MULTIPROCESSORS AND THREAD-LEVEL PARALLELISM

B649
Parallel Architectures and Programming

Motivation behind Multiprocessors

- Limitations of ILP (as already discussed)
- Growing interest in servers and server-performance
- Growth in data-intensive applications
- Increasing desktop performance relatively unimportant
- Effectiveness of multiprocessors for server applications
- Leveraging design investment by replication

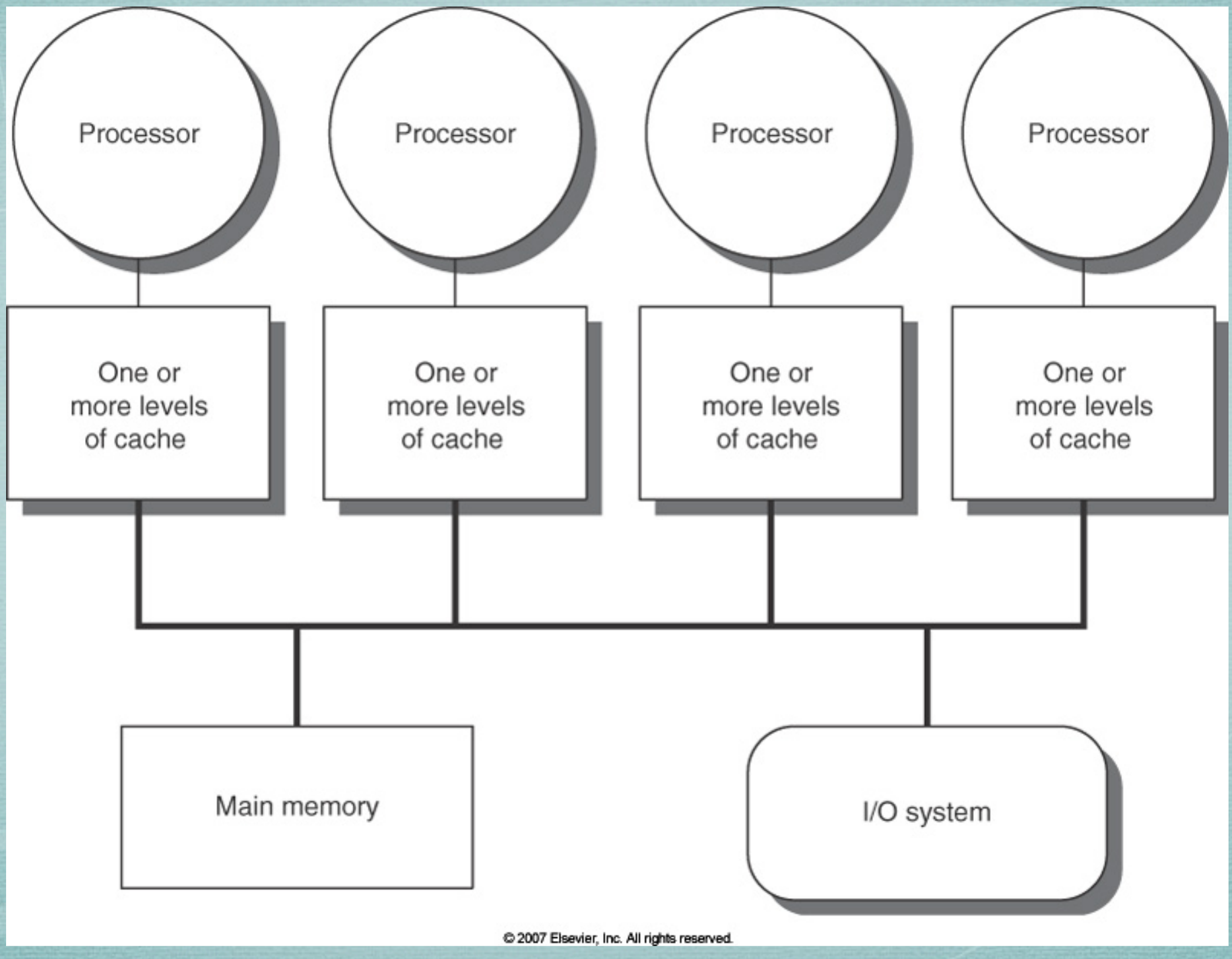
Flynn's Classification of Parallel Architectures

- SISD: Single Instruction Single Data stream
  - ⋆ uniprocessors
- SIMD: Single Instruction Multiple Data streams
  - ⋆ suitable for data parallelism
  - ⋆ Intel's multimedia extensions, vector processors
  - ⋆ growing popularity in graphics applications
- MISD: Multiple instruction Single Data stream
  - ⋆ no commercial multiprocessor to date
- MIMD: Multiple Instruction Multiple Data streams
  - ⋆ suitable for thread-level parallelism

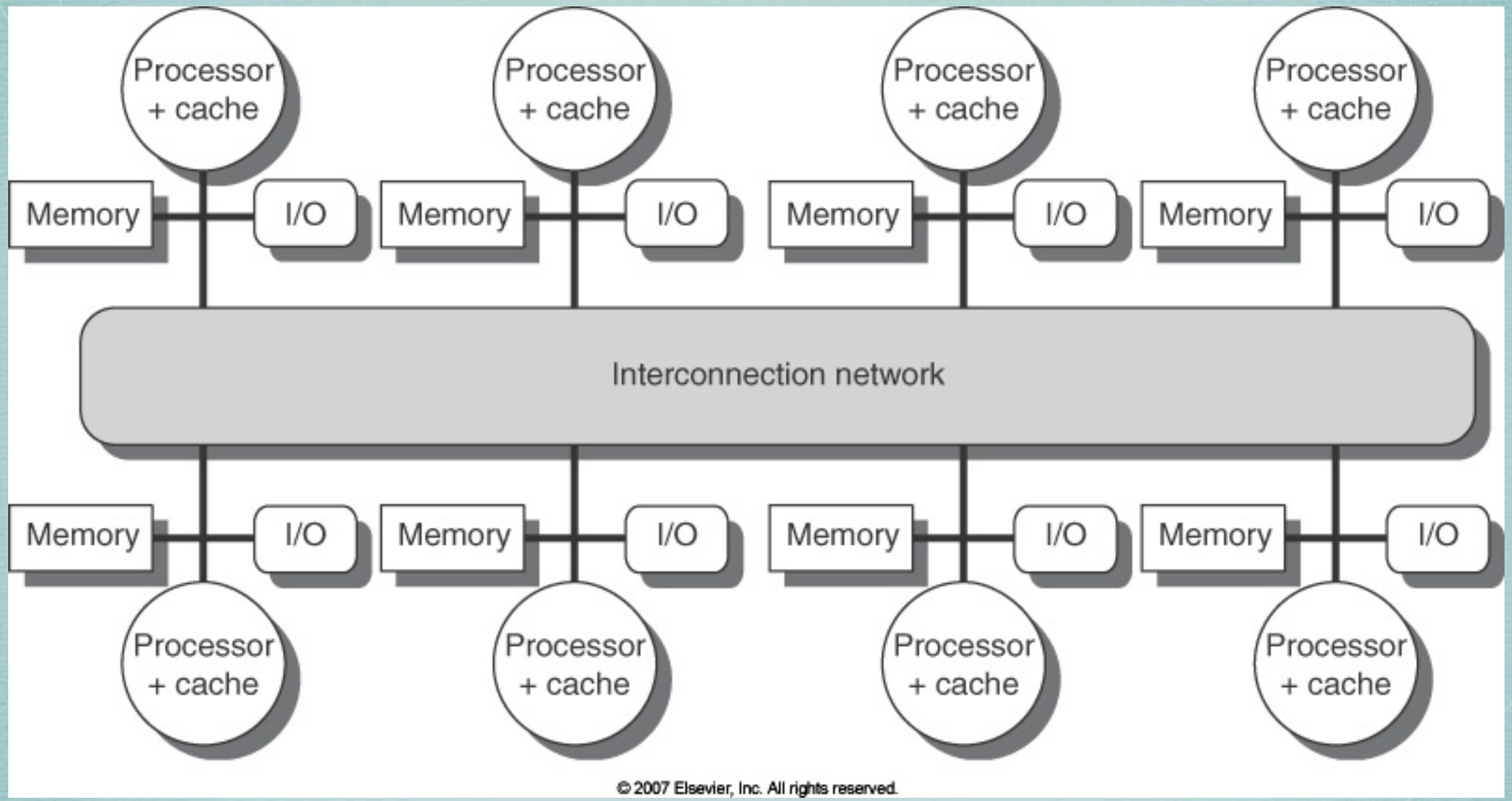
MIMD

- Architecture of choice for general-purpose multiprocessors
- Offers flexibility
- Can leverage the design investment in uniprocessors
- Can use off-the-shelf processors
  - ⋆ "COTS" (Commercial, Off-The-Shelf) processors
- Examples
  - ⋆ Clusters
    - ✳ *commodity* and *custom* clusters
  - ⋆ Multicores

Shared-Memory Multiprocessors



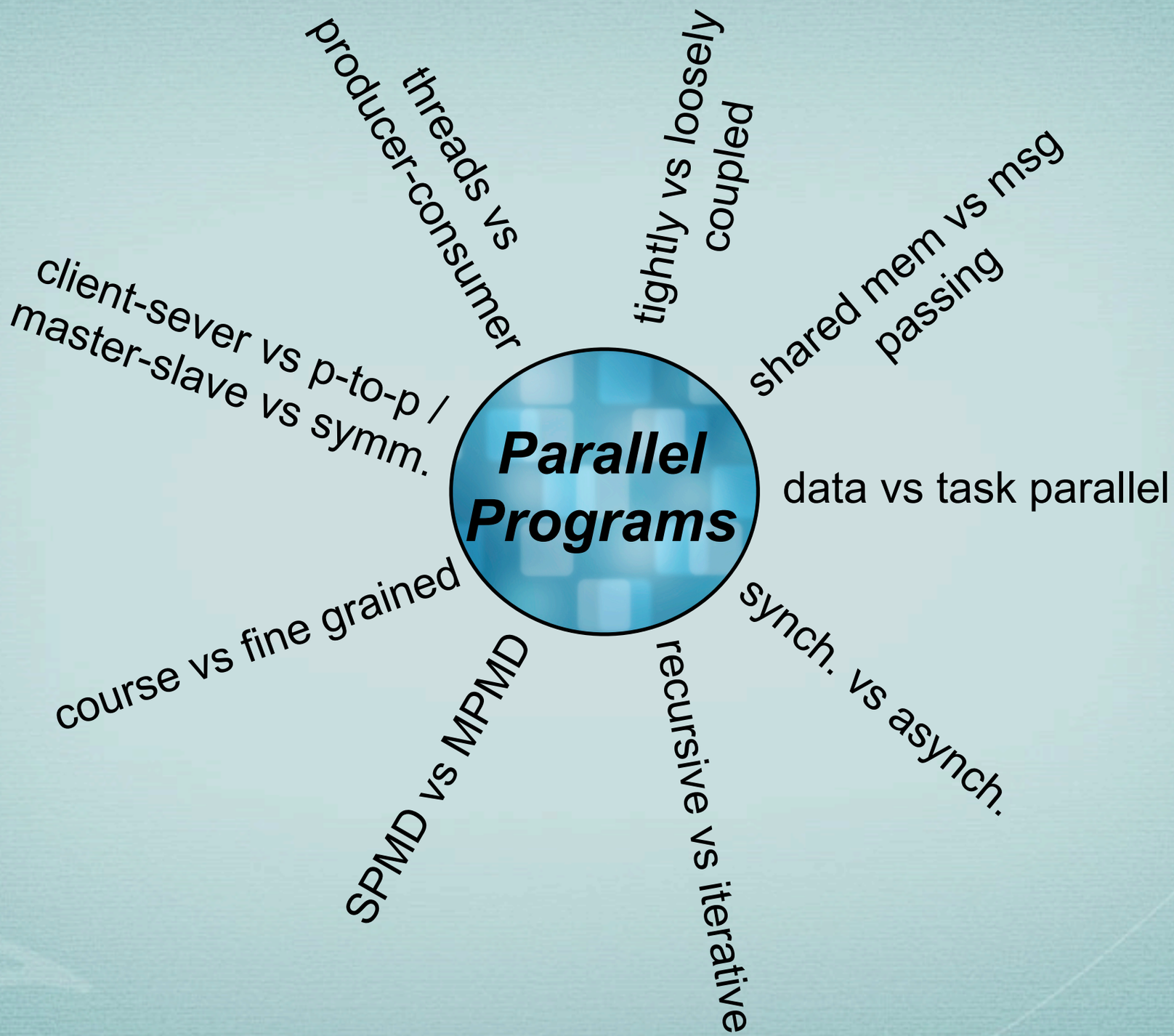
Distributed-Memory Multiprocessors



Models for Memory and Communication

- Memory architecture
  - ⋆ shared memory
    - ✳ Uniform Memory Access (UMA)
    - ✳ Symmetric (Shared-Memory) Multiprocessors (SMPs)
  - ⋆ distributed memory
    - ✳ Non-Uniform Memory Access (NUMA)
- Communication architecture (programming)
  - ⋆ shared memory
  - ⋆ message-passing

