Chapter 6 Multiprocessors and Thread-Level Parallelism

Outline

6.1 Introduction

Why Parallel

Greed for speed is a permanent malady – 2 basic options

- Build a faster uniprocessor
  - Advantages
    - *Programs don't need to change*
    - *Compilers may need to change to take advantage of intra-CPU parallelism*
  - Disadvantages
    - *Improved CPU performance is very costly - we already see diminishing returns*
    - *Very large memories are slow*
- Parallel Processors
  - Today implemented as an ensemble of microprocessors

Parallel Processors

- The high end requires this approach
- Advantages
  - Leverage the off-the-shelf technology
  - Huge partially unexplored set of options
- Disadvantages
  - Software: optimized balance and change are required
  - Overheads: a whole new set of organizational disasters are now possible

Types of Parallelism

- Pipelining & Speculation
- Vectorization
- Concurrency & simultaneity
- Data and control parallelism
- Partitioning & specialization
- Interleaving & overlapping of physical subsystems
- Multiplicity & replication
- Time & space sharing
- Multitasking & multiprogramming
- Multi-threading
- Distributed computing: for speed or availability

What changes when you get more than one?

- Communication
  - 2 aspects always are of concern: latency & bandwidth
  - Before: I/O meant disk/sec. = slow latency & OK bandwidth
  - Now: inter-processor communication = fast latency and high bandwidth $\rightarrow$ becomes as important as the CPU
- Resource Allocation
- Smart Programmer: programmed
- Smart Compiler: static
- Smart OS: dynamic
- Hybrid: some of all of the above is the likely balance point

A. A Taxonomy of Parallel Architectures — Flynn's Taxonomy

- Flynn's Taxonomy: 1972
  - Too simple but it's the only one that moderately works
- 4 Categories = (Single, Multiple) × (Data Stream, Instruction Stream)
- SISD: conventional uniprocessor system
  - Still a lot of intra-CPU parallelism options
- SIMD: vector and array style computers
  - First accepted multiple PE style systems
  - Now has fallen behind MIMD option
- MISD: no commercial products
- MIMD: intrinsic parallel computers
  - Lots of options → today's winner

MIMD options

- Heterogeneous vs. Homogeneous PE’s
- Communication Model
  - Implicit: shared-memory
  - Explicit: message passing
- Interconnection Topology: Ch8
  - Which PE gets to talk directly to which PE
  - Blocking vs. non-blocking
  - Packet vs. circuit switched
  - Wormhole (interconnect instantaneously) vs. store and forward
  - Synchronous vs. asynchronous

Why MIMD?

- MIMDs offer flexibility, can function as:
  - Single-user multiprocessors focusing on high performance for one AP
  - Multiprogrammed multiprocessors running many tasks simultaneously
  - Combination
- MIMDs can build on the cost-performance advantages of off-the-shelf microprocessors

B. Classes of Existing MIMD Multiprocessors

- Classes of existing MIMD multiprocessors:
  - Depend on the # of processors involved, which in turn dictate a memory organization and interconnect strategy.
- Two classes:
  - Centralized shared-memory architectures
  - Distributed-memory architectures

(a) Centralized Shared-Memory Architectures

- Symmetric (shared-memory) multiprocessors (SMPs)
  - There is a single main memory that has a symmetric relationship to all processors and a *uniform access time* (UMA) from any processor
- Uniform Memory Access
- Symmetric ➔ all PE's have same access to I/O, memory, executive (OS) capability etc.
- With large caches, the bus and the single memory, possibly with multiple banks, can satisfy the memory demands of a *small number of processors.*
- Shared memory CANNOT support the memory bandwidth demand of a larger number of processors without incurring excessively long access latency.

Basic Structure of a Centralized Shared-Memory Multiprocessor

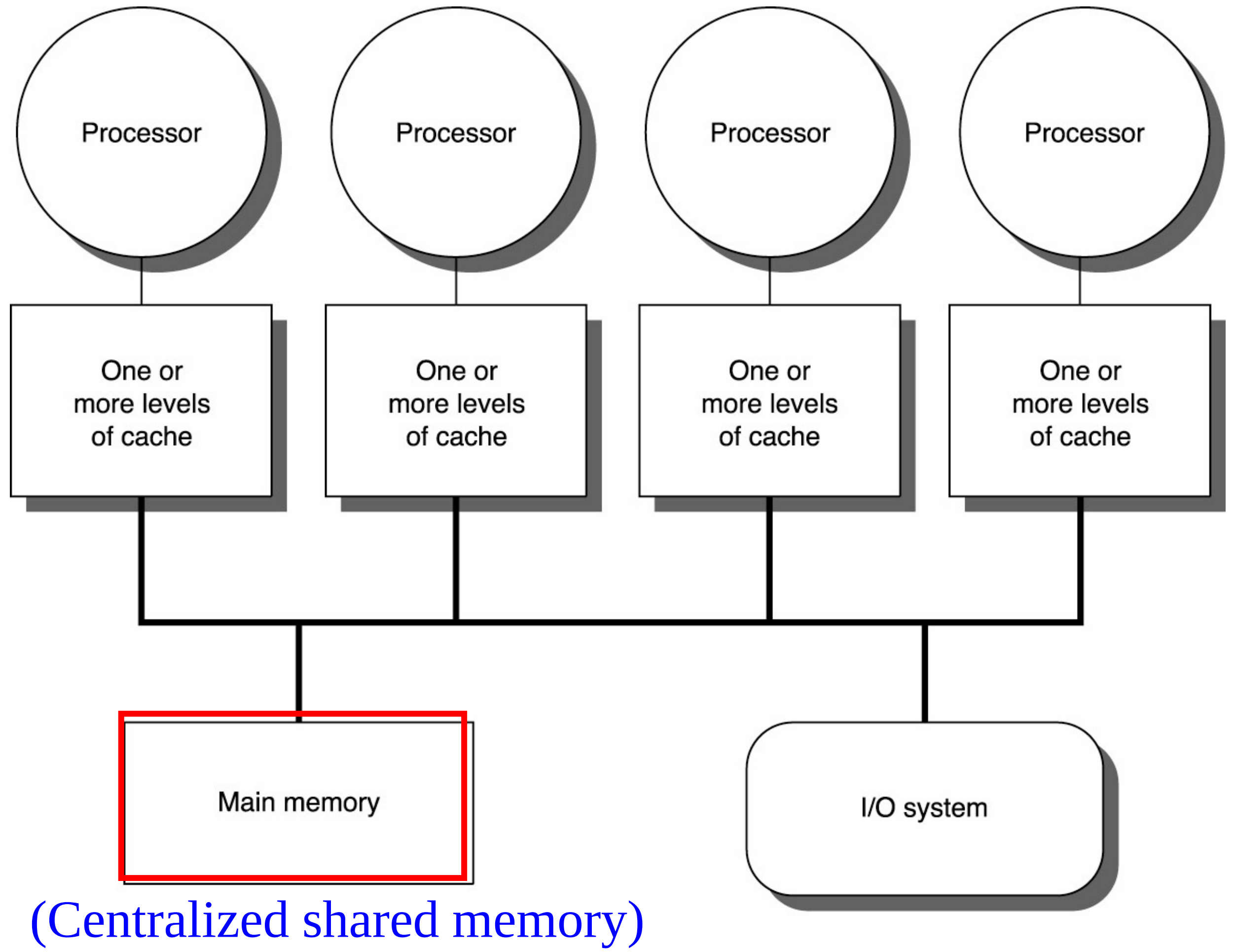
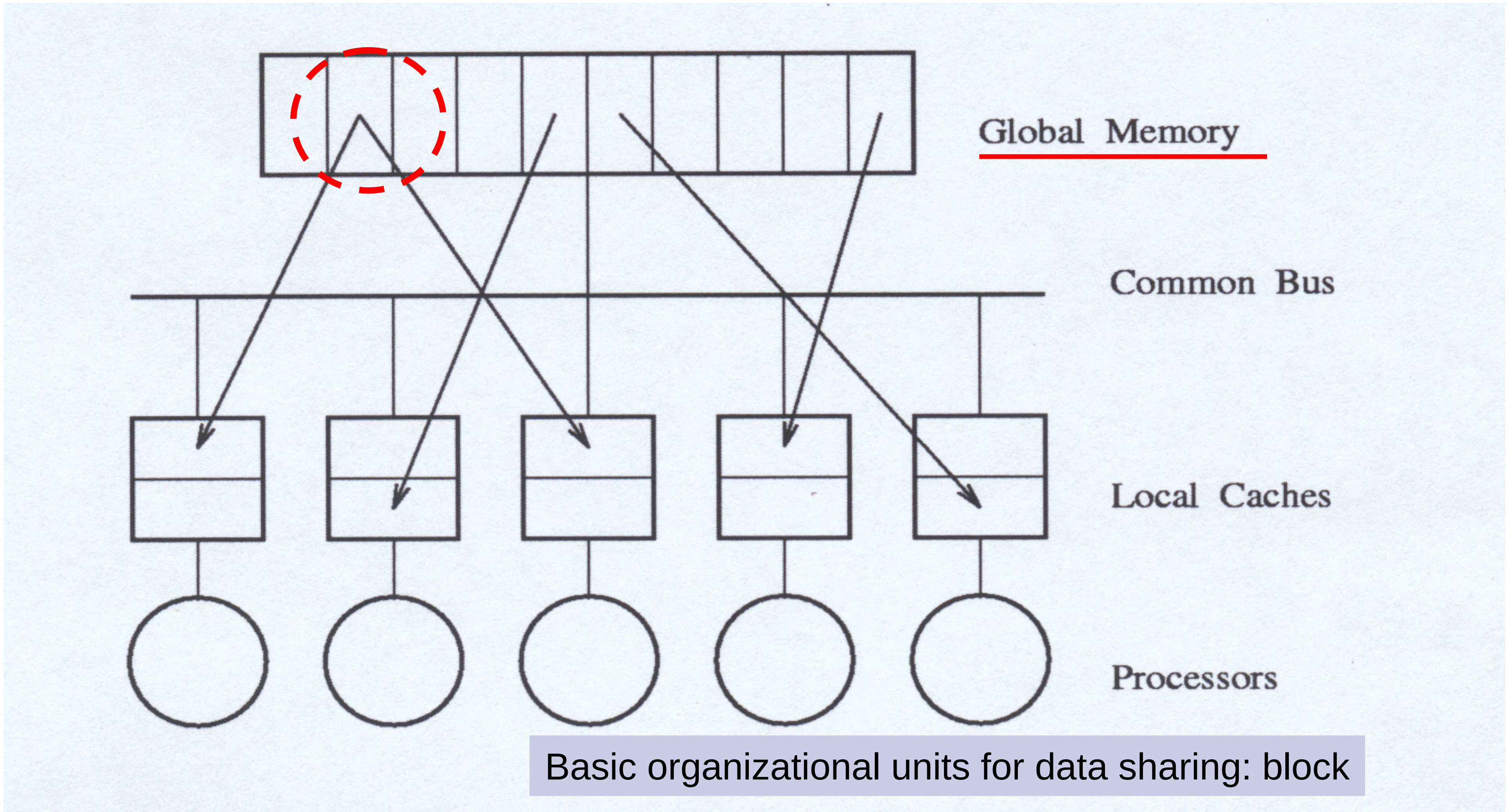

Processor
Processor
Processor
Processor
One or more levels of cache
One or more levels of cache
One or more levels of cache
One or more levels of cache
Main memory
I/O system
(Centralized shared memory)

Fig 6.1 Basic structure of a centralized shared-memory multiprocessor



Basic organizational units for data sharing: block

(a) Multiprocessor cache architecture

(b) Distributed-Memory Multiprocessor

- Multi-processors with physically distributed memory
- NUMA: Non-Uniform Memory Access
  - The access time depends on the location of a data word in memory.
- Support larger processor counts
- Raise the need for a high bandwidth interconnect
- Advantages
  - Cost-effective to scale the memory bandwidth if most of the accesses are to the local memory in the node
  - Reduce the latency for accesses to the local memory
- Disadvantages
  - Communicating data between processors becomes somewhat complex and has higher latency

