory:
  - local memory traffic is separated from remote memory traffic
  - bandwidth demands on the memory system and on the interconnection network is reduced



Distributed Shared-Memory

- The absence of any centralized data structure for caches is both
  - fundamental **advantage** of a snooping-based scheme - inexpensive
  - its **Achilles' heel** - scalability
- The introduction of distributed memory is useful if **broadcast on every cache miss** in the coherence protocol is **eliminated**



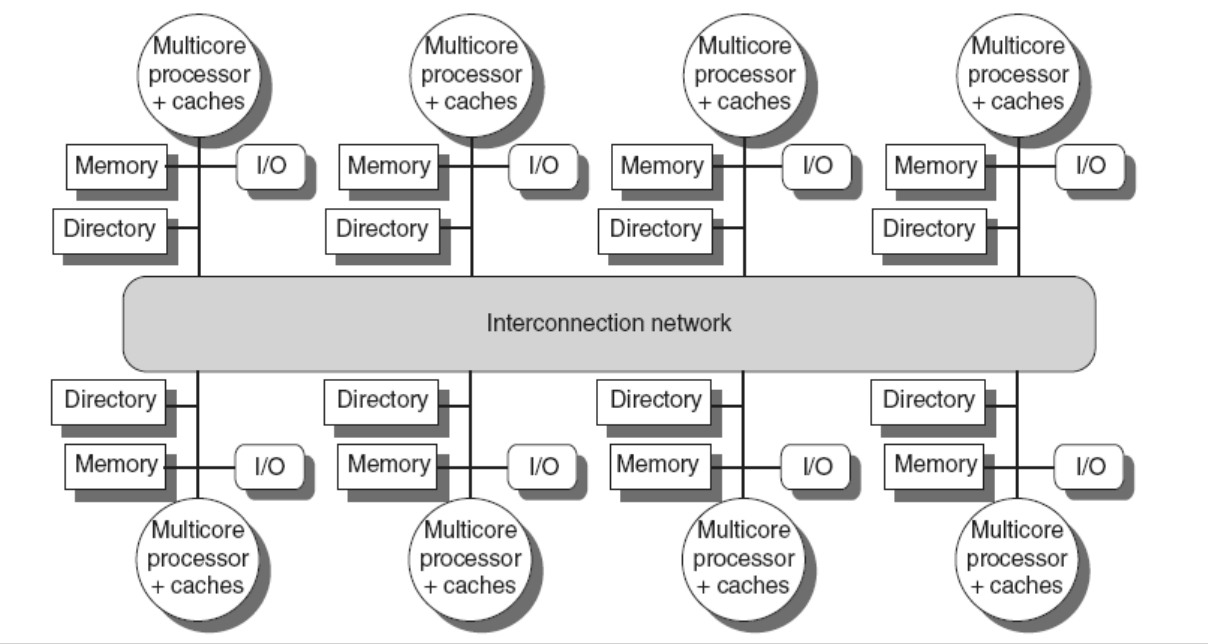
Directory-Based Coherence

- ***Directory protocols*** instead of snooping coherence protocol
- Directory keeps track of every block
  - Which caches have copy of the block
  - Dirty status of each block

- Implement in shared L3 cache
  - Keep bit vector of size = # cores for each block in L3
  - bit vector indicates which private caches may have copies of a block in L3
  - invalidations are only sent to those caches
  - (scheme is the one used in the Intel i7)