Other Ways to Categorize Parallel Programming



CACHES

Terminology

cache
virtual memory
memory stall cycles
direct mapped
valid bit
block address
write through
instruction cache
average memory access time
cache hit
page
miss penalty
fully associative
dirty bit
block offset
write back
data cache
hit time
cache miss
page fault
miss rate
n-way set associative
least-recently used
tag field
write allocate
unified cache
misses per instruction
block
locality
address trace
set
random replacement
index field
no-write allocate
write buffer
write stall

Memory Hierarchy

Level	1	2	3	4
Name	registers	cache	main memory	disk storage
Typical size	< 1 KB	< 16 MB	< 512 GB	> 1 TB
Implementation technology	custom memory with multiple ports, CMOS	on-chip or off-chip CMOS SRAM	CMOS DRAM	magnetic disk
Access time (ns)	0.25–0.5	0.5–25	50–250	5,000,000
Bandwidth (MB/sec)	50,000–500,000	5000–20,000	2500–10,000	50–500
Managed by	compiler	hardware	operating system	operating system/ operator
Backed by	cache	main memory	disk	CD or tape

Four Memory-Hierarchy Questions

- Where can a block be placed in the upper level?
  - ⋆ block placement
- How is a block found if it is in the upper level?
  - ⋆ block indentification
- Which block should be replaced on a miss?
  - ⋆ block replacement
- What happens on a write?
  - ⋆ write strategy

Where Can a Block Be Placed in a Cache?

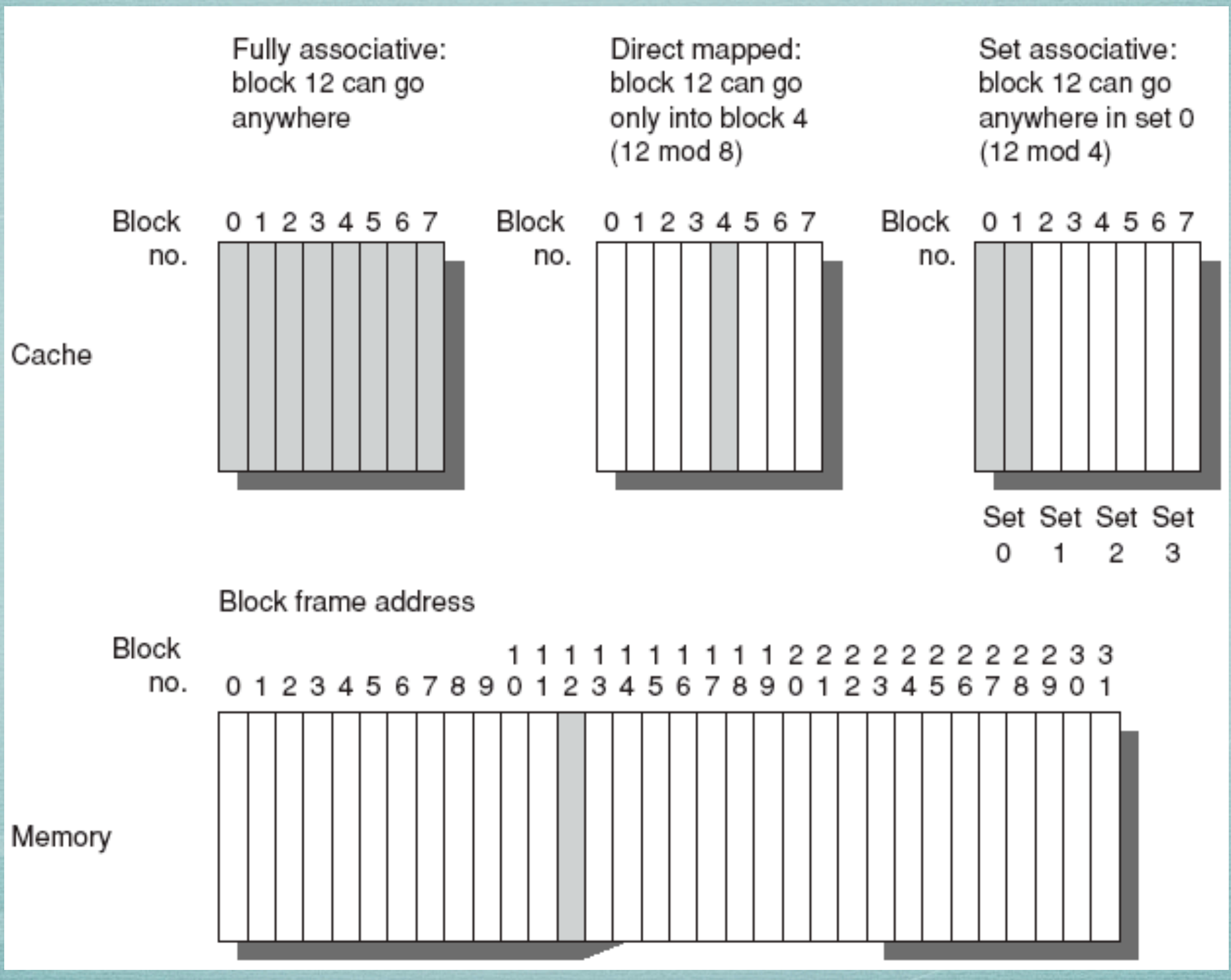
- Only one place for each block
  - ⋆ direct mapped

(Block address) MOD (Number of blocks in cache)

- Anywhere in the cache
  - ⋆ fully associative
- Restricted set of places
  - ⋆ set associative

(Block address) MOD (Number of sets in cache)

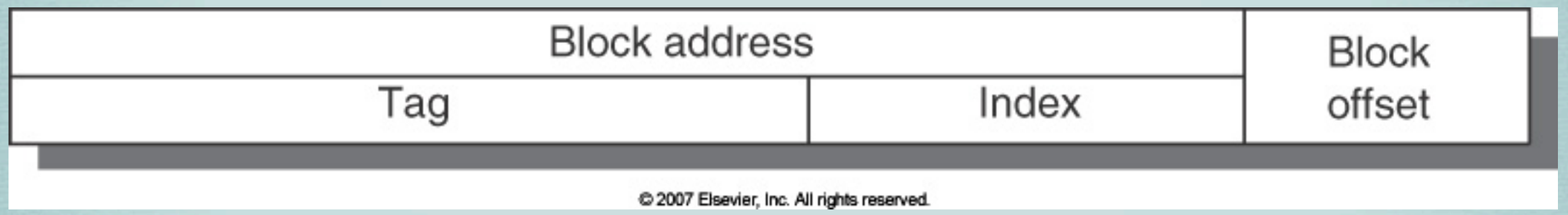
Example


Fully associative: block 12 can go anywhere
Direct mapped: block 12 can go only into block 4 (12 mod 8)
Set associative: block 12 can go anywhere in set 0 (12 mod 4)
Block no.
0 1 2 3 4 5 6 7
Cache
Set 0
Set 1
Set 2
Set 3
Block frame address
Block no.
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Memory

How is a Block Found if it is in Cache?

- "Tags" in each cache block gives the block address
  - ⋆ all possible tags searched in parallel (associative memory)
  - ⋆ *valid bit* tells whether a tag match valid

Fields in a memory address



- No "index" field in fully associative caches

Which Block Should be Replaced on a Miss?

- Random
  - ⋆ easy to implement
- Least-recently used (LRU)
  - ⋆ idea: rely on the past to predict the future
  - ⋆ replace the block unused for the longest time
- First in, First out (FIFO)
  - ⋆ approximates LRU (*oldest*, rather than least recently used)
  - ⋆ simpler to implement

Comparison of Replacement Policies

Data cache misses per 1000 instructions on five SPECint2000 and five SPECfp2000 benchmarks

	Associativity								
	Two-way			Four-way			Eight-way		
Size	LRU	Random	FIFO	LRU	Random	FIFO	LRU	Random	FIFO
16 KB	114.1	117.3	115.5	111.7	115.1	113.3	109.0	111.8	110.4
64 KB	103.4	104.3	103.9	102.4	102.3	103.1	99.7	100.5	100.3
256 KB	92.2	92.1	92.5	92.1	92.1	92.5	92.1	92.1	92.5