The Basic Structure of Distributed-Memory Multiprocessor

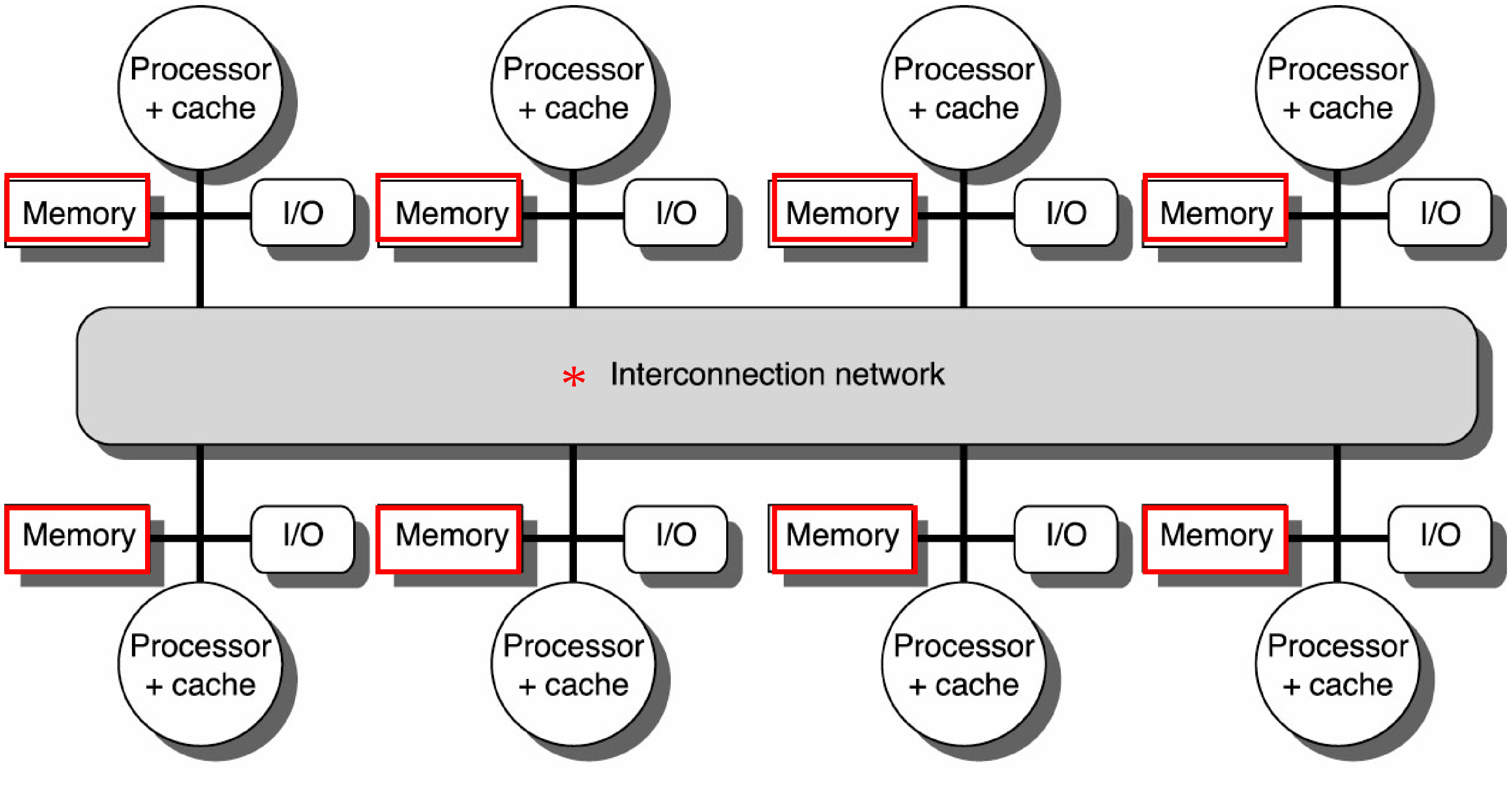
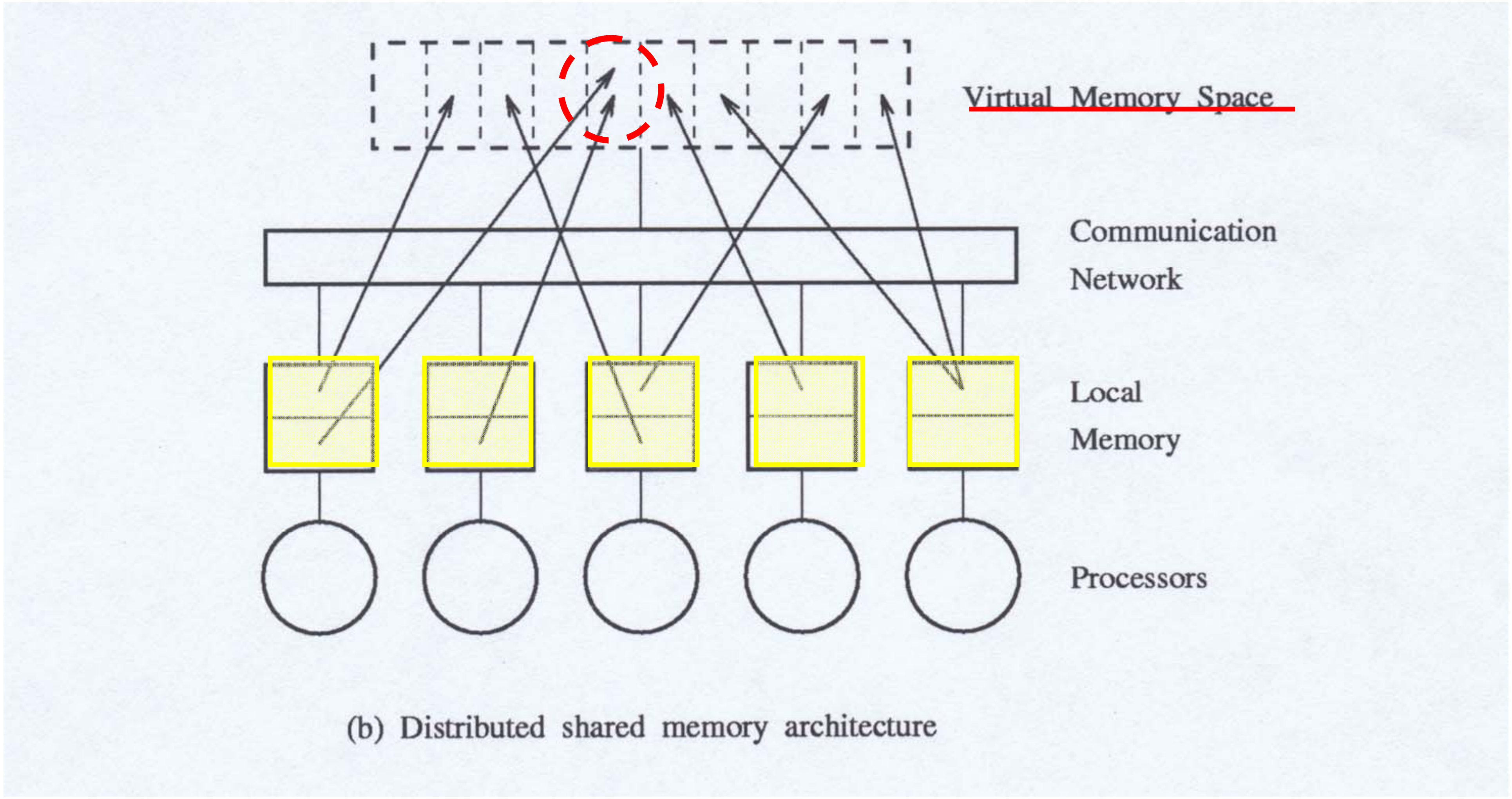


Fig 6.1 Basic structure of a distributed-memory multiprocessor

Inter-PE Communication

- Distributed shared-memory: implicit via memory
  - Distinction of local vs. remote
  - Implies some shared memory
  - Sharing model and access model must be consistent
- Message-passing: explicitly via send and receive
  - Need to know destination and what to send
  - Blocking vs. non-blocking option
  - Usually seen as message passing
  - High-level primitives: RPC (remote procedure call) …



(b) Distributed shared memory architecture

Figure Distributed shared memory

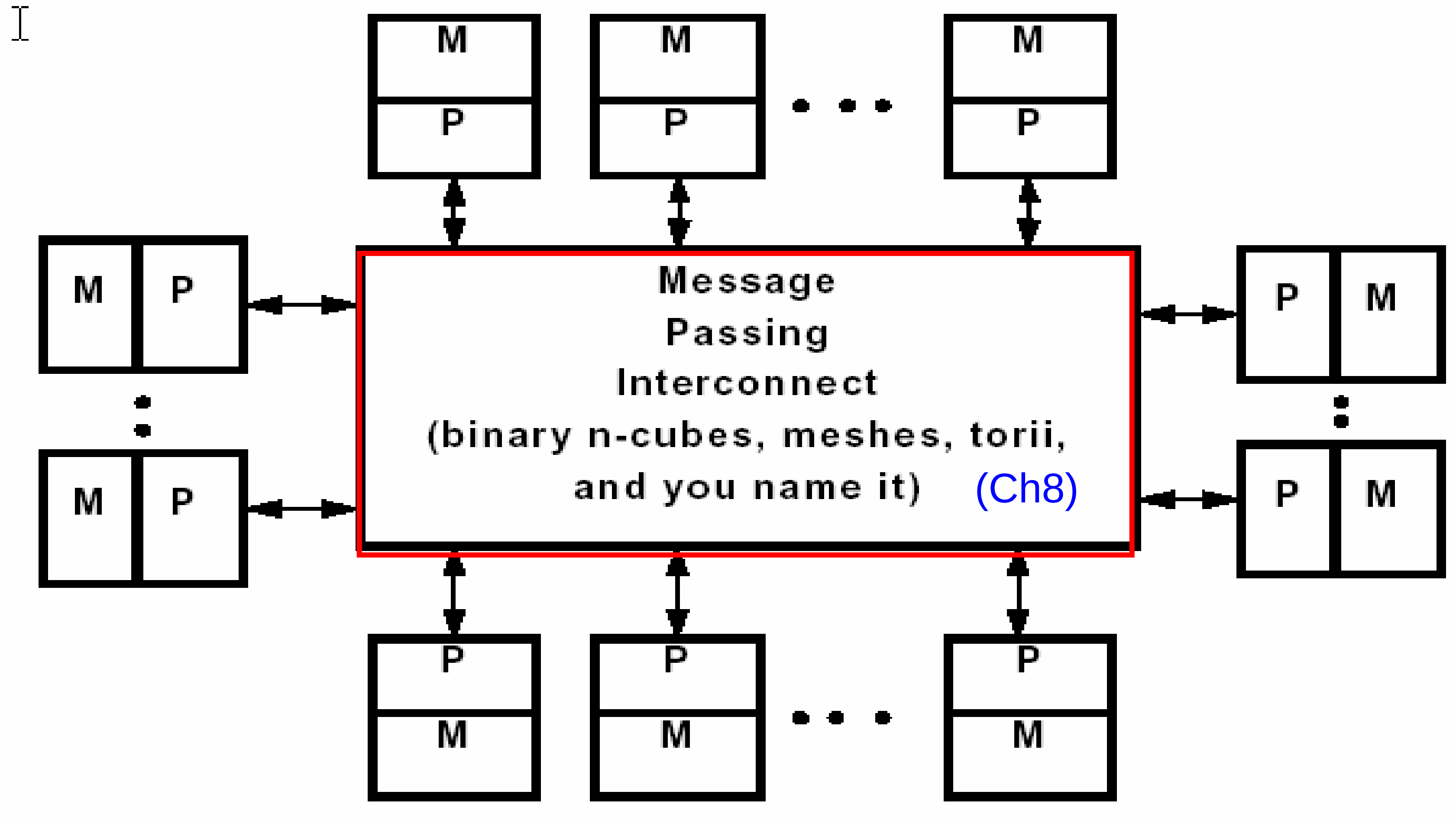


Figure Message passing MIMD machines

Performance Metrics for Communication Mechanisms

- 3 performance metrics:
  - Communication bandwidth
  - Communication latency
  - Communication latency hiding

Advantages of Different Communication Mechanisms

- Advantages of shared-memory communication:
  - Compatibility with the well-understand mechanisms in use in centralized multi-processors (shared-memory communication).
  - Ease of compiler design and programming when the communication patterns among processors are complex or vary dynamically during execution.
  - Ability to develop APs using the familiar shared-memory model.
  - Lower overhead for communication and better use of bandwidth when communication small items.
  - Ability to use HW caching to reduce the frequency of remote communication by supporting automatic caching of all data, both shared and private.

Advantages of Different Communication Mechanisms (Cont.)

- Advantages of message passing communication:
  - HW can be simpler – no need to cache remote data.
  - Communication is explicit – simpler to understand when communication occurs.
  - Explicit communication focuses programmer attention on this costly aspect of parallel computation, sometimes leading to improve structure in a multi-processor program.
  - Synchronization is naturally associated with sending messages, reducing the possibility for errors introduced by incorrect synchronization.
  - Easier to use send-initiated communication – may have some advantages in performance.

C. Challenges for Parallel Processing

- Challenge 1: Limited parallelism available in programs

  ⇒ Difficult to achieve good speedups.

  ⇒ Need new algorithms that can have better parallel performance.

Example: Parallelism

- Suppose you want to achieve a speedup of 80 with 100 processors. What fraction of the original computation can be sequential?
- Assumption: The program operates in only two modes
  - i. parallel w/ all processors fully used (enhanced mode)
  - ii. Serial w/ only one processor in use.

<Ans.>

Amdahl's Law:

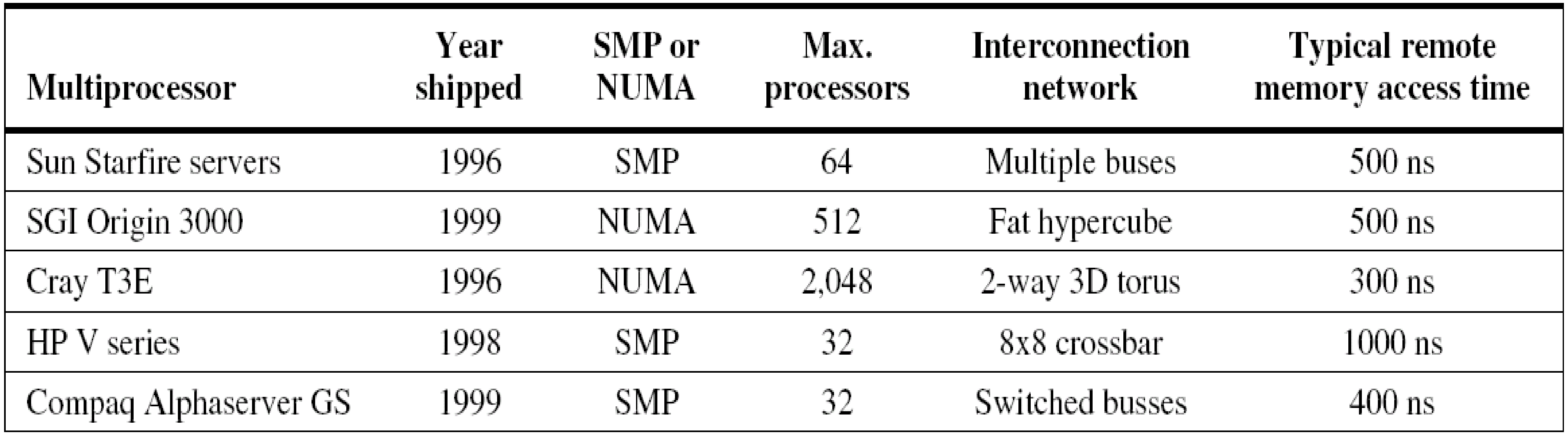
$$\boxed{Speedup_{Overall} = \frac{1}{(1 - Fraction_{enhanced}) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}}}}$$

$$80 = \frac{1}{(1 - Fraction_{Parallel}) + \frac{Fraction_{Parallel}}{100}} \Rightarrow Fraction_{Parallel} = 0.9975$$

Only 0.25% of original computation can be sequential

Challenges for Parallel Processing (Cont.)

- Challenge 2: Large latency of remote access in a parallel processor



Multiprocessor	Year shipped	SMP or NUMA	Max. processors	Interconnection network	Typical remote memory access time
Sun Starfire servers	1996	SMP	64	Multiple buses	500 ns
SGI Origin 3000	1999	NUMA	512	Fat hypercube	500 ns
Cray T3E	1996	NUMA	2,048	2-way 3D torus	300 ns
HP V series	1998	SMP	32	8x8 crossbar	1000 ns
Compaq Alphaserver GS	1999	SMP	32	Switched busses	400 ns

  - Possible solutions:
    - *HW: caching shared data…*
    - *SW: restructure the data to make more accesses local …*

Example: Effect of Long Communication Delays

- 32-processor multiprocessor. Clock rate = 1GHz (CC = 1ns)
- 400ns to handle reference to a remote memory
- Base IPC (all references hit in the cache) = 2
- Processors are stalled on a remote request.
- All the references except those involving communication hit in the local memory hierarchy.
- How much faster is the multiprocessor if there is no communication vs. if 0.2% of the instruction involve a remote communication?

