CS60003: High Performance Computer Architecture

Multiprocessor Computers and Thread-Level Parallelism

IIT KHARAGPUR

Instructor:
Prof. Rajat Subhra Chakraborty
Professor
Dept. of Computer Science and Engineering
Indian Institute of Technology Kharagpur
Kharagpur, West Bengal, India 721302
E-mail: rschakraborty@cse.iitkgp.ac.in

Instruction-level Parallelism: Advantages and Challenges

- Advantage: led to spectacular increase of throughput (1990 – mid-2000s), transparent to programmer

- Challenges:
  - For wide-issue processors with high clock-speed, off-chip memory access latencies cannot be hidden effectively!
  - Complexity: issue-width limited to 4-8 instructions per clock cycle
  - High-clock speed is no longer attractive for throughput improvement!
    - Higher power dissipation
    - Deeper pipeline: plenty of wasted effort for incorrect branch prediction!
    - Transistors do not scale (become smaller) and faster over generations as effectively as before!
    - No substantial increase in clock-speed of processors in the last 15 years!

- We must look for other sources of parallelism!

Types of Parallelism

Parallelism Exploited by Software Applications:

- **Data-level Parallelism (DLP): operate on several data items at the same time, in the *same task***
- **Task-level Parallelism (TLP): create and operate *multiple tasks*, that can operate and execute independently**

Parallelism Exploited by Hardware:

- **Instruction-level Parallelism (ILP): exploits DLP at modest levels**
  - **Intelligent compilers, pipelining, speculative execution etc.**
- **Vector Processors, GPUs, Multimedia Instruction Sets: exploits DLP**
  - **The same instruction is executed on multiple processors, on multiple sets of data, in parallel**
- **Thread-level Parallelism: exploits either DLP or TLP**
  - **Tightly-coupled hardware, allows interaction between parallel cooperating *threads***
- **Request-level Parallelism: exploits either DLP or TLP**
  - **Executes largely decoupled tasks**

Types of Computers: *Flynn's Taxonomy* [1966]

- Single Instruction Stream, Single Data Stream (SISD): uniprocessor, i.e. one-core processor, may have ILP exploitation ability
- Single Instruction Stream, Multiple Data Streams (SIMD): the same instruction is executed on multiple processors on different data streams, exploits DLP, e.g. Vector Processors, GPUs, multimedia extensions
  - Each processor has separate data memory
  - Only one instruction memory
  - Single *control processor* fetches and dispatches instructions
- Multiple Instruction Stream, Single Data Stream (MISD): no current processor
- Multiple Instruction Stream, Multiple Data Streams (MIMD): usually exploits TLP, can also exploit DLP
  - More expensive than SIMD
  - Tightly-coupled: exploits *thread-level parallelism*
    - e.g. multi-core processors
  - Loosely-coupled: exploits request-level parallelism
    - e.g. clusters and warehouse-scale computers

Thread-level Parallelism

- ***Thread***: similar to a process, with a private *state* and private PC, but sharing the address space of a common *process*
  - Each thread carries out an independent task, but may share variables and data structures
  - Rapid switching possible between threads (e.g. different thread executed every clock cycle)
  - No full-fledged *context switch* required
  - More effective at handling pipeline and memory latencies

- Growing popularity of multi-core processors and Graphics Processing Units (GPUs) => more opportunity to exploit multi-threaded programs
- Growing popularity of cloud computing, which is inherently benefited by a multi-threaded computation model
- First commercial multi-core general-purpose processor: IBM Power 4 (2 cores) [2001]
  - Based on academic design at Stanford University [1998]

Thread-level Parallelism: Challenges

- Issue of independent thread dependent on intelligent compiler, or by explicit programming!
    - Often requires special software framework to exploit
    - Often requires substantial effort from programmer (with steep learning curve)
    - Not of much use if program inherently has only one thread!
- Extremely complex hardware design!
    - Remember, each core still has all the ILP techniques in-built!
    - Each thread has its private set of renaming tables and PC!
    - Instructions from multiple threads may execute and commit simultaneously!
    - How do you maintain a coherent view of shared cache/memory?
    - How do you manage contention for shared buses (e.g. between processor and cache), for a large number of cores?
- Prices are inherently high
    - Each core is a complete processor: 4 cores => 4X price?
    - Overkill for low-cost industrial applications!

Thread-level Parallelism: Types

- Multithreading within single-core
    - Most common, has existed for several decades
    - Does not require a multi-core processor
    - Acceptable for low-cost processors targeting industrial automation
- Multithreading within multi-core processor, with 1 thread per core
- Multithreading within multi-core processor, with multiple threads per core
    - This is the most common option in current client/server products
    - Both general-purpose processors (client and server class), as well as GPUs

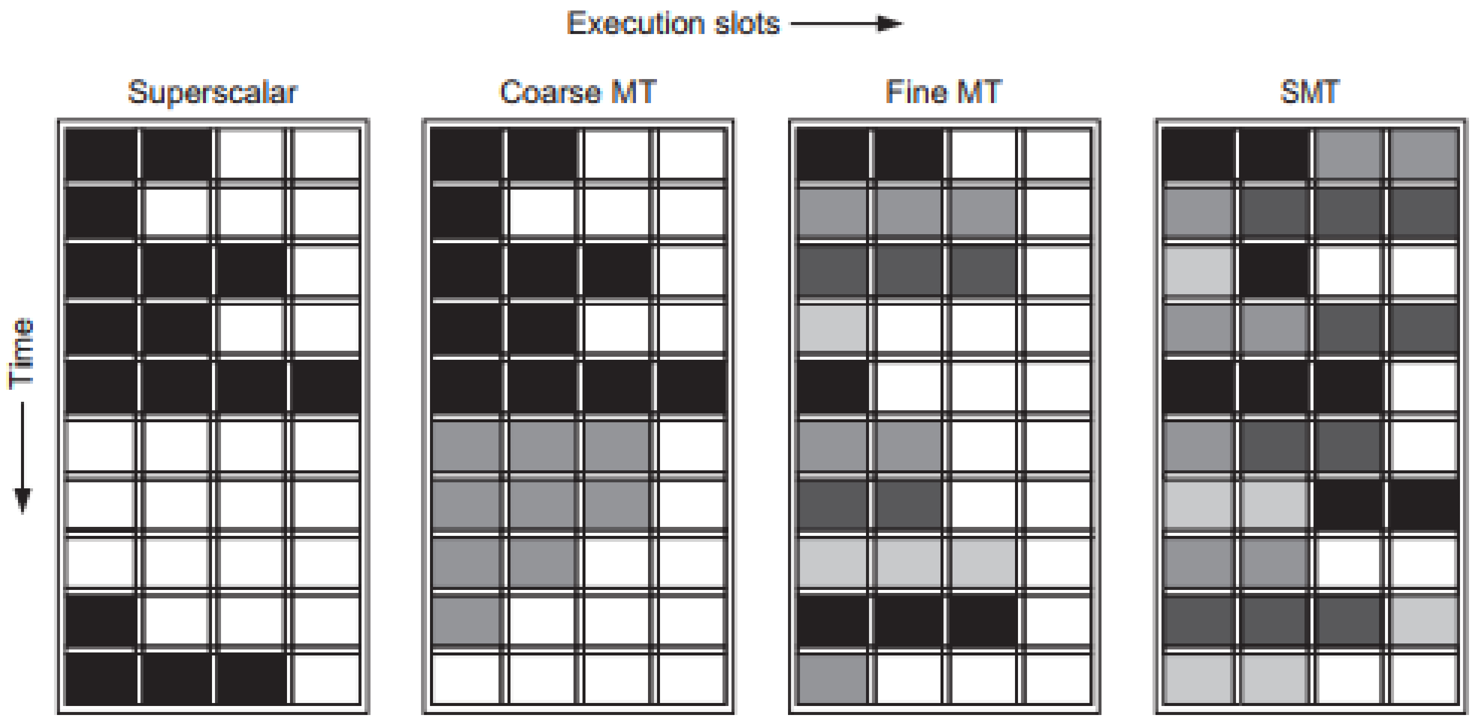
Types of Multithreading

- Fine-grained (a.k.a. “Fine”) MT
  - Mandatory switch of thread per clock cycle
  - Often round-robin scheduling
  - Skip-over stalled threads
  - All instructions executed belong to same thread in a given clock cycle
  - e.g.: Sun SPARC (T1 through T5)
  - e.g.: NVIDIA GPUs

- Simultaneous MT (SMT)
  - Fine MT + Dynamic Scheduling
  - Mixture of instructions from different threads executed in a given clock cycle!
  - Most common option
  - e.g. Intel Core i7 (2 threads per core)
  - e.g. IBM Power 7 (4 threads per core)

H&P CA:A QA (6th. Ed.)

Types of Multithreading (contd.)



Note: **all instructions issued in a given clock cycle belong to the *same thread*!**

Figure 3.31 How four different approaches use the functional unit execution slots of a superscalar processor. The horizontal dimension represents the instruction execution capability in each clock cycle. The vertical dimension represents a sequence of clock cycles. An empty (white) box indicates that the corresponding execution slot is unused in that clock cycle. The shades of gray and black correspond to four different threads in the multithreading processors. Black is also used to indicate the occupied issue slots in the case of the superscalar without multithreading support. The Sun T1 and T2 (aka Niagara) processors are fine-grained, multithreaded processors, while the Intel Core i7 and IBM Power7 processors use SMT. The T2 has 8 threads, the Power7 has 4, and the Intel i7 has 2. In all existing SMTs, instructions issue from only one thread at a time. The difference in SMT is that the subsequent decision to execute an instruction is decoupled and could execute the operations coming from several different instructions in the same clock cycle.