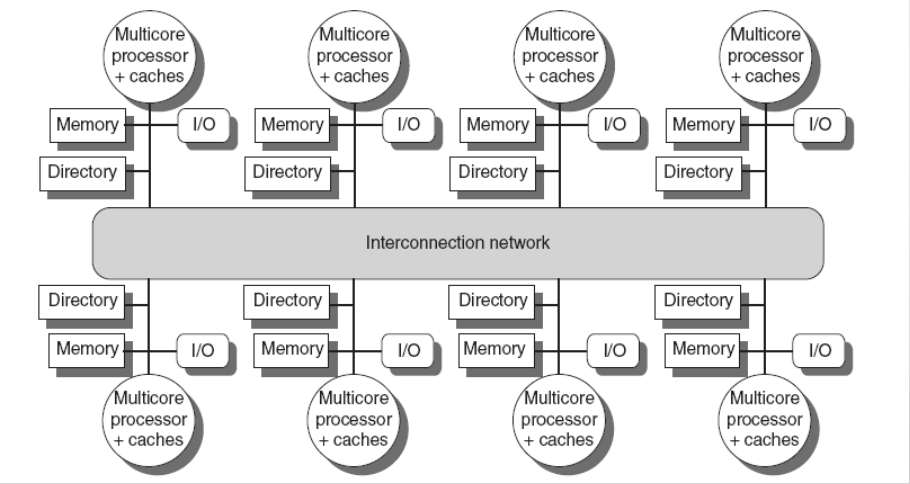
Directory-Based Coherence

- A **single directory** used in a multicore is **not scalable**, even though it avoids broadcast
- The **directory must be distributed**, along with the memory
  - different coherence requests can go to different directories, just as different memory requests go to different memories



Directory-Based Coherence

- The **distribution** must allow the coherence protocol knows **where to find the directory** information for any cached block to avoid broadcast
- The coherence protocol in a distributed directory is based on
  - the characteristic that the **sharing status of a block** is always in a **single known location**
  - the maintenance of information that says **what other nodes** may be caching the block



Directory-Based Coherence Protocols

- Just as with a snooping protocol, there are **two primary operations** that a directory protocol must implement:
  - handling a **read miss**
  - handling a **write** to a shared, clean cache block
- **Observe** that handling a **write miss** to a block that is currently shared is a simple combination of the previous two

- To implement these operations, a directory must track the state of each cache block

Directory-Based Coherence Protocols

- For each block, state could be:
  - **Shared**
    - One or more nodes have the block cached, value in memory is up-to-date
    - Set of node IDs
  - **Uncached**
    - No node has a copy of the cache block
  - **Modified**
    - Exactly one node has a copy of the cache block, value in memory is out-of-date
    - Owner node ID
- Directory maintains block states and sends invalidation messages

Directory-Based Coherence Protocols

- In addition to tracking the state of each potentially shared memory block, we must **track which nodes have copies of that block**, to invalidated them on a write

- The simplest way to do this is:
  - To keep a **bit vector for each memory block**
  - When the block is shared, each bit of the vector indicates whether the corresponding processor chip (which is likely a multicore) has a copy of that block
  - The bit vector keeps also track of the **owner** of the block when the block is in the **exclusive state**
  - The state of each cache block at the individual caches is also tracked

Directory-Based Coherence Protocols

- The states and transitions for the state machine at each cache
  - Are identical to what we used for the snooping cache
  - **But** the actions on a transition are slightly different
- Infact
  - The processes of **invalidating** and **locating an exclusive copy** of a data item are **different**, since they both involve communication between the requesting node and the directory and between the directory and one or more remote nodes
  - *In a snooping protocol, these two steps are combined through the use of a broadcast to all the nodes*