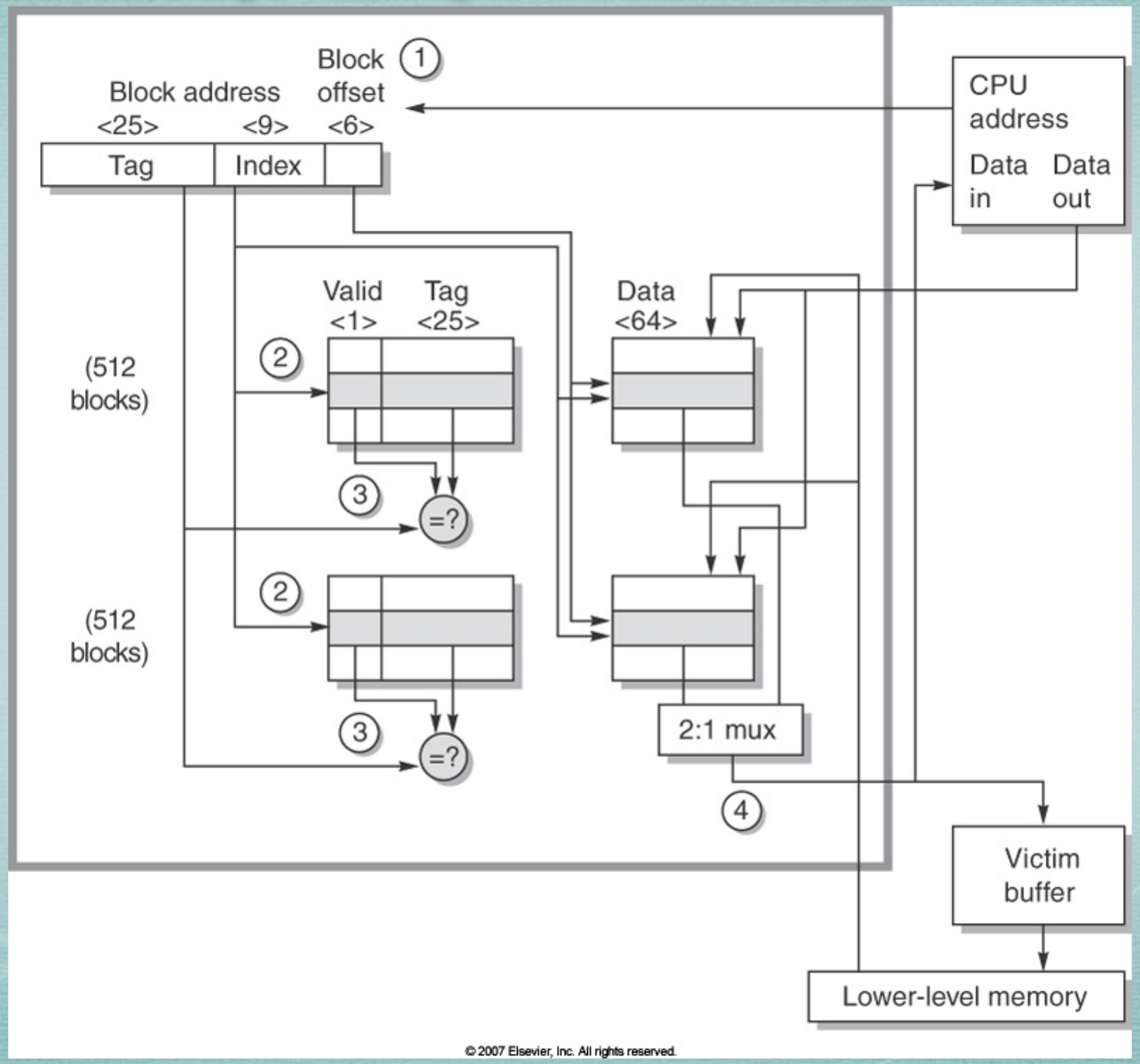
What Happens on a Write?

- Reads dominate
  - ⋆ 7% of the overall memory traffic are writes
  - ⋆ 28% of the data cache traffic are writes
- Write takes longer
  - ⋆ reading cache line and validity check can be parallel
  - ⋆ reads can read the whole line, write must modify only the specified bytes

Handling Writes

- Write strategy
  - ⋆ write through
    - ✳ write to cache block **and** to the block in the lower-level memory
  - ⋆ write back
    - ✳ write only to cache block, update the lower-level memory when block replaced
- Block allocation strategy
  - ⋆ write allocate
    - ✳ allocate a block on cache miss
  - ⋆ no-write allocate
    - ✳ do **not** allocate, no affect on cache

Example: Opteron Data Cache



Terminology

cache
virtual memory
memory stall cycles
direct mapped
valid bit
block address
write through
instruction cache
average memory access time
cache hit
page
miss penalty
fully associative
dirty bit
block offset
write back
data cache
hit time
cache miss
page fault
miss rate
n-way set associative
least-recently used
tag field
write allocate
unified cache
misses per instruction
block
locality
address trace
set
random replacement
index field
no-write allocate
write buffer
write stall

SYMMETRIC SHARED-MEMORY: CACHE COHERENCE

Quotes

Quotes

We are dedicating all of our future product development to multicore designs. We believe this is a key inflection point for the industry.

Intel President Paul Otellini,
describing Intel's future direction at the Intel Developers Forum in 2005

Quotes

> The turning away from conventional organization came in the middle 1960s, when the law of diminishing returns began to take effect in the effort to increase the operational speed of a computer ... Electronic circuits are ultimately limited in their speed of operation by the speed of light ... and many of the circuits were already operating in the nanosecond range.
>
> **W. Jack Bouknight et al.**
> *The Illiac IV System (1972)*

> We are dedicating all of our future product development to multicore designs. We believe this is a key inflection point for the industry.
>
> **Intel President Paul Otellini**,
> *describing Intel's future direction at the Intel Developers Forum in 2005*

Multiprocessor Cache Coherence

X is not in any cache initially. The caches are write-through.

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

Multiprocessor Cache Coherence

X is not in any cache initially. The caches are write-through.

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

Proposed definition: Memory system is coherent if any read of a data item returns the most recently written value of that data item.

Multiprocessor Cache Coherence

X is not in any cache initially. The caches are write-through.

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

Proposed definition: Memory system is coherent if any read of a data item returns the most recently written value of that data item.

Too simplistic!

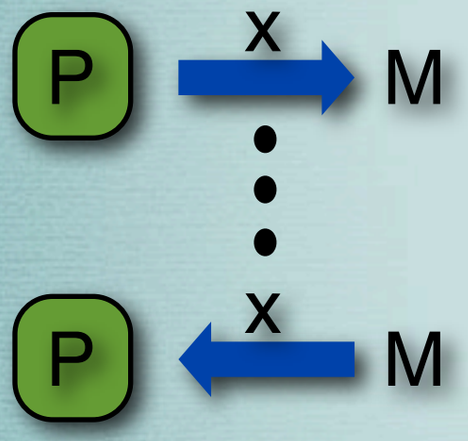
Coherency: Take 2

- A memory system is coherent if:

Coherency: Take 2

- A memory system is coherent if:

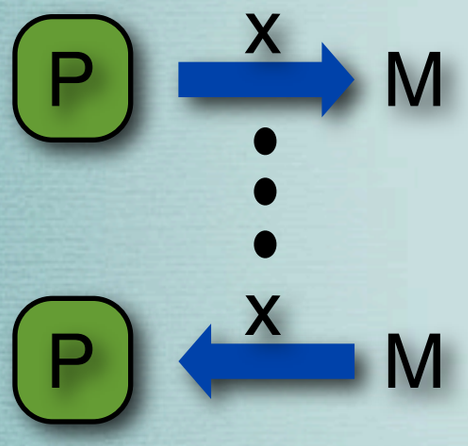
1. Writes and Reads
by one processor



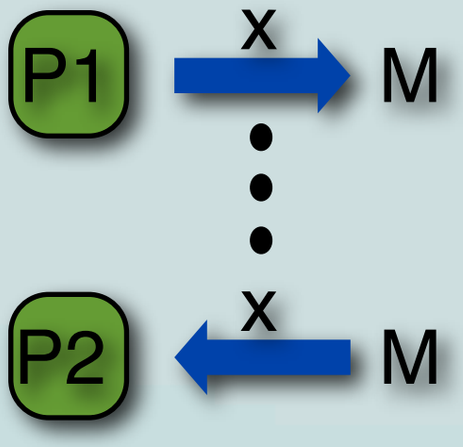
Coherency: Take 2

- A memory system is coherent if:

1. Writes and Reads by one processor



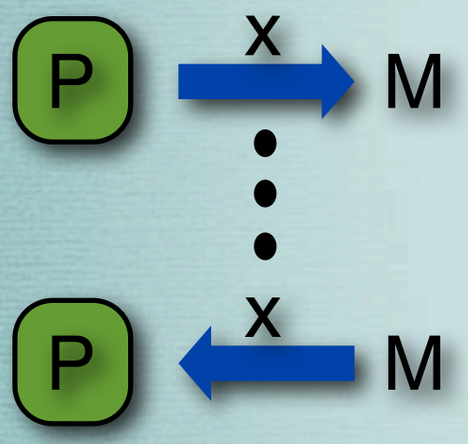
2. Writes and Reads by two processors



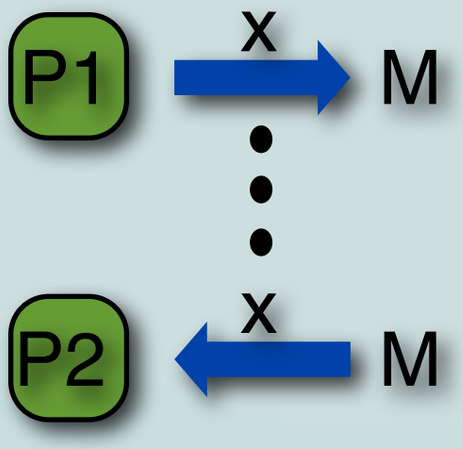
Coherency: Take 2

- A memory system is coherent if:

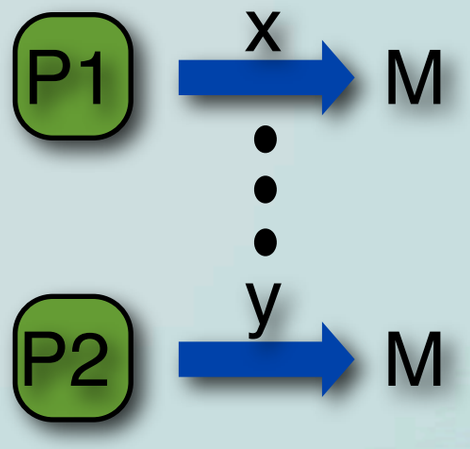
1. Writes and Reads by one processor

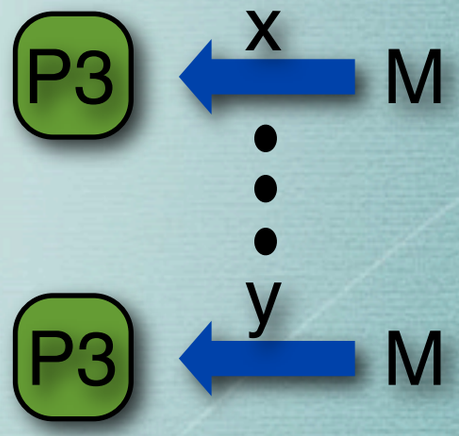


2. Writes and Reads by two processors



3. Writes by two processors (serialization)





Coherence vs Consistency

- Coherence defines the behavior of reads and writes to the same memory location
- Consistency defines the behavior of reads and writes with respect to accesses to other memory locations
  - ⋆ we will return to consistency later
- Working assumptions:
  - ⋆ a write does not complete until all processors have seen the effect of that write
  - ⋆ processor does not change the order of an write with respect to any other memory access

What Needs to Happen for Coherence?

