# Chapter 5
# Thread-Level Parallelism

## Introduction

- Multiprocessor was growing throughout the 1990s as severs and supercomputers were built
- Some factors reinforced the trend toward multiprocessor
  - A growing interest in severs and supercomputers
  - A growth in data-intensive applications
  - Increasing performance on the desktop is less important
  - More understanding of multiprocessors
  - Replication rather than unique design leverages a design investment
- Multiprocessor architecture is a large and diverse field, and much of the field is in its youth, with ideas coming and going and, until very recently, more architectures failing than succeeding

## Taxonomy of Parallel Architectures

- At 40 years ago, Flynn proposed a model of categorizing all computers that is still useful today
  - —*Single instruction stream*, *single data stream (SISD )*
  - —*Single instruction stream*, *multiple data streams (SIMD )*
  - —*Multiple instruction streams*, *single data stream (MISD )*
  - —*Multiple instruction streams*, *multiple data streams (MIMD )*
- In fact, some multiprocessors are hybrids of these categories

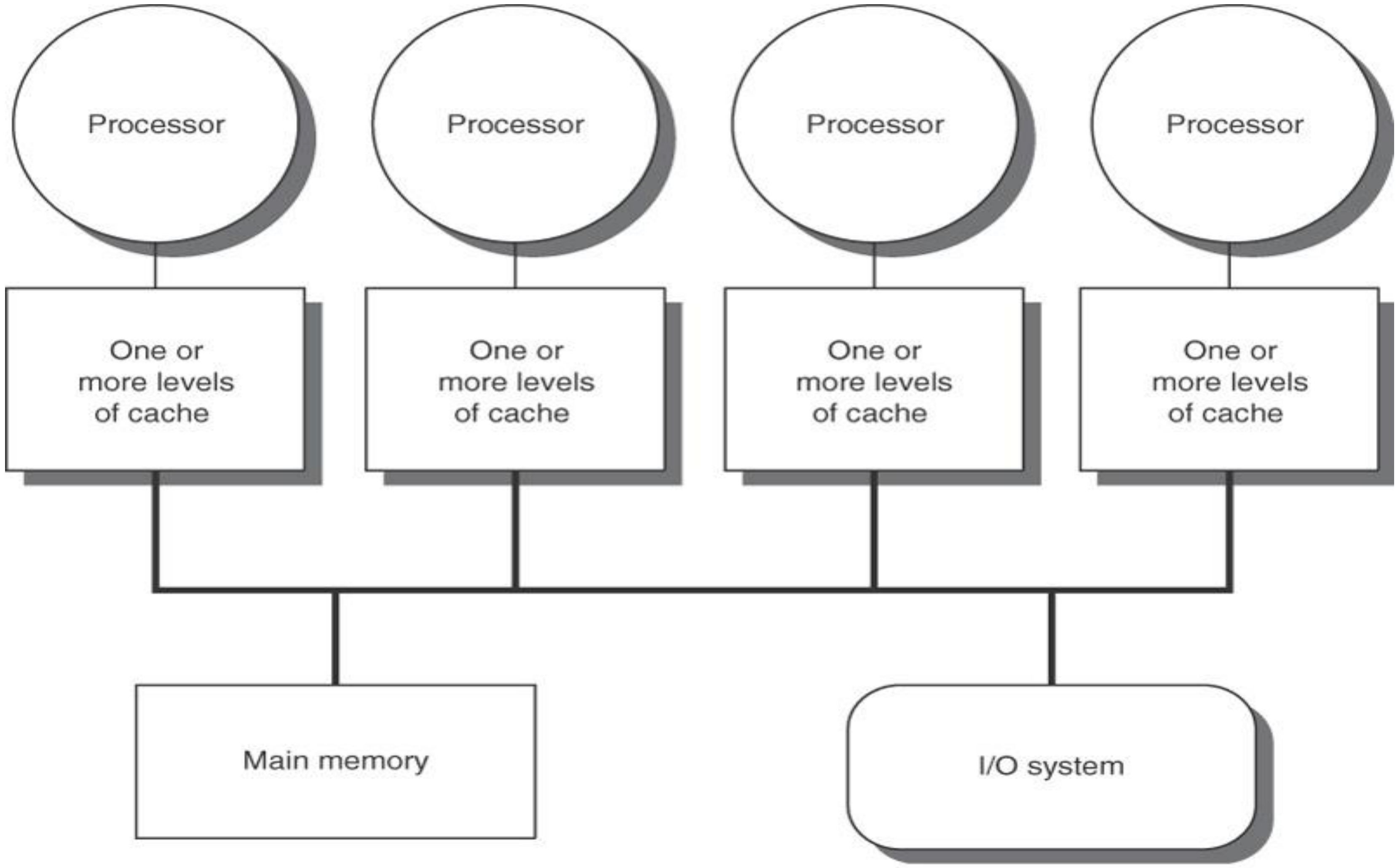
## MIMD (1)

- MIMID can exploit TLP
- Two factors arise MIMD
  - It offers flexibility
  - It has the edge of cost-performance advantages
- Clusters is a popular one of MIMDs
  - They focus on parallel applications
  - A significant amount of communications required during computations
- *On-chip multiprocessing*, *single-chip multiprocessing*, or *multicore*
  - IBM power5, Sun T1, Intel Pentium D, and Xeon-MP are multicore and multithreaded

## MIMD (2)

- With an MIMD, each processor has its own process and instruction stream
- A multiprocessor with *n* processors, *n* threads or processors are usually executed
  - *Threads*: multiple loci of execution
  - How to utilize threads?
  - *Grain size* (computation amount) is important to exploit TLP efficiently
- Two classes of MIMD multiprocessors, depending on # of processors, memory organization, and interconncet strategy
  - Centralized shared-memory architectures
  - Those with distributed memory

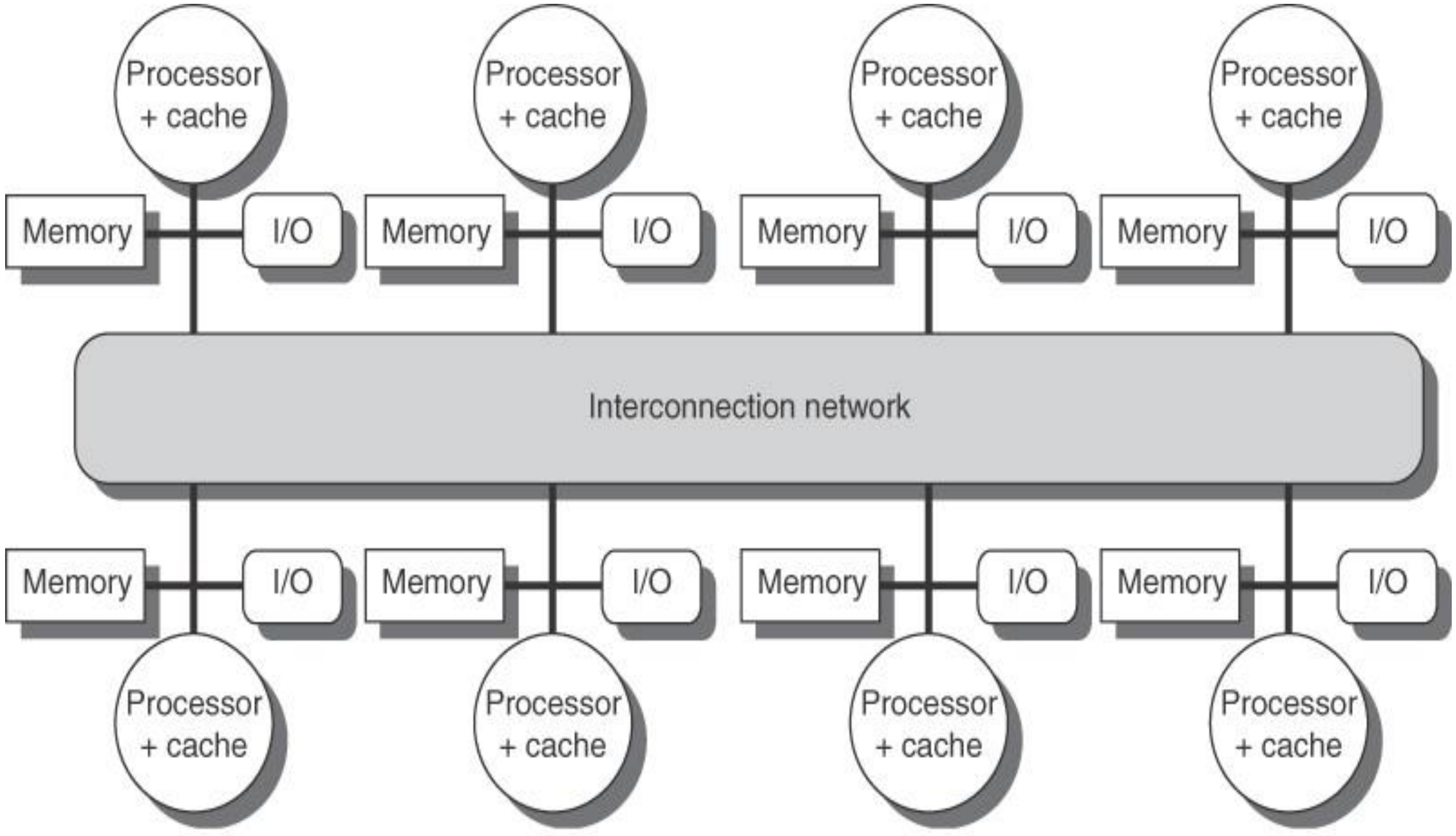
## Basic Structure of Centralized Shared-Memory Multiprocessors


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

## Features of Centralized Shared-Memory Multiprocessors

- It usually contains less than 100 cores
- For small scales, sharing a memory is possible
  - With large caches, a single memory, possibly with multiple banks, can satisfy the requirements
- Using multiple point-to-point connections, or switches, and adding additional memory banks, a centralized shared-memory design can be scaled to a few dozen processors
- *Symmetric (shared-memory) multiprocessors (SMPs)*, has a symmetric relationship to all processors and a uniform access time from any processor
  - This style of architecture is called *uniform memory access (UMA)*

## Basic Structure of Destributed-Memory Multiprocessors



## Features of Destributed-Memory Multiprocessors

- Two factors limit the size
  - Rapid increase in performance
  - Significant requirements of memory bandwidth
- The large number of processors raises the need for a high-performance interconnect
- Two advantages of distributing the memory among nodes
  - It is cost-effective to scale the memory bandwidth if most accesses are local
  - It reduces the latency for accesses to the local memory
- The key disadvantage is complicated communication

## Models for communication and Memory Architecture (1)

- The first one is the communication via shared-memory
  - These multiprocessors are called *distributed shared-memory (DSM)* or *nonuniform memory access (NUMAs)*
  - The physically separate memories can be addressed as one logically shared address space
  - Data are communicated implicitly via load and store operations

## Models for communication and Memory Architecture (2)

- The second one is *message-passing multiprocessor*
  - —The address space consisting of many private spaces are logically disjoint and cannot be addressed by a remote processor
  - —The same physical address on two processors refers to two different locations in two different memories

## Challenges of Parallel Processing

- The APs of multiprocessors ranges from running independent programs (no communication) to running parallel programs with communication.
- Two important hurdles
  - The limited parallelism available in programs
  - The high cost of communications
    - The cost ranges from 50 clock cycles (multicore) to over 1,000 clock cycles (large-scale multiprocessors), depending on the communication mechanism, the type of interconnection network, and the scale of the multiprocessor

## Example 1

- Suppose you want to achieve a speedup of 80 with 100 processors. What fraction of the original computation can be sequential?
- Answer:

$$\text{Speedup}_{\text{overall}} = \frac{\text{Execution time}_{\text{old}}}{\text{Execution time}_{\text{new}}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

$$80 = \frac{1}{\frac{\text{Fraction}_{\text{parallel}}}{100} + (1 - \text{Fraction}_{\text{parallel}})}$$

Simplifying this equation yields

$$\begin{aligned} 0.8 \times \text{Fraction}_{\text{parallel}} + 80 \times (1 - \text{Fraction}_{\text{parallel}}) &= 1 \\ 80 - 79.2 \times \text{Fraction}_{\text{parallel}} &= 1 \\ \text{Fraction}_{\text{parallel}} &= \frac{80 - 1}{79.2} \\ \text{Fraction}_{\text{parallel}} &= 0.9975 \end{aligned}$$

## Example 2

- Suppose we have an application running on a 32-processor multiprocessor, which has a 200ns time to handle reference to a remote memory. For this application, assume that all the references except those involving communication hit in the local memory hierarchy, which is slightly optimistic. Processors are stalled on a remote request, and the processor clock rate is 2 GHz. If the base CPI (assuming that all references hit in the cache) is 0.5, how much faster is the multiprocessor if there is no communication versus if 0.2% of the instructions involve a remote communication reference?

## Answer:

CPI = Base CPI + Remote request rate × Remote request cost

= 0.5 + 0.2% × Remote request cost

The remote request cost is

Remote access cost / Cycle time

= 200ns / 0.5ns = 400 cycles

Thus, CPI = 0.5 + 0.8 = 1.3

The multiprocessor with all local references is 1.3/0.5 = 2.6 times faster

## Two major problems

- Inefficient parallelism
  - It can be attacked primarily in software with new algorithms that can have better parallel performance
  - Multithreading
- Long-latency remote communication
  - It can be attacked by the architecture and by the programmer
  - HW: caching shared data, multithreading, prefetching
  - SW: restructuring the data, smarter algorithms

## Symmetric Shared-Memory Architectures (1)

- The use of large, multilevel caches substantially reduce the memory bandwidth demands of a processor
- For example
  - —IBM in 2000: on-chip multiprocessor
  - —AMD and Intel: two-processor
  - —Sun: T1 (eight-processor) in 2006

## Symmetric Shared-Memory Architectures (2)

- Small-scale shared-memory machines usually support the caching of both shared and private data
  - Private data is used by a single processor
  - Shared data is used by multiple processors, essentially providing communication among the processors through reads and writes of the shared data
- When a private item is cached, its location is migrated to the cache, reducing the average access time as well as the memory bandwidth required

## Multiprocessor Cache Coherence

- The introduction of caches caused a coherence problem for I/O operations
    - since the view of memory through the cache could be different from the view of memory obtained through the I/O subsystem
- The same problem exists in the case of multiprocessors, because the view of memory held by two different processors is through their individual caches

## Cache-coherence Problem

- It may be caused by two different processors accessing the same location see different values

| 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 |

## Cache Coherence (1)

- Informally, we could say that a memory system is coherent if any read of a data item returns the most recently written value of that data item
  - —This definition is intuitively appealing, vague and simplistic
  - —The reality is much more complex

## Cache Coherence (2)

- This simple definition contains two different aspects of memory system behavior, both of which are critical to writing correct shared-memory programs
- *Coherence* defines what values can be returned by a read
- *Consistence* determines when a written value will be returned by a read

## Cache Coherence (3)

- A memory system is coherent if
  - A read by a processor P to a location X that follows a write by P to X, with no writes of X by another processor occurring between the write and the read by P, always returns the value written by P
  - A read by a processor to a location 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 on other writes to X occur between the two accesses
  - 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 the values 1 and then 2 are written to a location, processors can never read the value of the location as 2 and then later read it as 1.

## Cache Coherence (4)

- The first preserves the program order
- The second defines the notion of memory coherence
- Serializing the writes ensures that every processor will see the write done by some processor at some point
- Suppose we did not serialize writes, and processor P1 writes location X followed by P2 writing location X

## Cache Coherence (5)

- Serializing the writes ensures that every processor will see the write done by P2 at some point
- If we did not serialize the writes, some processor could see the write of P2 first and then see the write of P1, maintaining the value written by P1 indefinitely
- *Write serialization*: The simplest way to avoid such difficulties is to serialize writes, so that all writes to the same location are seen in the same order

## Cache Coherence (6)

- The question of when a written value will be seen is also important
  - —For example, we cannot require that a read of X instantaneously see the value written for X by some other processor
  - —For example, a write of X on one processor precedes a read of X on another processor by a very small time
- *Memory consistency model* : The issue of exactly *when a written value must be seen by a reader*

## Cache Coherence (7)

- Coherence and consistency are complementary
  - 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.
- Assume
  - We require that a write does not complete until all processors have seen the effect of the write
  - The processor does not change the order of any write with any other memory access
- This allows the processor to reorder reads, but forces the processor to finish a write in program order

## Basic Schemes for Enforcing Coherence (1)

- The coherence problem for multiprocessors and I/O, although similar in origin, has different characteristics that affect the appropriate solution
- Unlike I/O, where multiple data copies are a rare event
- A program running on multiple processors will normally have copies of the same data in several caches
- In a coherent multiprocessor, the caches provide both *migration* and *replication* of shared data items

## Basic Schemes for Enforcing Coherence (2)

- Coherent caches provide migration, since a data item can be moved to a local cache and used there in a transparent fashion
  - This migration reduces both the latency to access a shared data item that is allocated remotely and the bandwidth demand on the shared memory
- Coherent caches also provide replication for shared data that is being simultaneously read, since the caches make a copy of the data item in the local cache

## Basic Schemes for Enforcing Coherence (3)

- Replication reduces both latency of access and contention for a read shared data item
- Supporting this migration and replication is critical to performance in accessing shared data
- Thus, rather than trying to solve the problem by avoiding it in software, small-scale ultiprocessors adopt a hardware solution by introducing a protocol to maintain coherent caches.