<Ans.>

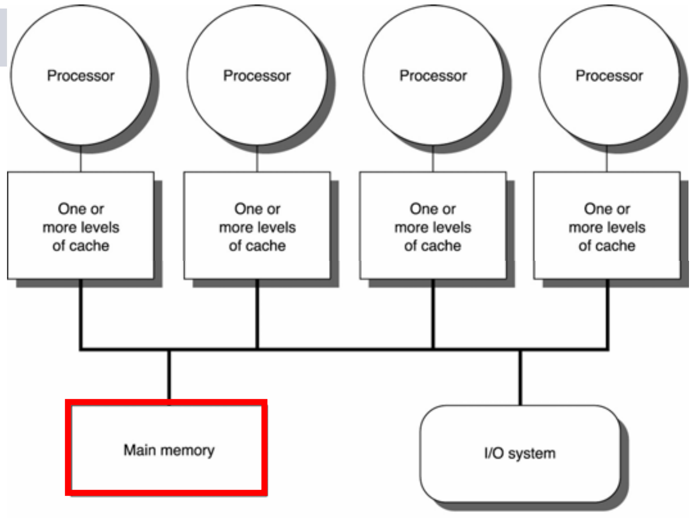
- Effect CPI (0.2% remote access)

  = Base CPI + Remote_request_rate × Remote_request_cost

  = ½ + 0.2% × 400 = 0.5 + 0.8 = 1.3
- The multiprocessor with all local reference = 1.3/0.5 = 2.6 times faster

6.3 Symmetric Shared-Memory Architectures

Overview


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
Main memory
I/O system

- The use of large, multi-level caches can substantially reduce memory bandwidth demands of a processor

  ⇒ Symmetric shared-memory architecture

  (Multi-processors, each having a local cache, share the same memory system)

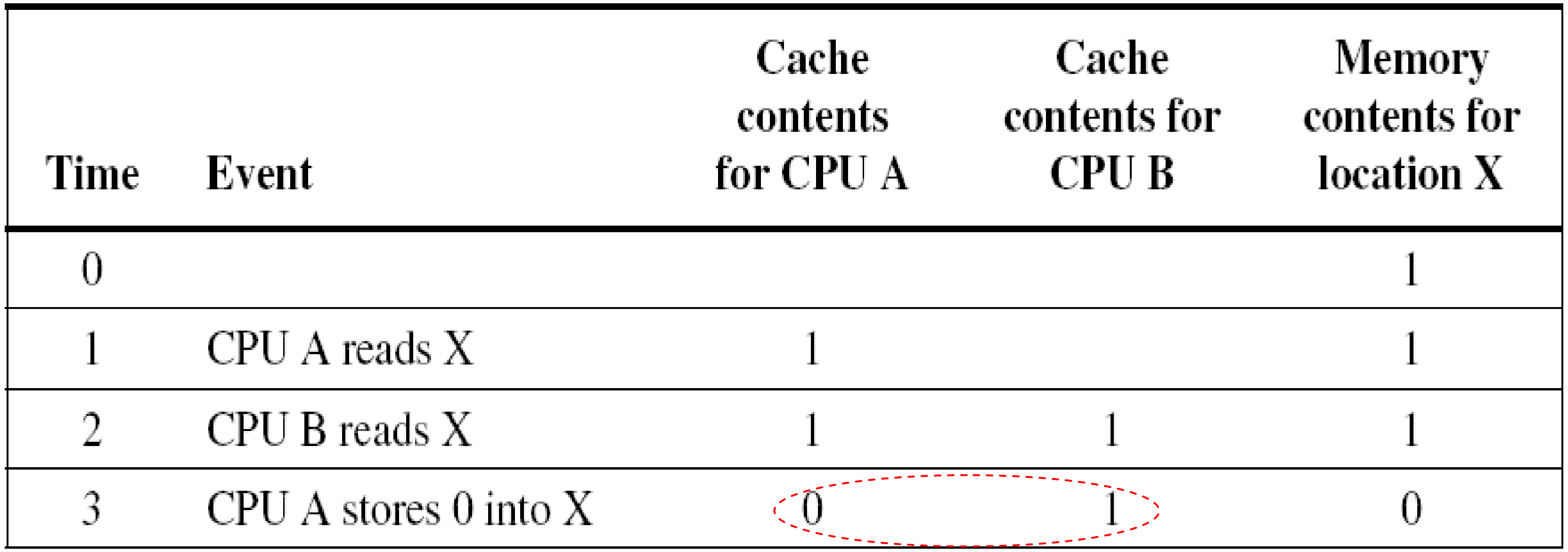
- Cache both *shared* and *private* data
  - *Private*: used by a single processor

    ➔ migrate to the cache
  - *Shared*: use by multiple processors

    ➔ replicate to the cache

* Cache coherence Problem

Example: Cache Coherence Problem



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

Write-through cache

Initially, the two caches do not contain X

Fig 6.7 The cache coherence problem for a single memory location (X), read and written by two processors (A and B).

Coherence and Consistency

- Coherence: what values can be returned by a read
- Consistency: when a written value will be returned by a read (§6.8)
  - Due to communication latency, the writes cannot be seen instantaneously
- Memory system is coherent if
  - A read by a processor P to a location X that follows a write by P to X always returns the value written by P, if no writes of X by another processor occurring between the write and the read by P
  - A read by a processor to X that follows a write by another processor to X returns the written value if the read and write are sufficiently separated in time and no other writes to X occur in-between
  - Writes to the same location are *serialized*; i.e. two writes to the same location by any two processors are seen in the same order by all processors

A. Basic Schemes for Enforcing Coherence

- Cache Coherence Protocol:
  – Key: tracking the state of any sharing of a data block
- 2 classes of cache coherence protocols:

  i. Directory based: (§6.5)
    - *The sharing status of a block of physical memory is kept in just one location, the* directory.

  ii. Snooping: popular for symmetric shared-mem arch (§6.3)
    - *Every cache that has a copy of the data from a block of physical memory also has a copy of the sharing status of the block, and no centralized state is kept.*
    - *Caches are usually on a shared-memory bus, and all cache controllers monitor or snoop on the bus to determine if they have a copy of a block that is requested on the bus*

Coherence enforcement strategy

- 2 ways to maintain the coherence requirement:
  - i. Write-invalidate: (✓)
    - *Writer sends invalidation to all caches whenever data is modified, i.e., all other cached copies of the item are invalidated.*
    - *By far the most common protocol, both for snooping and for directory schemes.*
  - ii. Write-update: *write-broadcast*
    - *Writer propagates the update, i.e., update all the cached copies of a data item when that item is written.*

Example: Write-Invalidate (Snooping Bus)

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

Write-back Cache

Fig 6.8 An example of an invalidation protocol working on a snooping bus for a single cache block (X) with write-back caches.

Example: Write-Update (Snooping Bus)

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

Write-back Cache

Fig 6.9 An example of a write update or broadcast protocol working on a snooping bus for a single cache block (X) with write-back caches.

Write-Update vs. Write-Invalidate

- Multiple writes to the same word with no intervening reads require multiple write broadcast in an update protocol, but only one initial invalidation in a write invalidate protocol.
- With multiword cache block, each word written in a cache block requires a write broadcast in an update protocol, although only the first write to any word in the block needs to generate an invalidate in an invalidation protocol.
  - Invalidation protocol works on cache blocks
  - Update protocol works on individual words

Write-Update vs. Write-Invalidate (Cont.)

- Delay between writing a word in one processor and reading the value in another processor is usually less in a write update scheme, since the written data are immediately updated in the reader's cache. In an invalidation protocol, the reader is invalidated first, then later reads the data and is stalled until a copy can be read and returned to the processor.
- Invalidate protocols generate less bus and memory traffic. (✓)
- Update protocols cause problems for memory consistency model.

B. Basic Snooping Implementation Techniques

- Implementation of a snooping protocol with write invalidate on a symmetric shared-memory architecture:
  - Use the bus to perform invalidates
    - *Acquire bus access and broadcast address to be invalidated on bus*
    - *All processors continuously snoop on the bus, watching the address*
    - *Check if the address on the bus is in their cache ➔ invalidate*
  - A write to a shared data item cannot complete until it obtains bus access.
  - Serialization of access enforced by bus also forces serialization of writes
    - *The first processor to obtain bus access will cause others' copies to be invalidated.*
  - Assume atomic operations. (Consider no consistency problem)

Basic Snooping Implementation Techniques (Cont.)

- How to locate a data when a cache miss occurs?
  - For a write-through cache: easy
    - *Memory always has the most updated data. ⇒ From which the most recent value of a data item can always be fetched.*
  - For a write-back cache: harder but generate lower requirements for memory bandwidth (✓)
    - *The most recent value of a data item can be in a cache rather than in memory.*
    - *Use the same snooping scheme both for cache misses and for writes. ⇒ Each processor snoops every addr placed on the 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 access to be aborted.*

      (Has a dirty copy of data → provide it and abort memory access)

Basic Snooping Implementation Techniques (Cont.)

- HW cache structure for implementing snooping:
  - tag: can be used to implement the process of snooping
  - valid bit: makes invalidation easy to implement
  - dirty bit
  - Shared bit: *shared mode* or *exclusive (unshared) mode*
    - *to track whether or not a cache block is shared $\Rightarrow$ 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 private. No further invalidations will be sent by that processor for that block. If another processor later requests this cache block, the state must be made shared again.*

Conceptual Write-Invalid Protocol with Snooping

- Read hit (valid): reads the data block and continues
- Read miss: does not hold the block or invalid block
  - Transfer the block from the shared memory (write-through, or write-back and clean), or from the copy-holder (write-back dirty)
  - Set the corresponding valid-bit and shared-mode bit
    - *The sole holder? Yes → set the shared-mode bit exclusive*
    - *If the block before the read is exclusive? Yes → set the shared-mode bit to shared*

Conceptual Write-Invalid Protocol with Snooping (Cont.)

- Write hit:
  - Write hit to an exclusive cache block (block owner)
    → proceeds and continues
  - Write hit to a shared-read-only block
    → need to obtain permission
    - *Invalidate all cache copies*
    - *Completion of invalidation → write data and set the exclusive bit*
      - *The processor becomes the sole owner of the cache block until other read accesses arrive from other processors*
        - *Can be detected by snooping*
        - *Then the block changes to the shared state*
- Write miss: action similar to that of a write hit, except
  - A block copy is transfer to the processor after the invalidation

C. An Example Snooping Protocol

- Cache states:
  - Invalid
  - Shared
  - Exclusive

An Example Snooping Protocol (Cont.)

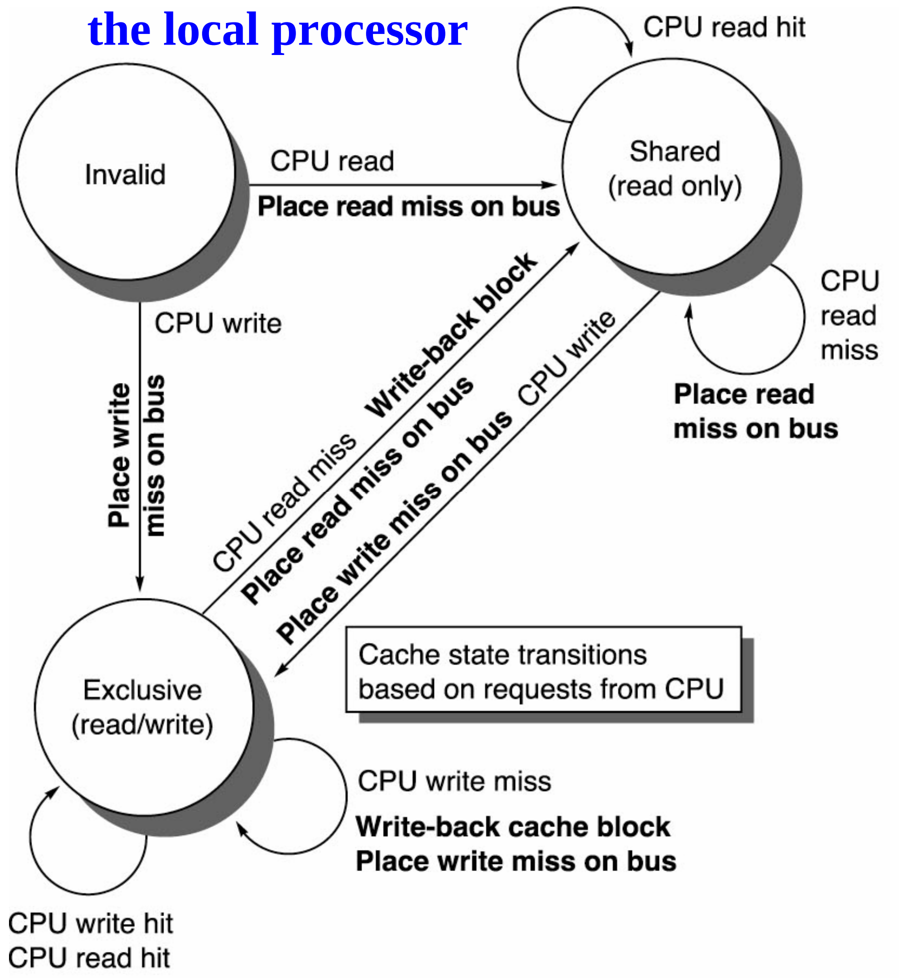
Request	Source	State of addressed cache block	Function and explanation
Read hit	Processor	Shared or Exclusive	Read data in cache
Read miss	Processor	Invalid	Place read miss on bus.
Read miss	Processor	Shared	Address conflict miss: place read miss on bus
Read miss	Processor	Exclusive	Address conflict miss: write back block, then place read miss on bus
Write hit	Processor	Exclusive	Write data in cache.
* Write hit	Processor	Shared	Place ~~write miss~~ on bus. (Write data in cache)
Write miss	Processor	Invalid	Place write miss on bus.
Write miss	Processor	Shared	Address conflict miss: place write miss on bus
Write miss	Processor	Exclusive	Address conflict miss: write back block, then place write miss on bus
Read Miss	Bus	Shared	No action; allow memory to service read miss.
Read Miss	Bus	Exclusive	Attempt to share data: place cache block on bus and change state to Shared.
Write miss	Bus	Shared	Attempt to write shared block; invalidate the block.
Write miss	Bus	Exclusive	Attempt to write block that is exclusive elsewhere: write back the cache block and make its state Invalid.

An Example Snooping Protocol (Cont.)

- Treat "write-hit to a shared cache block" as write-miss
  - Place write-miss on bus. Any processors with the block → invalidate
  - Reduce no. of different bus transactions and simplifies controller
    - *Only 2 different bus transactions: read miss & write miss*

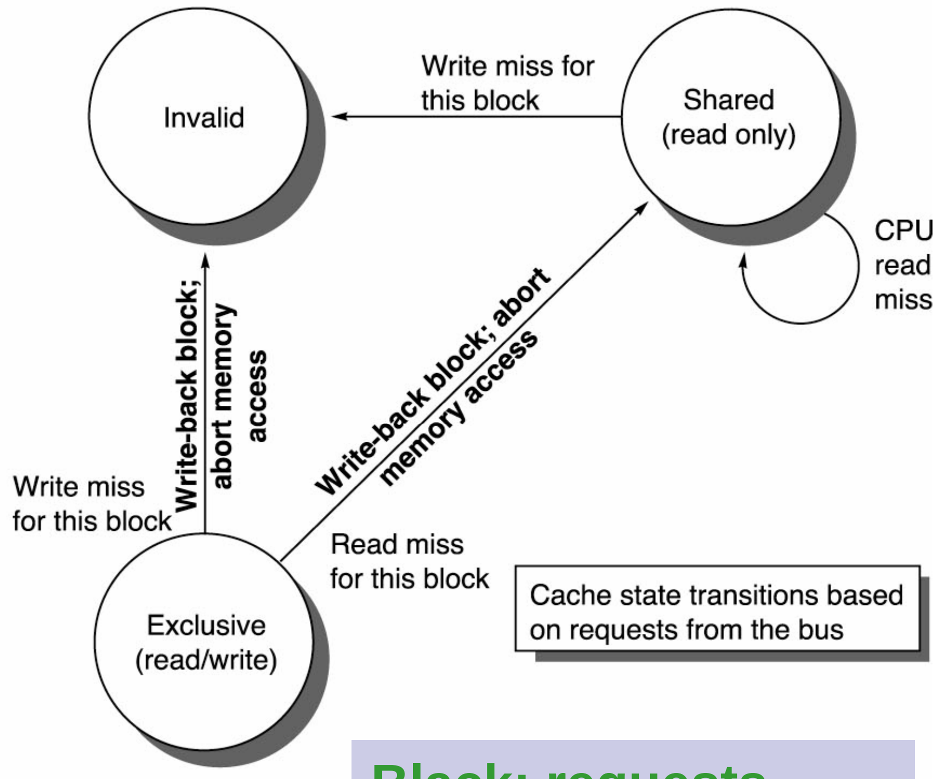
Write-Invalidate Cache Coherence Protocol for a Write-Back Cache

state transitions induced by the local processor



Invalid
Shared (read only)
Exclusive (read/write)
CPU read hit
CPU read
Place read miss on bus
CPU write
Place write miss on bus
CPU read miss
Write-back block
Place read miss on bus
CPU write
Place write miss on bus
CPU read miss
Place read miss on bus
Cache state transitions based on requests from CPU
CPU write miss
Write-back cache block
Place write miss on bus
CPU write hit
CPU read hit