- Migration
  - ⋆ data can move to local cache, when needed
- Replication
  - ⋆ data may be replicated in local cache, when needed
- Need a protocol to maintain the coherence property
  - ⋆ specialized hardware

Coherence Protocols

- Directory based
  - ⋆ central location (directory) maintains the sharing state of a block of physical memory
  - ⋆ slightly higher implementation cost
  - ⋆ scalable
  - ⋆ most used for distributed-memory multiprocessors
- Snooping
  - ⋆ each cache maintains sharing state of the blocks it contains
  - ⋆ shared broadcast medium (e.g., bus)
  - ⋆ each cache **snoops** on the medium to determine whether they a copy of the block that is requested
  - ⋆ most used for shared-memory multiprocessors

Snooping Protocols: Handling Writes

- Write invalidate
  - ⋆ write requires exclusive access
  - ⋆ any copy held by the reading processor is **invalidated**
  - ⋆ if two processors attempt to write, one wins the race
- Write update (or write broadcast)
  - ⋆ **update** all the cached copies of a data item when written
  - ⋆ consumes substantially more bandwidth than invalidating-based protocol
  - ⋆ not used in recent multiprocessors

Example of Invalidation Protocol

Write-back caches

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 Invalidate Protocol: Observations

- Serialization through access to the broadcast medium
- Need to locate data item upon miss
  - ⋆ simple on write-through caches
    - ✳ write-buffers may complicate this
    - ✳ write-through increases the memory bandwidth requirement
  - ⋆ more complex on write-back caches
    - ✳ caches snoop for read addresses, supply matching dirty block
    - ✳ no need to write back dirty block if cached elsewhere
    - ✳ preferred approach on most modern multiprocessors, due to lower memory bandwidth requirements
- Cache tags and valid bits can do double duty
- Additional bit to indicate whether block shared
- Desirable to reduce contention on cache between processor and snooping

Invalidation-Based Coherence Protocol for Write-Back Caches with Allocate on Write

Invalidation-Based Coherence Protocol for Write-Back Caches with Allocate on Write

- Idea: Shared read, exclusive write

Invalidation-Based Coherence Protocol for Write-Back Caches with Allocate on Write

- Idea: Shared read, exclusive write
- Read
  - ⋆ Hit: get from local cache
  - ⋆ Miss: get from memory or another processor's cache; write-back existing block if needed

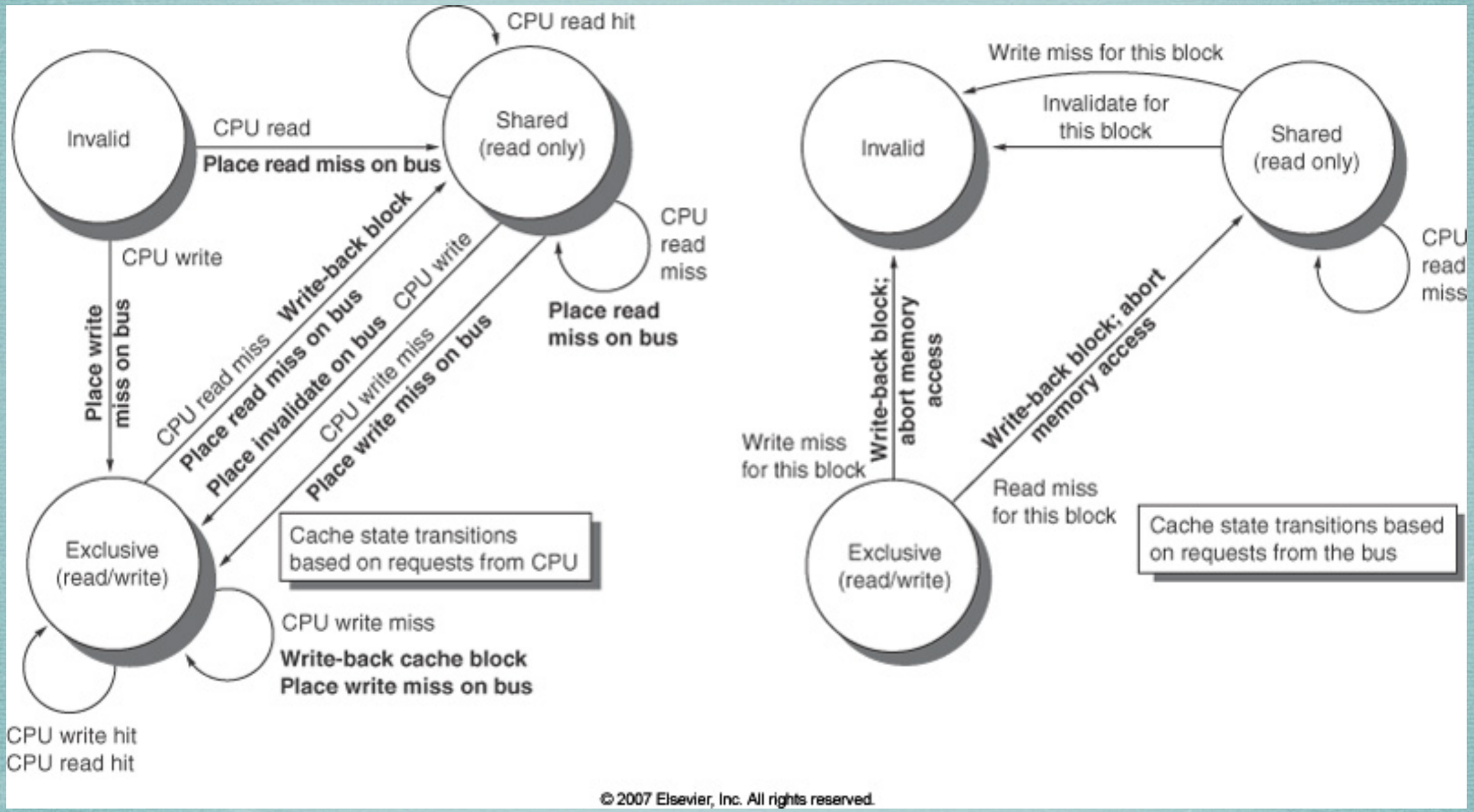
Invalidation-Based Coherence Protocol for Write-Back Caches with Allocate on Write

- Idea: Shared read, exclusive write
- Read
  - ⋆ Hit: get from local cache
  - ⋆ Miss: get from memory or another processor's cache; write-back existing block if needed
- Write
  - ⋆ Hit: write in cache, mark the block exclusive (invalidate copies at other processors)
  - ⋆ Miss: get from memory or another processor's cache; write-back existing block if needed

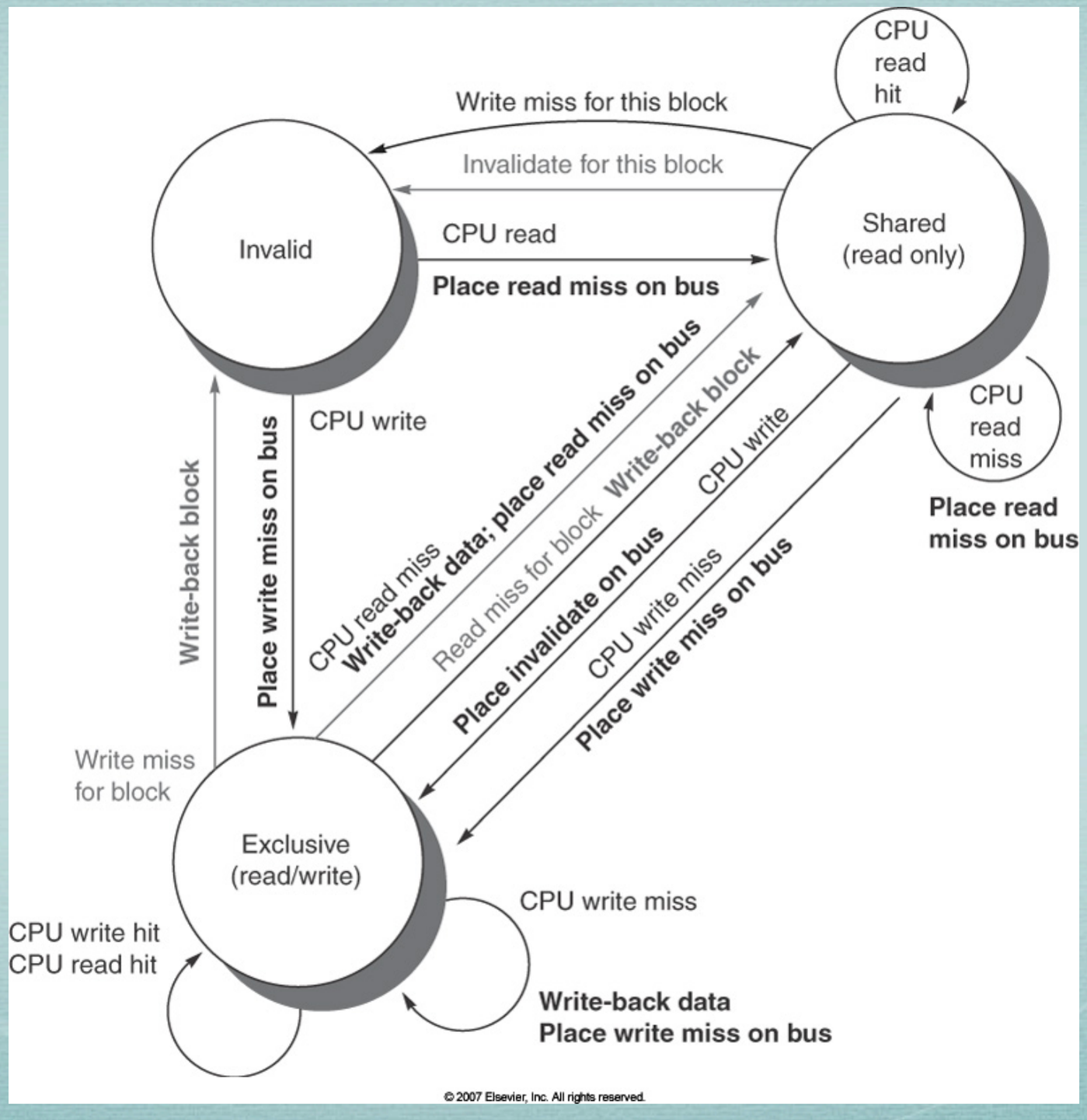
Cache Coherence Mechanism (MESI or MOESI)

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.

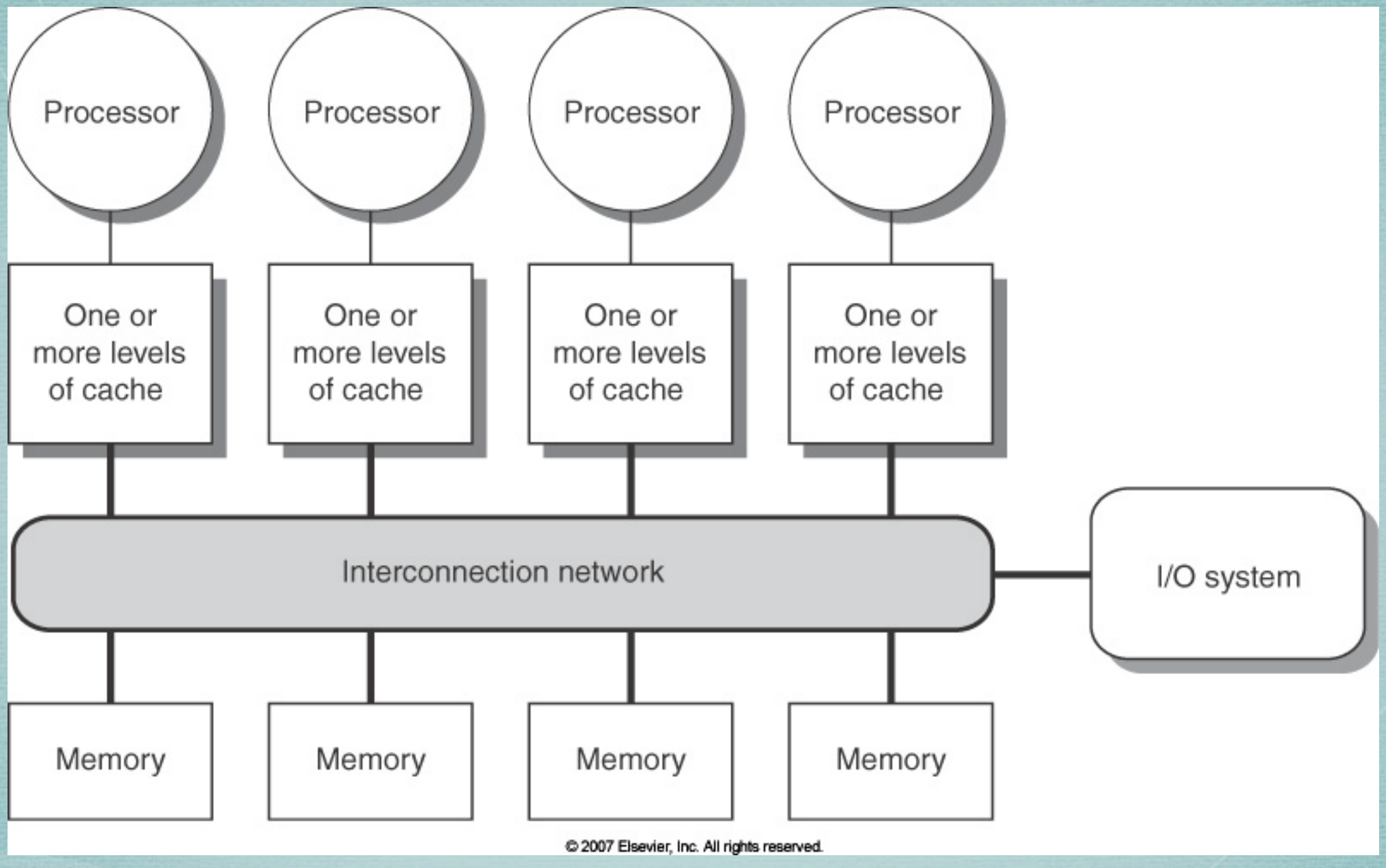
Write Invalidate Cache Coherence Protocol for Write-Back Caches