## Cache Coherence Protocols

- The protocols to maintain coherence for multiple processors are called *cache coherence protocols*
- Key to implementing a cache-coherence protocol is tracking the state of any sharing of a data block

## Two Classes of Protocols (1)

- *Directory based:* The sharing status of a block of physical memory is kept in just one location, called the *directory*
  - —It has slightly implementation overhead than snooping
  - —It can scale to larger processor counts

## Two Classes of Protocols (2)

- *Snooping:* 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
  - —The caches are usually on a shared-memory bus
  - —All cache controllers monitor or *snoop* on the bus to determine whether or not they have a copy of a block that is requested on the bus

## Snooping Protocols (1)

- Snooping protocols became popular with multiprocessors using microprocessors and caches attached to a single shared memory
- These protocols can use a preexisting physical connection—the bus to memory—to interrogate the status of the caches

## Snooping Protocols (2)

- There are two ways to maintain the coherence requirement
- The first one is *write invalidate*
  - —It ensures that a processor has exclusive access to a data item before it writes that item
  - —This style of protocol is called a *protocol* because it invalidates other copies on a write
  - —It can be used for snooping and for directory schemes
  - —It enforces write serialization
  - —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

## Snooping Protocols (3)

- To see how this protocol ensures coherence, 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 in the cache and is forced to fetch a new copy of the data
- For a write, we require that the writing processor have exclusive access

## Snooping Protocols (4)

- If two processors do attempt to write the same data simultaneously, one of them wins the race
  - —The other processor's copy must 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

## Snooping Protocols (5)

- It works on a snooping bus for a single cache block (X) with write-back caches.
- It may be caused by two different processors accessing the same location see different values

| Processor Activity | Event | 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 |

## Snooping Protocols (6)

- *Write update* or *write broadcast* protocol: The alternative to an invalidate protocol is to update all the cached copies of a data item when that item is **written**
- To keep the bandwidth requirements of this protocol under control it is useful to track whether or not a word in the cache is shared
- That is, it is contained in other caches
- If it is not, then there is no need to broadcast or update any other caches
- It is performance critical

## Basic Implementation Techniques (1)

- The key to implementing an invalidate protocol in a small-scale multiprocessor is the use of the bus to perform invalidates
- To perform an invalidate the processor simply acquires
  - —bus access
  - —broadcasts the address to be invalidated on the bus
- 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 is invalidated

## Basic Implementation Techniques (2)

- When a write to a block that is shared occurs, the writing processor must acquire bus access to broadcast its invalidation
- The serialization of access enforced by the bus also forces serialization of writes,
  - —Since when two processors compete to write to the same location, one must obtain bus access before the other
  - —The first processor to obtain bus access will cause the other processor’s copy to be invalidated, causing writes to be strictly serialized

## Basic Implementation Techniques (3)

- One implication of this scheme is that a write to a shared data item cannot complete until it obtains bus access
- All coherence schemes require some method of serializing accesses to the same cache block, either by
  - —serializing access to the communication medium or
  - — another shared structure

## Basic Implementation Techniques (4)

- 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
  - Since all written data are always sent to the memory, from which the most recent value of a data item can always be fetched

## Basic Implementation Techniques (5)

- For a write-back cache, however, the problem of finding the most recent data value is harder
  - —Since the most recent value of a data item can be in a cache rather than in memory
- Happily, write-back caches can use the same snooping scheme both for caches misses and for writes:
  - —Each processor snoops every address placed on the bus

## Basic Implementation Techniques (6)

- 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
- We focus on implementation with write-back caches since
  - —write-back caches generate lower requirements for memory bandwidth
  - —they are greatly preferable in a multiprocessor despite the slight increase in complexity

## Basic Implementation Techniques (7)

- Complexity arise from
  - Implementation to maintain coherence
  - Remote access to the cache block

## Basic Implementation Techniques (8)

- The normal cache tags can be used to implement the process of snooping, and the valid bit for each block makes invalidation easy to implement
- Read misses are also straightforward since they simply rely on the snooping capability

## Basic Implementation Techniques (9)

- For writes we'd like to know whether any other copies of the block are cached
- If there are no other cached copies, then the write need not be placed on the bus in a write-back cache
- Not sending the write reduces both the time taken by the write and the required bandwidth

## Basic Implementation Techniques (10)

- To track whether or not a cache block is shared, we can add an extra bit just like valid bit and 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 private
- No further invalidations will be sent by that processor for that block

## Basic Implementation Techniques (10)

- The processor with the sole copy of a cache block is normally 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

## Basic Implementation Techniques (11)

- 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
- Every bus transaction must check the cache-address tags, which could potentially interfere with CPU cache accesses
  - — This potential interference is reduced by duplicating the tags or employing a multilevel cache with *inclusion*

## Basic Implementation Techniques (12)

- *Inclusion*: whereby the levels closer to the CPU are a subset of those further away
- If the tags are duplicated, then the CPU and the snooping activity may proceed in parallel
- If the CPU uses a multilevel cache with the inclusion property, then every entry in the primary cache is required to be in the secondary cache
- Thus the snoop activity can be directed to the second-level cache, while most of the processor’s activity is directed to the primary cache

## Basic Implementation Techniques (13)

- If the snoop gets a hit in the secondary cache, then it must arbitrate for the primary cache to update the state and possibly retrieve the data, which usually requires a stall of the processor.
- Since many multiprocessors use a multilevel cache to decrease the bandwidth demands of the individual processors, this solution has been adopted in many designs
- Sometimes it may even be useful to duplicate the tags of the secondary cache to further decrease contention between the CPU and the snooping activity

## Example Protocol (1)

- A snooping coherence protocol is usually implemented by incorporating a finite state controller in each node
- This controller responds to requests from the processor and from the bus, changing the state of the selected cache block, as well as using the bus to access data or to invalidate it
- Snooping operations or cache requests for different blocks can proceed independently

## Example Protocol (2)

- In actual implementations, a single controller allows multiple operations to distinct blocks to proceed in interleaved fashion
- The simple protocol has three states: invalid, shared, and modified

## Example Protocol (3): Three States

- *Invalid*: the block is invalid for other processors
- *Shared*: the block is potentially shared
- *Modified*: the block has been updated in the cache
  - It implies the block is exclusive

## Example Protocol (4)

- When the write miss is placed on the bus, any processors with copies of the cache block invalidate it
- In a write-through cache, the data for write miss can always be retrieved from the memory
- In a write-back cache, if the block is exclusive in just one cache, that cache also writes back the block ; otherwise, the data can always be read from the memory

## Example Protocol (5)

- Treating write hits to shared blocks as cache misses reduces the number of different bus transactions and simplifies the controller
- In more sophisticated protocols, these “misses” are treated as upgrade requests that generate a bus transaction and an invalidate
- But it does not actually transfer the data, since the copy in he cache is up-to-date

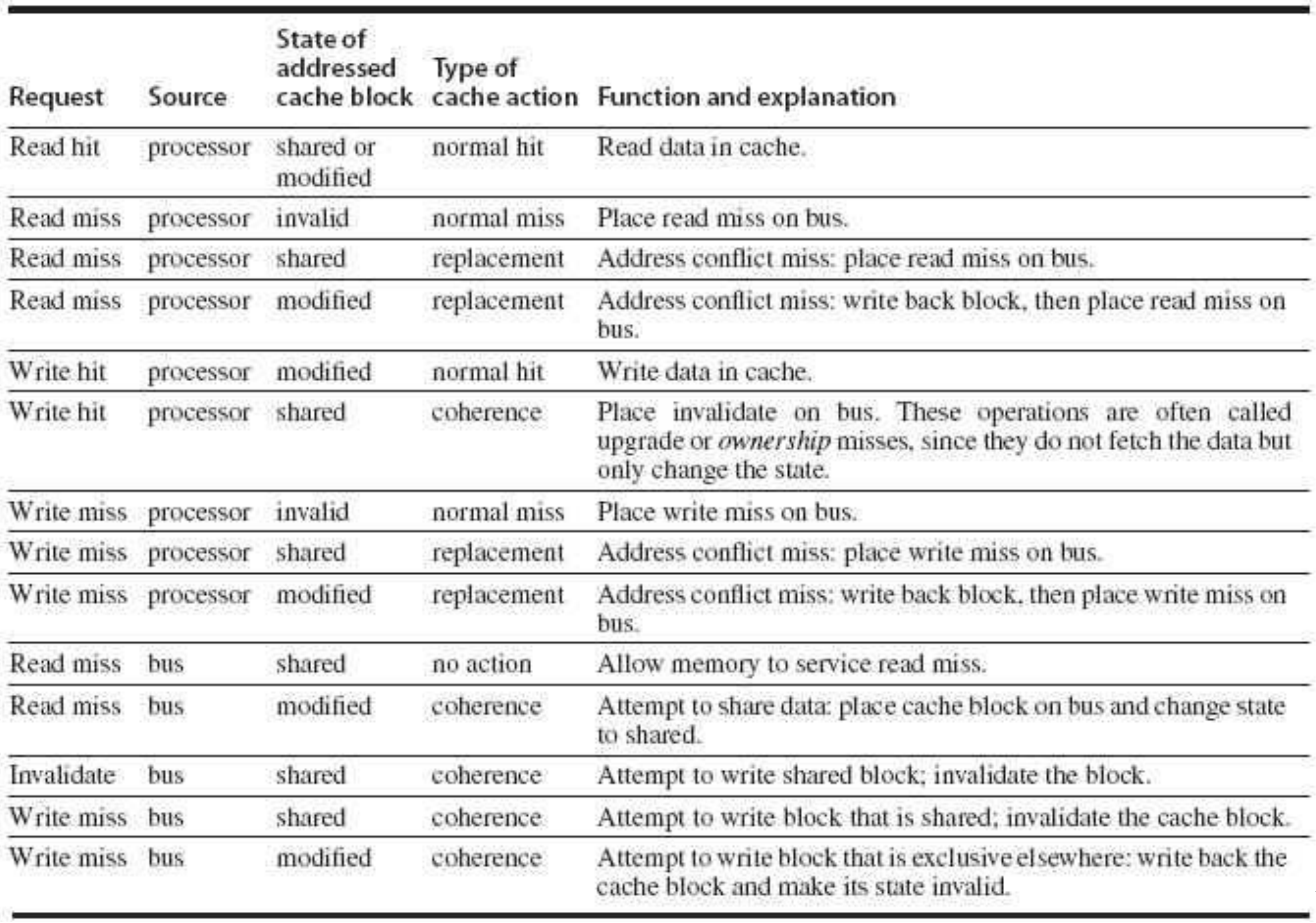
## Example Protocol (6)

- Any valid cache block is either in the shared state in multiple caches or in the exclusive state in exactly one cache
- Any transition to the exclusive state (which is required for a processor to write to the block) requires a write miss to be placed on the bus, causing all caches to make the block invalid

## Example Protocol (7)

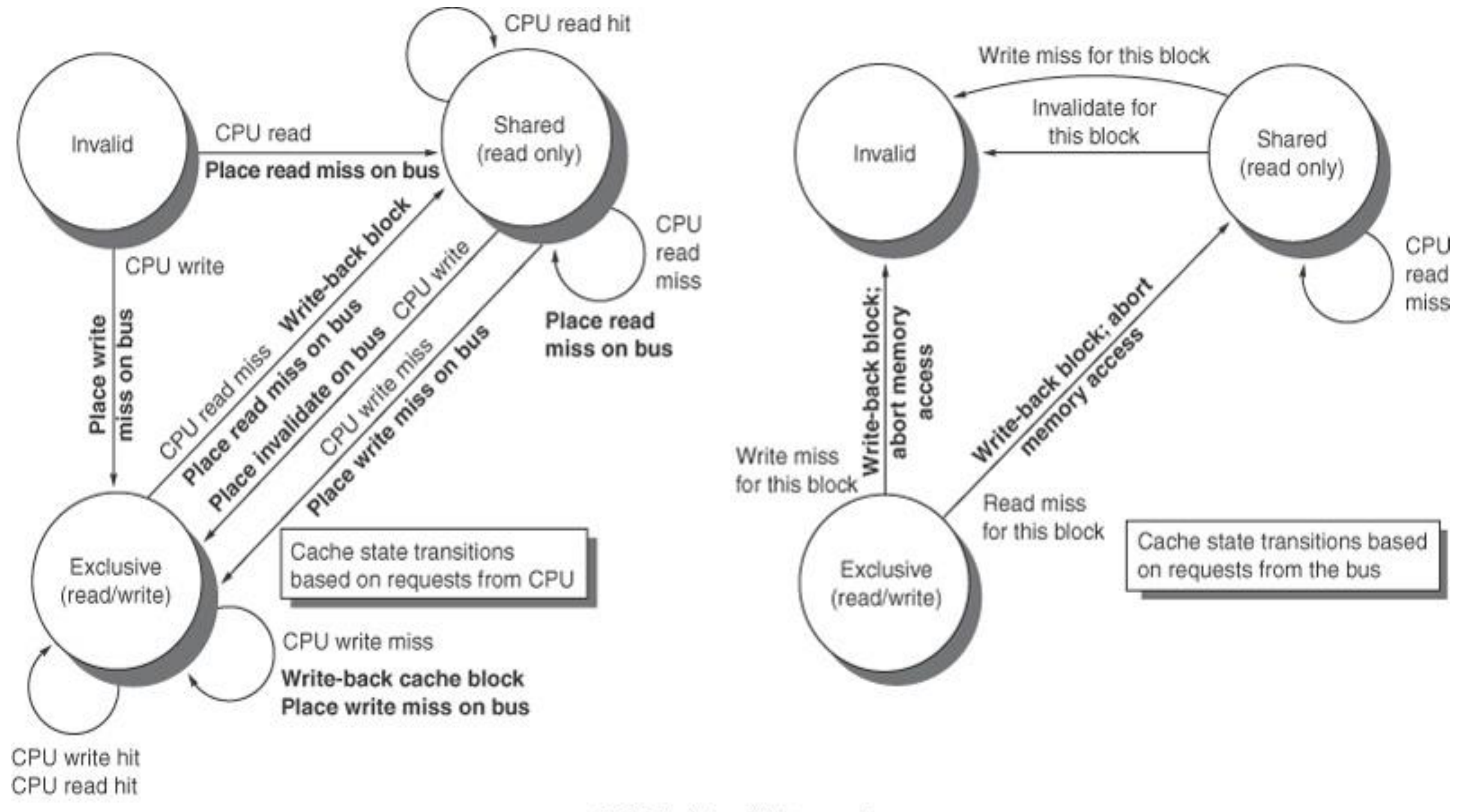
- In addition, if some other cache had the block in exclusive state, that cache generates a write back with the desired address
- Finally, if a read miss occurs on the bus to a block in the exclusive state, the owning cache also makes its state shared, forcing a subsequent write to require exclusive ownership