state transitions induced by the bus activities

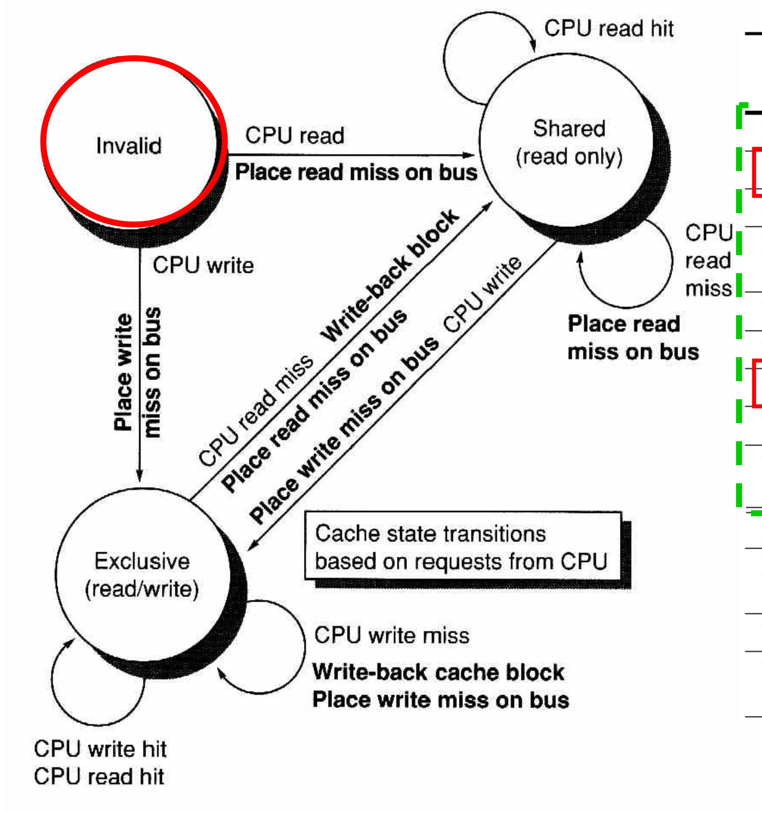


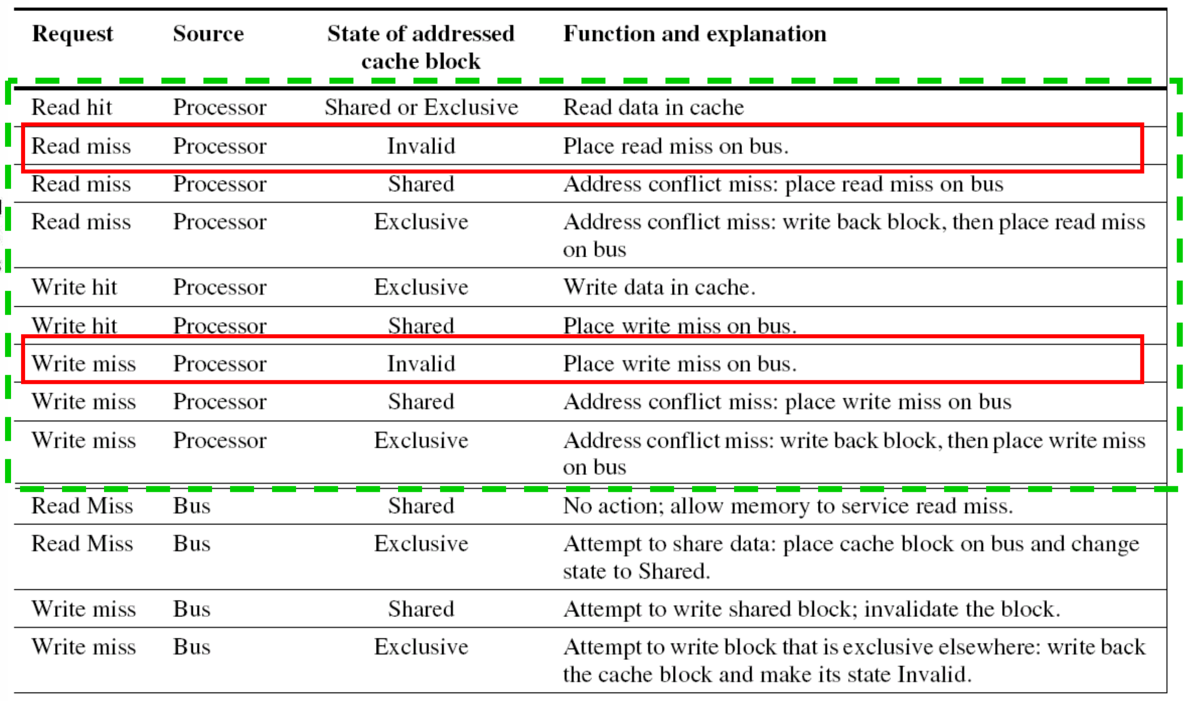
Invalid
Shared (read only)
Exclusive (read/write)
Write miss for this block
CPU read miss
Write miss for this block
Write-back block; abort memory access
Write-back block; abort memory access
Read miss for this block
Cache state transitions based on requests from the bus

Black: requests
Bold: action

Fig 6.11 A write invalidate, cache coherence protocol for a write-back cache showing the states and state transitions for each block in the cache.

state transitions induced by the local processor

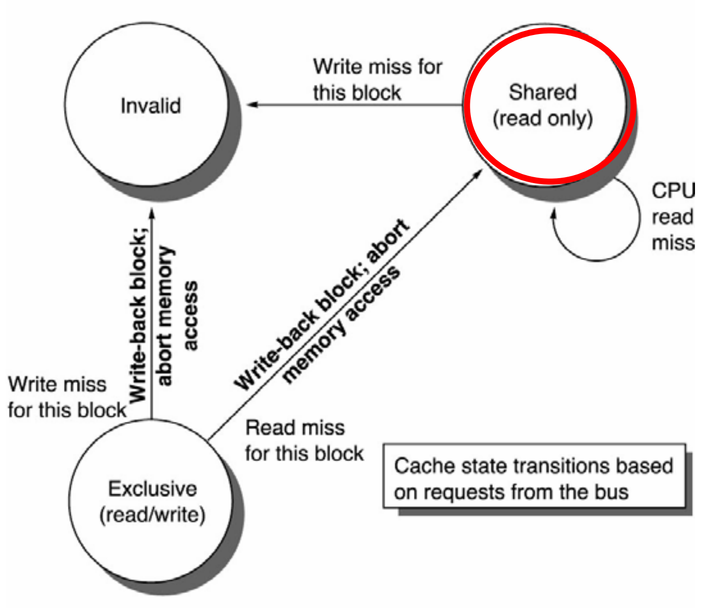




Request	Source	State of addressed cache block	Function and explanation
Read hit	Processor	Shared or Exclusive	Read data in cache
Read miss	Processor	Invalid	Place read miss on bus.
Read miss	Processor	Shared	Address conflict miss: place read miss on bus
Read miss	Processor	Exclusive	Address conflict miss: write back block, then place read miss on bus
Write hit	Processor	Exclusive	Write data in cache.
Write hit	Processor	Shared	Place write miss on bus.
Write miss	Processor	Invalid	Place write miss on bus.
Write miss	Processor	Shared	Address conflict miss: place write miss on bus
Write miss	Processor	Exclusive	Address conflict miss: write back block, then place write miss on bus
Read Miss	Bus	Shared	No action; allow memory to service read miss.
Read Miss	Bus	Exclusive	Attempt to share data: place cache block on bus and change state to Shared.
Write miss	Bus	Shared	Attempt to write shared block; invalidate the block.
Write miss	Bus	Exclusive	Attempt to write block that is exclusive elsewhere: write back the cache block and make its state Invalid.

Fig 6.11 (a) A write invalidate, cache coherence protocol for a write-back cache showing the states and state transitions for each block in the cache.

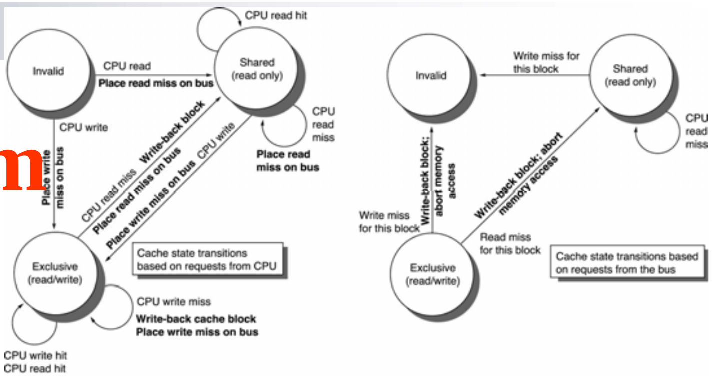
state transitions induced by the bus activities


Invalid
Write miss for this block
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

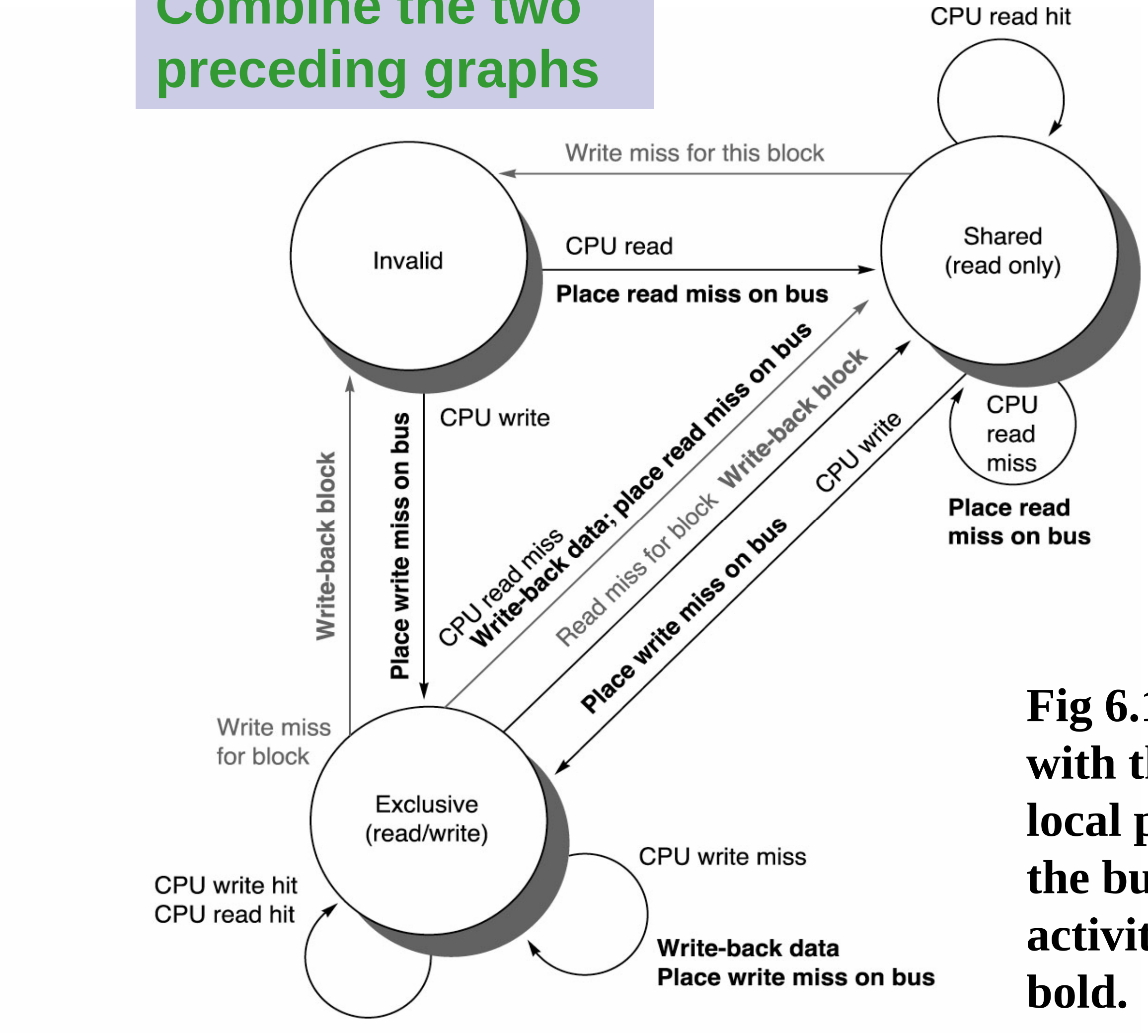
Request	Source	State of addressed cache block	Function and explanation
Read hit	Processor	Shared or Exclusive	Read data in cache
Read miss	Processor	Invalid	Place read miss on bus.
Read miss	Processor	Shared	Address conflict miss: place read miss on bus
Read miss	Processor	Exclusive	Address conflict miss: write back block, then place read miss on bus
Write hit	Processor	Exclusive	Write data in cache.
Write hit	Processor	Shared	Place write miss on bus.
Write miss	Processor	Invalid	Place write miss on bus.
Write miss	Processor	Shared	Address conflict miss: place write miss on bus
Write miss	Processor	Exclusive	Address conflict miss: write back block, then place write miss on bus
Read Miss	Bus	Shared	No action; allow memory to service read miss.
Read Miss	Bus	Exclusive	Attempt to share data: place cache block on bus and change state to Shared.
Write miss	Bus	Shared	Attempt to write shared block; invalidate the block.
Write miss	Bus	Exclusive	Attempt to write block that is exclusive elsewhere: write back the cache block and make its state Invalid.

Fig 6.11 (b) A write invalidate, cache coherence protocol for a write-back cache showing the states and state transitions for each block in the cache.

Cache Coherence State Diagram



Combine the two preceding graphs



Request induced by the local processor shown in black and by the bus activities shown in gray

Fig 6.12 Cache coherence state diagram with the state transitions induced by the local processor (shown in black) and by the bus activities (shown in gray). The activities on a transition are shown in bold.

6.4 Performance of Symmetric Shard-Memory Multiprocessors

Performance Measurement

- Overall cache performance is a combination of
  - Uniprocessor cache miss traffic
  - Traffic caused by communication: invalidation and subsequent cache misses
- Uniprocessor miss rate: compulsory, capacity, conflict
- Communication miss rate: coherence misses
  - True sharing misses + false sharing misses
- Changing the processor count, cache size, and block size can affect these two components of miss rate

True and False Sharing Miss

- True sharing miss:
  - arise from the communication of data through the cache coherence mechanism
  1. The first write by a PE to a shared cache block causes an invalidation to establish ownership of that block.
  2. When another PE attempts to read a modified word in that cache block, a miss occurs and the resultant block is transferred.
- False sharing miss:
  - arise from the use of an invalidation-based coherence algorithm w/ a single valid bit per cache block.
  - Occur when a block is invalidate (and a subsequent reference causes a miss) because some word in the block, other than the one being read, is written to.
  - **The block is shared, but no word in the cache is actually shared, and this miss would not occur if the block size were a single word.**

Example: True and False Sharing Miss

- Assume that words x1 and x2 are in the same cache block, which is in the shared state in the caches of P1 and P2. Assuming the following sequence of events, identify each miss as a true sharing miss or a false sharing miss.