H&P CA:A QA (6th. Ed.)

Computers with Multiple Processors: Common Types

- **Single multi-core processor: most common in personal computing**
  - **Almost all modern desktops/laptops/tabs/smartphones etc.**
- **Generic Standalone Servers:**
  - **4 to 16 *separate* microprocessor chips (“sockets” on “boards”), *each chip* being multi-core**
  - **4 to 256 *total* cores**
- **Supercomputer-scale Clusters**
  - **100 to 1000 server “nodes”**
  - **Usually combination of CPUs and GPUs**
  - **Common in Scientific Computing (e.g. *PARAM Shakti* at IIT KGP with 442 [CPU + GPU] “nodes”), 1.66 petaFLOPs rated performance**
  - **Less strongly coupled than generic servers**
  - **Requires substantial software effort to leverage**
- **Ultrascale Computing (aka “Warehouse-scale Computers”)**
  - **Extremely large clusters (> 10,000 servers)**
  - **Requires substantial software effort to leverage**
  - **Amazon, Facebook, Google, Microsoft, etc..**

Symmetric (shared-memory) Multiprocessors (SMP)

- **Multiple tightly-coupled cores on a single chip**
  - **e.g.: Intel Core i7, Intel Xeon**
- **No. of cores: usually <= 32**
  - **GPUs are different, thousands of cores!!**
- **Each core: one/more levels of private cache**
- **Cores (Symmetrically) Share Last Level Cache (LLC)**
- **Cores (Symmetrically) Share Main memory**
  - **Uniform Memory Access (UMA)**
- **Note: very rarely, the LLC is *distributed* across the cores!**
  - **e.g. IBM Power 7, with distributed L3 cache**
  - **Non-uniform Cache Access (NUCA)**

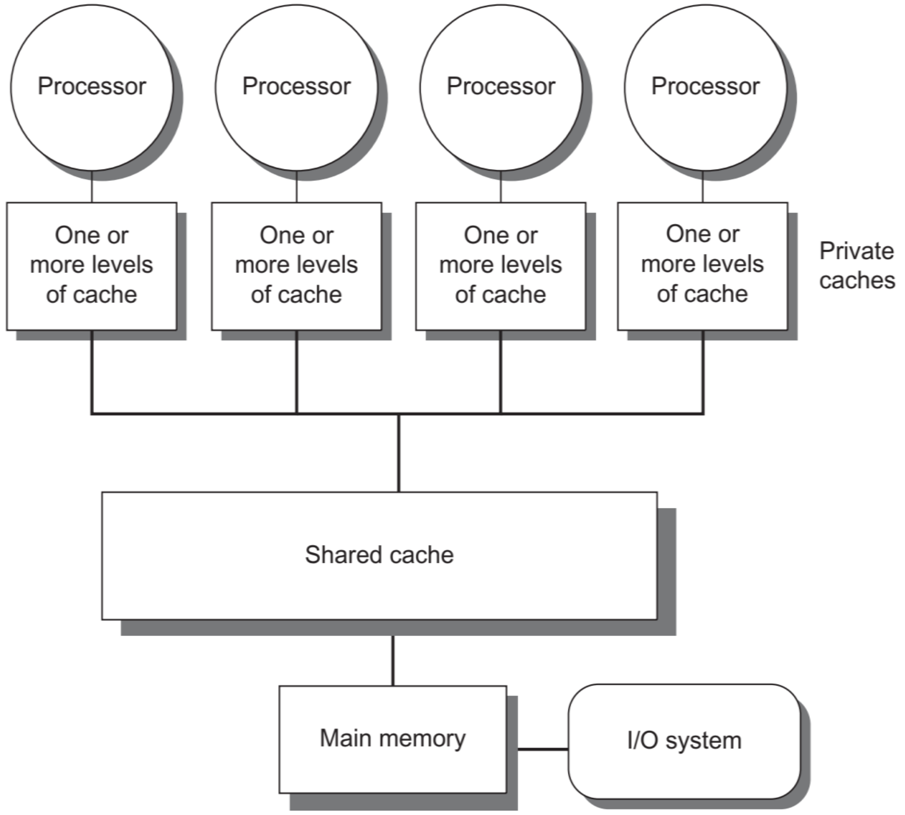

Processor
Processor
Processor
Processor
One or more levels of cache
One or more levels of cache
One or more levels of cache
One or more levels of cache
Private caches
Shared cache
Main memory
I/O system

Figure 5.1 Basic structure of a centralized shared-memory multiprocessor based on a multicore chip. Multiple processor-cache subsystems share the same physical memory, typically with one level of shared cache on the multicore, and one or more levels of private per-core cache. The key architectural property is the uniform access time to all of the memory from all of the processors. In a multichip design, an interconnection network links the processors and the memory, which may be one or more banks. In a single-chip multicore, the interconnection network is simply the memory bus.

H&P CA:A QA (6th. Ed.)

Distributed (shared-memory) Multiprocessors (DSM)