# Cache Coherence Mechanism

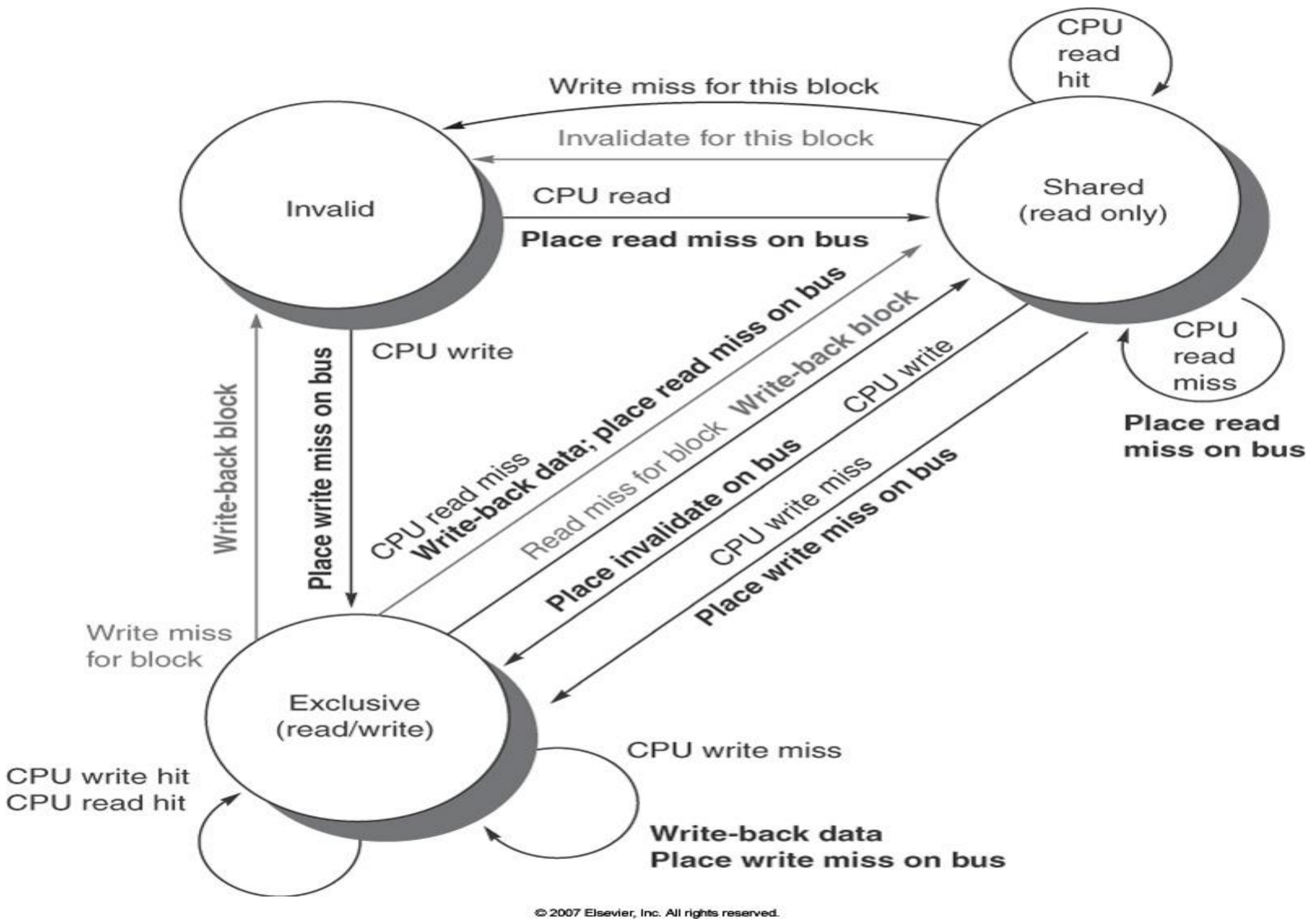


| Request | Source | State of addressed cache block | Type of cache action | Function and explanation |
|---|---|---|---|---|
| Read hit | processor | shared or modified | normal hit | Read data in cache. |
| Read miss | processor | invalid | normal miss | Place read miss on bus. |
| Read miss | processor | 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 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. |
| Read miss | bus | shared | no action | Allow 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 block that is shared; 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. |

## A Finite-State Transition Diagram



## A Cache-Coherence State Diagram



## Some Issues (1)

- Although our simple cache protocol is correct, it omits a number of complications that make the implementation much trickier
- The most important of these is that the protocol assumes that operations are *atomic*
  - For example, the protocol described assumes that write misses can be detected, acquire the bus, and receive a response as a single atomic action
- In reality this is not true
- Similarly, if we used a split transaction bus, as most modern bus based multiprocessors do, then even read misses would also not be atomic

## Some Issues (2)

- Nonatomic actions introduce the possibility that the protocol can *deadlock*
- Constructing small (2-4) processor bus-based multiprocessors has become very easy
  - —F. g., Pentium 4 and AMD Opteron
  - —They support snooping for 2-4 processors
  - —They also have larger caches to reduce bus utilization

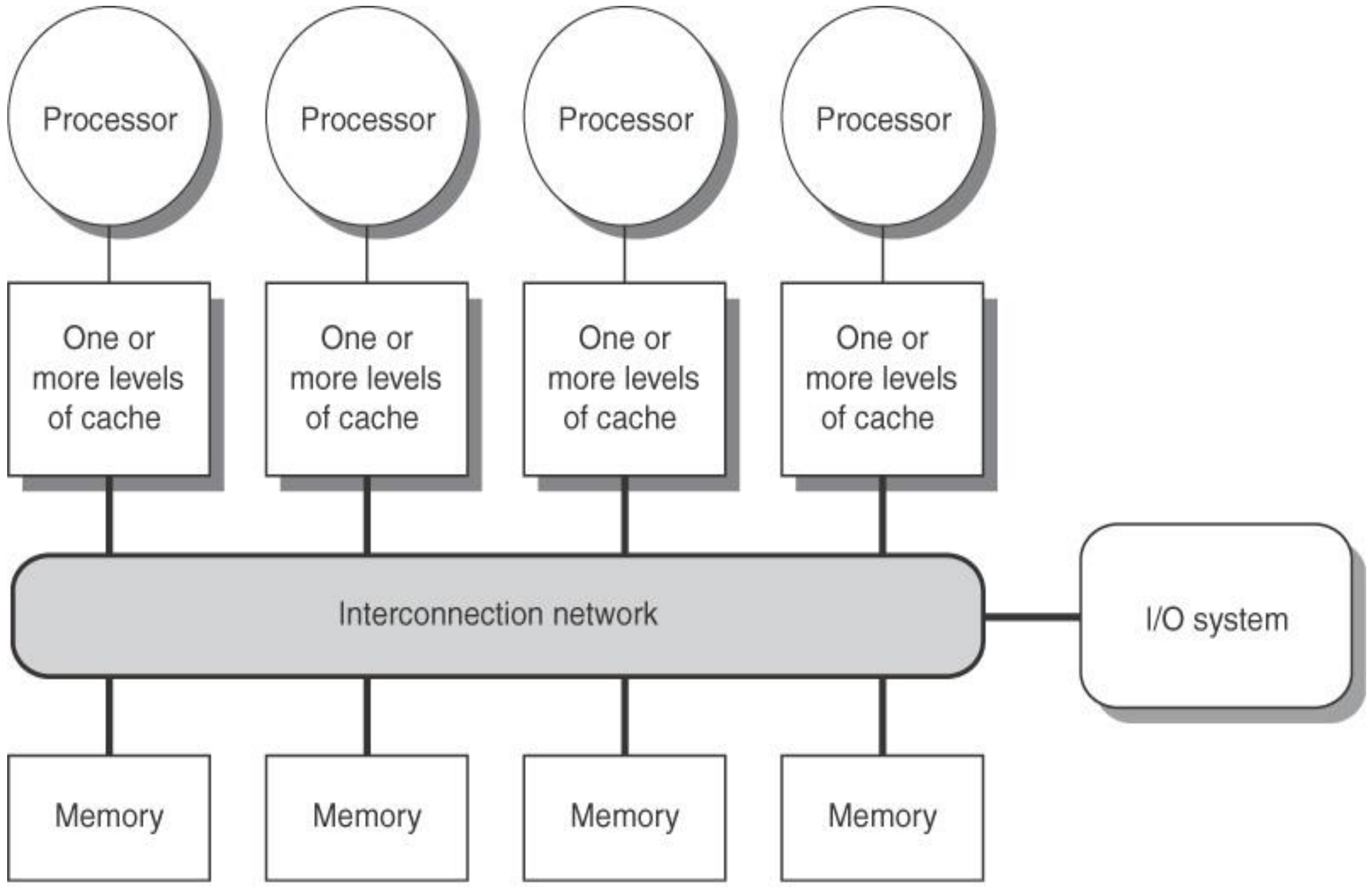
## Some Issues (3)

- Many modern microprocessors provide basic support for cache coherency and also allow the construction of a shared memory bus by direct connection of the external memory bus of two processors
- These capabilities reduce the number of chips required to build a small-scale multiprocessor

## Limitations in Symmetric Shared-Memory Multiprocessors and Snooping Protocols

- An centralized resource may become a bottleneck for many processors and heavy memory requirement
- How can a designer increase the memory bandwidth to support either more or faster processors?
  - —Multiple buses
  - —Interconnection network
- For example, AMD Opteron 4 dual-processors

## A Multiprocessor with Uniform Memory Acces


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
Interconnection network
I/O system
Memory
Memory
Memory
Memory

## Performance of Symmetric Shared-Memory Multiprocessors

- In a multiprocessor using a snooping protocol, several different phenomena combine to determine performance
  - In particular, the overall cache performance is a combination of the behavior of uniprocessor cache miss traffic and the traffic caused by communication, which results in invalidations and subsequent cache misses
  - Changing the processor count, cache size, and block size can affect these two components of the miss rate in different ways, leading to overall system behavior that is a combination of the two effects.

## Coherence Misses

- Uniprocessor miss rate into the 3C classification could provide insight into both application behavior and potential improvements to the cache design
- Similarly, the misses that arise from interprocessor communication, which are often called *coherence misses*,
- Coherence misses, can be broken into two separate sources
  - *True sharing misses* arises from the communication of data through the cache coherence mechanism
  - *False sharing* arises from the use of an invalidation-based coherence algorithm with a single valid bit per cache block

## True sharing misses

- In an invalidation based protocol, the first write by a processor to a shared cache block causes an invalidation to establish ownership of that block
- Additionally, when another processor attempts to read a modified word in that cache block, a miss occurs and the resultant block is transferred
- Both these misses are classified as true sharing misses since they directly arise from the sharing of data among processors.

## False sharing (1)

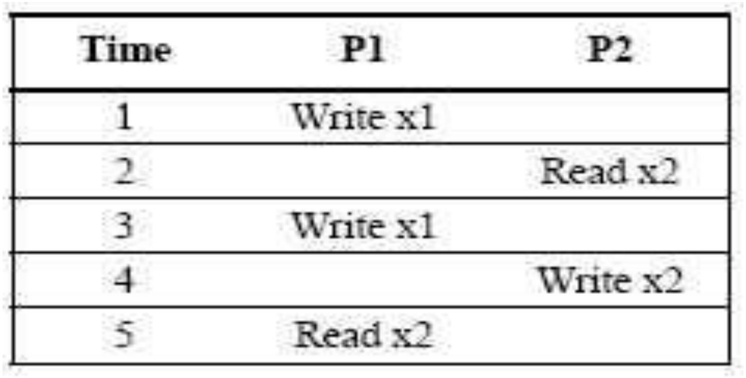
- It arises from the use of an invalidation-based coherence algorithm with a single valid bit per cache block
- False sharing occurs when a block is invalidated (and a subsequent reference causes a miss) because some word in the block, other than the one being read, is written into
- If the word written into is actually used by the processor that received the invalidate, then the reference was a true sharing reference and would have caused a miss independent of the block size or position of words

## False sharing (2)

- If, however, the word being written and the word read are different and the invalidation does not cause a new value to be communicated, but only causes an extra cache miss, then it is a false sharing miss
- In a false sharing miss, the block is shared, but no word in the cache is actually shared, and the miss would not occur if the block size were a single word

## Example

- 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, a false sharing miss, or a hit. Any miss that would occur if the block size were one word is designated a true sharing miss.



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

## Answer (1)

1. This event is a true sharing miss, since x1 was read by P2 and needs to be invalidated from P2.
2. This event is a false sharing miss, since x2 was invalidated by the write of x1 in P1, but that value of x1 is not used in P2.
3. This event is a false sharing miss, since the block containing x1 is marked shared due to the read in P2, but P2 did not read x1. The cache block containing x1 will be in the shared state after the read by P2; a write miss is required to obtain exclusive access to the block. In some protocols this will be handled as an *upgrade request*, which generates a bus invalidate, but does not transfer the cache block.

## Answer (2)

4. This event is a false sharing miss for the same reason as step 3.
5. This event is a true sharing miss, since the value being read was written by P2.

## Performance Measurements of the Commercial Workload

- The target is Alphaserver 4100, or using a configurable simulator modeled after the Alphaserver 4100
  - The Alphaserver 4100 used for these measurements has four processors, each of which is an Alpha 21164 running at 300 MHz
- Each processor has a three-level cache hierarchy:
  - L1 consist of a pair of 8 KB direct-mapped on-chip caches, one for instruction and one for data. The block size is 32-bytes, and the data cache is write-through to L2, using a write buffer
  - L2 is a 96 KB on-chip unified 3-way set associative cache with a 32-byte block size, using write-back
  - L3 is an off-chip, combined, direct-mapped 2 MB caches with 64-byte blocks also using write-back
- The latency for an access to L2 is 7 cycles, to L3 it is 21 cycles, and to main memory it is 80 clock cycles (typical without contention)
- Cache to cache transfers, which occur on a miss to an exclusive block held in another cache, require 125 clock cycles

## A Commercial Workload

- Our commercial workload consists of three applications:
  - Online transaction processing workload (OLTP)
  - Decision support system (DSS) workload
  - Web index search (Altavista) benchmark

## A Commercial Workload-OLTP

- OLTP modeled after TPC-B and using Oracle 7.3.2 as the underlying database
  - It consists of a set of client processes that generate requests and a set of servers that handle them
  - The server processes consume 85% of the user time, with the remaining going to the clients
  - Although the I/O latency is hidden by careful tuning and enough requests to keep the CPU busy, the server processes typically block for I/O after about 25,000 instructions.

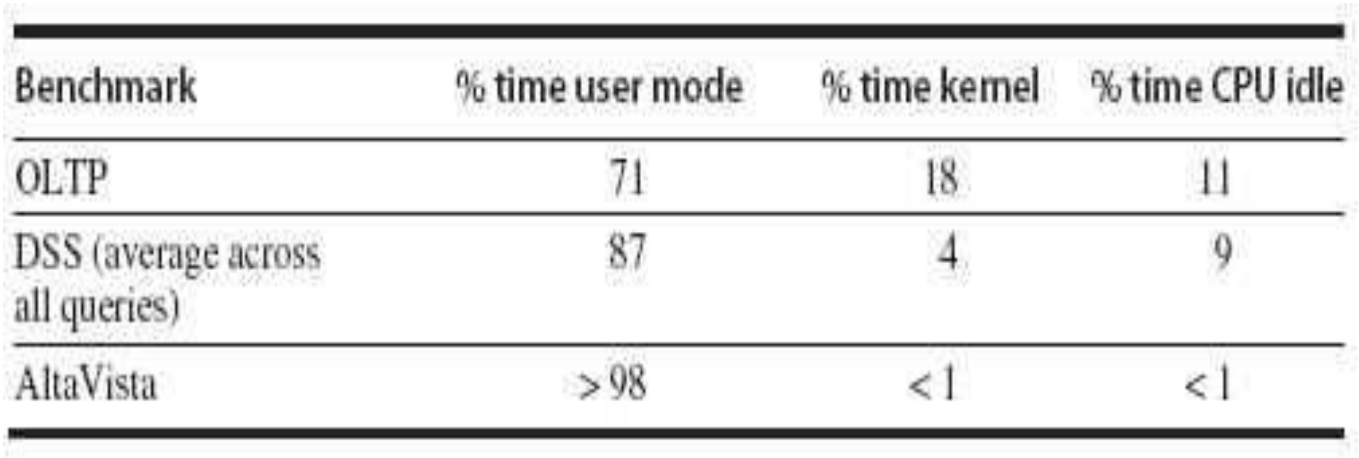
## A Commercial Workload-DSS

- DSS workload based on TPC-D and also using Oracle 7.3.2 as the underlying database
  - It includes only 6 of the 17 read queries in TPC-D, although the 6 queries examined in the benchmark span the range of activities in the entire benchmark.
  - To hide the I/O latency, parallelism is exploited both within queries, where parallelism is detected during a query formulation process, and across queries.
  - Blocking calls are much less frequent than in the OLTP benchmark; the 6 queries average about 1.5 million instructions before blocking.