Time	P1	P2
1	Write x1	
2		Read x2
3	Write x1	
4		Write x2
5	Read x2	

Time	P1	P2
1	Write x1	
2		Read x2
3	Write x1	
4		Write x2
5	Read x2	

<Ans.>

- 1: True sharing miss (invalidate P2)
- 2: False sharing miss
  - x2 was invalidated by the write of P1, but that value of x1 is not used in P2
- 3: False sharing miss
  - The block containing x1 is marked shared due to the read in P2, but P2 did not read x1. A write miss is required to obtain exclusive access to the block
- 4: False sharing miss
- 5: True sharing miss

Performance Measurements

- Commercial Workload
- **Multiprogramming and OS Workload**
- Scientific/Technical Workload

Multiprogramming and OS Workload

- Two independent copies of the compile phase of Andrew benchmark
  - A parallel make using eight processors
  - Run for 5.24 seconds on 8 processors, creating 203 processes and performing 787 disk requests on three different file systems
  - Run with 128MB of memory, and no paging activity
- Three distinct phases of the workload:
  - Compiling the benchmarks: substantial compute activity
  - Installing the object files in a binary: I/O plays a major role and the CPU is largely idle
  - Removing the object files: dominated by I/O and only 2 processes are active

Multiprogramming and OS Workload (Cont.)

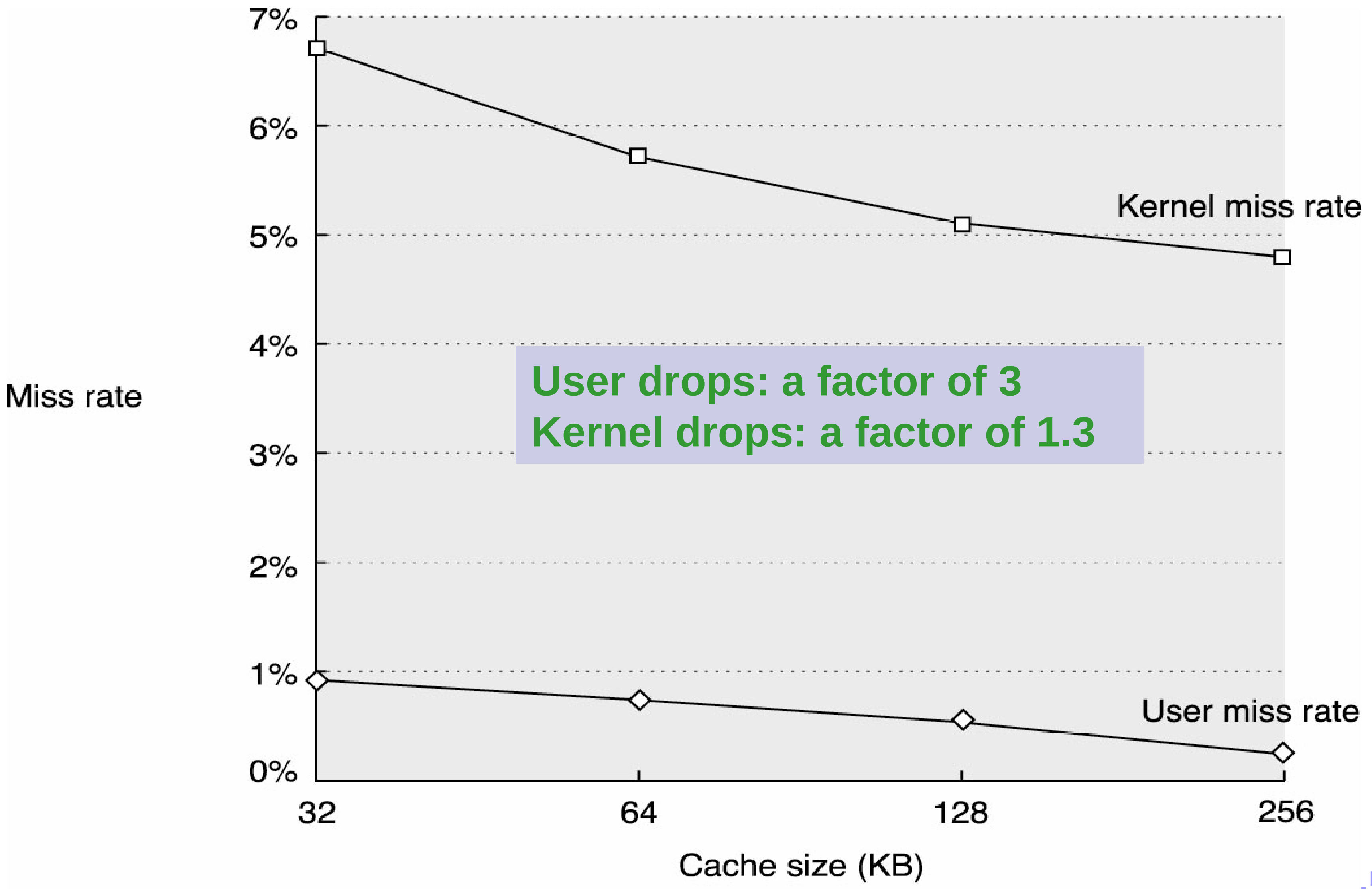
- Assumptions of the memory and I/O systems:
  - L1 I-cache: 32KB, 2-way set associative with 64-byte block, 1 CC hit time
  - L1 D-cache: 32KB, 2-way set associative with 32-byte block, 1 CC hit time
  - L2 cache: 1MB unified, 2-way set associative with 128-byte block, 10 CC hit time
  - Main memory: 128MB, single memory on a bus with an access time of 100 CC
  - Disk system: fixed-access latency of 3 ms (less than normal to reduce idle time)

Distribution of Execution Time in the Multiprogrammed Parallel Make Workload

	User Execution	Kernel Execution	Synchronization Wait	CPU idle for I/O
% instruction executed	27	3	1	69
% execution time	27	7	2	64

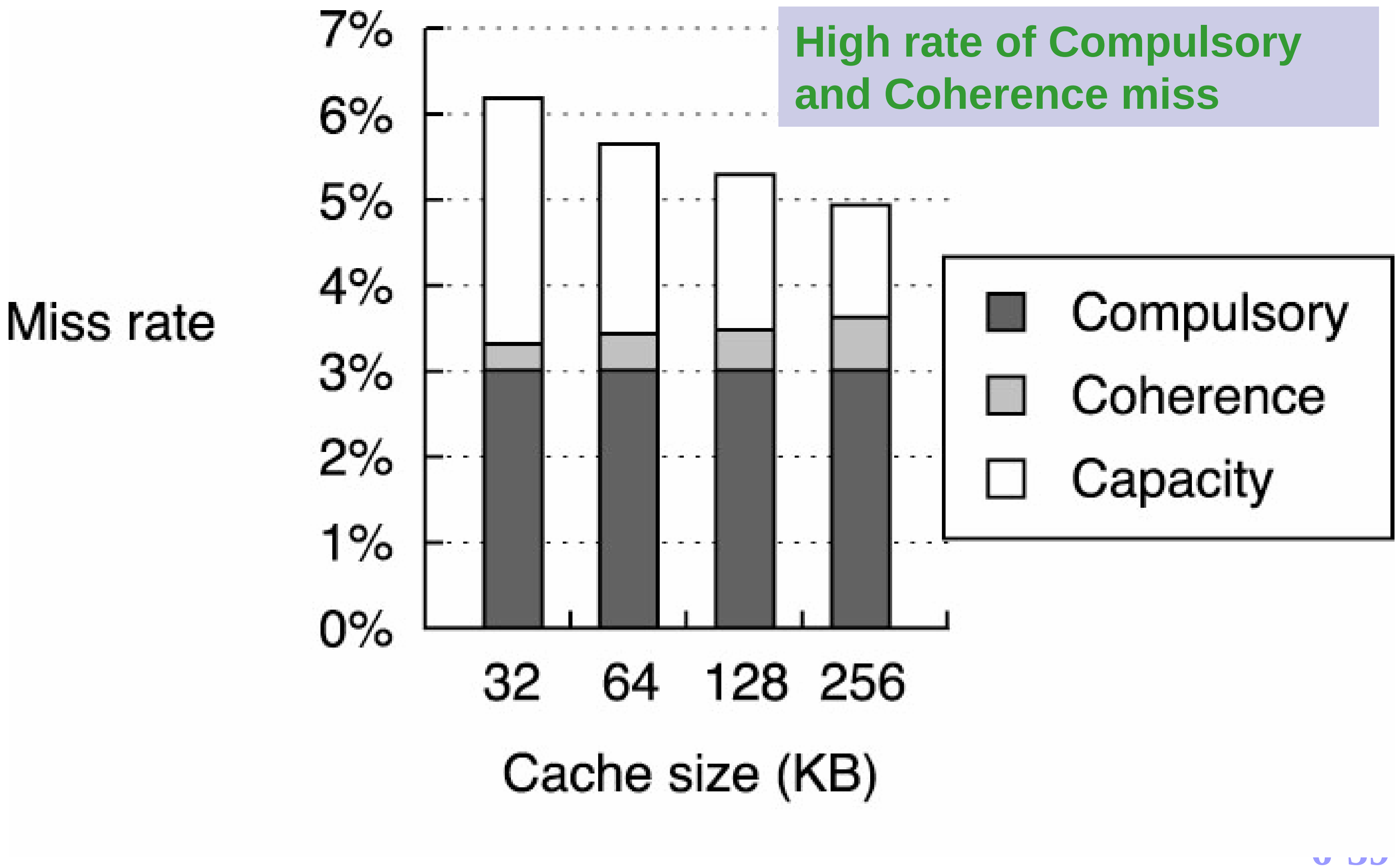
- A significant I-cache performance loss (at least for OS)
  - I-cache miss rate in OS for a 64-byte block size, 2-way se associative: 1.7% (32KB) ~ 0.2% (256KB)
  - I-cache miss rate in user-level: 1/6 of OS rate

Data Miss Rate vs. Data Cache Size



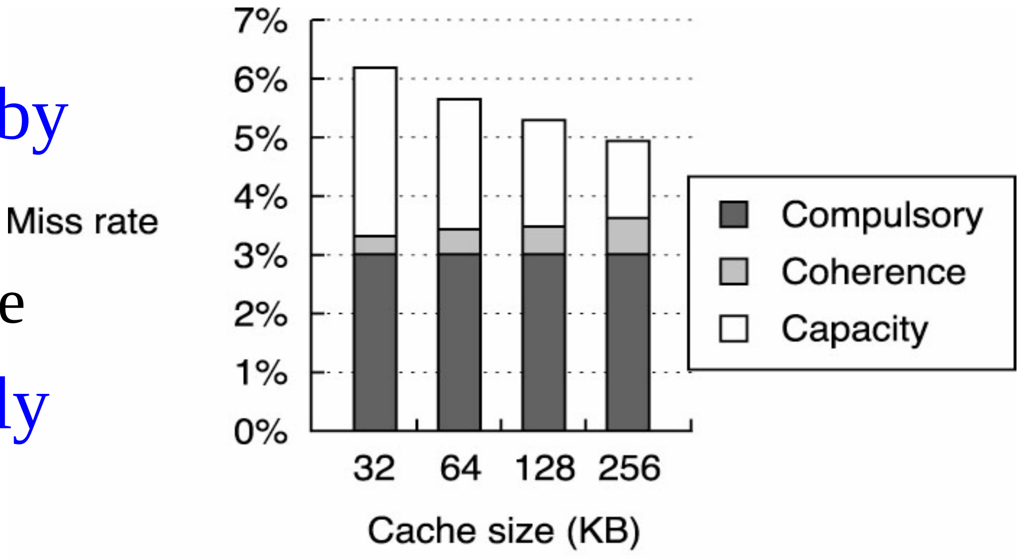
User drops: a factor of 3
Kernel drops: a factor of 1.3

Components of Kernel Data Miss Rate



Components of Kernel Data Miss Rate

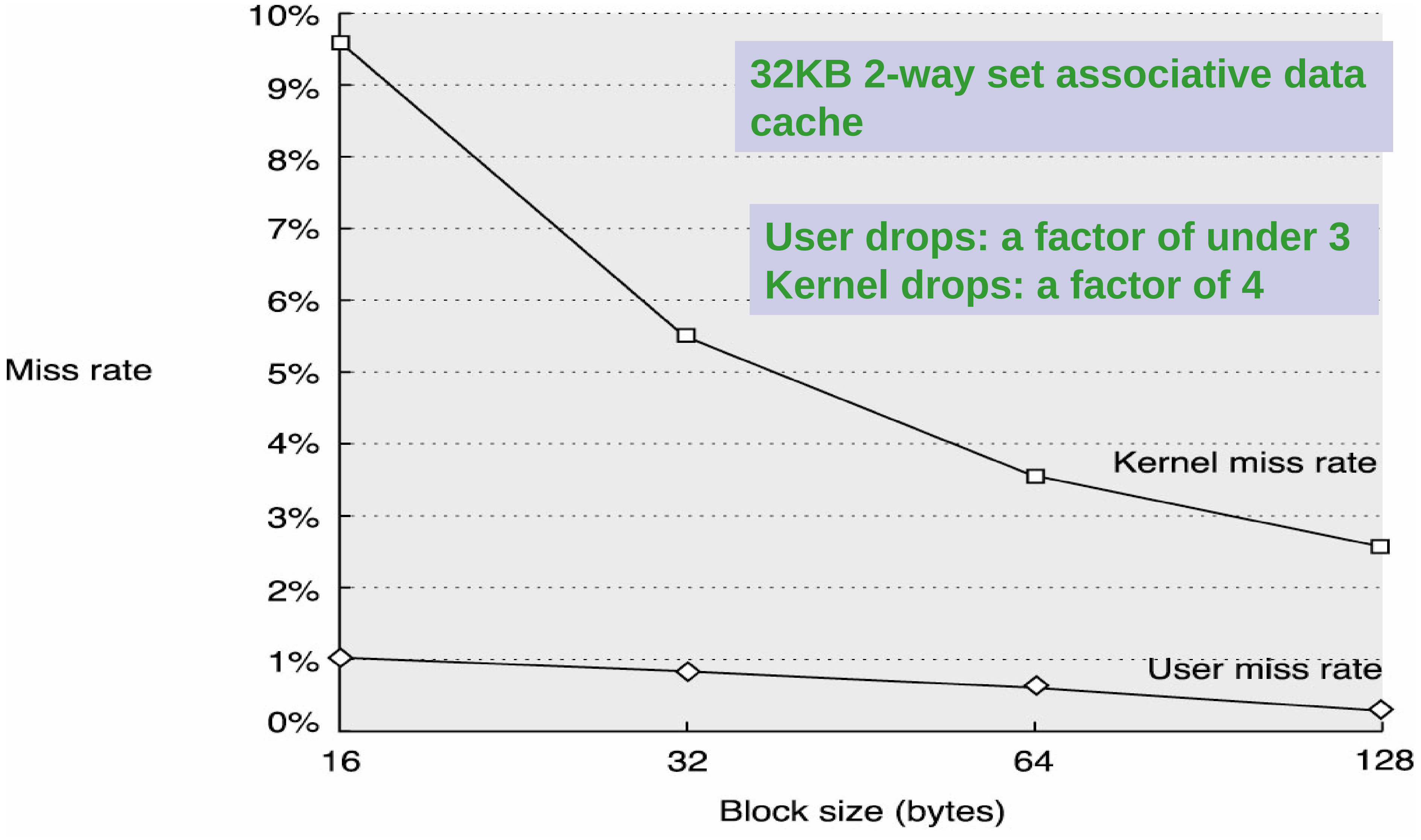
- Compulsory miss rate stays constant
- Capacity miss rate drops by more than a factor of 2
  - Including conflict miss rate
- Coherence miss rate nearly doubles
  - The probability of a miss being caused by an invalidation increases with cache size



Kernel and User Behavior

- Kernel behavior
  - Initialize all pages before allocating them to user → compulsory miss
  - Kernel actually shares data → coherence miss
- User process behavior
  - Cause coherence miss only when the process is scheduled on a different processor ➔ small miss rate