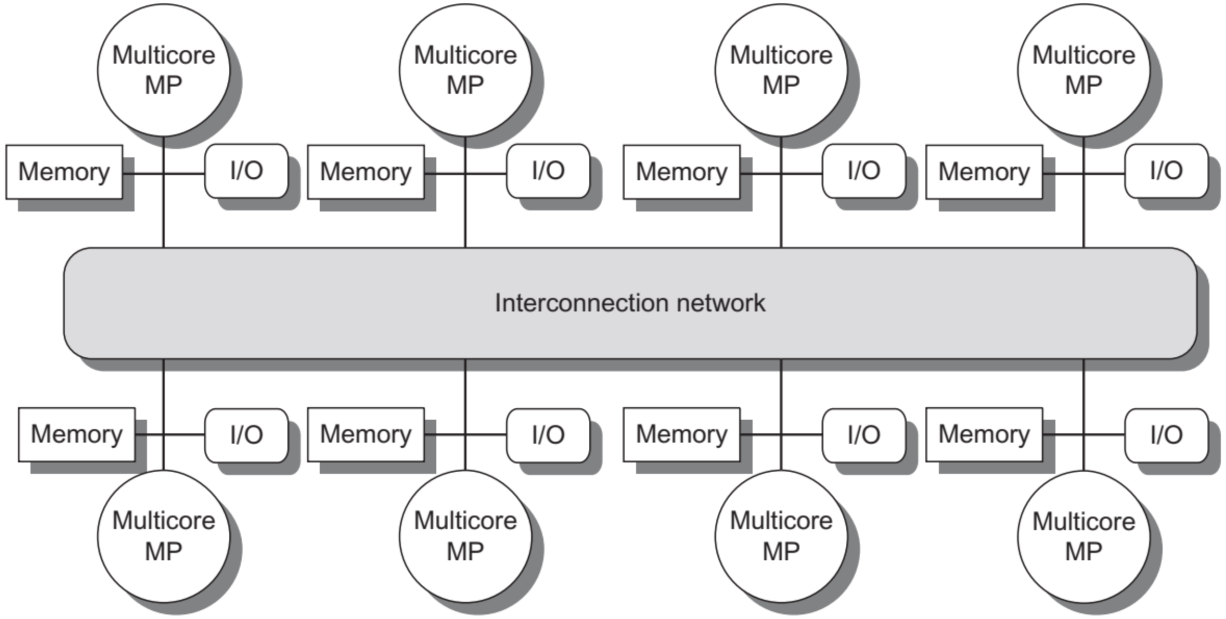


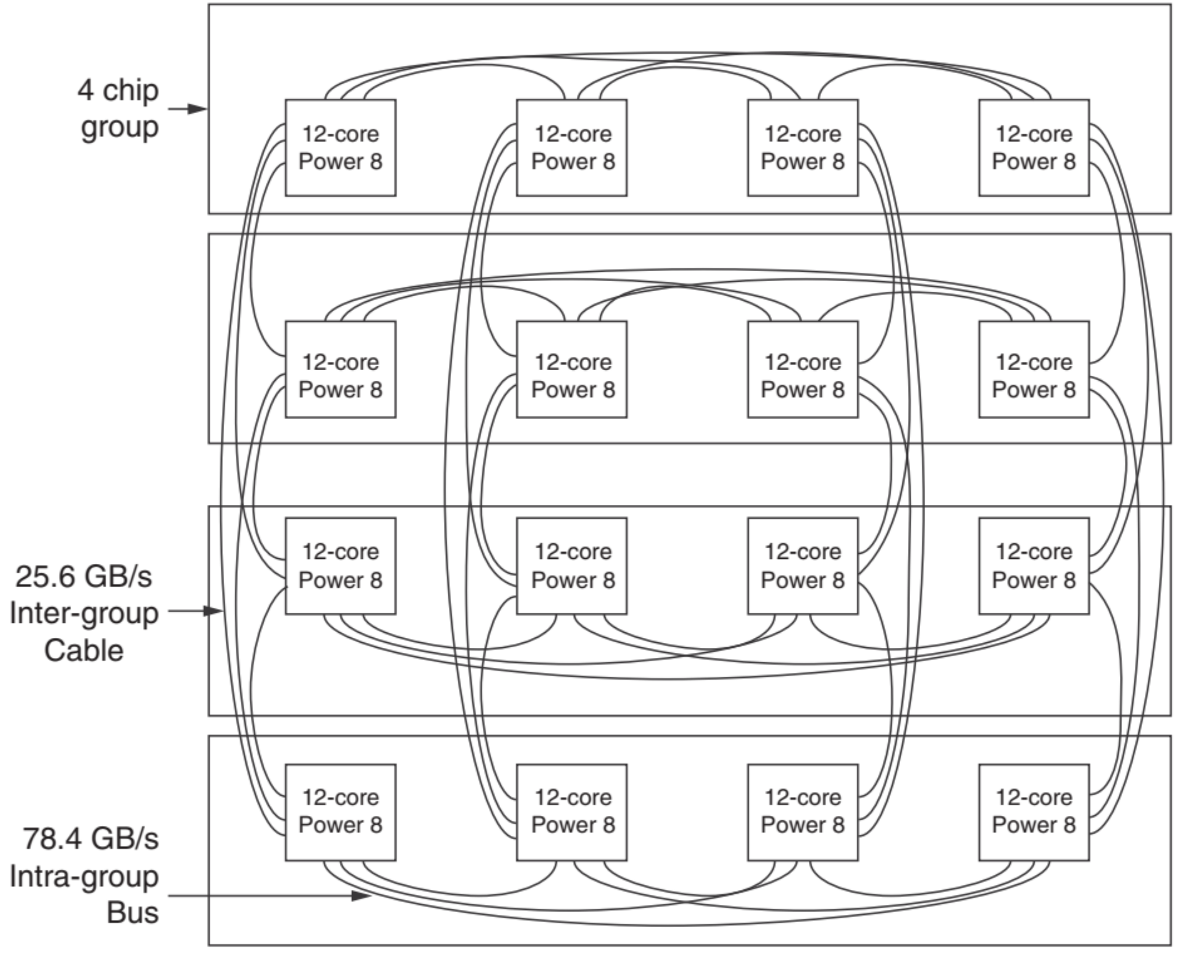
Figure 5.2 **The basic architecture of a distributed-memory multiprocessor in 2017 typically consists of a multicore multiprocessor chip with memory and possibly I/O attached and an interface to an interconnection network that connects all the nodes.** Each processor core shares the entire memory, although the access time to the local memory attached to the core's chip will be much faster than the access time to remote memories.

Note: **"shared memory" => address space is shared.**
In both SMP/DSM: Threads communicated through shared address address space => a thread can access any memory location, provide it has the proper permissions!

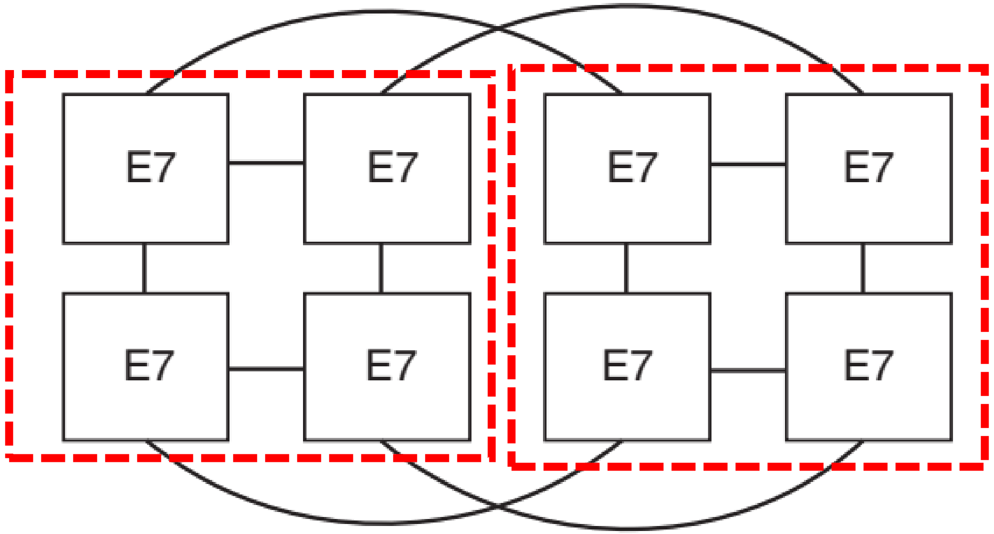
- **Servers usually employ DSMs**
  - **e.g.: Intel Xeon based servers**
- **Multiple (multi-core) processors**
  - **Multiple "sockets" on a multiprocessor board**
  - **Multiple such boards can be connected!**
- **Distributed Memory**
  - **(Asymmetrically) Shared DRAM modules**
  - **Local DRAM and Remote DRAM**
  - **Local DRAM: connected directly through *backside bus*, less access time**
  - **Remote DRAM: More access time!**
  - **Non-uniform Memory Access (NUMA)**

H&P CA:A QA (6th. Ed.)

Typical NUMA DSM Configurations: Power8 and Xeon E7



(A) Power8 system with up to 16 chips.



(B) Xeon E7 system showing with up to 8 chips.

4-chip group

- **4 Xeon E7 processors connected using an advanced interconnect technology called *QuickPath Interconnect* (QPI)**
- **Multiple such groups can be connected!**
- **Every processor is connected to another through maximum two hops!**

H&P CA:A QA (6th. Ed.)

Challenges of Parallel Processing

- **Lack of sufficient parallelism to exploit**
  - ***Amdahl's Law!***
  - **Soln.: algorithmic techniques, intelligent compliers to exploit parallelism**
- **Long-latency of remote memory access**
  - **For NUMA, DRAM can be far away, and only accessible through host processor**
  - **Many processors trying to access shared bus => contention related latency**
  - **Solns.:**
    - **Basic idea: reduce need to access remote data**
    - **Hardware technique: have many levels of local cache, prefetch data**
    - **Hardware technique: connect DRAM locally to microprocessor through "backside bus"**
    - **Software techniques: reorganize data (so that most accesses are local)**
    - **Hybrid: multi-threaded programming**

Multiprocessor *Cache Coherence Problem*

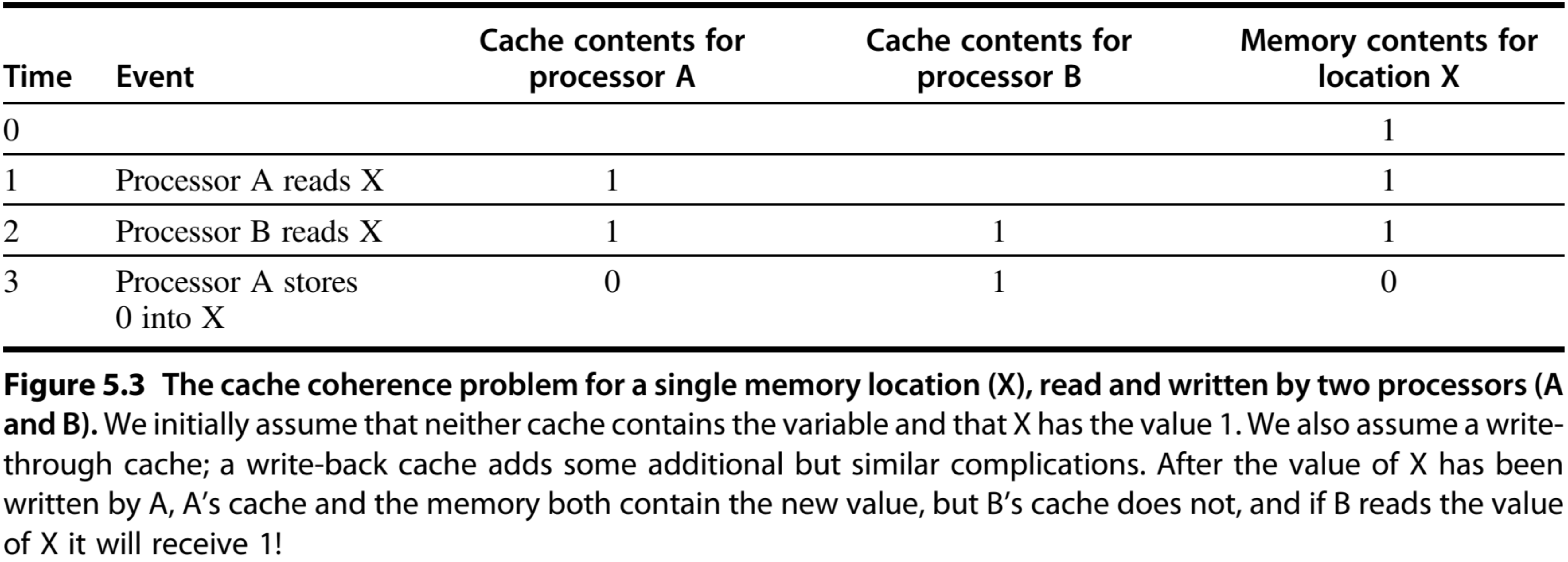
- ***Private data***: **data used by a single core/processor**
  - **If only one core has access to some data => Less complications!**

- ***Shared data***: **data used by multiple cores/processors**
  - **Allows communication between the cores/processors**
  - **Multiple cores/processors read and write the shared data**
  - ***Global state***: **data in shared main memory**
  - ***Local state***: **data in private caches**
  - **Observation: shared data may be replicated in multiple caches!**
  - **Observation: the view of memory is *through* individual caches**

- **Observation: each core/processor sees *different values for the same memory location***

- **Cache Coherence Problem: how to maintain a *coherent view* of shared memory?**

Multiprocessor Cache Coherence Problem: Example



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

Figure 5.3 The cache coherence problem for a single memory location (X), read and written by two processors (A and B). We initially assume that neither cache contains the variable and that X has the value 1. We also assume a write-through cache; a write-back cache adds some additional but similar complications. 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!