## A Commercial Workload-AltaVista

- A Web index search (AltaVista) benchmark based on a search of a memory mapped version of the AltaVista database (200 GB)
  - —The inner loop is heavily optimized. Because the search structure is static
  - —Little synchronization is needed among the threads

## Workload Comparison (1/2)

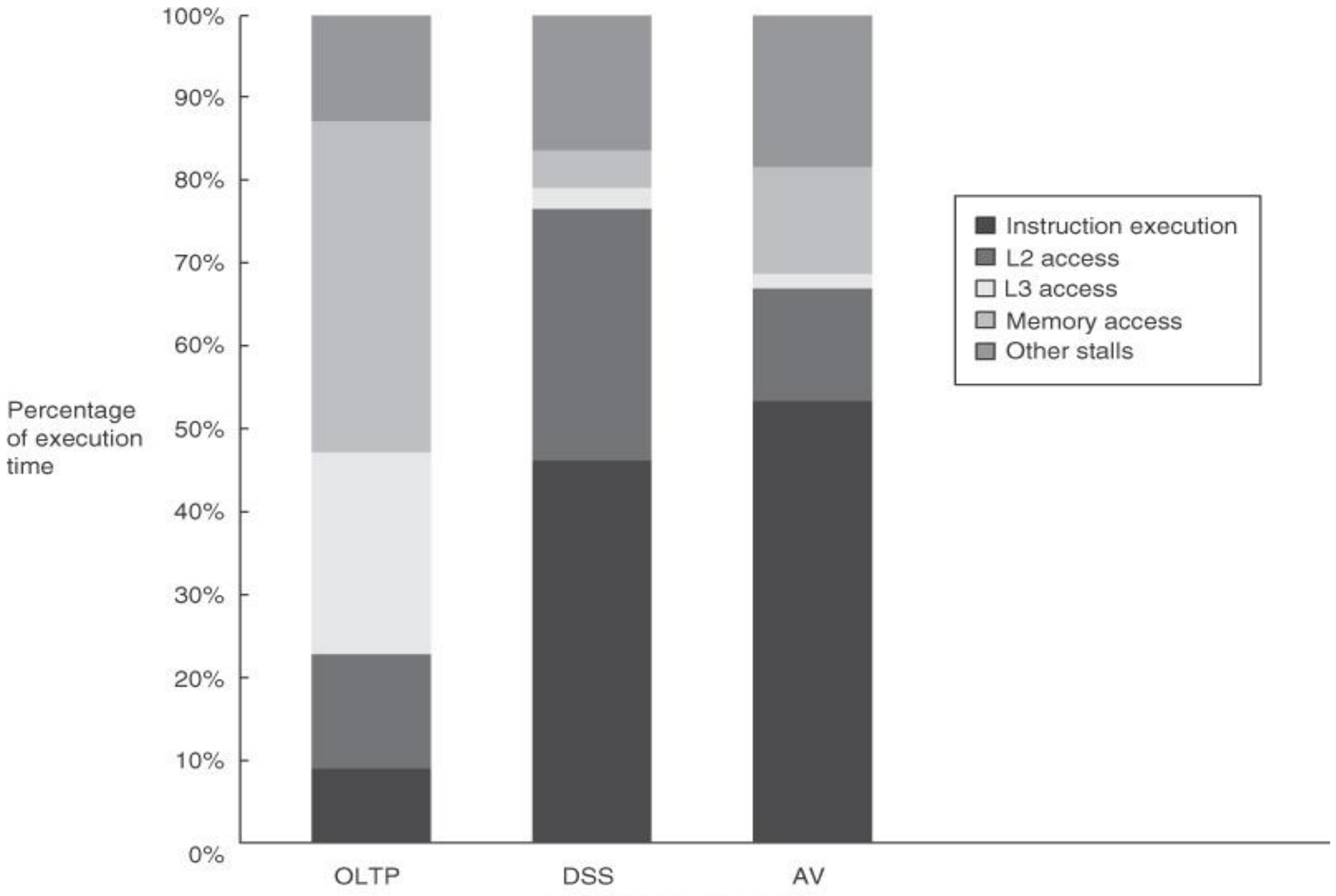


| Benchmark | % time user mode | % time kernel | % time CPU idle |
|---|---|---|---|
| OLTP | 71 | 18 | 11 |
| DSS (average across all queries) | 87 | 4 | 9 |
| AltaVista | > 98 | < 1 | < 1 |

## Workload Comparison (2/2)

- We start by looking at the overall CPU execution for these benchmarks on the four-processor system
- They include substantial I/O time, which is ignored in the CPU time measurements
- The effective CPI varies widely for these benchmarks
  - —from a CPI of 1.3 for the AltaVista Web search,
  - —to an average CPI of 1.6 for the DSS workload,
  - —to 7.0 for the OLTP workload

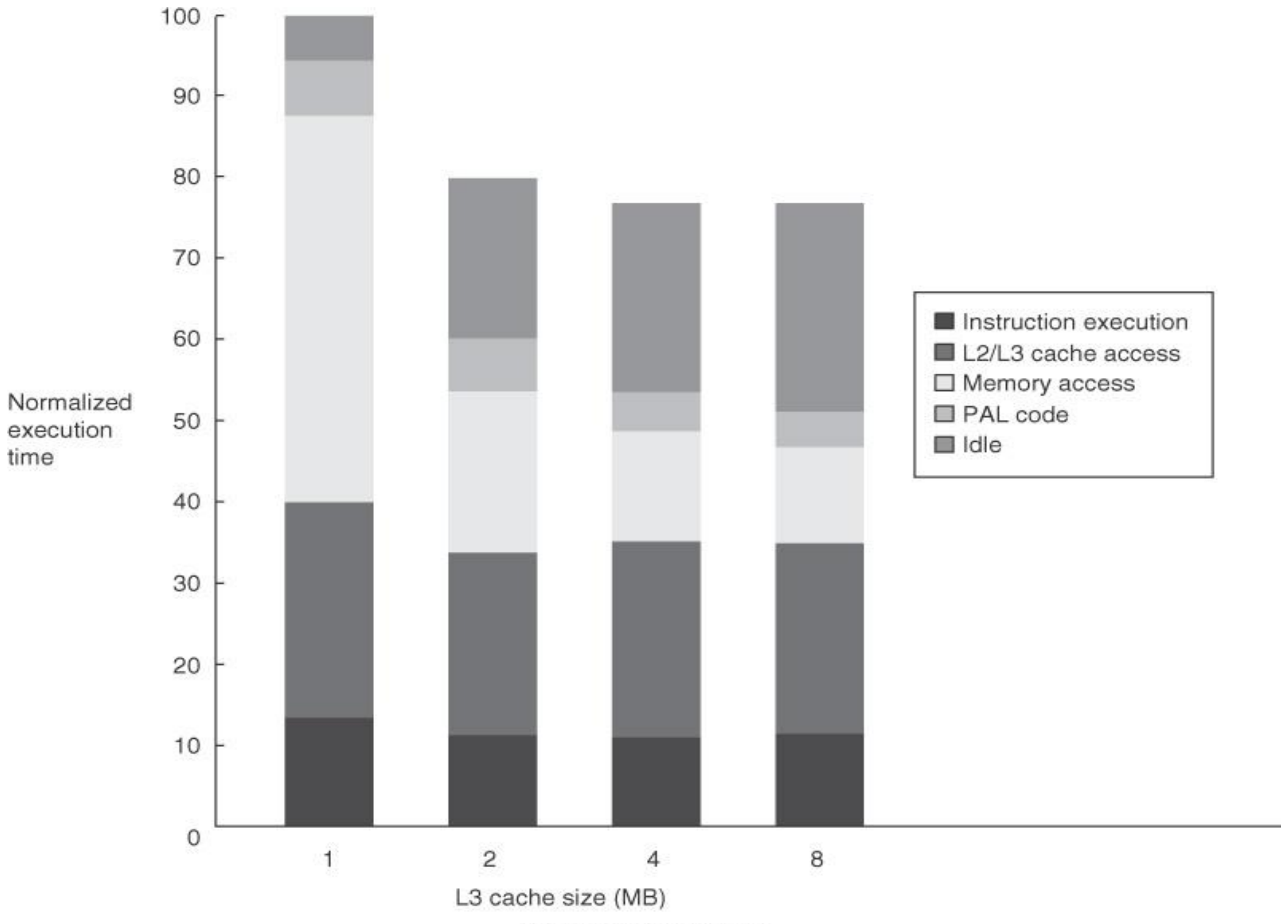
## Execution Time Breakdown



## Observation (1)

- Only OLTP has a significant fraction of kernel accesses, and the kernel accesses tend to be better behaved than the user accesses
- Since the OLTP workload demands the most from the memory system with large numbers of expensive L3 misses
  - We focus on examining the impact of L3 cache size, processor count, and block size on the OLTP benchmark.
- The execution time is improved as the L3 cache grows due to the reduction in L3 misses

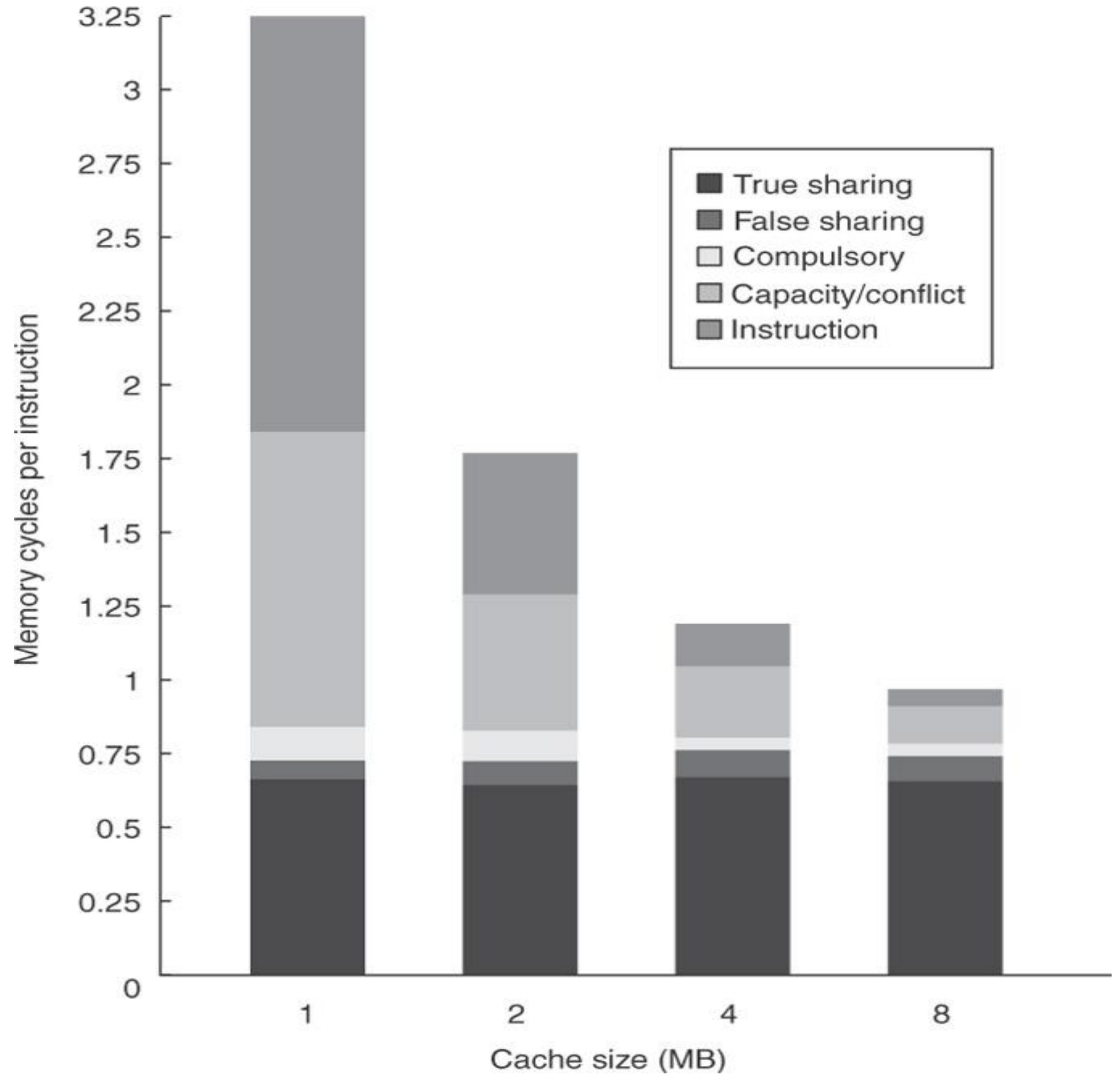
## Relative OLTP Performance for L3 Cache



## Observation (2)

- Almost all of the gain occurs in going from 1 to 2 MB, with little additional gain beyond that
- Despite the fact that cache misses are still a cause of significant performance loss with 2 MB and 4 MB caches
- We need to determine what factors contribute to the L3 miss rate and how they change as the L3 cache grows
- Figure 4.12 shows this data, displaying the number of memory access cycles contributed per instruction from five sources

## Memory Access Cycles for Various Cache Sizes



## Observation (3)-1

- Increasing the cache size eliminates most of the uniprocessor misses, while leaving the multiprocessor misses untouched.
- How does increasing the processor count affect different types of misses?
- Figure 4.13 shows this data assuming a base configuration with a 2 MB, two-way set associative L3 cache.
- The increase in the true sharing miss rate, which is not compensated for by any decrease in the uniprocessor misses, leads to an overall increase in the memory access cycles per instruction

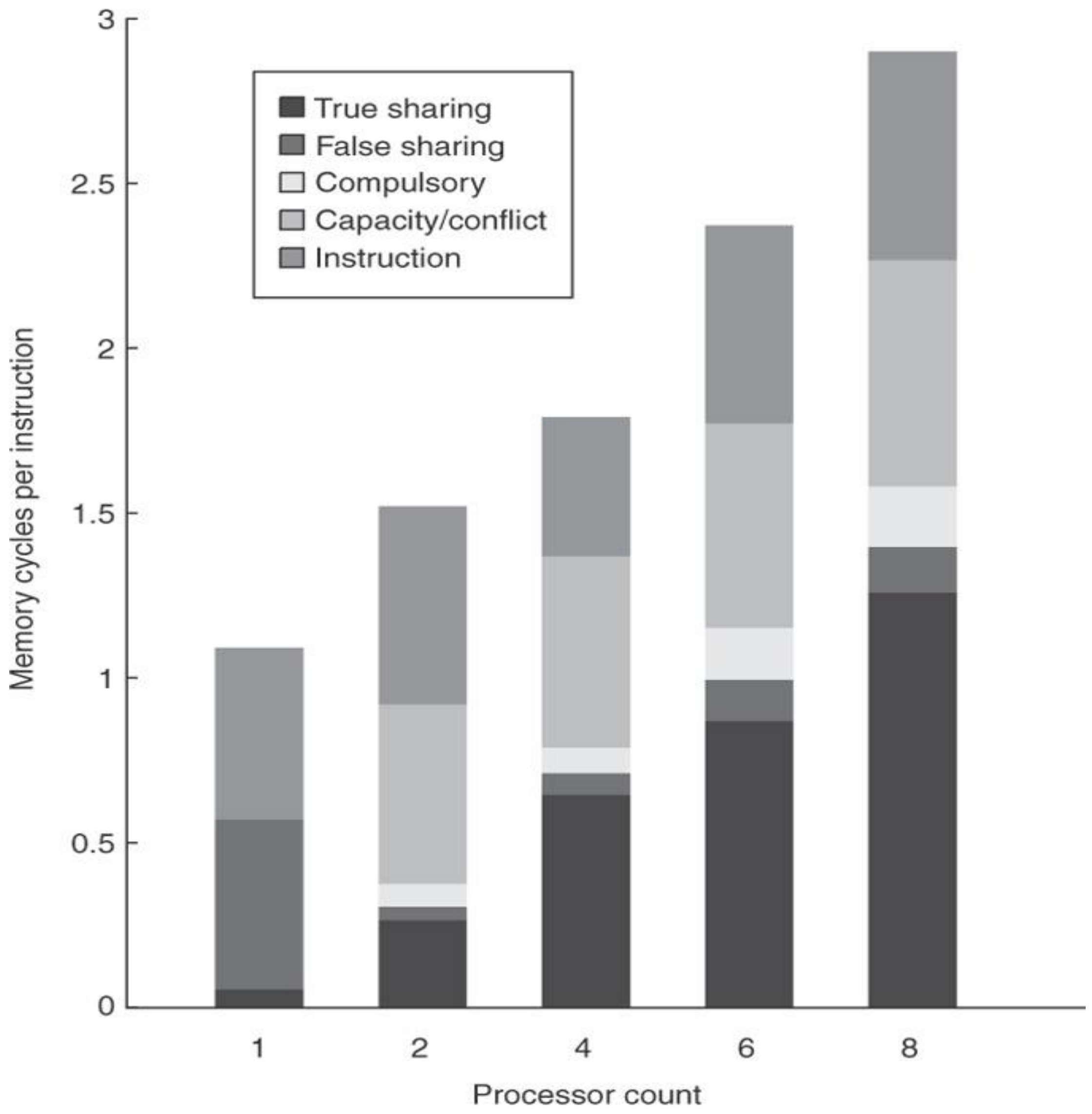
## Observation (3)-2

- The final question we examine is whether increasing the block size
- It should decrease the instruction and cold miss rate and, within limits, also reduce the capacity/conflict miss rate and possibly the true sharing miss rate—is helpful for this workload
- Figure 4.14 shows the number of misses per 1000 instructions as the block size is increased from 32 to 256. Increasing the block size from 32 to 256 affects four of the miss rate components:

## Observation (3)-3

- The true sharing miss rate decreases by more than a factor of 2, indicating some locality in the true sharing patterns
- The compulsory miss rate significantly decreases, as we would expect.
- The conflict/capacity misses show a small decrease (a factor of 1.26 compared to a factor of 8 increase in block size), indicating that the spatial locality is not high in the uniprocessor misses that occur with L3 caches larger than 2 MB.
- The false sharing miss rate, although small in absolute terms, nearly doubles.