Write Invalidate Cache Coherence Protocol for Write-Back Caches



Interconnection Network, Instead of Bus



SYMMETRIC SHARED-MEMORY: PERFORMANCE

Performance Issues

- Cache misses
  - ⋆ capacity
  - ⋆ compulsory
  - ⋆ conflict
- Cache coherence
  - ⋆ true-sharing misses
  - ⋆ false-sharing

Performance Issues

- Cache misses
  - ⋆ capacity
  - ⋆ compulsory
  - ⋆ conflict
- Cache coherence
  - ⋆ true-sharing misses
  - ⋆ false-sharing

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

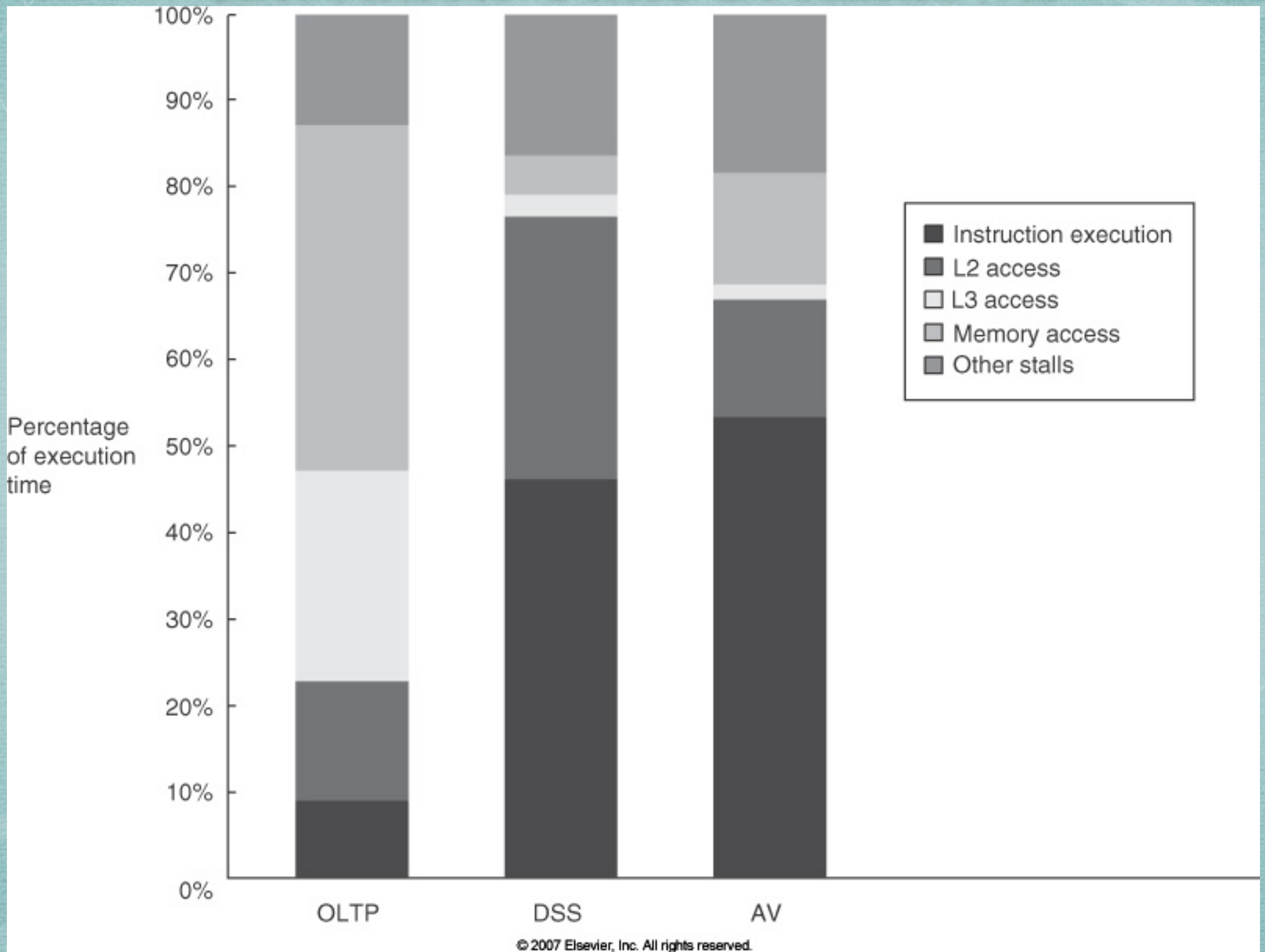
Performance: Alpha-Server

- Machine
  - ⋆ Alpha-Server 4100, 4 processors: processor Alpha 21164 (four issue)
- Three-level cache
  - ⋆ L1: 8KB direct-mapped, separate instruction and data, 32-byte block size, write through, on-chip
  - ⋆ L2: 96KB 3-way set-associative, unified, 32-byte block size, write back, on-chip
  - ⋆ L3: 2MB direct-mapped, unified, 64-byte block size, write back, off-chip
- Latencies
  - ⋆ L2: 7 cycles, L3: 21 cycles, memory: 80 cycles

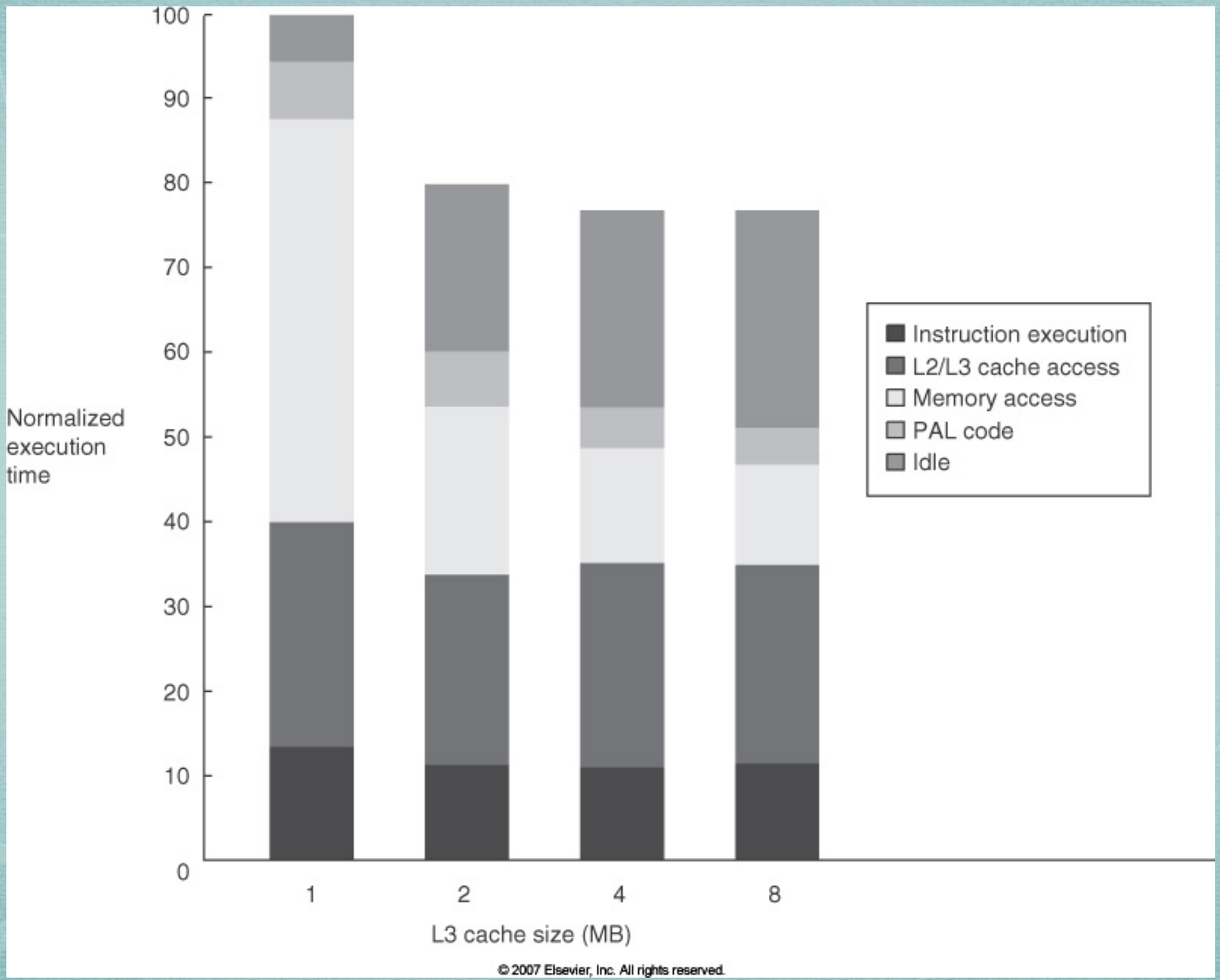
Performance: Commercial Workload

- Online Transaction-Processing (OLTP)
  - ⋆ modeled after TPC-B
  - ⋆ client-server
- Decision Support System (DSS)
  - ⋆ modeled after TPC-D
  - ⋆ long-running queries against large complex data structures (obsoleted)
- Web index search
  - ⋆ AltaVista, using 200 GB memory-mapped database
- I/O time ignored (substantial for these applications)