H&P CA:A QA (6th. Ed.)

Cache Coherence: Basic Definition

- **(Informally) A memory system is *coherent* if any read of a data item returns the *most recently written value* of that data item**
  - **Data is returned from a shared memory reference *in the same way* that would have happened in a system *without any cache*!**

- **Two necessary properties of a shared memory system: *coherence* + *consistency***
  - **Complimentary aspects**
- **Coherence: defines *what value* can be returned by a read**
  - **Ensures that local cached values are used and modified *transparently* to others**
- **Consistency: determines *when* a written value would be returned by a read**
  - **More complicated topic!**

Coherence: Property-1

- Consider time interval $[t_1, t_2]$
- Processor/core “P” writes to location “X” at time = t_1
- No other processor/core writes to location “X” in interval $[t_1, t_2]$
- Processor/core “P” reads from location “X” at time = t_2
- Then, “P” receives the value written by it at time = t_1

- Comment: this is expected behavior, even in a single-core uniprocessor!
  - Load after a Store is executed in-order
  - Load is guaranteed to return latest value (remember “Write Buffer” complication issues?)

Coherence: Property-2

- Consider time interval [t_1, t_2], *sufficiently large*
- Processor/core "P_1" writes to location "X" at time = t_1
- No other processor/core writes to location "X" in interval [t_1, t_2]
- Processor/core "P_2" ($\neq P_1$) reads from location "X" at time = t_2
- Then, "P_2" receives the value written by "P_1" at time = t_1

- Comment: this is the defining property of a "coherent" memory system!
- Comment: if the interval is small, then maintaining this property requires *consistency*!

Coherence: Property-3

- Consider time interval [t_1, t_2], sufficiently large
- Processor “P_1” writes to location “X” at time = t_1
- Processor “P_2” ($\neq P_1$) writes to location “X” at time = t_2
- Then, any processor “P_i”:
  - reads the value written by “P_1” in the interval (t_1, t_2]
  - reads the value written by “P_2” for time > t_2

- Comment: this is called “write serialization”!

Assumptions about Operations (to ensure coherent memory system)

- **A write operation (“store”) is not considered complete, until all processors/cores have seen the effect of that write**
  - **Next store operation in code must wait till assurance is obtained about this**

- **(Seen earlier) A processor/core does not re-order a store operation with respect to any other load/store**
  - **Writes (stores) are in strict program order, w.r.t. both loads and stores!**
  - **A collection of (only) loads can be re-ordered arbitrarily!**

- **This implies:**
  - **Suppose a processor “P” writes to two locations: “A” , “B” (in that order)**
  - **Then, a processor that sees updated value at “B” must also see updated value at “A”**

Cache Coherence: Basic Schemes

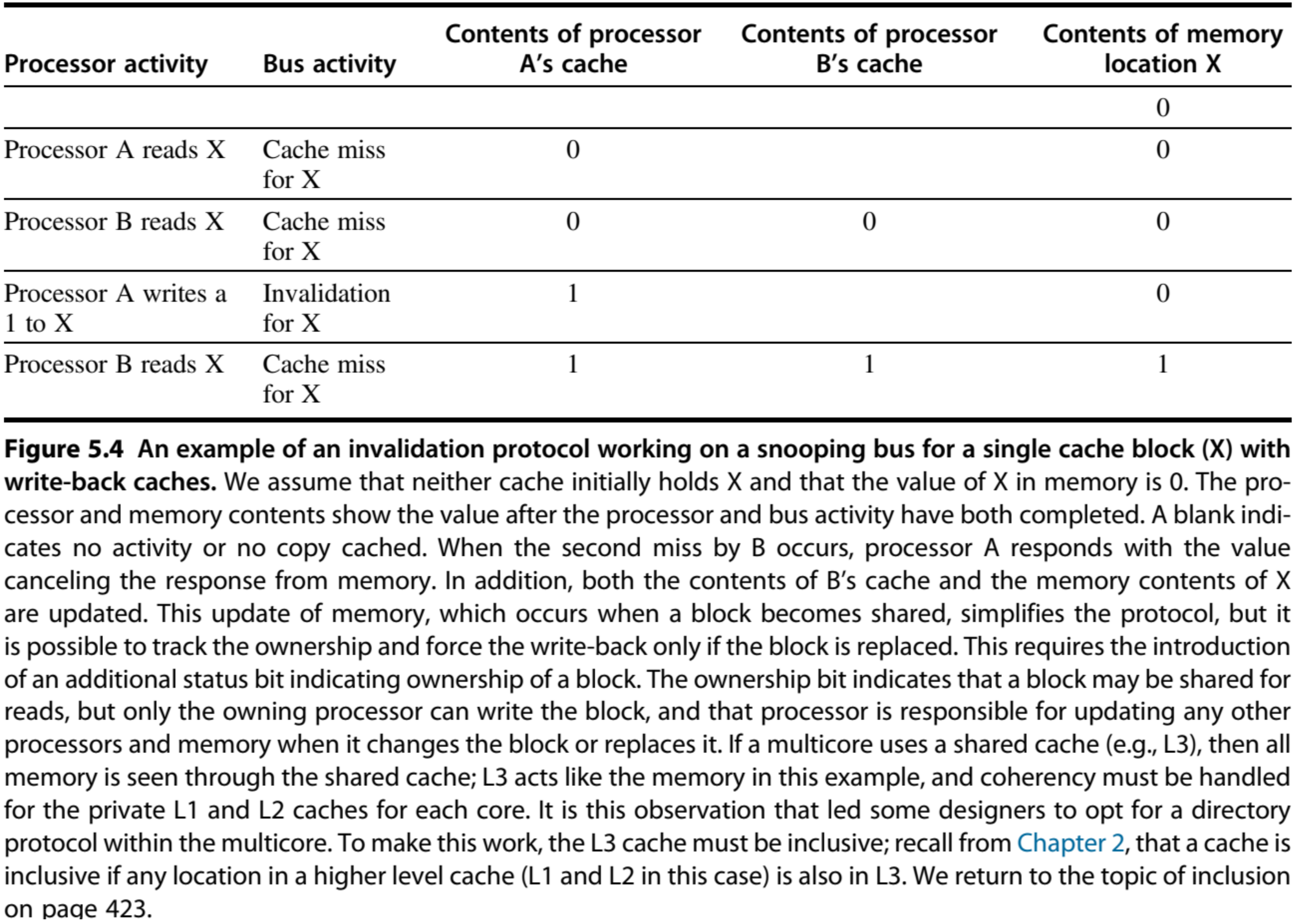
- Coherence allows: Migration + Replication
  - Migration: allows data to be moved to local cache and be used there *transparently*
  - Replication: allows maintaining copies of shared data in local cache, so that read accesses can be satisfied locally
- Cache coherence schemes are implemented in hardware inside microprocessors
- Cache coherence schemes: Snooping and Directory-based
- Snooping:
  - All blocks in cache associated with a *block state* (“Modified”/”Shared”/”Invalid”)
  - All cores monitor (*snoop)* on a common broadcast medium
  - Changes block status depending on signals received from broadcast medium
  - Common in multi-core processors, but increasingly difficult to scale!
- Directory-based:
  - Maintain the sharing status of a block of physical memory at a central location (“Directory”)
  - Relatively more scalable, must for multi-processor systems like large servers
- Modern schemes on multi-core processors: *combination* of the two schemes

Snooping Protocol: Variants

- Two basic schemes: (a) Write Invalidate ; (b) Write Update / Write Broadcast
- Write Invalidate
  - A processor acquires exclusive access to a data item before writing to it
  - All other cached copies of the data item are invalidated
  - Ensures no readable/writable copy of an item exists when write occurs
  - Requires broadcast of {block address, block state} in local cache (only metadata)
  - Relatively low bandwidth requirement
  - Widely used in modern microprocessors

- Write Update / Write Broadcast
  - Updates all cached copies of a data item when the item is written
  - Requires broadcast of {block data, block address, block state}
  - Requires tremendously large bandwidth
  - Not used any modern microprocessor

An Invalidate based Protocol: Example



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

Figure 5.4 An example of an invalidation protocol working on a snooping bus for a single cache block (X) with write-back caches. We assume that neither cache initially holds X and that the value of X in memory is 0. The processor and memory contents show the value after the processor and bus activity have both completed. A blank indicates no activity or no copy cached. 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. This update of memory, which occurs when a block becomes shared, simplifies the protocol, but it is possible to track the ownership and force the write-back only if the block is replaced. This requires the introduction of an additional status bit indicating ownership of a block. The ownership bit indicates that a block may be shared for reads, but only the owning processor can write the block, and that processor is responsible for updating any other processors and memory when it changes the block or replaces it. If a multicore uses a shared cache (e.g., L3), then all memory is seen through the shared cache; L3 acts like the memory in this example, and coherency must be handled for the private L1 and L2 caches for each core. It is this observation that led some designers to opt for a directory protocol within the multicore. To make this work, the L3 cache must be inclusive; recall from Chapter 2, that a cache is inclusive if any location in a higher level cache (L1 and L2 in this case) is also in L3. We return to the topic of inclusion on page 423.

Basic Snooping Protocol: “Modified/Shared/Invalid” (MSI)

- **Basic idea:** maintain a finite state machine (FSM) controller in each core

- Each core continuously snoops on the broadcast medium (bus)

- The controller responds to requests from the core circuitry and from the bus
  - Considers {source of request, block address, block state} for each bus transaction
  - Takes appropriate action (e.g. initiate memory access or perform cache invalidate) as a function of the above 3-tuple

- Logically, we can think that *each cache block* has a dedicated controller
  - Actually, there is only one controller per core, which serves multiple blocks in the local caches in an interleaved manner!