## Memory Access Cycles for Processor Count



## Multiprogramming and OS Workload (1)

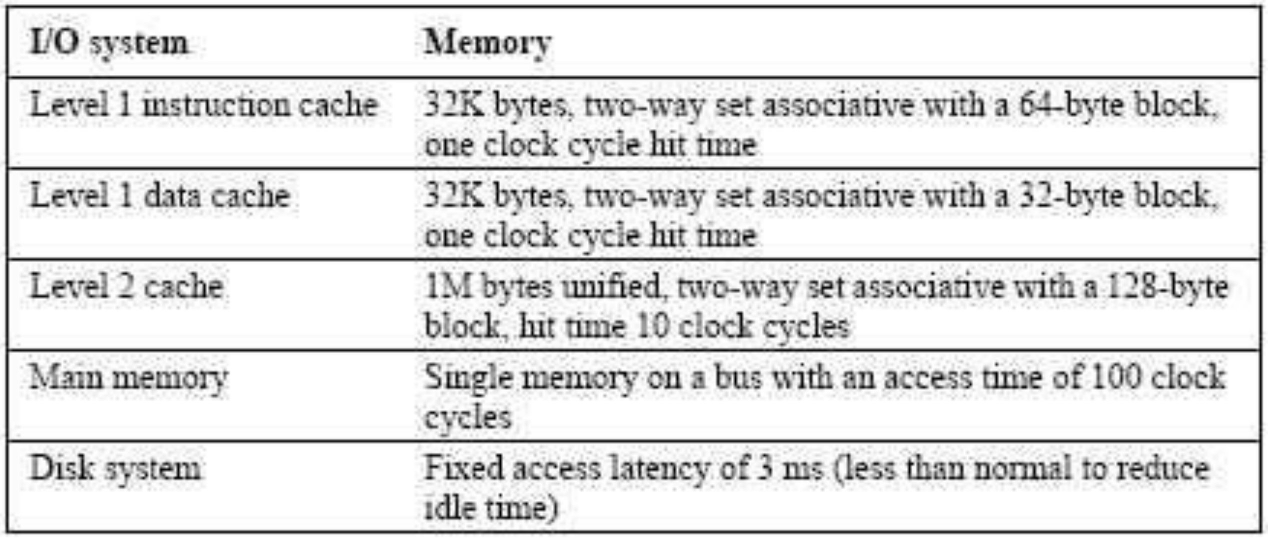
- A multiprogrammed workload consists of both user activity and OS activity
  - The workload used is two independent copies of the compile phase of the Andrew benchmark.
  - The compile phase consists of a parallel make using eight processors
  - The workload runs for 5.24 seconds on eight processors, creating 203 processes and performing 787 disk requests on three different file systems
  - The workload is run with 128 MB of memory, and no paging activity takes place

## Multiprogramming and OS Workload (2)

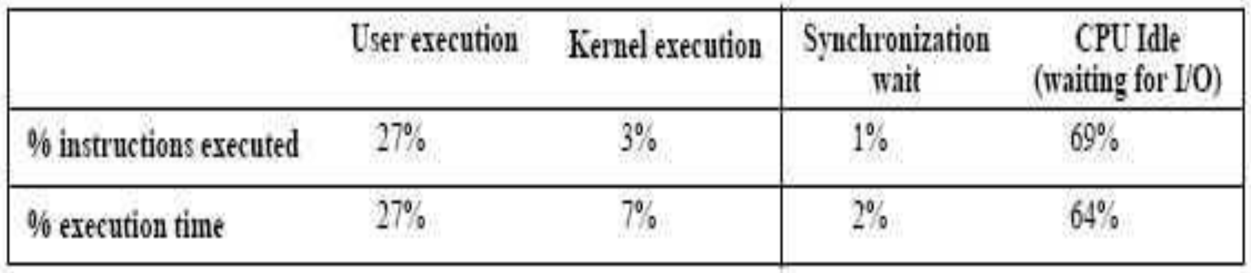
- The workload has three distinct phases:
  - Compiling the benchmarks involving substantial compute activity
  - Installing the object files in a library
  - Removing the object files
- CPU idle time and instruction cache performance are important in this workload

## Multiprogramming and OS Workload (3)

- We assume the following memory and I/O systems:

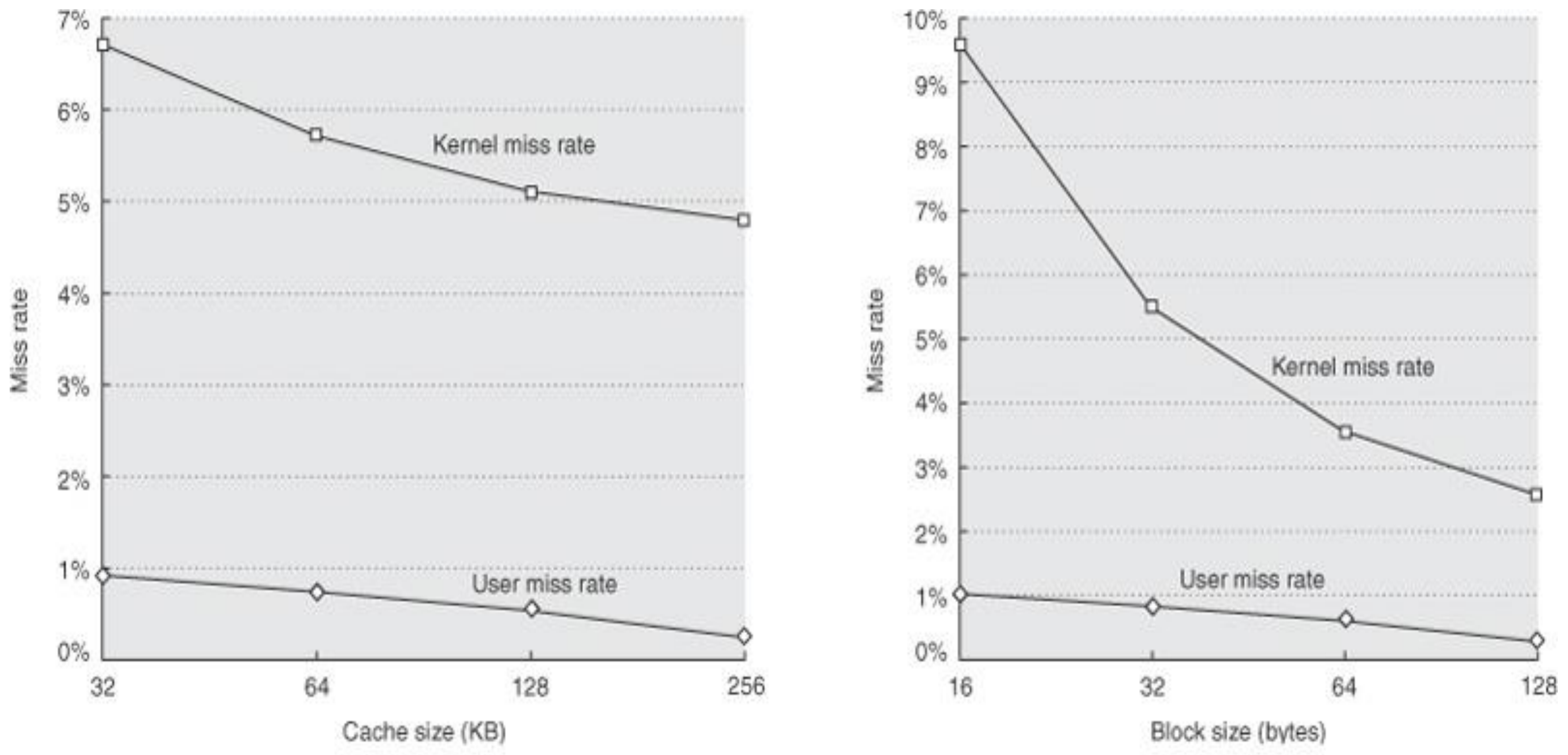


| I/O system | Memory |
|---|---|
| Level 1 instruction cache | 32K bytes, two-way set associative with a 64-byte block, one clock cycle hit time |
| Level 1 data cache | 32K bytes, two-way set associative with a 32-byte block, one clock cycle hit time |
| Level 2 cache | 1M bytes unified, two-way set associative with a 128-byte block, hit time 10 clock cycles |
| Main memory | Single memory on a bus with an access time of 100 clock cycles |
| Disk system | Fixed access latency of 3 ms (less than normal to reduce idle time) |

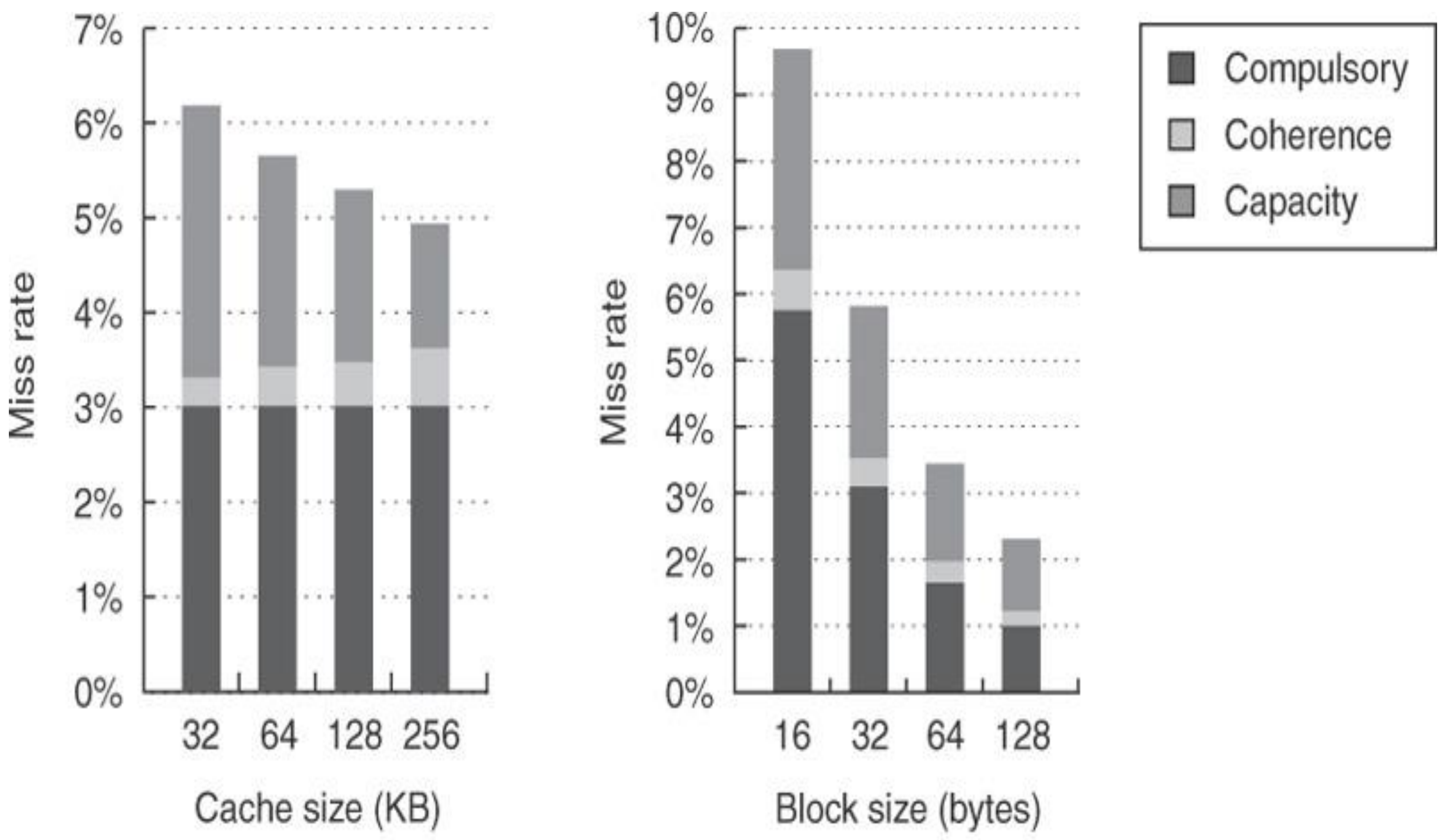


| | User execution | Kernel execution | Synchronization wait | CPU Idle (waiting for I/O) |
|---|---|---|---|---|
| % instructions executed | 27% | 3% | 1% | 69% |
| % execution time | 27% | 7% | 2% | 64% |

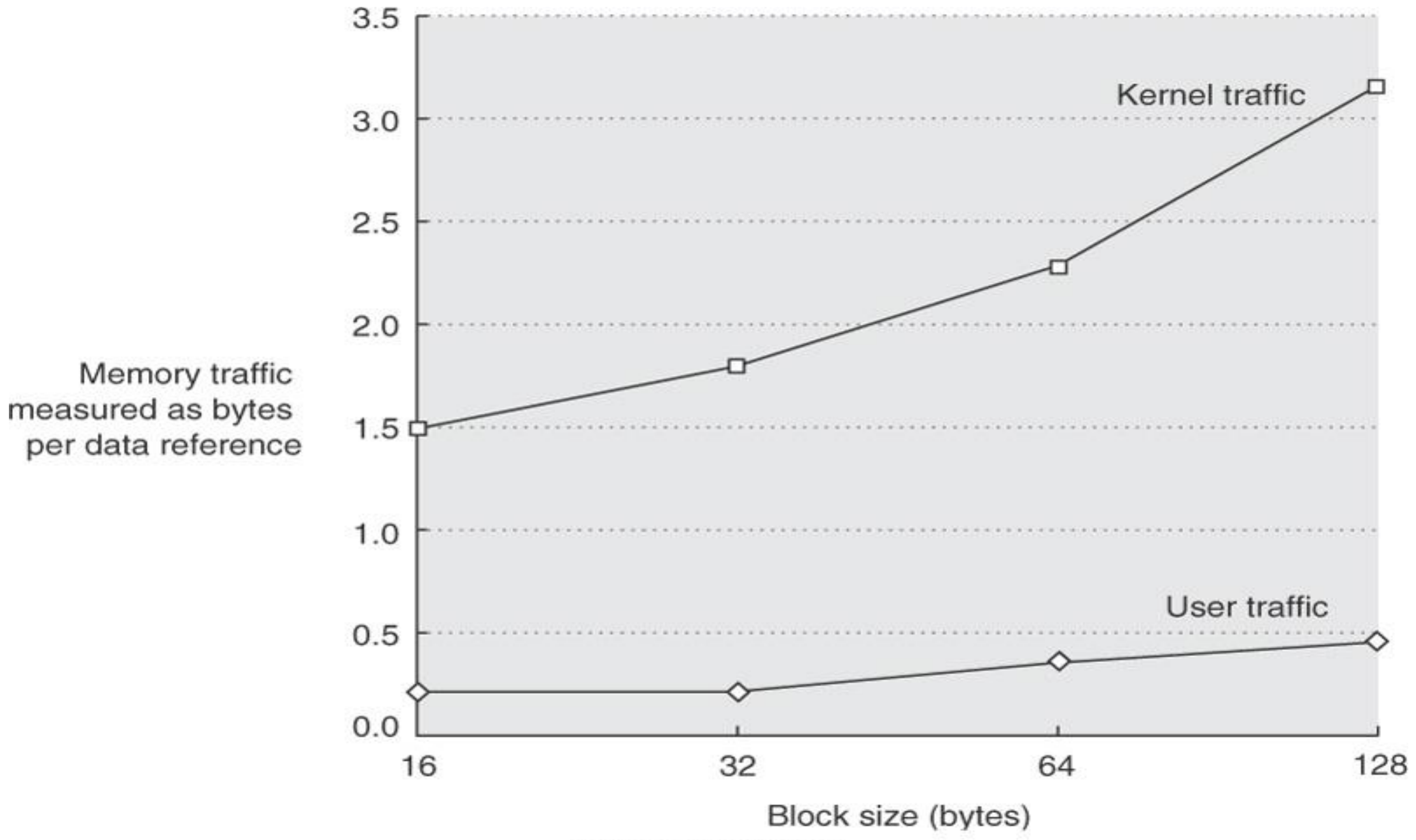
## Multiprogramming and OS Workload (4): Miss Rates for Kernel and User Components