Messages

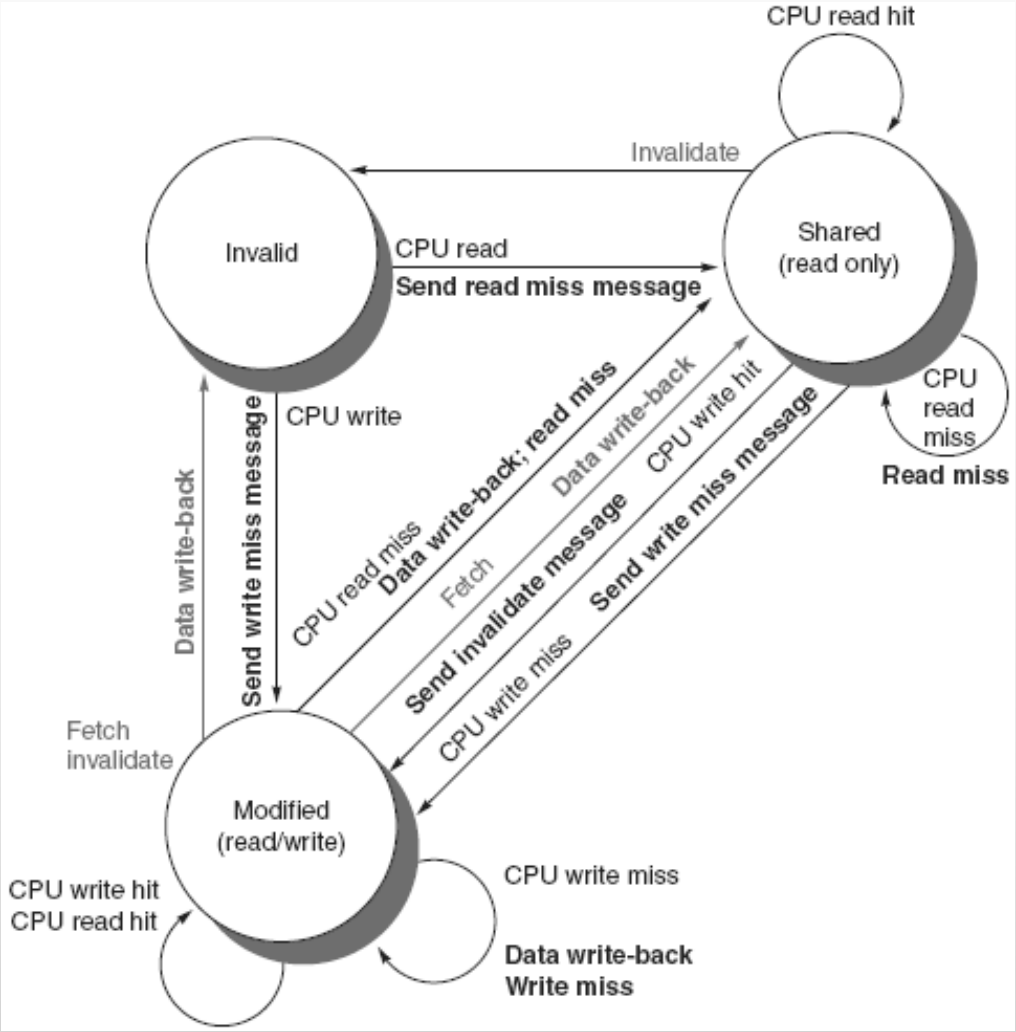
The possible messages sent among nodes to maintain coherence, with source and destination node, contents, and function of the message

P = requesting node number, A = requested address, and D = data contents

Message type	Source	Destination	Message contents	Function of this message
Read miss	Local cache	Home directory	P, A	Node P has a read miss at address A; request data and make P a read sharer.
Write miss	Local cache	Home directory	P, A	Node P has a write miss at address A; request data and make P the exclusive owner.
Invalidate	Local cache	Home directory	A	Request to send invalidates to all remote caches that are caching the block at address A.
Invalidate	Home directory	Remote cache	A	Invalidate a shared copy of data at address A.
Fetch	Home directory	Remote cache	A	Fetch the block at address A and send it to its home directory; change the state of A in the remote cache to shared.
Fetch/invalidate	Home directory	Remote cache	A	Fetch the block at address A and send it to its home directory; invalidate the block in the cache.
Data value reply	Home directory	Local cache	D	Return a data value from the home memory.
Data write-back	Remote cache	Home directory	A, D	Write-back a data value for address A.