Basic Snooping Protocol: “Modified/Shared/Invalid” (MSI)

- Three states of a cache block: Modified, Shared, Invalid
- Invalid: usual meaning
- Shared: the block in the private cache is *possibly shared* with other cores
  - A block in *shared* state is read-only
  - It does not imply that *actually any other core has a copy of it*!
- Modified: the block in the private cache *has been modified*
  - “*Modified*” state of a cache block implies “*Exclusive*” right to a block for the host core
  - This host core with the *Exclusive* right is called the *owner* of the block
- Observation: any valid cache block is in “Shared” state in one or more private caches, or in “Modified” state in exactly one private cache!

MSI Protocol: Example Actions in a Write-Back Cache

- <u>Situation-1:</u> Copy of block B_j in local cache of a core C_i is in ***Shared*** state, Write Hit occurs
  - C_i acquires bus access, possibly after winning contention from another core
  - C_i places address of B_j and *Invalidate* on bus (i.e. asks other cores having local copies of B_j to invalidate them)
  - All other cores (including some others who are also waiting to write to the same shared block B_j) snooping the bus, having local copies of B_j,invalidate them
  - C_i updates the state of the local copy of B_j to ***Modified*** (i.e. becomes *owner* of B_j)
  - C_i modifies the local copy of B_j

- <u>Situation-2:</u> Copy of block B_j in local cache of a core C_i has ***Modified*** state, Write Miss occurs
  - C_i addresses the conflict miss
  - C_i writes back the modified (dirty) local copy of block B_j to LLC
  - C_i keeps state of local copy of block B_j unchanged at ***Modified***

MSI Protocol: Example Actions in a Write-Back Cache (contd.)

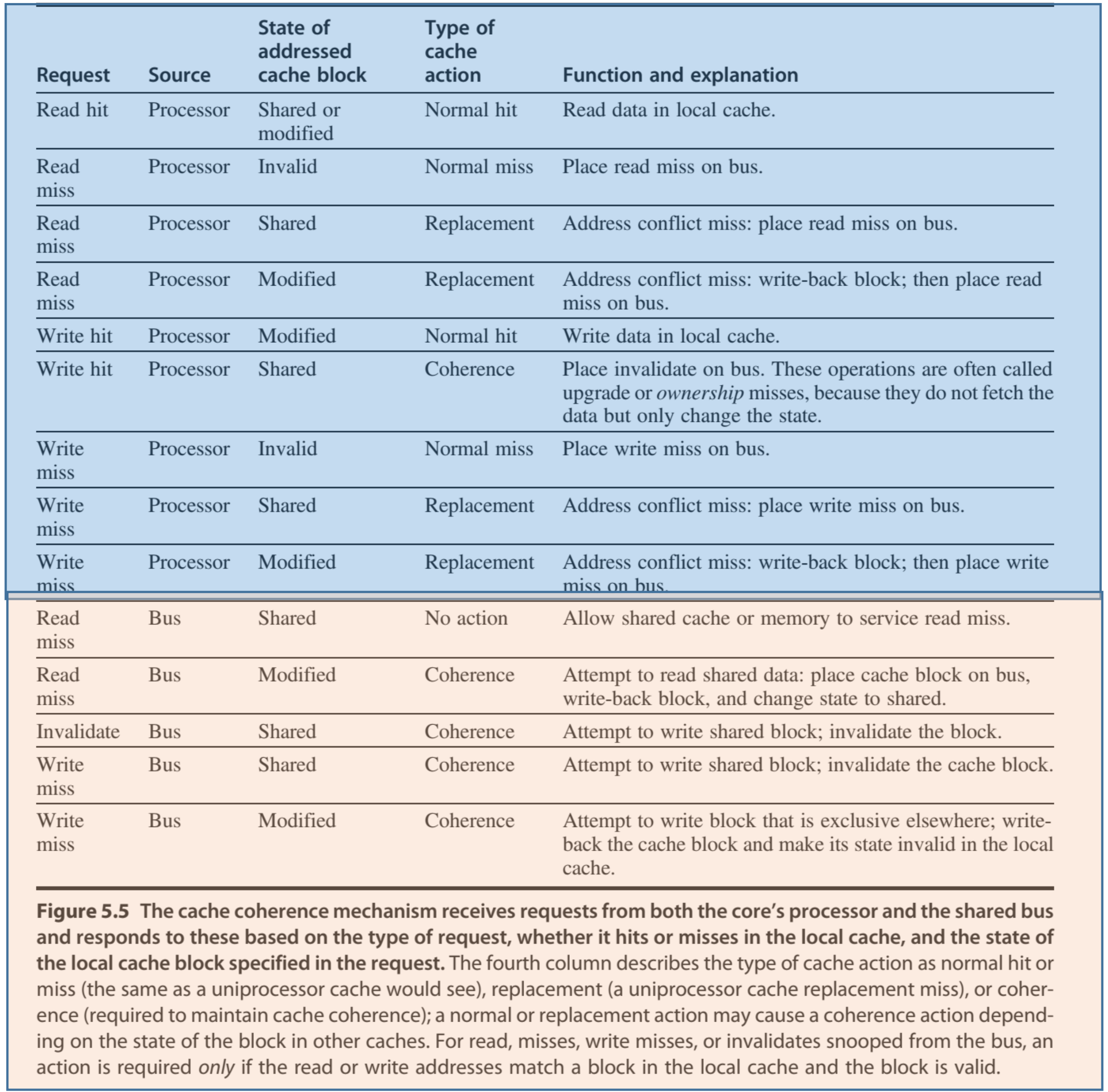
- <u>Situation-3:</u> Block B_j in local cache of a core C_k has ***Shared*** state, receives Write Miss from (a different) core C_i over bus
  - C_k invalidates local copy of B_j
- <u>Situation-4:</u> Copy of block B_j in local cache of a core C_i has ***Shared*** state, Write Miss occurs
  - C_i addresses the conflict miss
  - ~~C_i writes back the modified (dirty) local copy of block B_j to LLC~~ (no need!!)
  - C_i places Write Miss on bus

- <u>Situation-5:</u> Block B_j in local cache of a core C_k has ***Modified*** state, receives Read Miss from (a different) core C_i over bus
  - C_i places block (and its address) on bus for the core wanting it to collect
  - C_i writes back the modified block
  - C_i changes state of local block to *Shared*
  - C_i forces the memory access request of core C_k to abort!

Basic Snooping Protocol: “Modified/Shared/Invalid” (MSI)

Received from processor

Received from bus



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, because 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.
Read miss	Bus	Shared	No action	Allow shared cache or memory to service read miss.
Read miss	Bus	Modified	Coherence	Attempt to read shared data: place cache block on bus, write-back block, 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.

Figure 5.5 The cache coherence mechanism receives requests from both the core's processor and the shared bus and responds to these based on the type of request, whether it hits or misses in the local cache, and the state of the local cache block specified in the request. The fourth column describes the type of cache action as normal hit or miss (the same as a uniprocessor cache would see), replacement (a uniprocessor cache replacement miss), or coherence (required to maintain cache coherence); a normal or replacement action may cause a coherence action depending on the state of the block in other caches. For read, misses, write misses, or invalidates snooped from the bus, an action is required *only* if the read or write addresses match a block in the local cache and the block is valid.

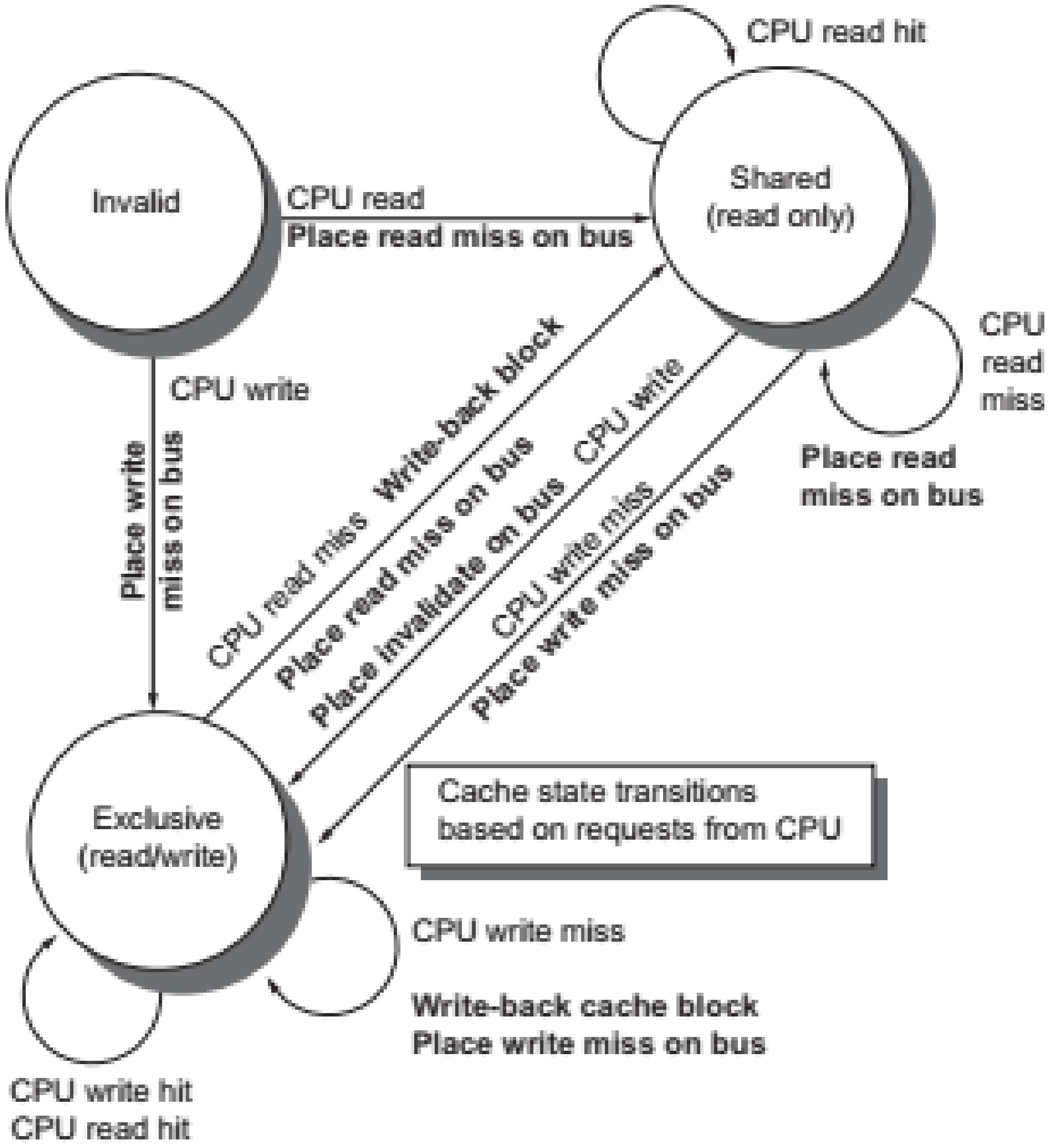
Note:
“Modified” state of a cache block implies “Exclusive” right to the block