## Multiprogramming and OS Workload (5): Kernel Misses vs. Cache Size and Block Size



## Multiprogramming and OS Workload (5): Byres Needed per Data Reference



## Distributed Shared Memory and Directory-Based Coherence (1)

- Snooping protocol requires communications with all caches on every misses
- The alternative to a snooping protocol is directory protocol
  - —A directory keeps the state of every cached block
  - —Information in the directory includes caches having copies of the block
  - —It can also reduce the bandwidth demands in a centralized shard-memory machine

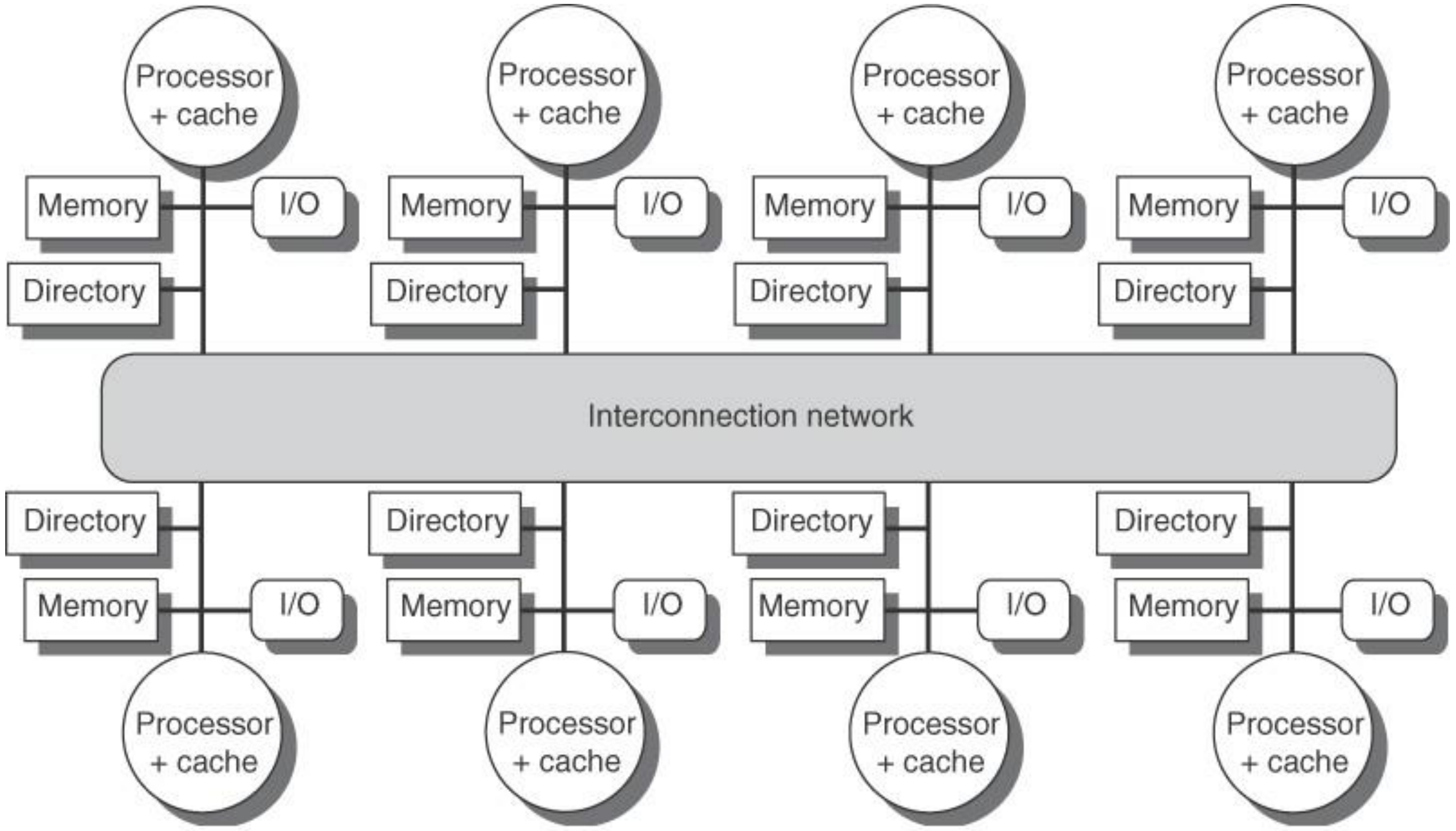
## Distributed Shared Memory and Directory-Based Coherence (2)

- The simplest one associates an entry in the directory with each memory block
  - The amount of information is proportional to the product of the number of memory blocks and the number of processors
- For larger multiprocessors, methods are needed to allow the directory structure to be efficiently scaled
- The methods that have been proposed either try to keep information for fewer blocks or try to keep fewer bits per entry
  - E.g., only those in caches rather than all memory blocks

## Distributed Shared Memory and Directory-Based Coherence (3)

- To prevent the directory from becoming the bottleneck, directory entries can be distributed along with the memory
  - —Different directory accesses can go to different locations, just as different memory requests go to different memories
- A distributed directory retains the characteristic that the sharing status of a block is always in a single known location

## Distributed Shared Memory and Directory-Based Coherence (3)



## Directory-Based Cache Coherence Protocol (1)

- Two primary operations in a directory protocol
  - —handling a read miss
  - —handling a write to a shared and clean cache block
- To implement these operations, a directory must track the state of each cache block
- Handling a write miss to a shared block is a simple combination of these two

## Directory-Based Cache Coherence Protocol (2)

- In a simple protocol, these states could be the following:
  - —*Shared:* One or more processors have the block cached, and the value in memory is up to date
  - —*Uncached:* No processor has a copy of the cache block
  - —*Modified*: Exactly one processor 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 block*

## Directory-Based Cache Coherence Protocol (3)

- In addition to tracking the state of each cache block, we must track the processors that have copies of the block when it is shared, since they will need to be invalidated 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 has a copy of that block
  - We can also use the bit vector to keep track of the owner of the block when the block is in the exclusive state
  - For efficiency reasons, we also track the state of each cache block at the individual caches

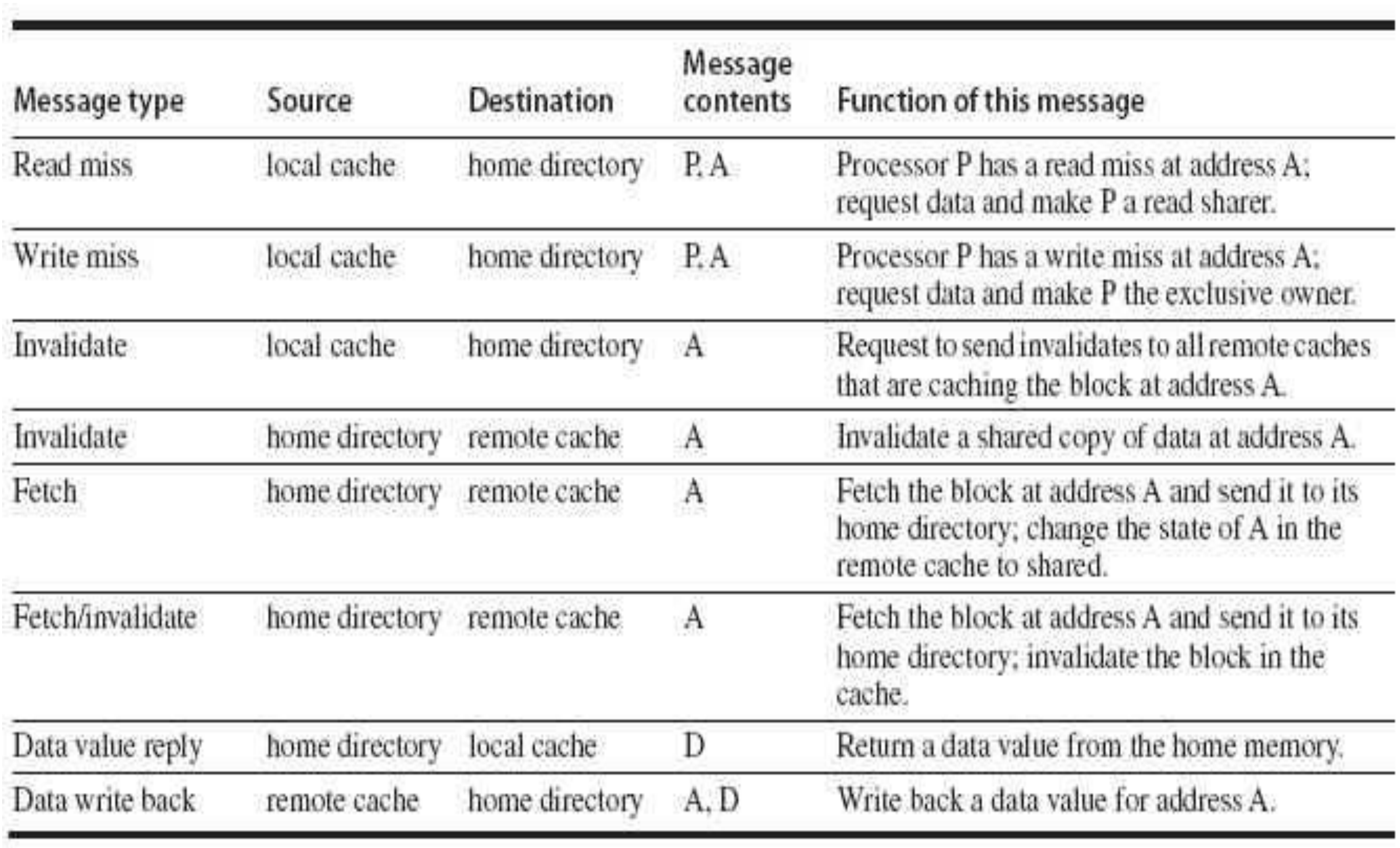
## Directory-Based Cache Coherence Protocol (4)

- It is useful to examine a catalog of the message types that may be sent between the processors and the directories
- The *local* node is the node where a request originates
- The *home* node is the node where the memory location and the directory entry of an address reside
  - The physical address space is statically distributed, so the node that contains the memory and directory for a given physical address is known
- A *remote* node is the node that has a copy of a cache block, whether exclusive (in which case it is the only copy) or shared

## Directory-Based Cache Coherence Protocol (5)

- The states and transitions for the state machine at each cache are identical to what we used for the snooping cache, although the actions on a transition are slightly different
- The invalidating and locating an exclusive copy are different due to the communication
  - —the requesting node and the directory
  - —the directory and the remote nodes
- These two steps are combined through the broadcast

## Directory-Based Cache Coherence Protocol (6)

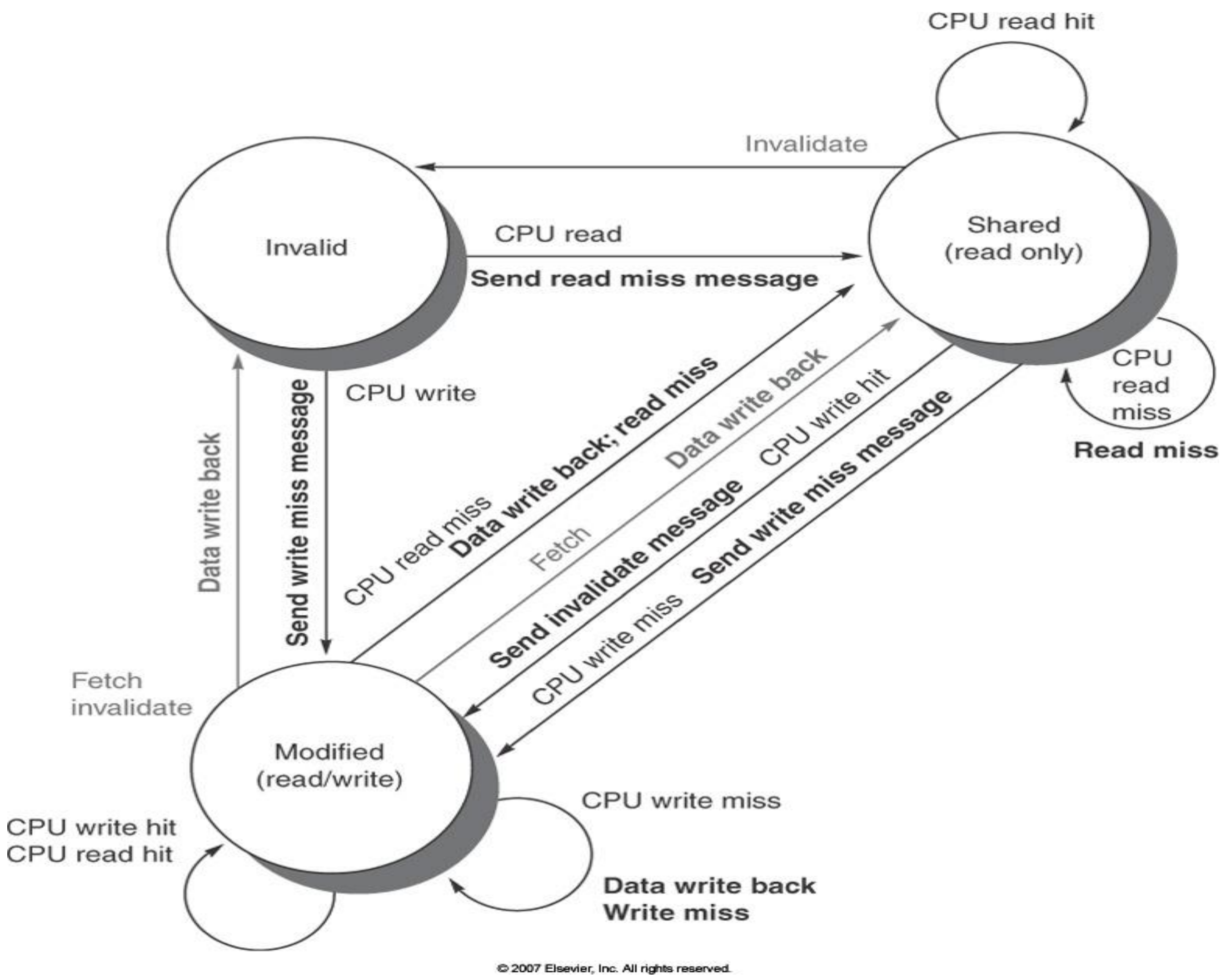


| Message type | Source | Destination | Message contents | Function of this message |
|---|---|---|---|---|
| Read miss | local cache | home directory | P, A | Processor P has a read miss at address A; request data and make P a read sharer. |
| Write miss | local cache | home directory | P, A | Processor P has a write miss at address A; request data and make P the exclusive owner. |
| Invalidate | 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 Cache Coherence Protocol (7)

- 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

## State Transition Diagram for An individual Cache Block



## Actions taken at the Directory (1)

- When a block is in the uncached state, the copy in memory is the current value, so the only possible requests for that block are
  - —Read miss: the requesting processor is sent the requested data from memory, and the requestor is made the only sharing node. The state of the block is made shared.
  - —Write miss: the requesting processor is sent the value and becomes the sharing node. The block is made exclusive to indicate that the only valid copy is cached
  - —Sharers indicates the identity of the owner.

## Actions taken at the Directory (2)

- When the block is in the shared state, the memory value is up to date, so the same two requests can occur:
  - Read miss: the requesting processor is sent the requested data from memory, and the requesting processor is added to the sharing set.
  - Write miss: the requesting processor is sent the value. All processors in the set Sharers are sent invalidate messages, and the Sharers set is to contain the identity of the requesting processor. The state of the block is made exclusive.