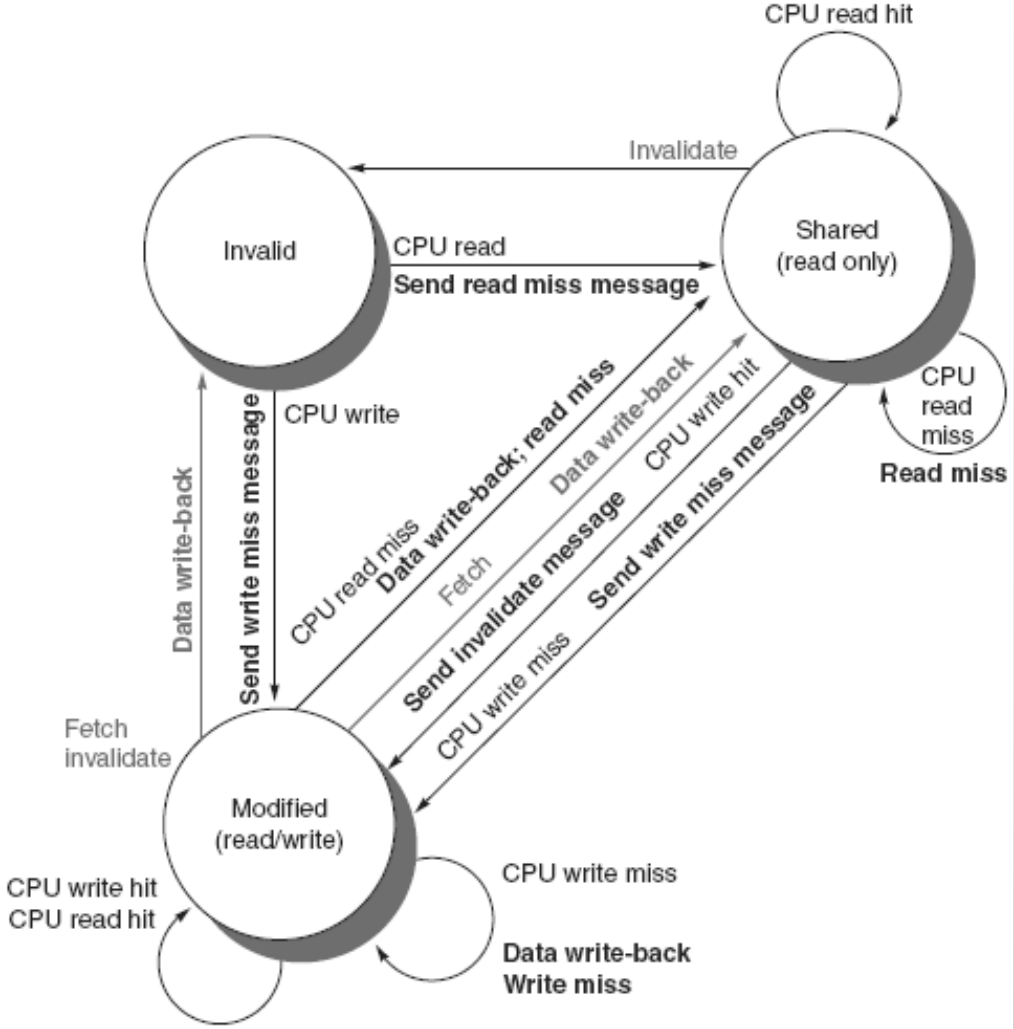
Directory-Based Coherence Protocols

- **State transition diagram** for an **individual cache block** in a **directory based system**
- **Observe** that there is also a **state diagram for the directory** entry corresponding to each block in memory
- Requests
  - by the local processor are **black**
  - from the home directory **gray**