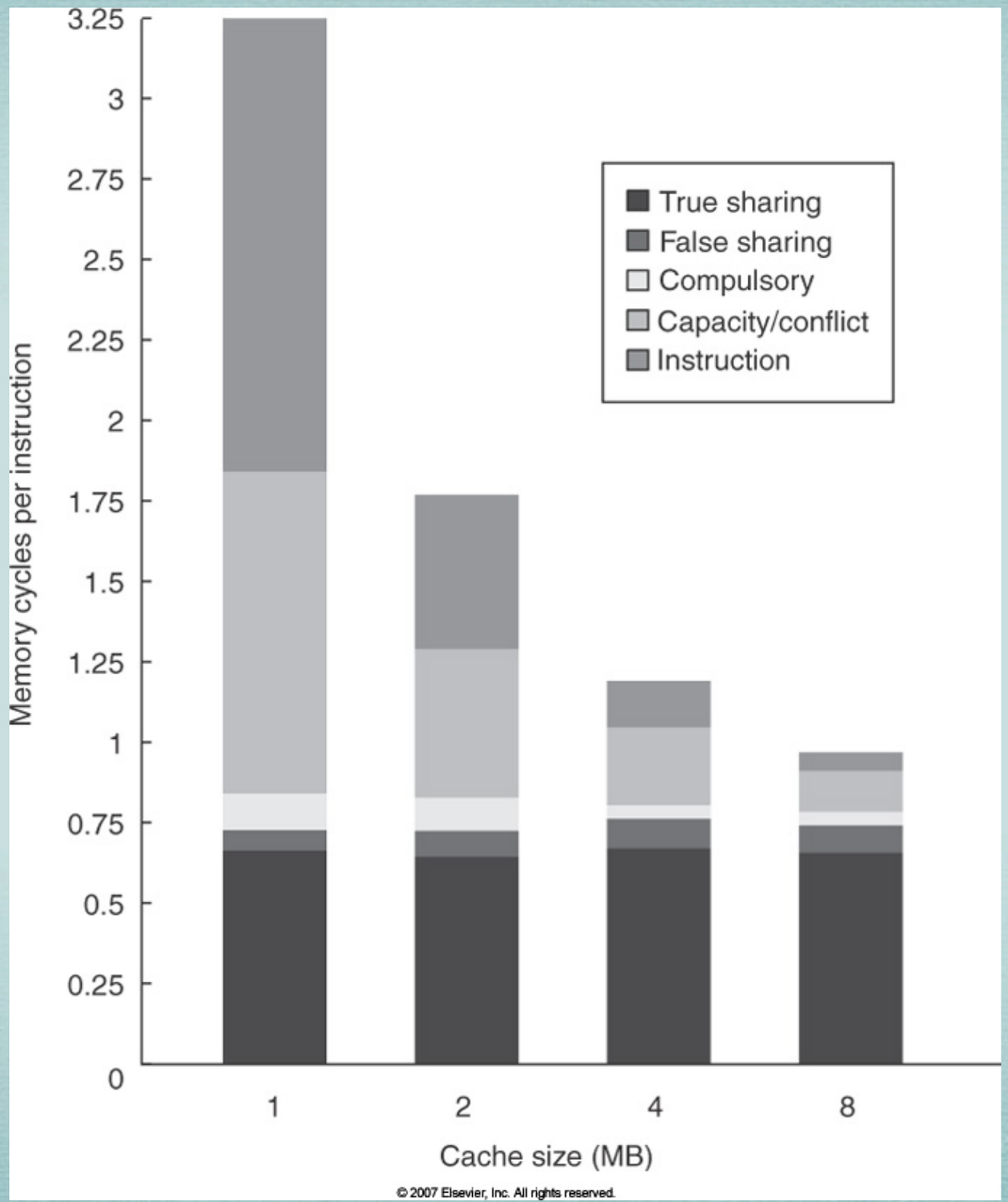
Execution Time Breakdown



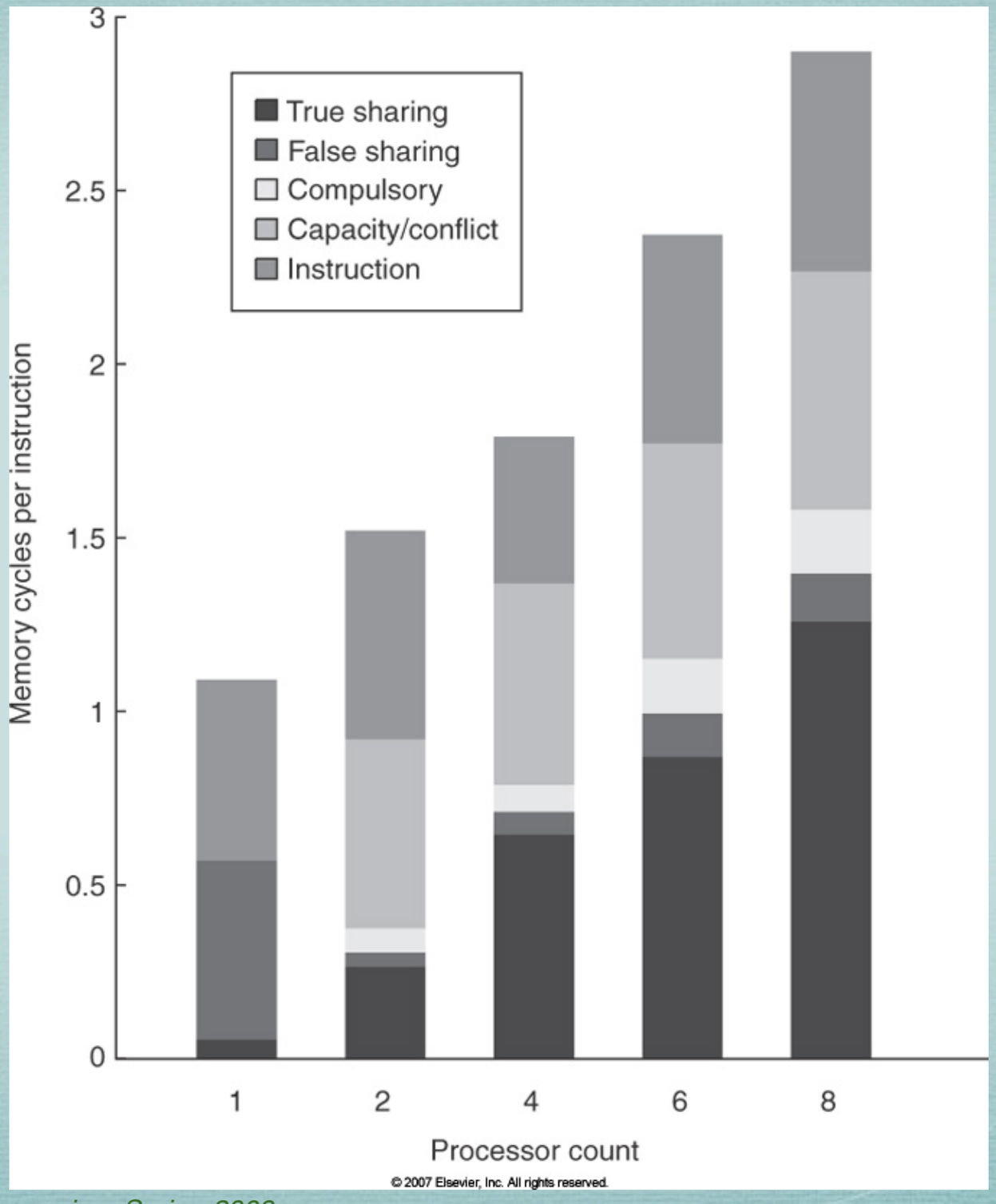
OLTP with Different Cache Sizes



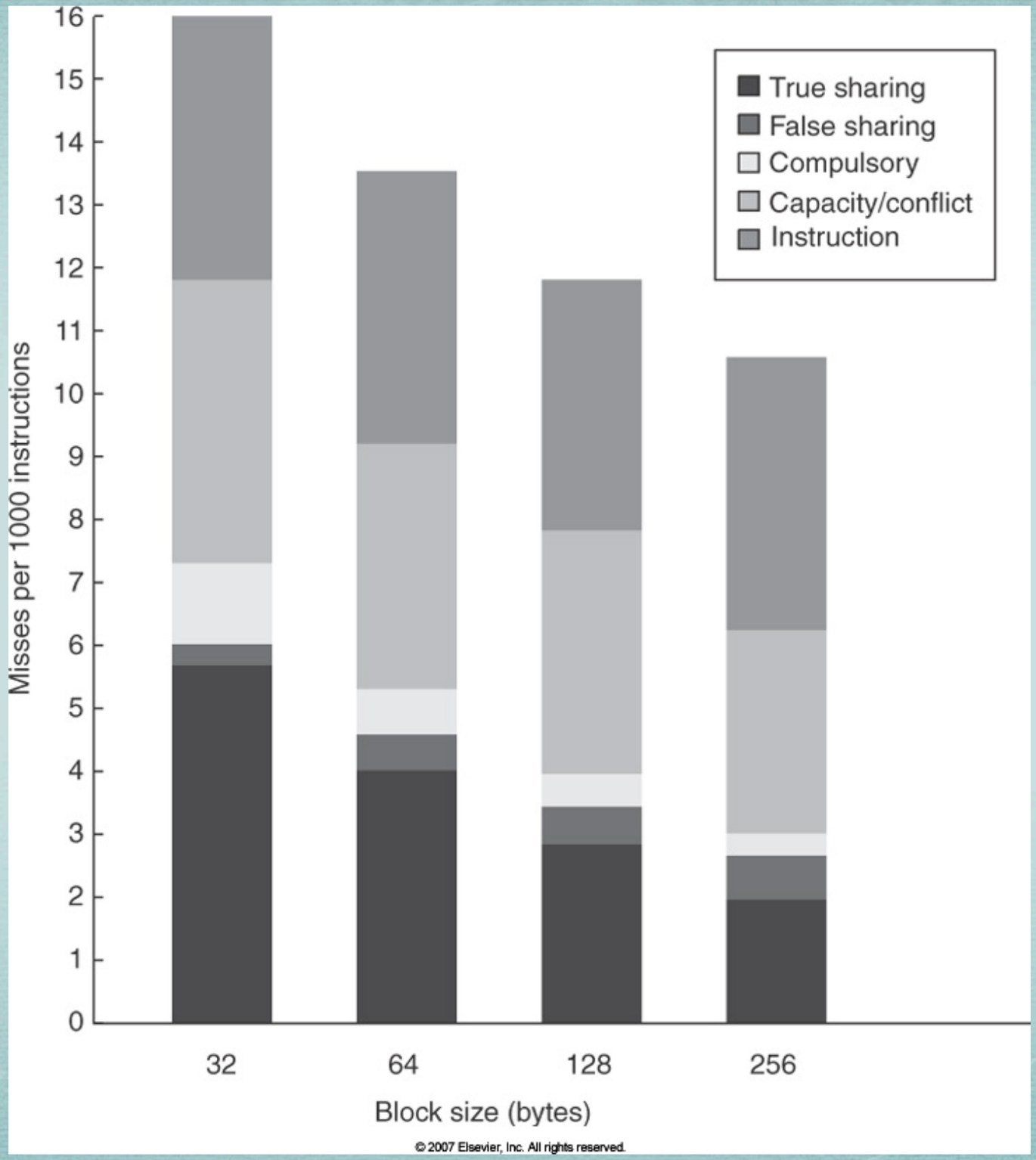
OLTP: Contributing Causes of L3 Cycles



OLTP Mem. Access Cycles with Processor Count



OLPT Misses with L3 Cache Block Size



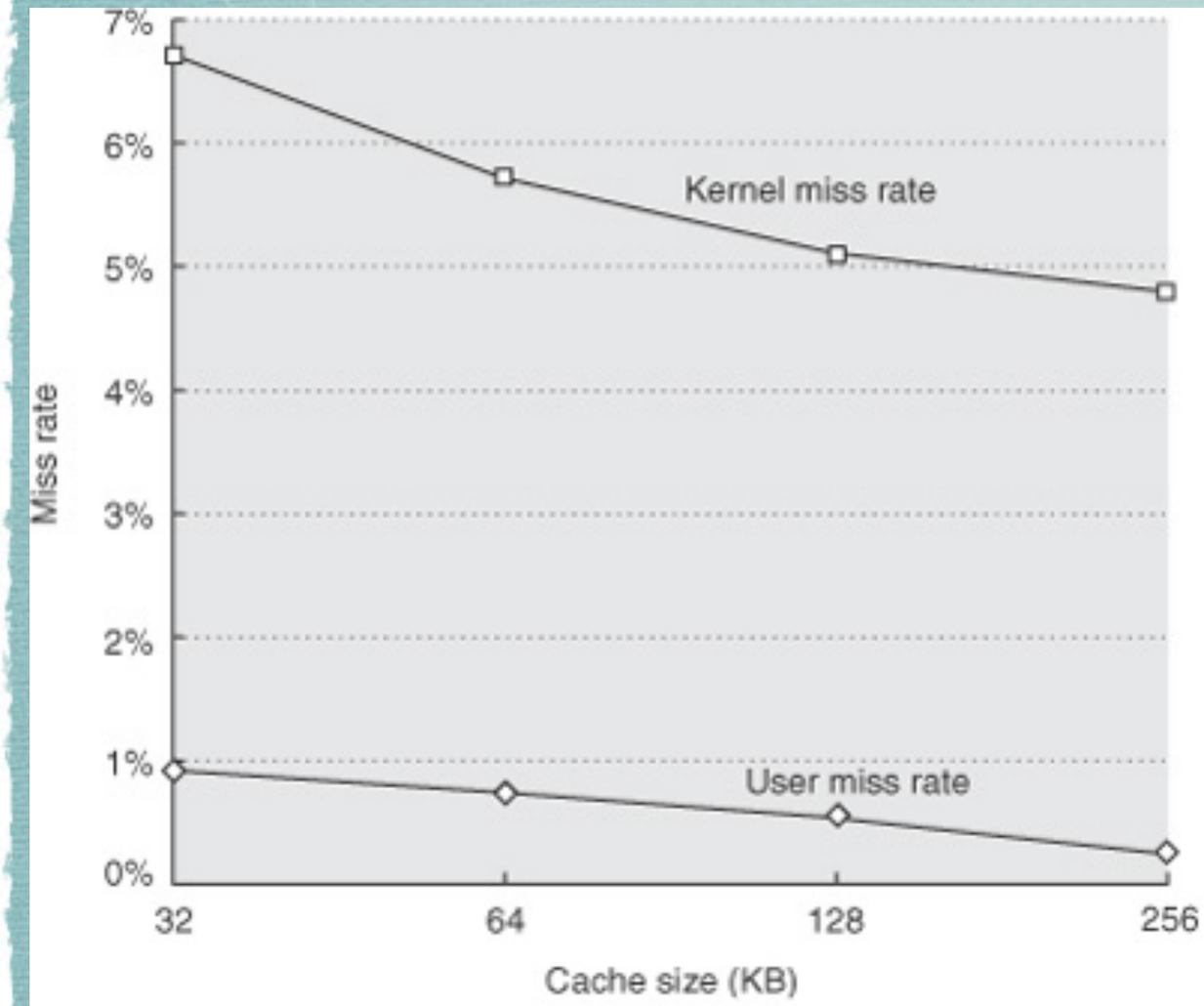
Multiprogramming and OS Workload

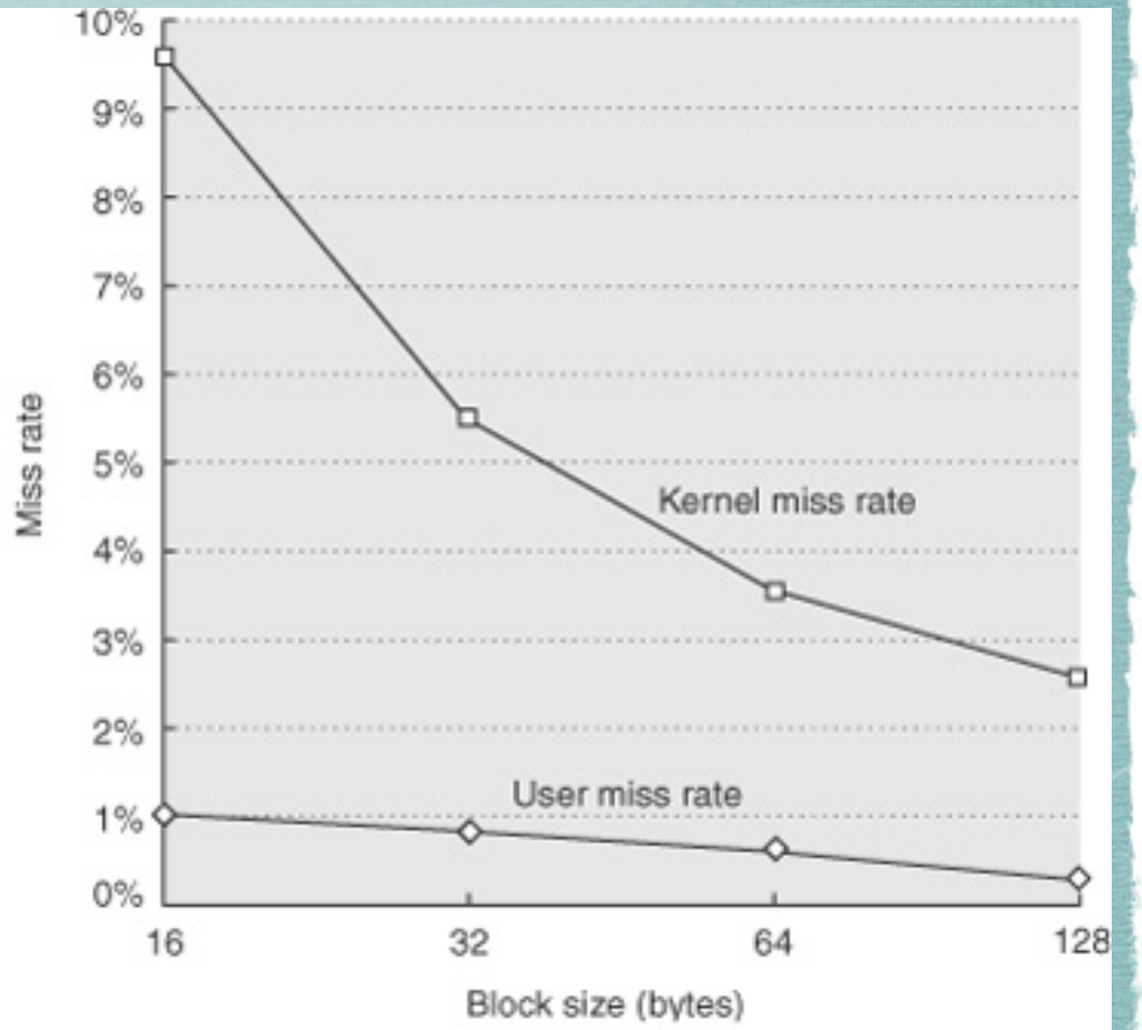
- Models user and OS activities
- Andrew benchmark, emulates software development
  - ⋆ compiling