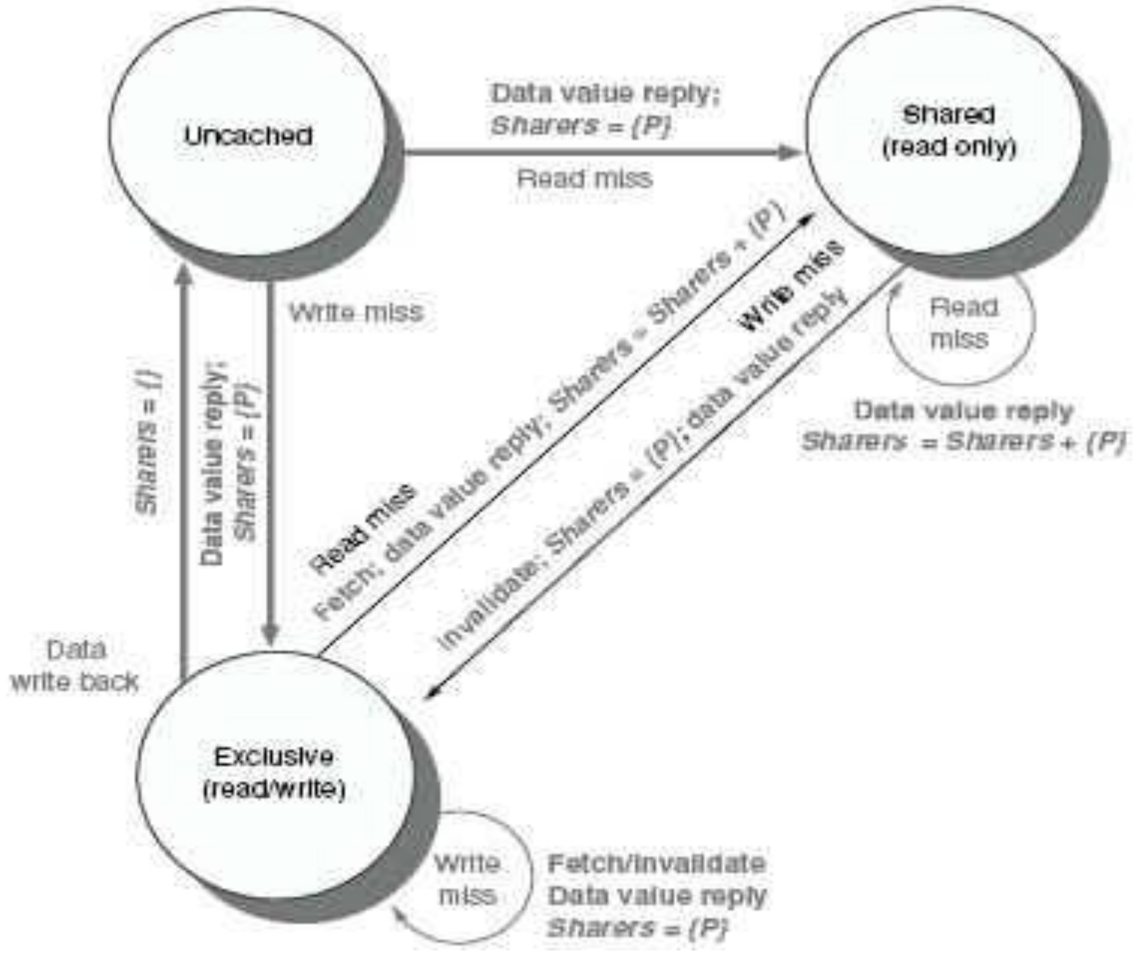
## Actions taken at the Directory (3)

- When the block is in the exclusive state, the current value of the block is held in the cache of the processor identified by the set Sharers, so there are three possible directory requests:
- Read miss:
  - —the owner processor is sent a data fetch message
  - —it causes the state of the block in the owner's cache to transition to shared
  - —It causes the owner to send the data to the directory, where it is written to memory and sent back to the requesting processor
  - —the identity of the requesting processor is added to the set Sharers

## Actions taken at the Directory (4)

- Data write back:
  - —the owner processor is replacing the block and therefore must write it back
  - —this write back makes the memory copy up to date
  - —the block is now uncached, and the Sharers set is empty
- Write miss:
  - —the block has a new owner
  - —A message is sent to the old owner, causing the cache to invalidate the block and send the value to the directory, from which it is sent to the requesting processor, which becomes the new owner
  - —Sharers is set to the identity of the new owner, and the state of the block remains exclusive

# Actions taken at the Directory (5)



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

## Synchronization

- Synchronization mechanisms are typically built with user-level software routines that rely on hardware-supplied synchronization instructions
- For smaller multiprocessors or low-contention situations
  - The key hardware capability is an uninterruptible instruction or instruction sequence capable of atomically retrieving and changing a value
  - Software synchronization mechanisms are then constructed using this capability
- In larger-scale multiprocessors or high-contention situations
  - synchronization can become a performance bottleneck, because contention introduces additional delays and because latency is potentially greater in such a multiprocessor

## Basic Hardware Primitives (1)

- The key ability we require to implement synchronization in a multiprocessor is a set of hardware primitives with the ability to atomically read and modify a memory location
- Without such a capability, the cost of building basic synchronization primitives will be too high and will increase as the processor count increases
- These hardware primitives are the basic building blocks that are used to build a wide variety of user-level synchronization operations, including things such as locks and barriers

## Basic Hardware Primitives (2)

- In general, architects do not expect users to employ the basic hardware primitives, but instead expect that the primitives will be used by system programmers to build a synchronization library
- *Atomic exchange* interchanges a value in a register for a value in memory
  - We want to build a simple lock where the value 0 is used to indicate that the lock is free
  - A 1 is used to indicate that the lock is unavailable. A processor tries to set the lock by doing an exchange of 1, which is in a register, with the memory address corresponding to the lock
- The value returned from the exchange instruction is 1 if some other processor had already claimed access and 0 otherwise

## Basic Hardware Primitives (3)

- Atomic primitives have the *atomic* property
- *test-and-set*
  - It tests a value and sets it if the value passes the test
- *fetch-and-increment*
  - It returns the value of a memory location and atomically increments it
- A slightly different approach to providing this atomic read-and-update operation has been used in some recent multiprocessors
- Implementing a single atomic memory operation introduces some challenges, since it requires both a memory read and a write in a single, uninterruptible instruction

## Basic Hardware Primitives (4)

- An alternative is to have a pair of instructions as if the instructions were atomic
  - The pair of instructions includes a special load called a *load linked* or *load locked* and a special store called a *store conditional*
- These instructions are used in sequence
  - If the contents of the memory location specified by the load linked are changed before the store conditional to the same address occurs, then the store conditional fails
  - If the processor does a context switch between the two instructions, then the store conditional also fails

## Basic Hardware Primitives (5)

- The store conditional is defined to return a value indicating whether or not the store was successful
- Since the load linked returns the initial value and the store conditional returns 1 if it succeeds and 0 otherwise

-
  ```
  try: MOV R3,R4 ;        mov exchange value
       LL R2,0(R1) ;      load linked
       SC R3,0(R1) ;      store conditional
       BEQZ R3,try ;      branch store fails
       MOV R4,R2 ;        put load value in R4
  ```

## Basic Hardware Primitives (6)

- An advantage of the load linked/store conditional mechanism is that it can be used to build other synchronization primitives
  - For example, here is an atomic fetch-and-increment:

```
try: LL R2,0(R1) ;          load linked
     DADDUI R3,R2,#1 ;      increment
     SC R3,0(R1) ;          store conditional
     BEQZ R3,try ;          branch store fails
```

## Implementing Locks Using Coherence (1)

- Once we have an atomic operation, we can use the coherence mechanisms of a multiprocessor to implement *spin locks:*
  - —Locks that a processor continuously tries to acquire, spinning around a loop until it succeeds
  - —Spin locks are used when a programmer expects the lock to be held for a very short amount of time and when she wants the process of locking to be low latency when the lock is available
  - —Because spin locks tie up the processor, waiting in a loop for the lock to become free, they are inappropriate in some circumstances

## Implementing Locks Using Coherence (2)

- The simplest implementation, which we would use if there were no cache coherence, would keep the lock variables in memory
- A processor could continually try to acquire the lock using an atomic operation, say exchange, and test whether the exchange returned the lock as free
- To release the lock, the processor simply stores the value 0 to the lock

## Implementing Locks Using Coherence (3)

- Here is the code sequence to lock a spin lock whose address is in R1 using an atomic exchange:

```
        DADDUI R2,R0,#1
lockit: EXCH R2,0(R1) ;   atomic exchange
        BNEZ R2,lockit ;  already locked?
```

## Implementing Locks Using Coherence (4)

- If our multiprocessor supports cache coherence, we can cache the locks using the coherence mechanism to maintain the lock value coherently
- Caching locks has two advantages
  - First, it allows an implementation where the process of "spinning" could be done on a local cached copy rather than requiring a global memory access on each attempt to acquire the lock
  - The second advantage comes from the observation that there is often locality in lock accesses

## Implementing Locks Using Coherence (5)

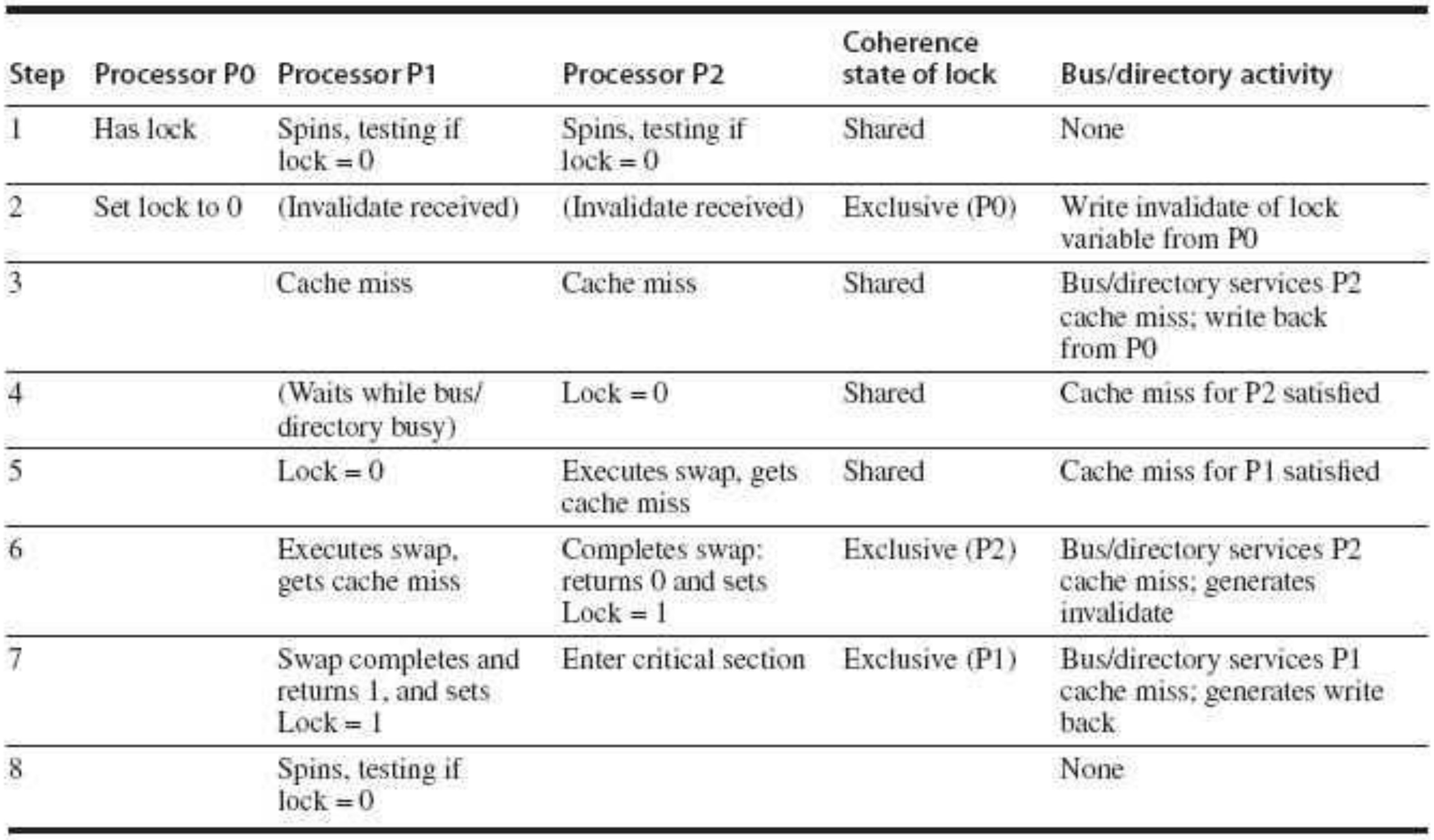
- If multiple processors are attempting to get the lock, each will generate the write
  - —Most of these writes will lead to write misses, since each processor is trying to obtain the lock variable in an exclusive state
- Thus we should modify our spin-lock procedure so that it spins by doing reads on a local copy of the lock until it successfully sees that the lock is available
- Then it attempts to acquire the lock by doing a swap operation
  - —A processor first reads the lock variable to test its state. A processor keeps reading and testing until the value of the read indicates that the lock is unlocked

## Implementing Locks Using Coherence (6)

- The processor then races against all other processes that were similarly "spin waiting" to see who can lock the variable first
- All processes use a swap instruction that reads the old value and stores a 1 into the lock variable
- The single winner will see the 0, and the losers will see a 1 that was placed there by the winner

```
lockit:  LD R2,0(R1) ;          load of lock
         BNEZ R2,lockit ;       not available-spin
         DADDUI R2,R0,#1 ;      load locked value
         EXCH R2,0(R1) ;        swap
         BNEZ R2,lockit ;       branch if lock wasn't 0
```

## Cache-Coherence Steps and Bus Traffic



| Step | Processor P0 | Processor P1 | Processor P2 | Coherence state of lock | Bus/directory activity |
|---|---|---|---|---|---|
| 1 | Has lock | Spins, testing if lock = 0 | Spins, testing if lock = 0 | Shared | None |
| 2 | Set lock to 0 | (Invalidate received) | (Invalidate received) | Exclusive (P0) | Write invalidate of lock variable from P0 |
| 3 | | Cache miss | Cache miss | Shared | Bus/directory services P2 cache miss; write back from P0 |
| 4 | | (Waits while bus/ directory busy) | Lock = 0 | Shared | Cache miss for P2 satisfied |
| 5 | | Lock = 0 | Executes swap, gets cache miss | Shared | Cache miss for P1 satisfied |
| 6 | | Executes swap, gets cache miss | Completes swap: returns 0 and sets Lock = 1 | Exclusive (P2) | Bus/directory services P2 cache miss; generates invalidate |
| 7 | | Swap completes and returns 1, and sets Lock = 1 | Enter critical section | Exclusive (P1) | Bus/directory services P1 cache miss; generates write back |
| 8 | | Spins, testing if lock = 0 | | | None |

## Example

- This example shows another advantage of the load-linked/store-conditional primitives
  - The read and write operation are explicitly separated

```
lockit: LL R2,0(R1) ;          load linked
        BNEZ R2,lockit ;       not available-spin
        DADDUIR2,R0,#1 ;       locked value
        SC R2,0(R1) ;          store
        BEQZ R2,lockit ;       branch if store fails
```

- Although our spin lock scheme is simple and compelling, it has difficulty scaling up to handle many processors because of the communication traffic generated when the lock is released

## Models of Memory Consistency: An Introduction

- Cache coherence ensures that multiple processors see a consistent view of memory
  - How consistent the view of memory must be
  - When must a processor see a value that has been updated by another processor?
- Since processors communicate through shared variables
  - In what order must a processor observe the data writes of another processor?

## Example

- Although the question of how consistent memory be seems simple, it is remarkably complicated

```
P1:     A = 0; P2: B = 0;
        ..... .....
        A = 1; B = 1;
L1:     if (B == 0) ... L2: if (A == 0)...
```

## Memory Consistency (1)

- Suppose the write invalidate is delayed, and the processor is allowed to continue during this delay
- P1 and P2 may have not seen the invalidations for B and A (respectively) *before* they attempt to read the values
  - Should this behavior be allowed, and if so, under what conditions?

## Memory Consistency (2)

- The most straightforward model for memory consistency is called *sequential consistency*
  - The simplest way to implement sequential consistency is to require a processor to delay the completion of any memory access until all the invalidations caused by that access are completed
  - It is equally effective to delay the next memory access until the previous one is completed

## Memory Consistency (3)

- Under sequential consistency, we cannot, for example, simply place the write in a write buffer and continue with the read
- Although sequential consistency presents a simple programming paradigm, it reduces potential performance
  - especially in a multiprocessor with a large number of processors or long interconnect delays

## Memory Consistency (4)

- To provide better performance, researchers and architects have explored two different routes
  - —First, they developed ambitious implementations that preserve sequential consistency but use latency hiding techniques to reduce the penalty
  - —Second, they developed less restrictive memory consistency models that allow for faster hardware

## Memory Consistency (5): Example

- Suppose we have a processor where a write miss takes 50 cycles to establish ownership, 10 cycles to issue each invalidate after ownership is established, and 80 cycles for an invalidate to complete and be acknowledged once it is issued. Assuming that four other processors share a cache block, how long does a write miss stall the writing processor if the processor is sequentially consistent? Assume that the invalidates must be explicitly acknowledged before the directory controller knows they are completed. Suppose we could continue executing after obtaining ownership for the write miss without waiting for the invalidates; how long would the write take?