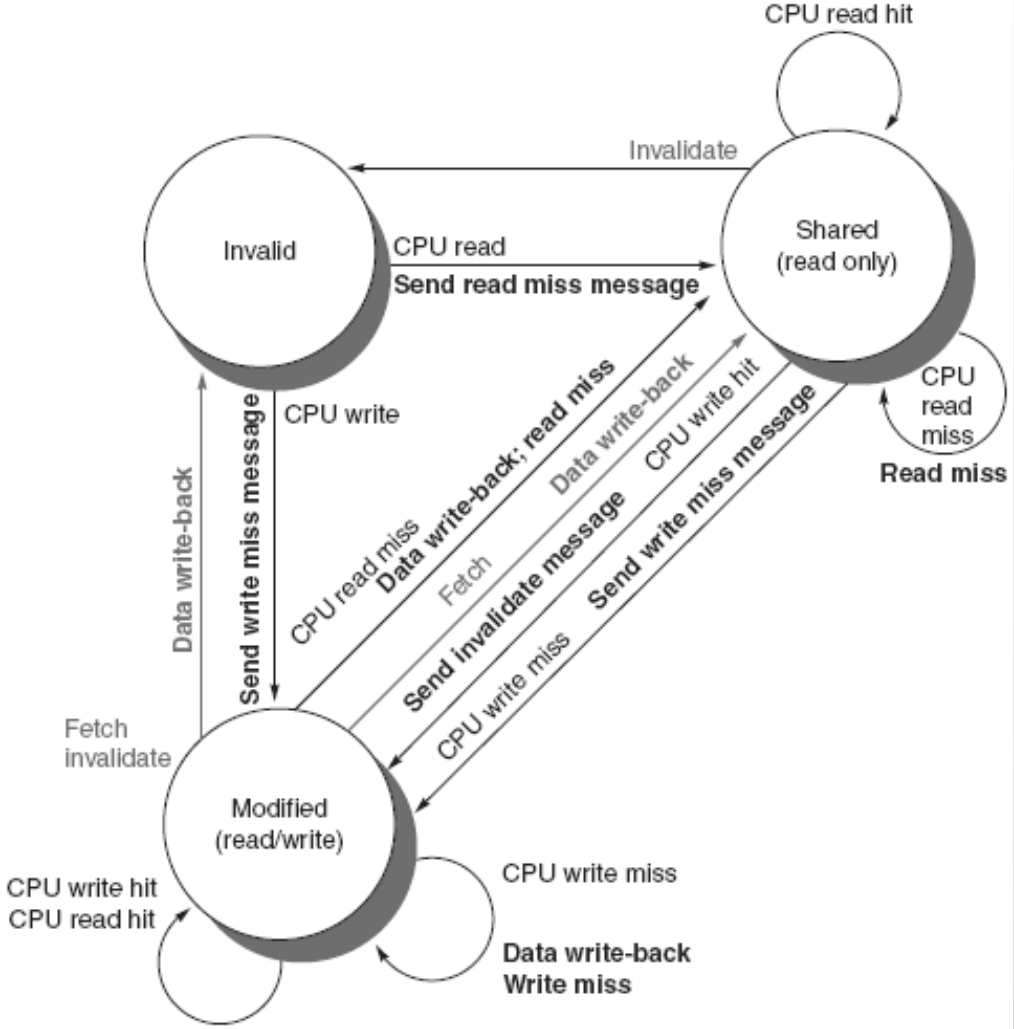
Directory-Based Coherence Protocols

- **States** are identical to those in the snooping case
- **Transitions** are similar, with explicit invalidate and write-back requests replacing the write misses that were broadcast on the bus
- An attempt to **write a shared cache block is treated as a miss**



Directory-Based Coherence Protocols

- The state transitions for an **individual cache** are caused by read misses, write misses, invalidates, and data fetch requests
- An individual cache also generates read miss, write miss, and invalidate messages that are sent to the **home directory**



Directory-Based Coherence Protocols

- Like the snooping protocol
  - any cache block must be in the exclusive state when it is written, and
  - any shared block must be up to date in memory
- In many multicore processors, the **outermost level in the processor cache is shared among the cores** (L3 in the Intel i7, the AMD Opteron, and the IBM Power7), and hardware at that level maintains **coherence among the private caches** of each core on the same chip, using either an **internal directory or snooping**
- In a directory-based protocol, the **directory implements the other half of the coherence protocol**

Directory-Based Coherence Protocols

- A message sent to a directory causes two different types of actions:
  - updating the directory state
  - sending additional messages to satisfy the request
- The states in the directory represent the three standard states for a block
- Unlike in a snooping scheme, however, the directory state indicates the state of **all the cached copies** of a memory block, rather than for a single cache block
- The memory block may be
  - **uncached** by any node,
  - **cached in multiple nodes** and readable (shared), or
  - **cached exclusively** and writable in exactly one node

Directory-Based Coherence Protocols

- For uncached block:
  - Read miss
    - Requesting node is sent the requested data and is made the only sharing node, block is now shared
  - Write miss
    - The requesting node is sent the requested data and becomes the sharing node, block is now exclusive
- For shared block:
  - Read miss
    - The requesting node is sent the requested data from memory, node is added to sharing set
  - Write miss
    - The requesting node is sent the value, all nodes in the sharing set are sent invalidate messages, sharing set only contains requesting node, block is now exclusive

Directory-Based Coherence Protocols

- For exclusive block:
  - Read miss
    - The owner is sent a data fetch message, block becomes shared, owner sends data to the directory, data written back to memory, sharers set contains old owner and requestor
  - Data write back
    - Block becomes uncached, sharer set is empty
  - Write miss
    - Message is sent to old owner to invalidate and send the value to the directory, requestor becomes new owner, block remains exclusive