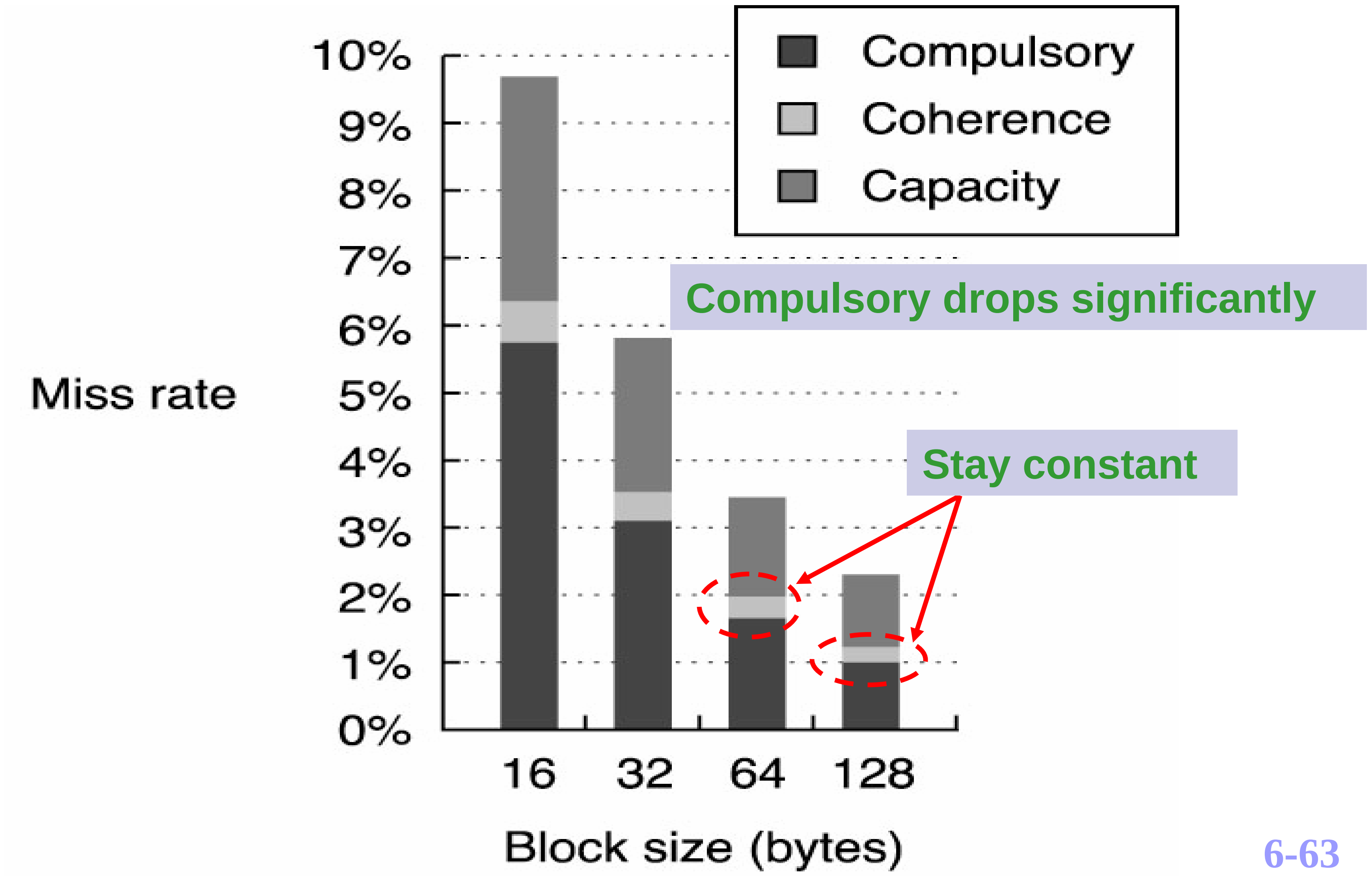
Data Miss Rate vs. Block Size



32KB 2-way set associative data cache

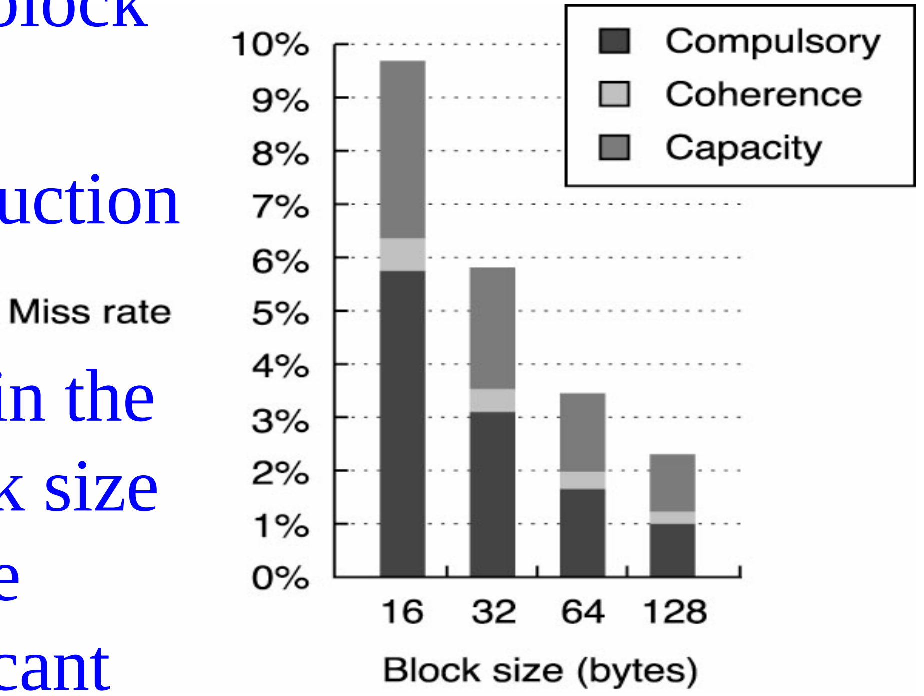
User drops: a factor of under 3
Kernel drops: a factor of 4

Data Miss Rate vs. Block Size for Kernel

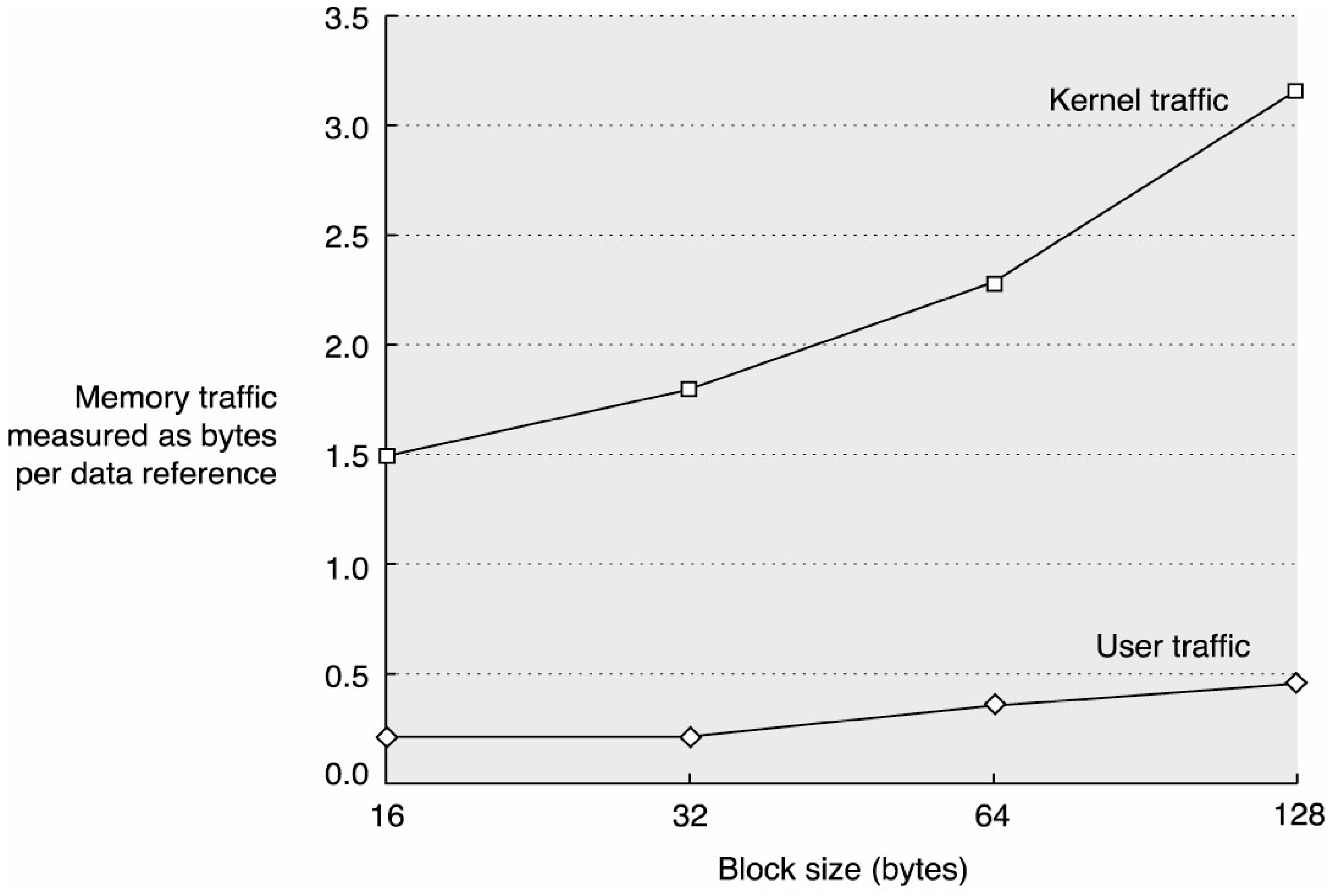


Data Miss Rate vs. Block Size for Kernel (Cont.)

- Compulsory and capacity miss can be reduced with larger block sizes
- Largest improvement is reduction of compulsory miss rate
- Absence of large increases in the coherence miss rate as block size is increased means that false sharing effects are insignificant



Memory Traffic Measured as Bytes per Data Reference



6.5 Distributed Shared-Memory Architecture

The Basic Structure of Distributed-Memory Multiprocessor

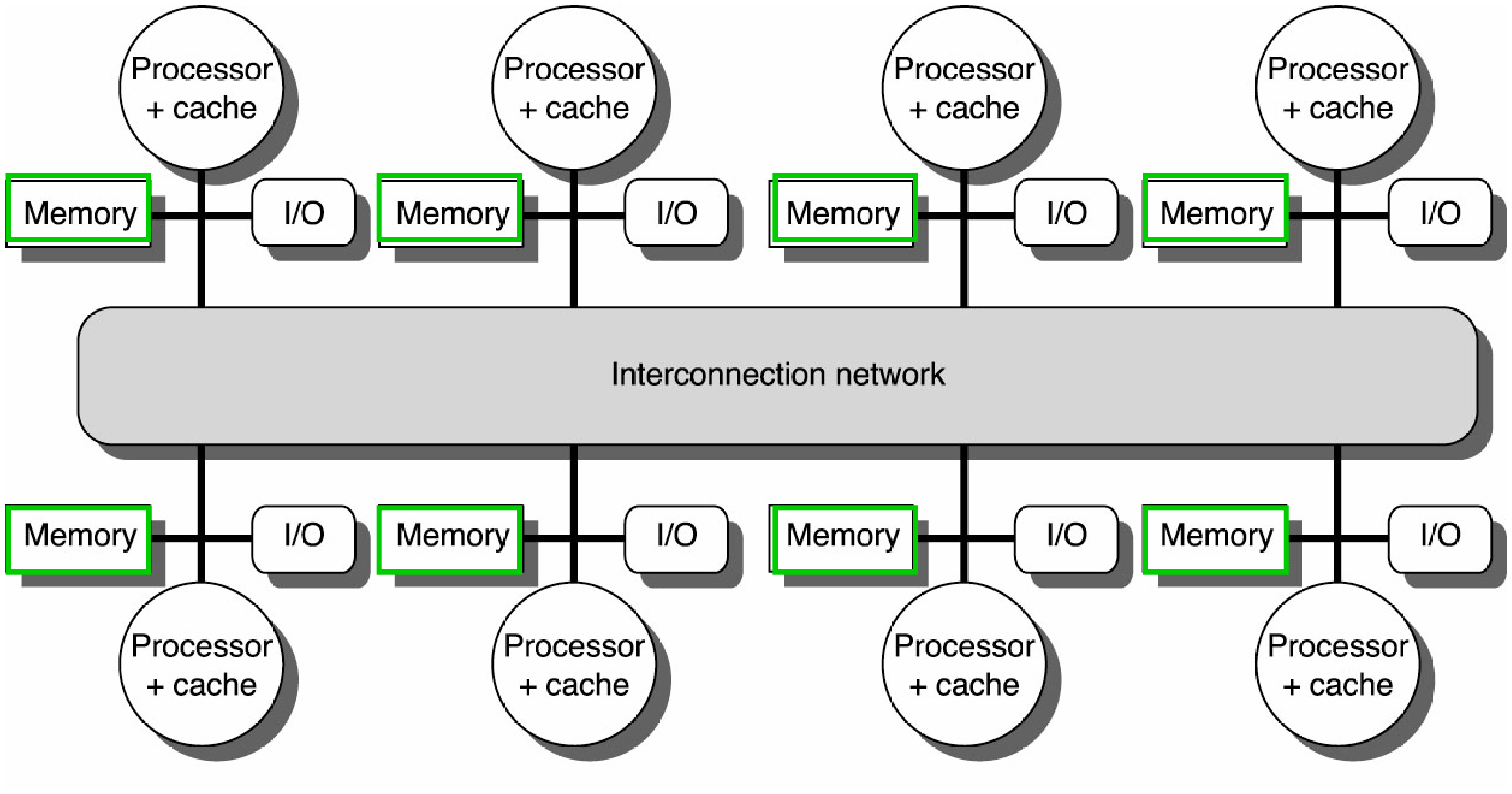


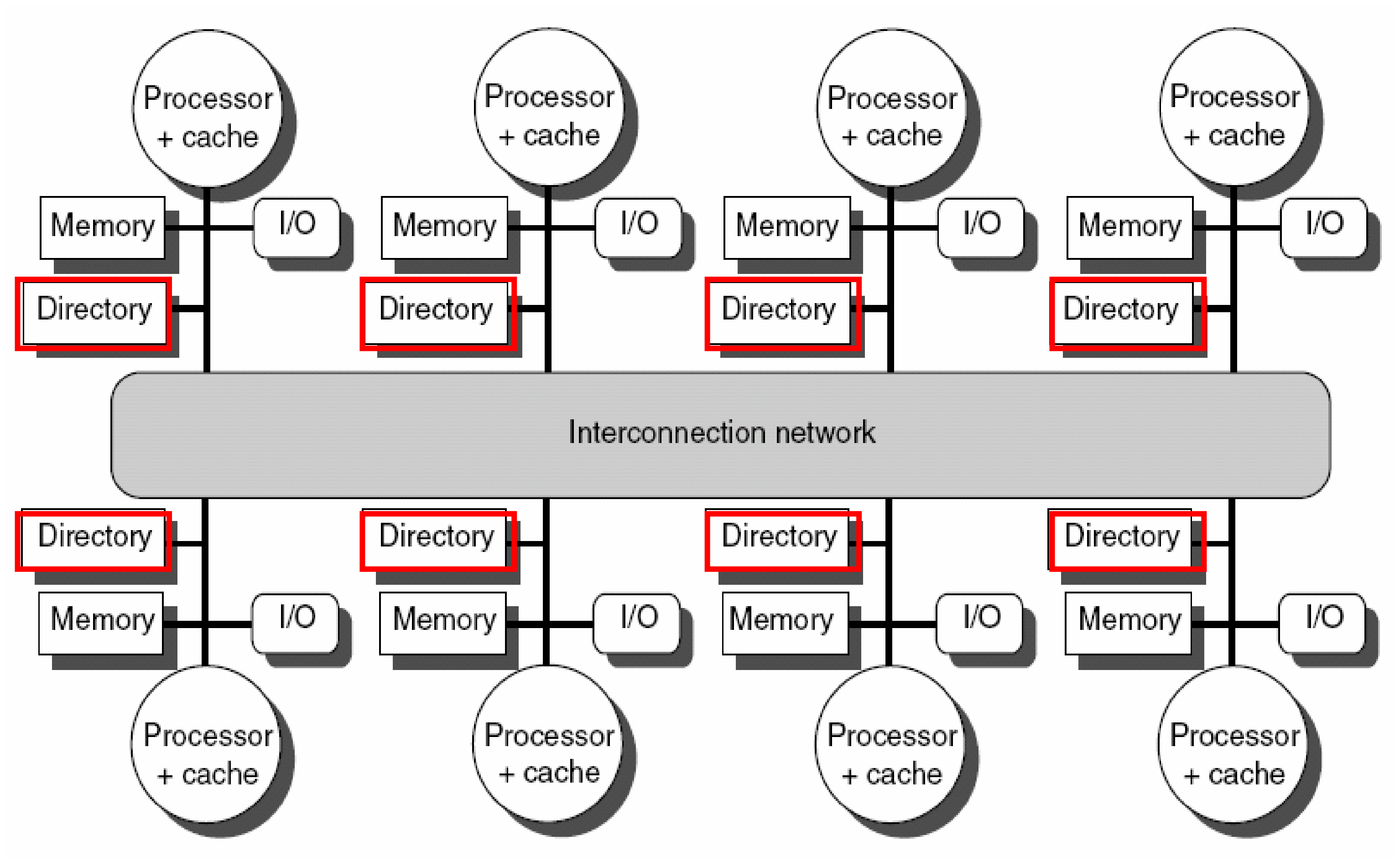
Fig 6.1 Basic structure of a distributed-memory multiprocessor