  - ⋆ installing object files in a library
  - ⋆ removing object files
- Memory hierarchy
  - ⋆ L1 instruction cache: 32KB, 2-way set-associative, 64-byte block; L1 data cache: 32KB 2-way set-associative, 32-byte block
  - ⋆ L2: 1MB unified, 2-way set assoc., 128-byte block
  - ⋆ Memory: 100 clock cycles access

Distribution of Execution Time

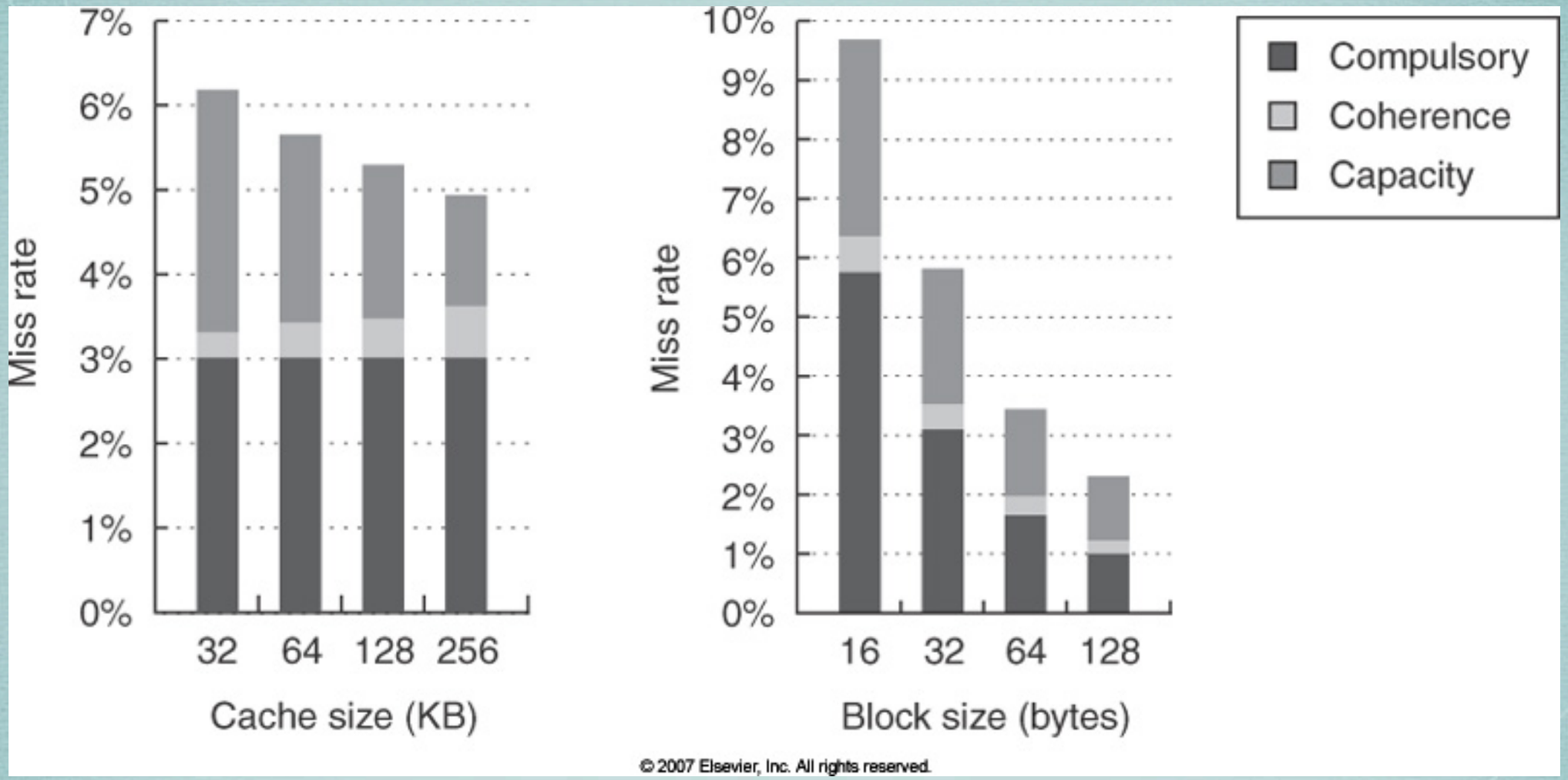
	User execution	Kernel execution	Synchronization wait	CPU idle (waiting for I/O)
% instructions executed	27	3	1	69
% execution time	27	7	2	64