H&P CA:A QA (6th. Ed.)

MSI FSM State Diagram

Note:

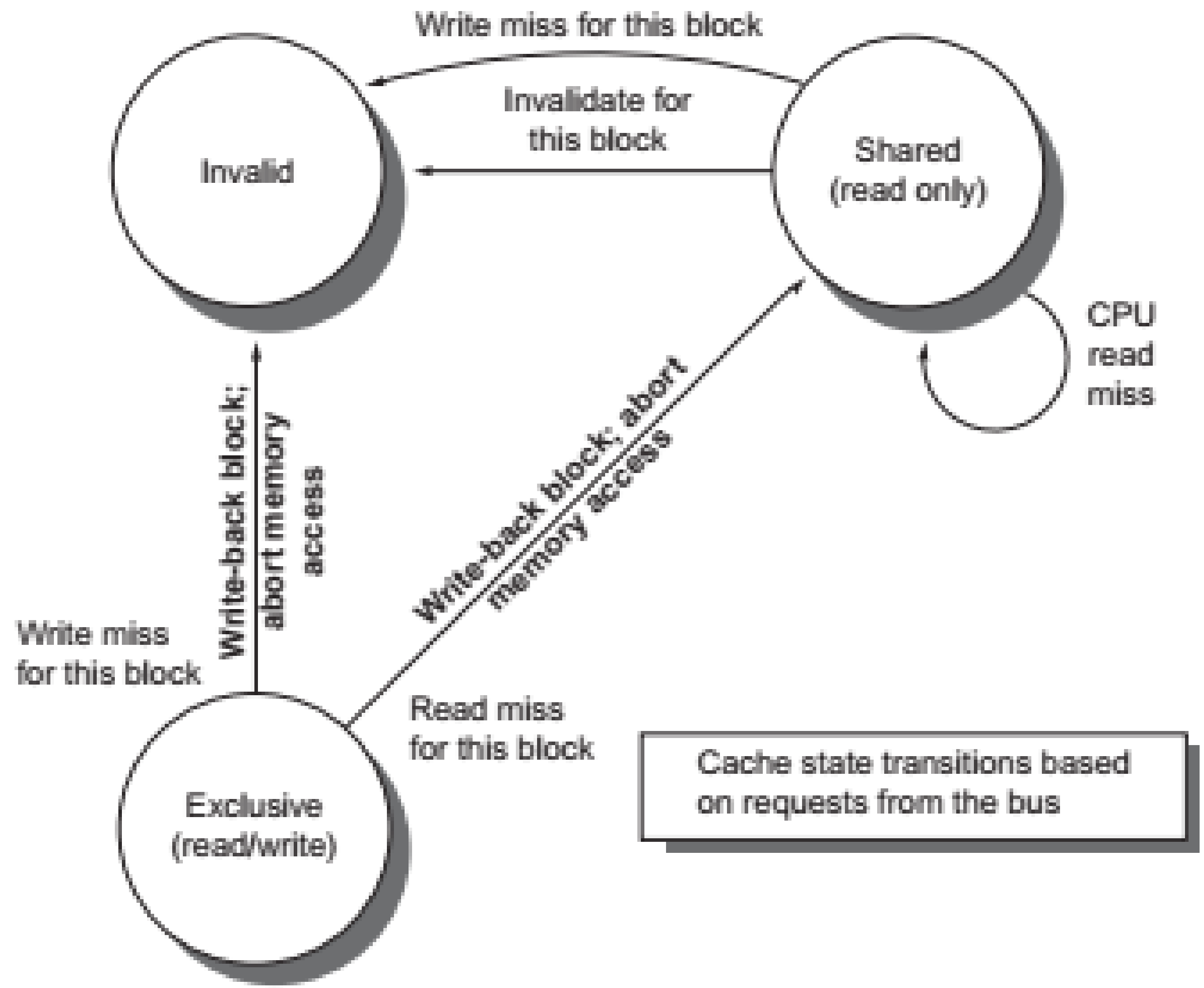
Here "Exclusive: means "Modified"!



H&P CA:A QA (6th. Ed.)

MSI FSM State Diagram

Note: Here “Exclusive: means “Modified”!



Write miss for this block
Invalidate for this block
Invalid
Shared (read only)
CPU read miss
Write-back block; abort memory access
Write-back block; abort memory access
Write miss for this block
Read miss for this block
Exclusive (read/write)
Cache state transitions based on requests from the bus

H&P CA:A QA (6th. Ed.)

MSI Protocol Optimization: MESI

- MSI Protocol Shortcoming:
  - "Shared" state does not necessarily mean that the any other core has a copy of this block!
  - Still, Write on a local block in "Shared" state generates an Invalidate on bus!
  - Hence, we can try to optimize the following case: only one core (the owner) currently has the block (in *Shared* state), and wants to Write on the block!

- Optimization: four states of a cache block: Modified, Exclusive, Shared, Invalid
  - *MESI* Protocol
  - This Exclusive state is different from the previous *Exclusive* state!
  - Exclusive: the block is held by *only this* (owner) cache, *and* is not dirty
  - Local block in ***Exclusive*** state => can be written without trying to acquire bus access, or generating an Invalidate on bus!
  - After the write, state of block changes to ***Modified***
  - A version of the *MESI* Protocol is used in the Intel Core i7

Coherence Misses

- Comparable to Cache Misses
- Two types: True Sharing Miss and False Sharing Miss

- Situation: block B_k is initially in *Shared* state between two different cores: C_i and C_m
- True Sharing Miss:
  - Core C_i writes to word W_j in block B_k, becomes owner of B_k
    - All other copies of B_k in other cores are invalidated
  - When core C_m wants to read the same word W_j of block B_k, has Read Miss!

- False Sharing Miss:
  - Core C_i writes to word W_j in block B_k, becomes owner of B_k
    - All other copies of B_k in other cores are invalidated
  - When core C_m wants to read a different word W_n in block B_k, has Read Miss because of Invalid block!

Directory-based Coherence Protocols

- Snoopy scheme disadvantage:
  - *Every cache miss* requires communication with *all* cores!
    - Where is the needed block?
  - Especially high amount of communication when a *Shared* block is written!
- Especially challenging if there are many cores/processors!
  - Common bus cannot satisfy the high bandwidth demand!
  - Difficult to support in modern multi-core processors with many cores
  - Impossible to support in modern multi-processor systems
- Soln.: Directory-based Coherence Protocol!
  - A *directory* preserves information about *every local memory block* that may be cached
  - More scalable solution, since only local directory, or only one remote directory has to be consulted to get information about block (avoid broadcast)!
- We consider two cases:
  - Case-(1): Directory-based Coherence Protocol for Multi-core Processors
  - Case-(2): Directory-based Coherence Protocol for Multi-processor Systems

Directory-based Coherence Protocol for Multi-core Processors

- LLC (usually L3) is “inclusive”
  - Guaranteed to contain {blocks in L1} ∪ {blocks in L2} for all cores!
- The directory is maintained in LLC, one entry *for every LLC block*
  - No need to store tag corresponding to block starting address!
- For each LLC block, store the following in directory:
  - A bit vector, with # of bits = # of cores
  - Bit in bit vector denotes whether the L2 cache of the corresponding core has a private copy of the block
- On Write to a *Shared* block, consult directory in LLC, and send invalidates to *only those* cores each of which has a local copy (saves communication bandwidth)
- Size of LLC directory: O(*pq*), *p*: # of LLC blocks, *q*: # of cores
- Intel Core i7 uses this protocol, built on top of a snoopy protocol
  - LLC (L3) filters snoop requests from L2 caches, based on whether L2 has the relevant block or not, and only allows the relevant cores to snoop