Directory Protocol

- An alternative coherence protocol
- Directory: track the state of each cache block
  - keeps the state of every block that may be cached
    - *Which caches have copies of the block, whether if is dirty, …*
- Existing directory implementations:
  - Associate an entry in the directory with each memory block
    - *Directory size* ➔ *# memory block $\times$# PEs $\times$ information size*
    - *OK for multiprocessors with less than about 200 PEs*
  - Some method exists for handling more than 200 PEs
  - Some method exists to prevent the directory from becoming the bottleneck
    - *Each PE has a directory to handle its physical memory*

      $\Rightarrow$ *Distributed directory*

Structure of Distributed-Memory Multiprocessor with Distributed Directory

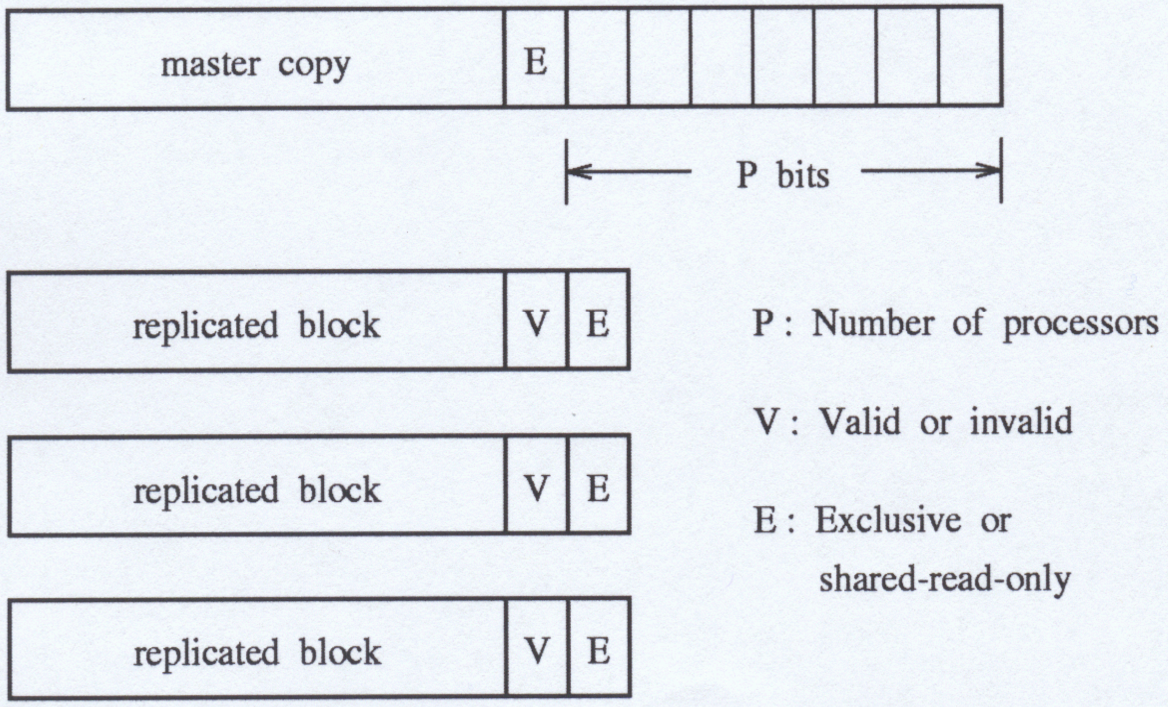


A. Directory-Based Cache Coherence Protocols: The Basics

- Two primary operations
  - Handling a read miss
  - Handling a write to a shared, clean cache block
  - * Handling a write miss to a shared block is the combination of these two.
- Block states in Directory:
  - *Shared*: one or more PEs have the block cached, and the value in memory is update to date (as well as in all caches)
  - *Uncached*: no PE has a copy of the cache block
  - *Exclusive*: exactly one PE has a copy of the cache block, and it has written the block, so the memory copy is out of date
    - *The processor is called the owner of the cache block.*

Directory Structure

- Directory:
  - track the state of each cache block
  - track the processors that have copies of the block when it is shared
  - * *For efficiency, track the state of each cache block at the individual caches.*
  - E.g.:



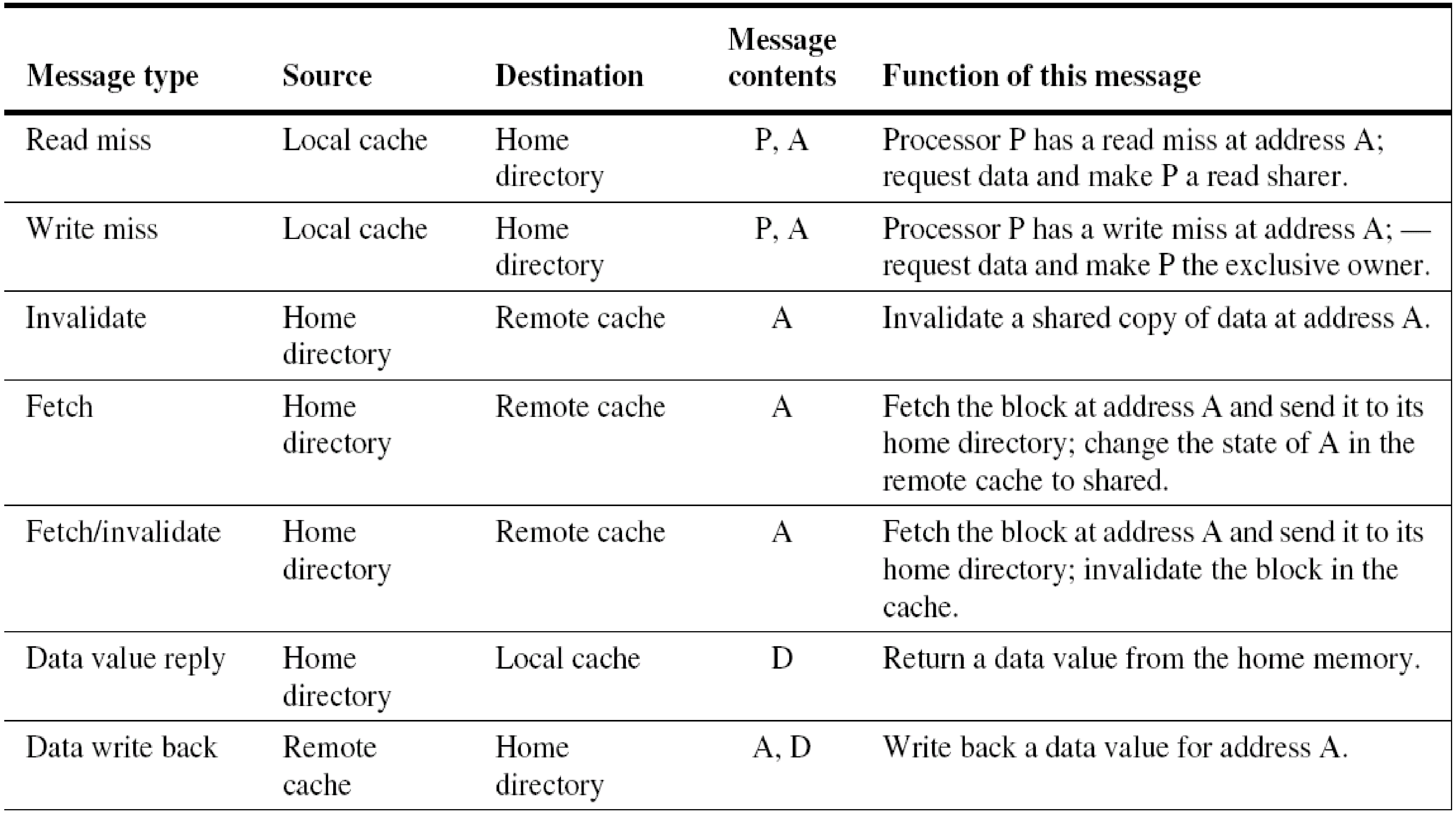
- Assumptions: the same as that made in snooping
  - Attempts to write data that are not exclusive in the writer's cache always generate write misses.
  - The processors block until an access completes.

- Differences b/t directory and snooping:
  - The interconnection is no longer a bus
    - *The interconnect can not be used as a single point of arbitration*
  - No broadcast
    - *Message oriented* $\rightarrow$ *many messages must have explicit responses*
    - *Assumption: all messages will be received and acted upon in the same order they are sent*
      - *Ensure that invalidates sent by a PE are honored immediately*

Types of Nodes

- *Local node*: the node where a request originates
- *Home node*: the node where the memory location and the directory entry of an address reside
  - The physical addr space is statically distributed, so the node that contains the memory and directory for a given physical addr is known.
  - The local node may also be the home node.
  - The directory must be accessed when the home node is the local node.
- *Remote node*: the node that has a copy of a cache block, whether exclusive or shared
  - A remote node may be the same as either the local or the home node.
    - *In such cases, the basic protocol does not change, but inter-processor messages may be replaced w/ intra-processor messages.*

Types of Message Sent Among Nodes

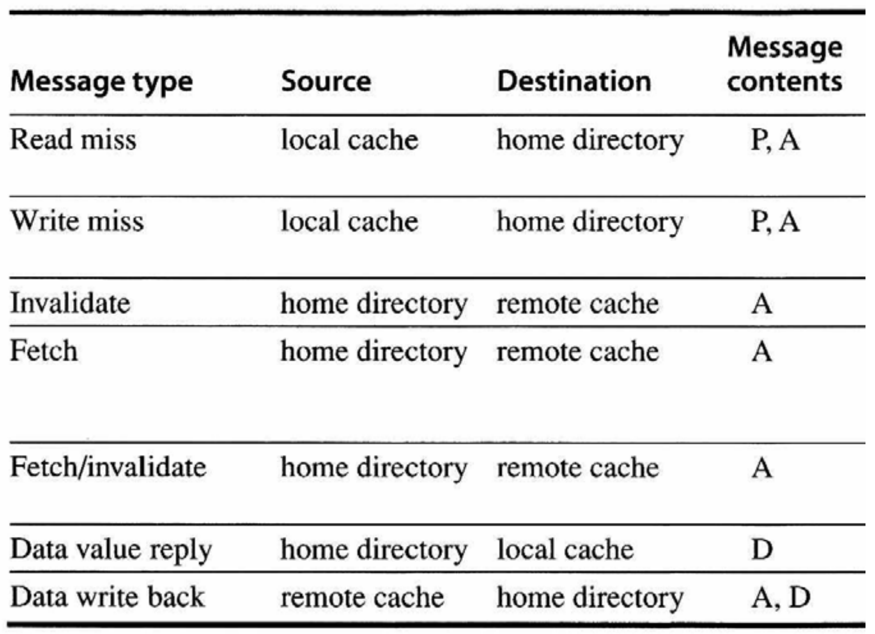


Message type	Source	Destination	Message contents	Function of this message
Read miss	Local cache	Home directory	P, A	Processor P has a read miss at address A; request data and make P a read sharer.
Write miss	Local cache	Home directory	P, A	Processor P has a write miss at address A; — request data and make P the exclusive owner.
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.

Fig 6.28 The possible messages sent among nodes to maintain coherence, along with the source and destination node, the contents (P = requesting processor #, A = requested addr, D = data contents), and the function of the message.

Types of Message Sent Among Nodes (Cont.)