## Answer

- When we wait for invalidates,
  - each write time = ownership time + time to complete the invalidates
- Since the invalidates can overlap, we need only worry about the last one
  - which starts 10 + 10 + 10 + 10 = 40 cycles after ownership is established
- Hence the total time for the write is 50 + 40 + 80 = 170 cycles
- In comparison, the ownership time is only 50 cycles
- With appropriate write-buffer implementations it is even possible to continue before ownership is established

## The Programmer’s View (1)

- Although the sequential consistency model has a performance disadvantage, from the viewpoint of the programmer it has the advantage of simplicity
- The challenge is to develop a programming model that is simple to explain and yet allows a high performance implementation

## The Programmer's View (2)

- A model assume that programs are *synchronized*
- A program is synchronized if all access to shared data is ordered by synchronization operation
- A data reference is ordered by a synchronization operation
  - —In every possible execution, a write of a variable by one processor and an access of that variable by another processor are separated by a pair of synchronization operations
  - —One executed after the write by the writing processor and one executed before the access by the second processor

## The Programmer’s View (3)

- Cases where variables may be updated without ordering by synchronization are called *data races*
    - Because the execution outcome depends on the relative speed of the processors
    - Like races in hardware design, the outcome is unpredictable, which leads to another name for synchronized programs: *data-race-free*
- If the accesses were unsynchronized, the behavior of the program would be quite difficult to determine because of the speeds of various processor executions

## Relaxed Consistency Models (1)

- *Relaxed consistency models* allow reads and writes to complete out of order, but to use synchronization operations to enforce ordering
- A synchronized program behaves as if the processor were sequentially consistent
- There are a variety of relaxed models that are classified according to what orderings they relax

## Relaxed Consistency Models (2)

- Three major sets are:
  - The ***W→R ordering***: which yields a model known as *total store ordering* or processor consistency
  - The ***W→W ordering***: which yields a model known as *partial store order*
  - ***The R→W*** and ***R→R orderings***: which yields a variety of models including *weak ordering*

## Final Remarks on Consistency Models (1)

- Many multiprocessors being built support some sort of relaxed consistency model, varying from processor consistency to release consistency
- Synchronization is highly multiprocessor specific and error prone
- Most programmers use standard synchronization libraries and write synchronized programs, making the choice of a weak consistency model invisible to the programmer and yielding higher performance

## Final Remarks on Consistency Models (2)

- An alternative viewpoint argues that with speculation much of the performance advantage of relaxed consistency models can be obtained with sequential or processor consistency
- A key part of this argument in favor of relaxed consistency revolves the role of the compiler and its ability to optimize memory access to potentially shared variables

## Crosscutting Issues

- Compiler optimization and consistency model
- Using speculation to hide latency in strict consistency models
- Inclusion and its implementation

## Compiler optimization and consistency model (1)

- The range of legal compiler optimizations that can be performed on shared data
- In explicitly parallel programs, unless the synchronization points are clearly defined and the programs are synchronized, the compiler could not interchange a read and a write of two different shared data items
  - —This prevents even relatively simple optimizations, such as register allocation of shared data

## Compiler optimization and consistency model (2)

- In implicitly parallelized programs—for example, those written in High Performance FORTRAN (HPF)—programs must be synchronized and the synchronization points are known, so this issue does not arise

## Using Speculation to Hide Latency in Strict Consistency Models (1)

- It can also be used to hide latency arising from a strict consistency model, giving much of the benefit of a relaxed memory
- The key idea is for:
  - the processor to use dynamic scheduling to reorder memory references, letting them possibly execute out-of-order
- Executing the memory references out-of-order may generate violations of sequential consistency, which might affect the execution of the program

## Using Speculation to Hide Latency in Strict Consistency Models (2)

- This possibility is avoided by using the delayed commit feature of a speculative processor
- Assume the coherency protocol is based on invalidation
  - If the processor receives an invalidation for a memory reference before the memory reference is committed, the processor uses speculation recovery to back-out the computation and restart with the memory reference whose address was invalidated

## Using Speculation to Hide Latency in Strict Consistency Models (3)

- If the reordering of memory requests by the processor yields an execution order that could result in an outcome that differs from what would have been seen under sequential consistency, the processor will redo the execution
- The approach is attractive because the speculative restart will rarely be triggered
  - It will only be triggered when there are unsynchronized access that actually cause a race
  - F.g., Hill (1998) and MIPS R10000

## Using Speculation to Hide Latency in Strict Consistency Models (4)

- One open question is how successful compiler technology will be in optimizing memory references to shared variables
  - The state of optimization technology and the fact that shared data is often accessed via pointers or array indexing has limited the use of such optimizations
  - If this technology became available and led to significant performance advantages, compiler writers would want to be able to take advantage of a more relaxed programming model

## Inclusion and its implementation (1)

- All multiprocessors use multilevel cache hierarchies to reduce both the demand on the global interconnect and the latency of cache misses
- If the cache also provides *multilevel inclusion* — then we can use the multilevel structure to reduce the contention between coherence traffic and processor traffic
  - —Thus most multiprocessors with multilevel caches enforce the *inclusion property*
  - —This restriction is also called the *subset property*

## Inclusion and its implementation (2)

- At first glance, preserving the multilevel inclusion property seems trivial
- Consider a two-level example:
  - Any miss in L1 either hits in L2 or generates a miss in L2, causing it to be brought into both L1 and L2
  - Likewise, any invalidate that hits in L2 must be sent to L1, where it will cause the block to be invalidated, if it exists

## Inclusion and its implementation (3)

- The catch is what happens when the block size of L1 and L2 are different
- Choosing different block sizes is quite reasonable, since L2 will be much larger and have a much longer latency component in its miss penalty, and thus will want to use a larger block size
- What happens to our "automatic" enforcement of inclusion when the block sizes differ?

## Inclusion and its implementation (4)

- A block in L2 represents multiple blocks in L1, and a miss in L2 causes the replacement of data that is equivalent to multiple L1 blocks
- For example, if the block size of L2 is four times that of L1, then a miss in L2 will replace the equivalent of four L1 blocks

## Example

- Assume that L2 has a block size four times that of L1. Show how a miss for an address that causes a replacement in L1 and L2 can lead to violation of the inclusion property.

## Answer (1)

- Assume
  - L1 and L2 are direct mapped
  - the block size of L1 is $b$ bytes and the block size of L2 is $4b$ bytes
- Suppose
  - L1 contains two blocks with starting addresses $x$ and $x + b$
  - $x \bmod 4b = 0$, meaning that $x$ also is the starting address of a block in L2

## Answer (2)

- Single block in L2 contains the L1 blocks $x$, $x + b$, $x + 2b$, and $x + 3b$
- Suppose the processor generates a reference to block $y$ that maps to the block containing $x$ in both caches and hence misses
- Since L2 missed, it fetches $4b$ bytes and replaces the block containing $x$, $x + b$, $x + 2b$, and $x + 3b$, while L1 takes $b$ bytes and replaces the block containing $x$
- Since L1 still contains $x + b$, but L2 does not, the inclusion property no longer holds

## Pitfall 1

- *Measuring performance of multiprocessors by linear speedup versus execution time*
  - —Is the power of the processors being scaled?
  - —The parallel program may be slower on a uniprocessor than a sequential version
  - —*Relative speedup* (same program) and *true speedup* (best program) are sometimes used
  - —*Super-linear*
  - —Comparing performance by comparing speedups is at best tricky and at worst misleading

## Fallacy 1

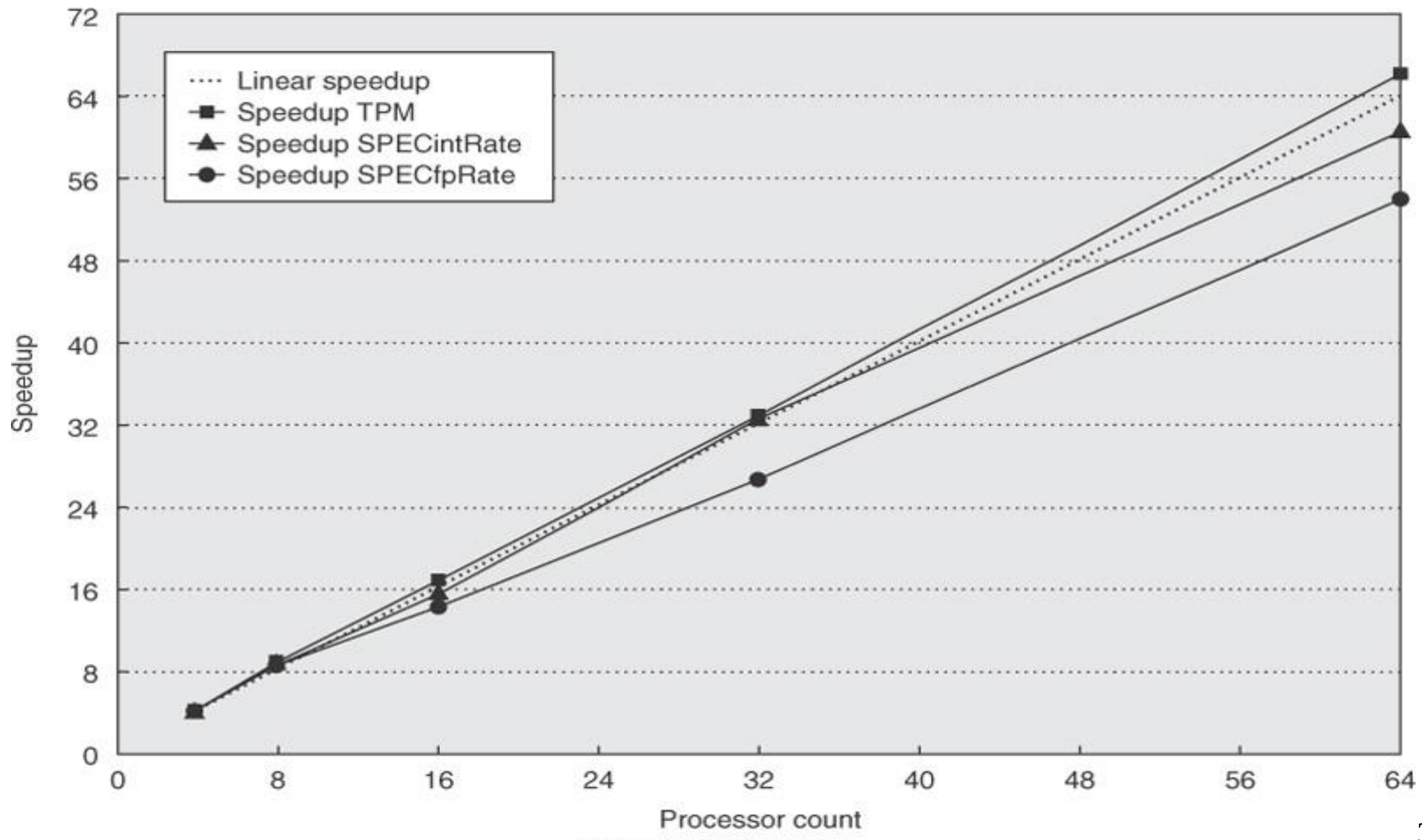
- *Amdahl's Law doesn't apply to parallel computers*
- That Amdahl's Law had been "overcome" was the use of *scaled speedup*
  - —The researchers scaled the benchmark to have a data set size that is 1000 times larger and compared the uniprocessor and parallel execution times of the scaled benchmark

## Fallacy 2 (1)

- *Linear speedups are needed to make multiprocessors cost-effective*
  - —Parallel processors cannot be as cost-effective as uniprocessors unless they can achieve perfect linear speedup
  - —The problem with this argument is that cost is not only a function of processor count, but also depends on memory and I/O

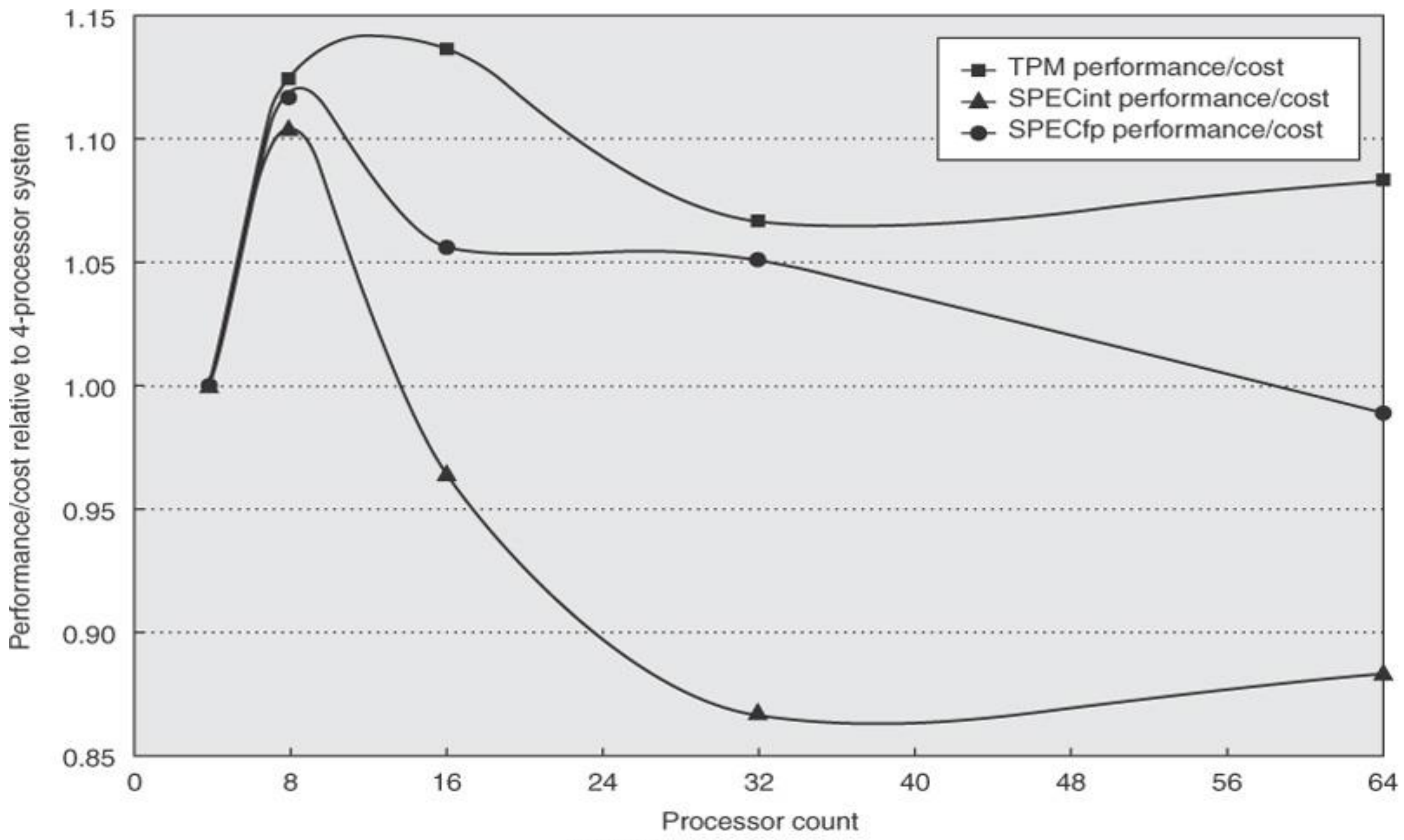
## Fallacy 2 (2)

- On IBM eserver p5 multiprocessor with 4 to 64 processors



## Fallacy 2 (3)

- Performance/cost relative to a 4-processor system



## Fallacy 3

- *Scalability is almost free*
  - It was widely held that you could build scalability into a multiprocessor and then simply offer the multiprocessor at any point on the scale from a small to large number of processors without sacrificing cost effectiveness
  - The difficulty is that multiprocessors require substantially more investment in the *interprocessor communication* network
  - Scalability is also not free in software due to
    - load balance, locality, potential contention for shared resources, and the serial (or partly parallel) portions of the program

## Pitfall 2

- *Not developing the software to take advantage of, or optimize for, a multiprocessor architecture*
  - —There is a long history of software lagging behind on massively parallel processors, possibly because the software problems are much harder
  - —Can we adapt software designed for a uniprocessor to a multiprocessor?
  - —F.g., SGI uses a lock to protect its page table structure, which is critical for multiprocessors
    - – Threads must be serialized by OS to access this page table structure
  - —But page table serialization eliminates parallelism