Directory-based Coherence Protocol for Multi-processor Systems

- Single directory based solution not scalable to DSM systems
  - Requires a "Distributed Directory" protocol!
- Physical memory is *statically distributed*
  - e.g. higher-order address bits denote node #
  - Easy to find out the DRAM and the corresponding directory for any address!
- A (local) directory is attached with every "node" (multi-core processor)
  - Local directory may be hosted in LLC of node, or as a separate DRAM in large systems (as shown in figure in next slide)
  - Every local directory contains information about the blocks in corresponding local memory
  - LLC in each multi-core processor may still have a directory addressing the cores!
- Node-wise directory size: $O(mn)$
- *Total* size of distributed directories: $O(mn^2)$

m: # of blocks in each local DRAM (assuming each DRAM is of same size)

n: # of nodes

Directory-based Coherence Protocol: Details

- Three states of a cache block: **Shared, Uncached, Modified**
  - **Shared:** one or more nodes have the block cached, and the value in memory (as well as in all the caches) is up to date
  - **Uncached:** no node has a copy of the block
  - **Modified:** exactly one node (the "owner") has a copy of the memory block, it has written the block, and **the memory copy is out of date**

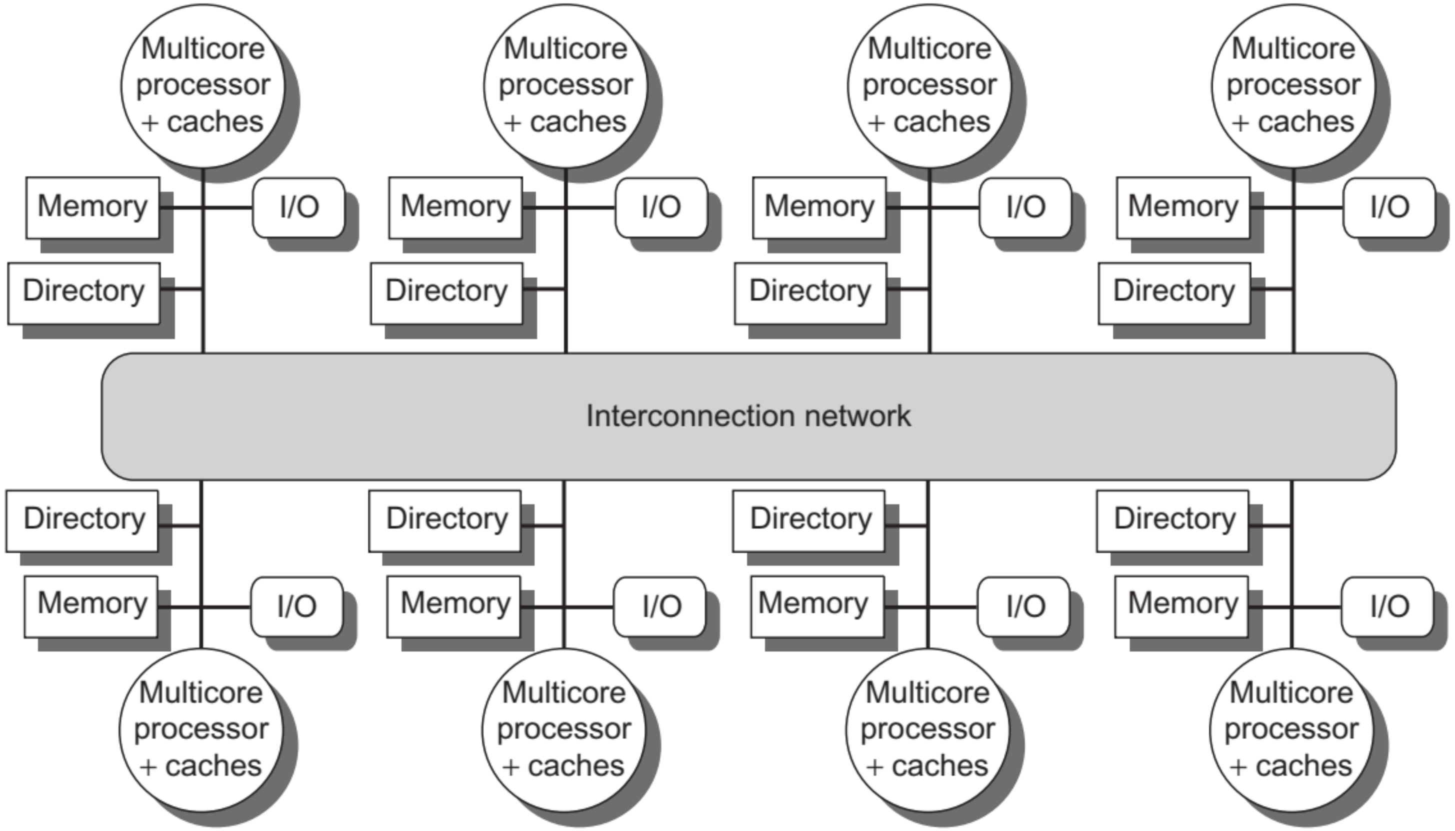


Figure 5.18 A directory is added to each node to implement cache coherence in a distributed-memory multiprocessor. In this case, a node is shown as a single multicore chip, and the directory information for the associated memory may reside either on or off the multicore. Each directory is responsible for tracking the caches that share the memory addresses of the portion of memory in the node. The coherence mechanism will handle both the maintenance of the directory information and any coherence actions needed within the multicore node.

Directory-based Coherence Protocol: Details (contd.)

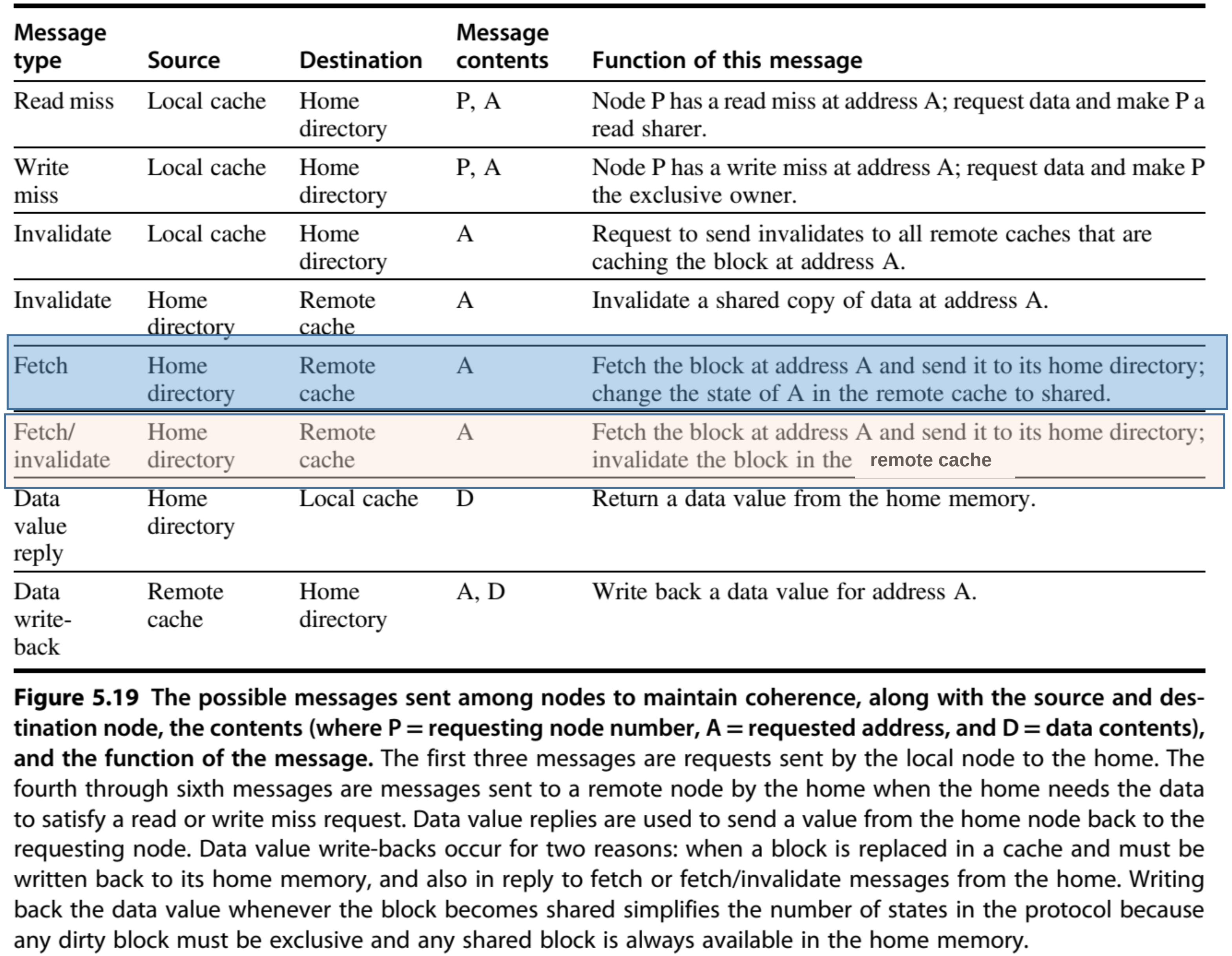
- Three types of nodes: Local, Home, Remote
  - Local: node where a request originates
  - Home: node to which the relevant DRAM for the memory location, and corresponding directory, is attached
    - The *Local node* can also be the *Home node*!
  - Remote: node which contains copy of a (local DRAM) block in its local cache
    - e.g.: Block address *A* is mapped to DRAM attached to Processor P_5 (home/local node), but processor P_8 contains a copy of the same block in its LLC
    - It is possible that a *Remote node* has the only cached copy currently!