- 1 & 2: miss requests sent by the local cache to the home
- 3 ~ 5: messages sent to a remote cache by the home when the home needs the data to satisfy a read or write miss
- 6: send a value from home back to requesting node
- 7: Data value write backs occur for two reasons
  - A block is replaced in a cache and must be written back to home
  - In reply to fetch or fetch/invalidate messages from home



Message type	Source	Destination	Message contents
Read miss	local cache	home directory	P, A
Write miss	local cache	home directory	P, A
Invalidate	home directory	remote cache	A
Fetch	home directory	remote cache	A
Fetch/invalidate	home directory	remote cache	A
Data value reply	home directory	local cache	D
Data write back	remote cache	home directory	A, D

- P: requesting PE #
- A: requested address
- D: data contents

State Transition Diagram for an Individual Cache Block

Request induced by the local processor shown in black and by the directory shown in gray

Message received:
- read misses
- write misses
- invalidates
- data fetch requests

Message generated:
- read misses
- write misses

(sent to the home directory)

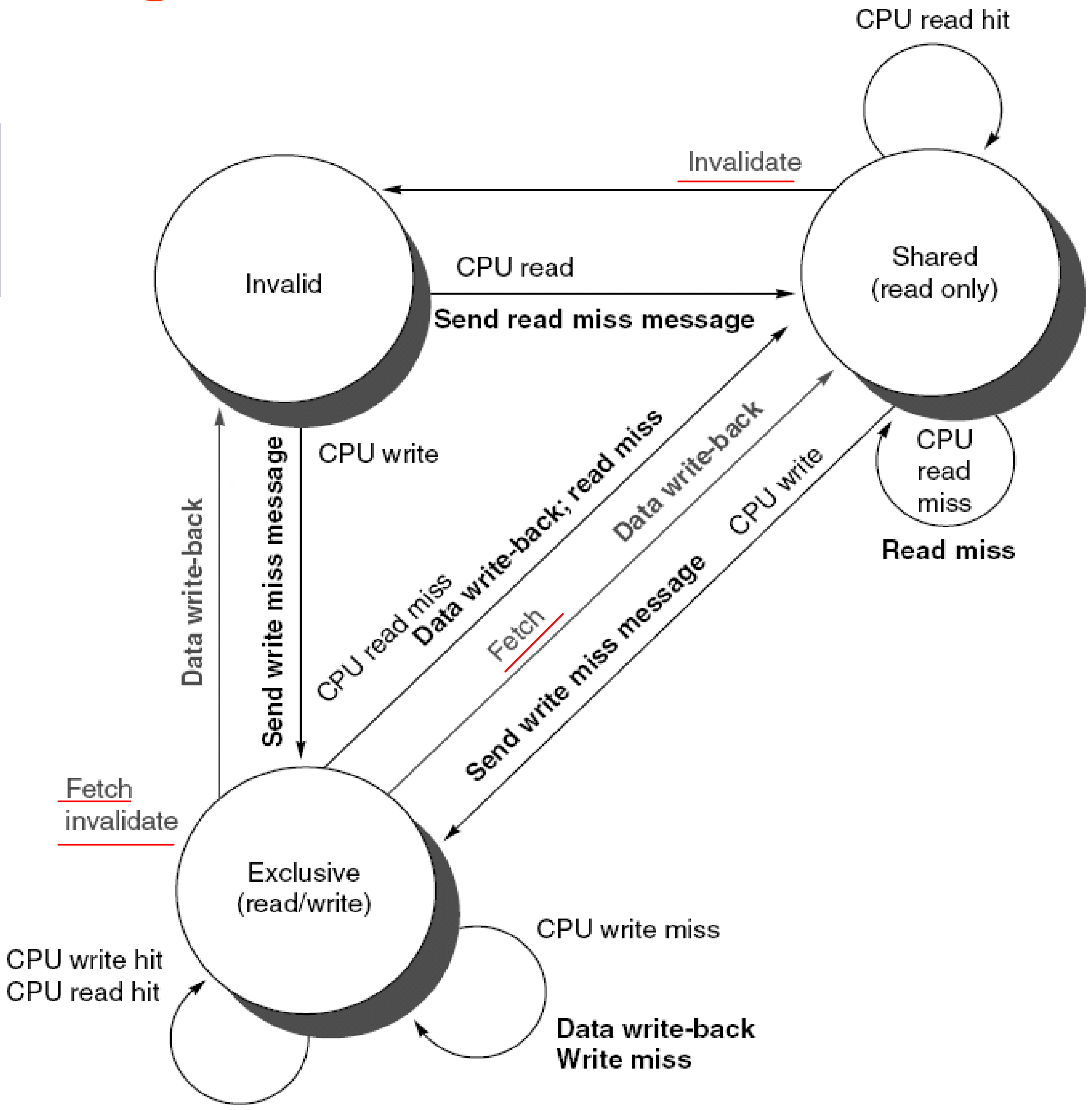
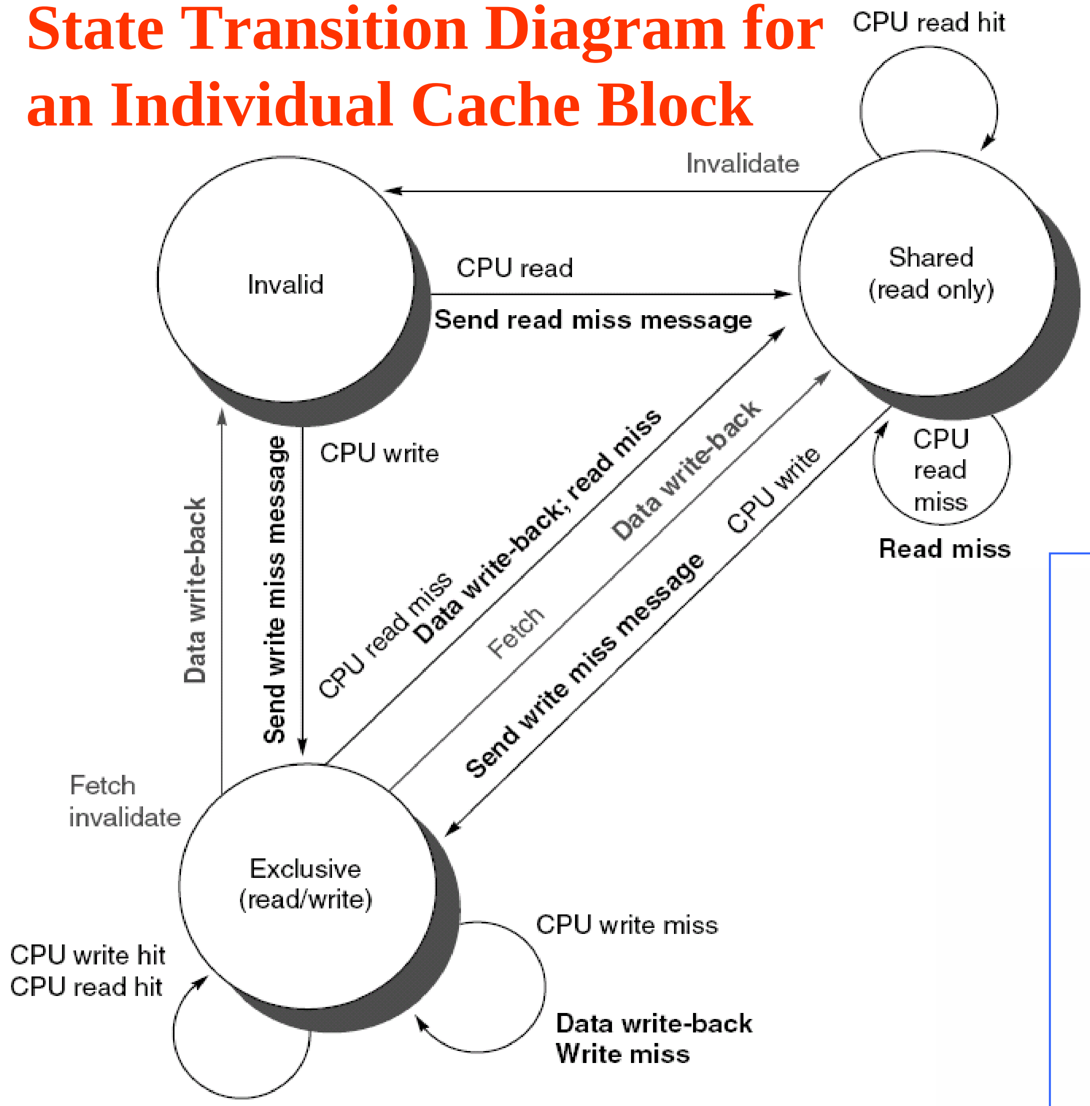


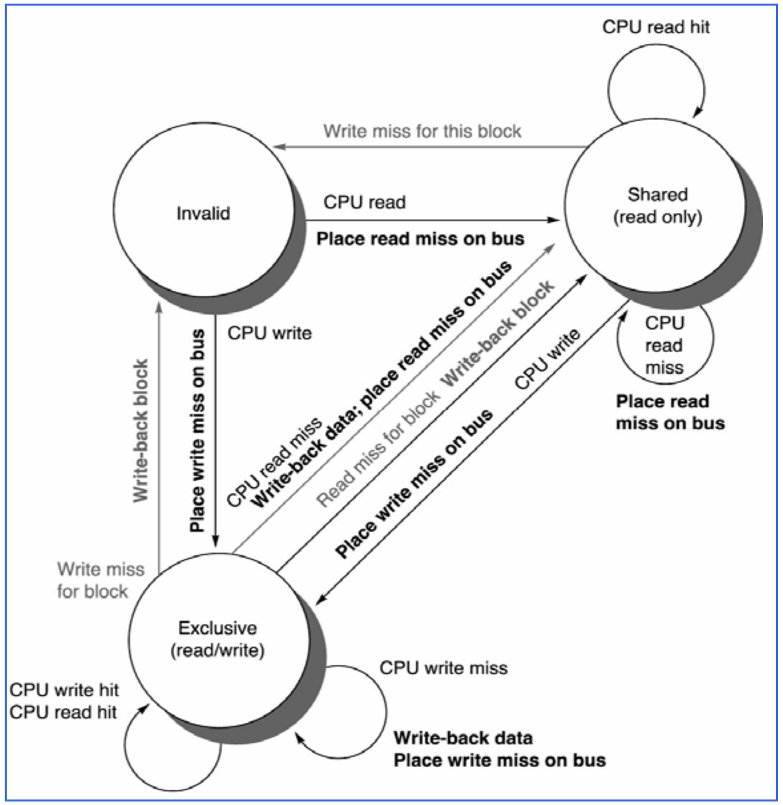
Fig 6.29 State transition diagram for an individual cache block in a directory-based system.

State Transition Diagram for an Individual Cache Block



Request induced by the local processor shown in black and by the directory shown in gray

Cache Coherence State Diagram (*Snooping*) p.6-48

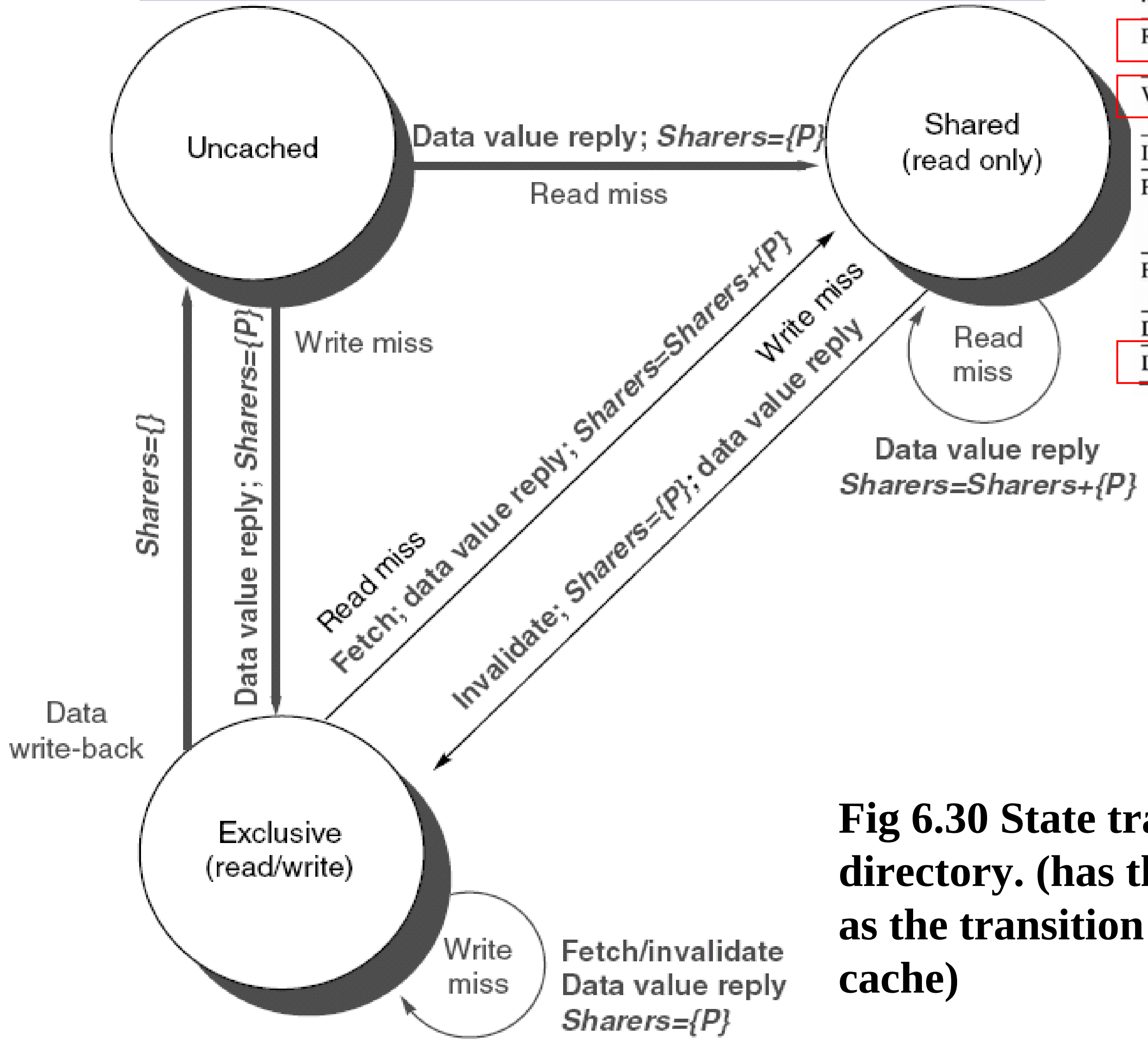


State Transition Diagram for An Individual Cache Block (Cont.)

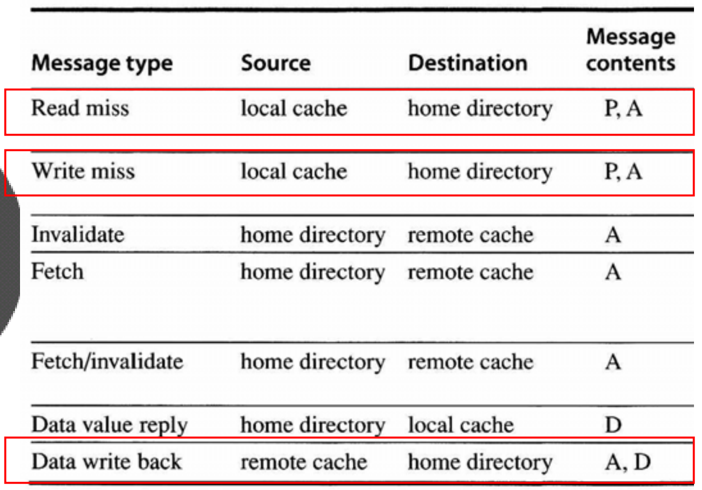
- An attempt to write a shared cache block is treated as a miss. (the same as for snooping)
- Any cache block must be in the exclusive state when it is written, and any shared block must be up to date in memory. (the same as snooping)
- Explicit invalidate and write-back requests replacing the write misses that were formerly broadcast on bus in the snooping scheme. (different from that of snooping)
  - Data fetch + invalidate operations that are selectively sent by the directory controller.

State Transition Diagram for the Directory

Sharers: PEs having the cache block



Uncached
Data value reply; Sharers={P}
Read miss
Shared (read only)
Read miss
Data value reply Sharers=Sharers+{P}
Write miss
Data value reply; Sharers={P}
Sharers={}
Data write-back
Read miss Fetch; data value reply; Sharers=Sharers+{P}
Write miss Invalidate; Sharers={P}; data value reply
Exclusive (read/write)
Write miss
Fetch/invalidate Data value reply Sharers={P}



Message type	Source	Destination	Message contents
Read miss	local cache	home directory	P, A
Write miss	local cache	home directory	P, A
Invalidate	home directory	remote cache	A
Fetch	home directory	remote cache	A
Fetch/invalidate	home directory	remote cache	A
Data value reply	home directory	local cache	D
Data write back	remote cache	home directory	A, D

Message received:
- read misses
- write misses
- data write back

Fig 6.30 State transition diagram for the directory. (has the same states and structure as the transition diagram for an individual cache)

State Transition Diagram for the Directory (Cont.)

- A msg sent to a directory causes 2 different types of actions: updates of the directory state and sending additional msgs to satisfy the request.
- The directory state indicates the state of all the cached copies of a memory block, rather than for a single cache block.
- The directory must track the set (*Sharers*) of processors that have a copy of a block.
- Assumption: some actions are atomic, e.g.: requesting a value and sending it to another node (not realistic)