Directory-based Coherence Protocol: Messages

- **P: requesting node number**
- **A: requested address**
- **D: block content**

In response to Read Miss from Local Cache

In response to Write Miss from Local Cache



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 remote 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.

Figure 5.19 The possible messages sent among nodes to maintain coherence, along with the source and destination node, the contents (where P = requesting node number, A = requested address, and D = data contents), and the function of the message. The first three messages are requests sent by the local node to the home. The fourth through sixth messages are messages sent to a remote node by the home when the home needs the data to satisfy a read or write miss request. Data value replies are used to send a value from the home node back to the requesting node. Data value write-backs occur for two reasons: when a block is replaced in a cache and must be written back to its home memory, and also in reply to fetch or fetch/invalidate messages from the home. Writing back the data value whenever the block becomes shared simplifies the number of states in the protocol because any dirty block must be exclusive and any shared block is always available in the home memory.

H&P CA:A QA (6th. Ed.)

Directory-based Coherence Protocol: Cache Block State Diagram

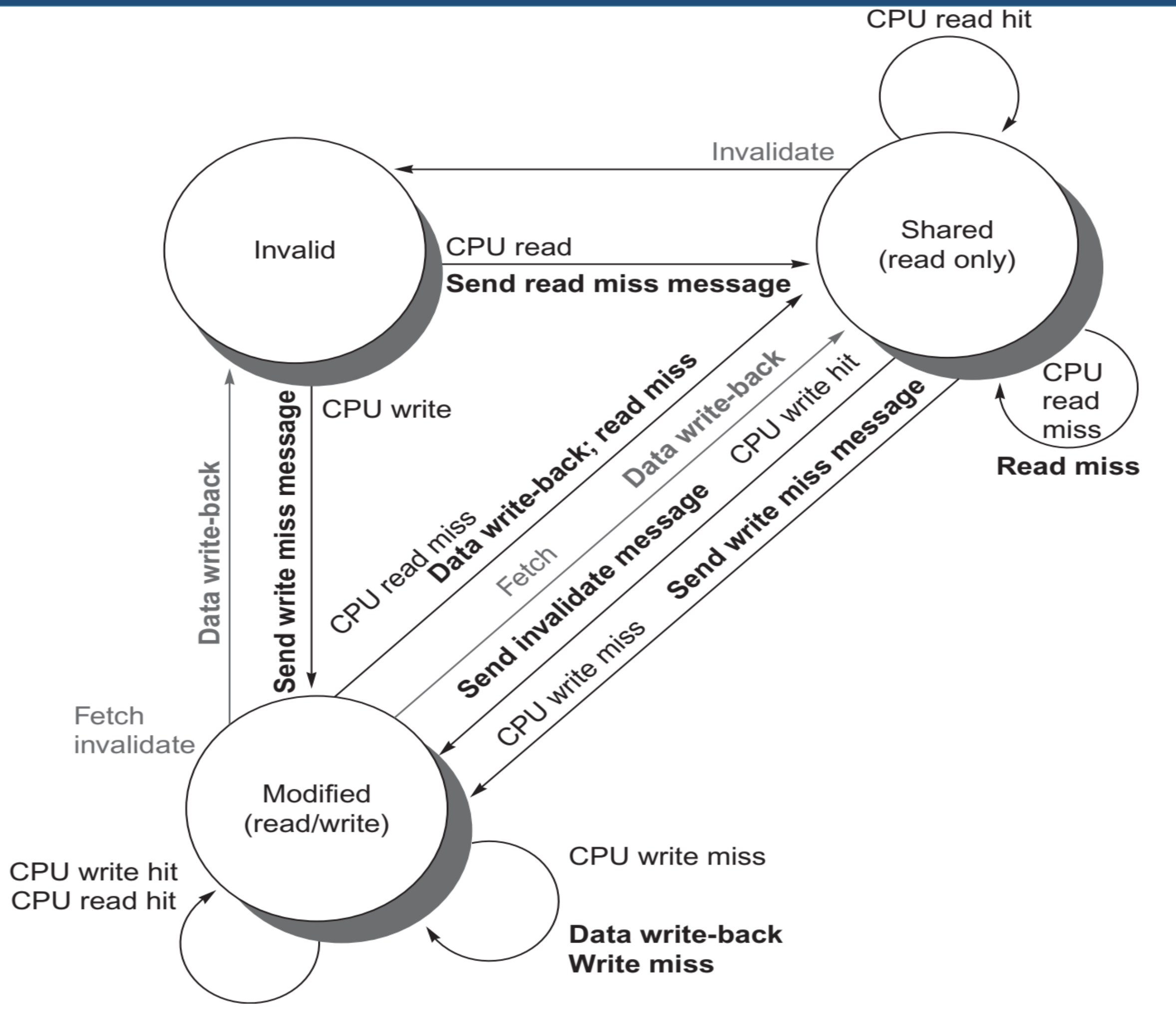


Figure 5.20 State transition diagram for an individual cache block in a directory-based system. Requests by the local processor are shown in *black*, and those from the home directory are shown in *gray*. The states are identical to those in the snooping case, and the transactions are very similar, with explicit invalidate and write-back requests replacing the write misses that were formerly broadcast on the bus. As we did for the snooping controller, we assume that an attempt to write a shared cache block is treated as a miss; in practice, such a transaction can be treated as an ownership request or upgrade request and can deliver ownership without requiring that the cache block be fetched.

H&P CA:A QA (6th. Ed.)

Directory-based Coherence Protocol: Directory State Diagram

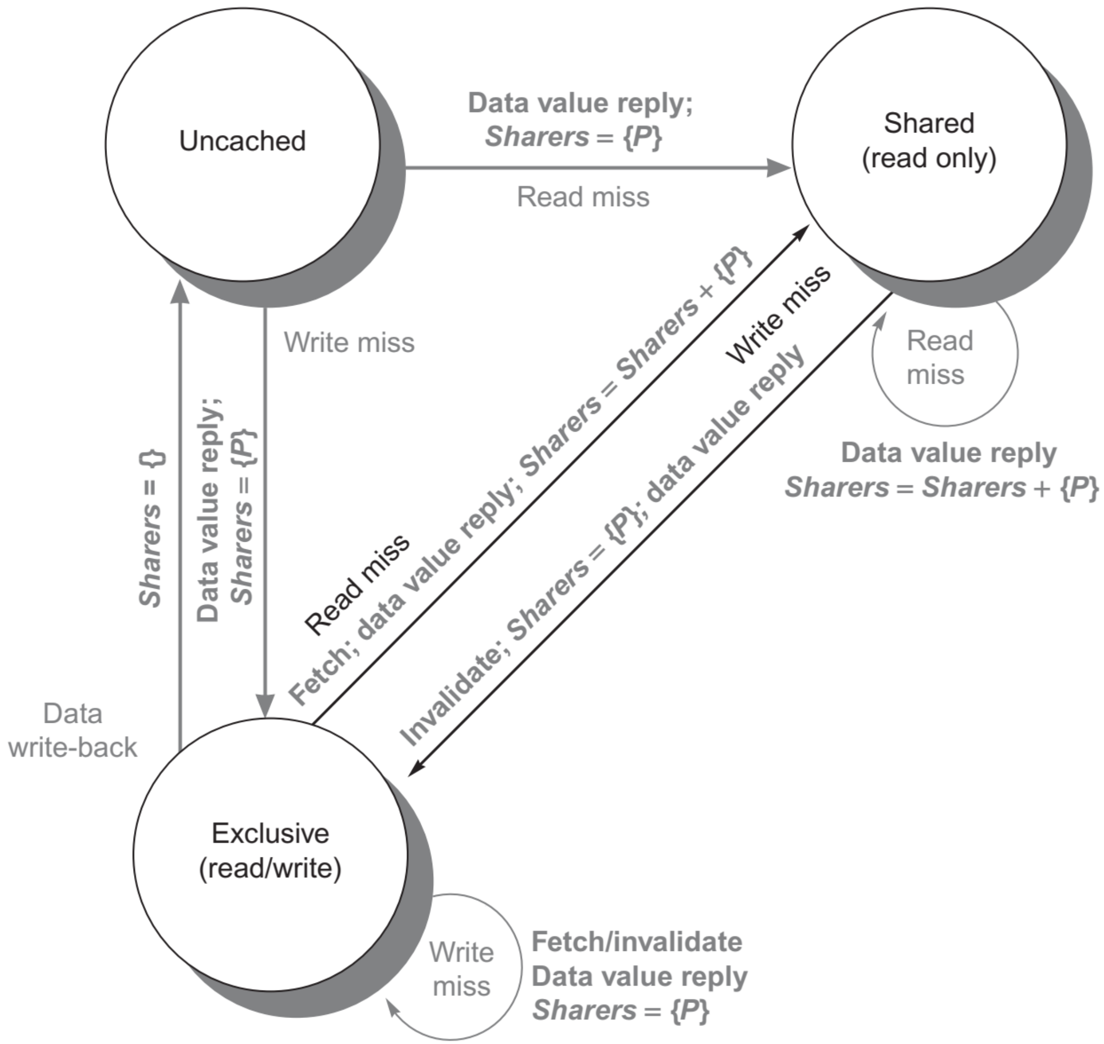


Figure 5.21 The state transition diagram for the directory has the same states and structure as the transition diagram for an individual cache. All actions are in gray because they are all externally caused. *Bold* indicates the action taken by the directory in response to the request.

H&P CA:A QA (6th. Ed.)

Thank you