L1 Size and L1 Block Size

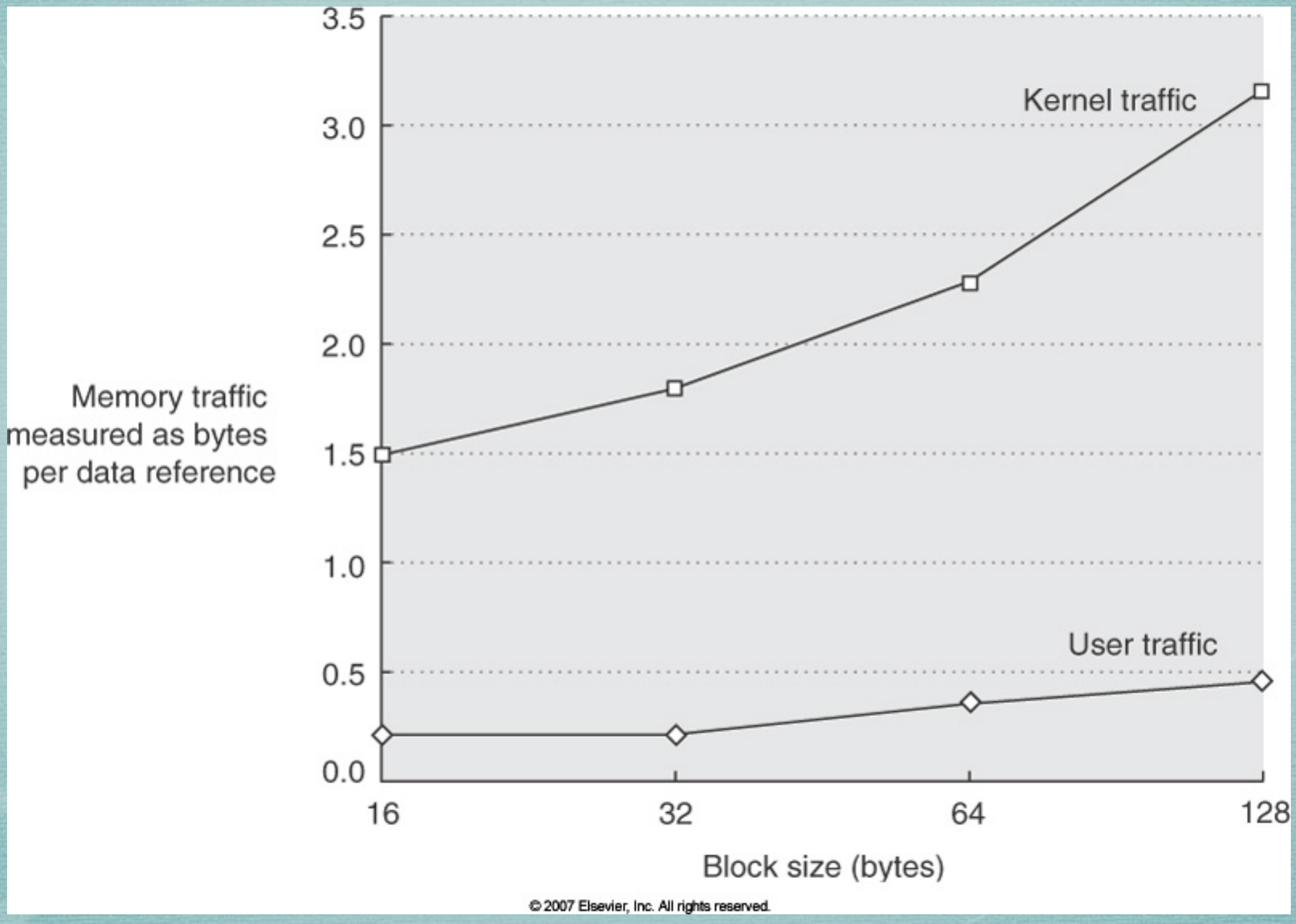




Kernel Data Cache Miss Rates

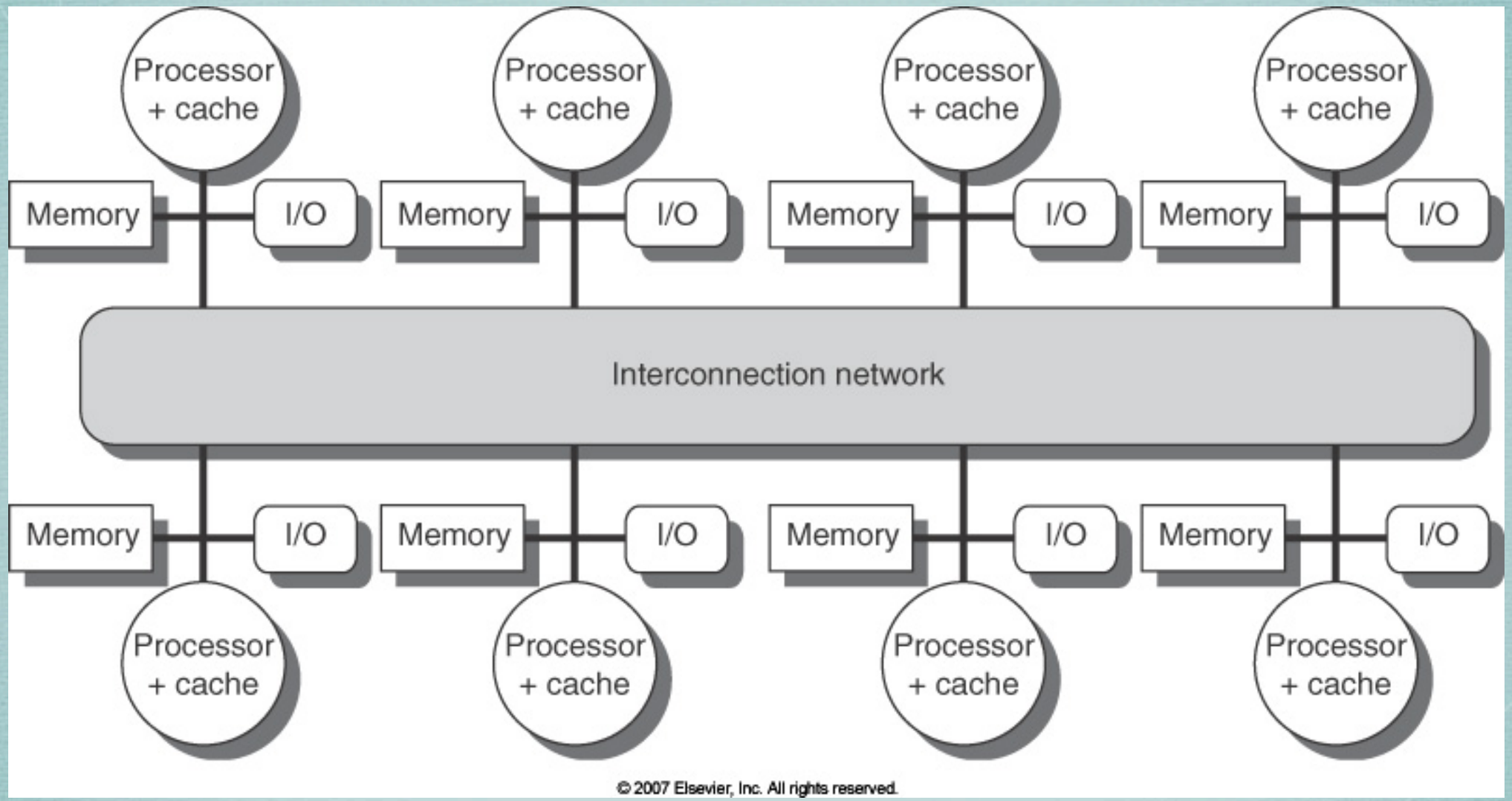


Memory Traffic Per Data Reference

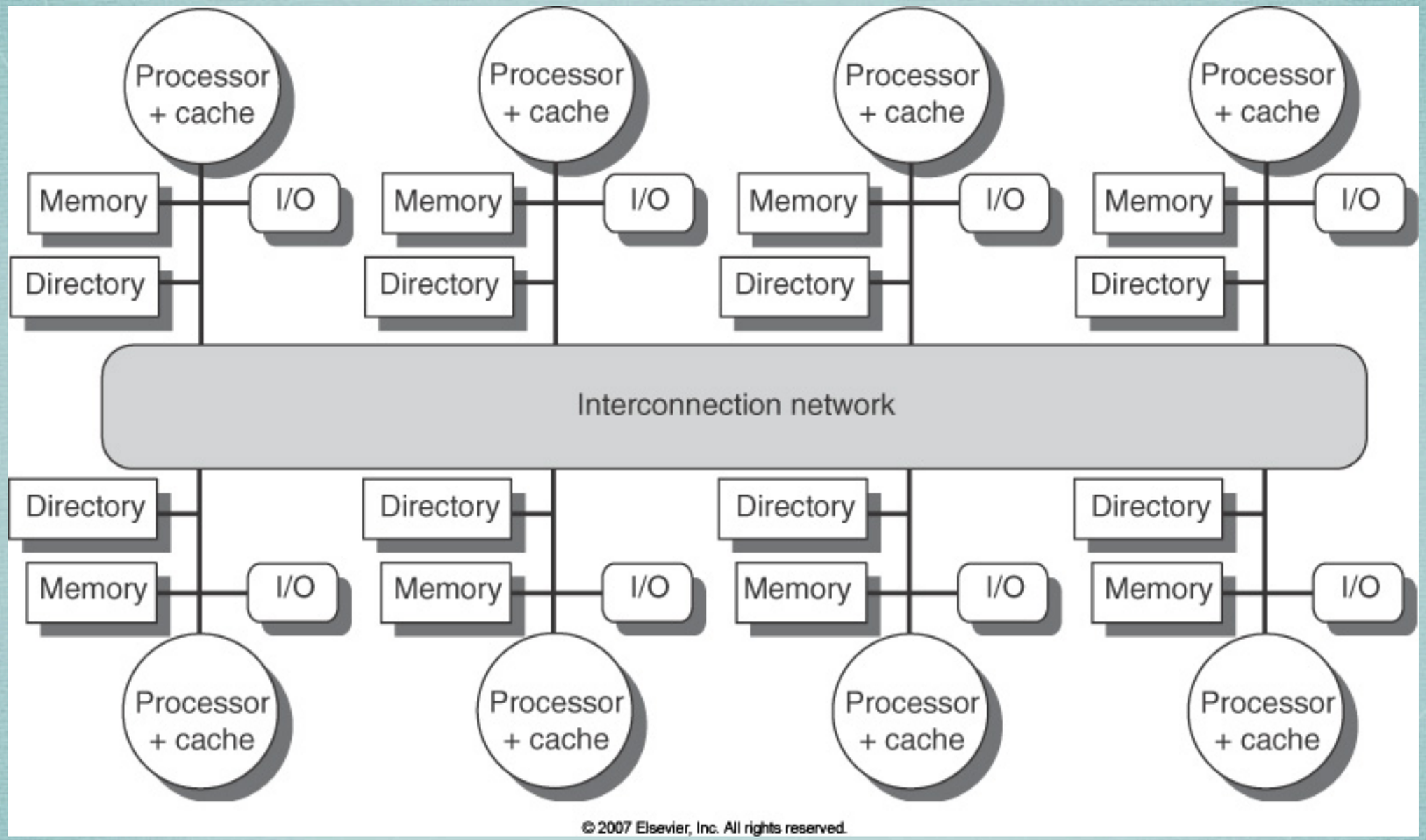


DISTRIBUTED SHARED-MEMORY: DIRECTORY-BASED COHERENCE

Distributed Memory



Distributed Memory+Directories



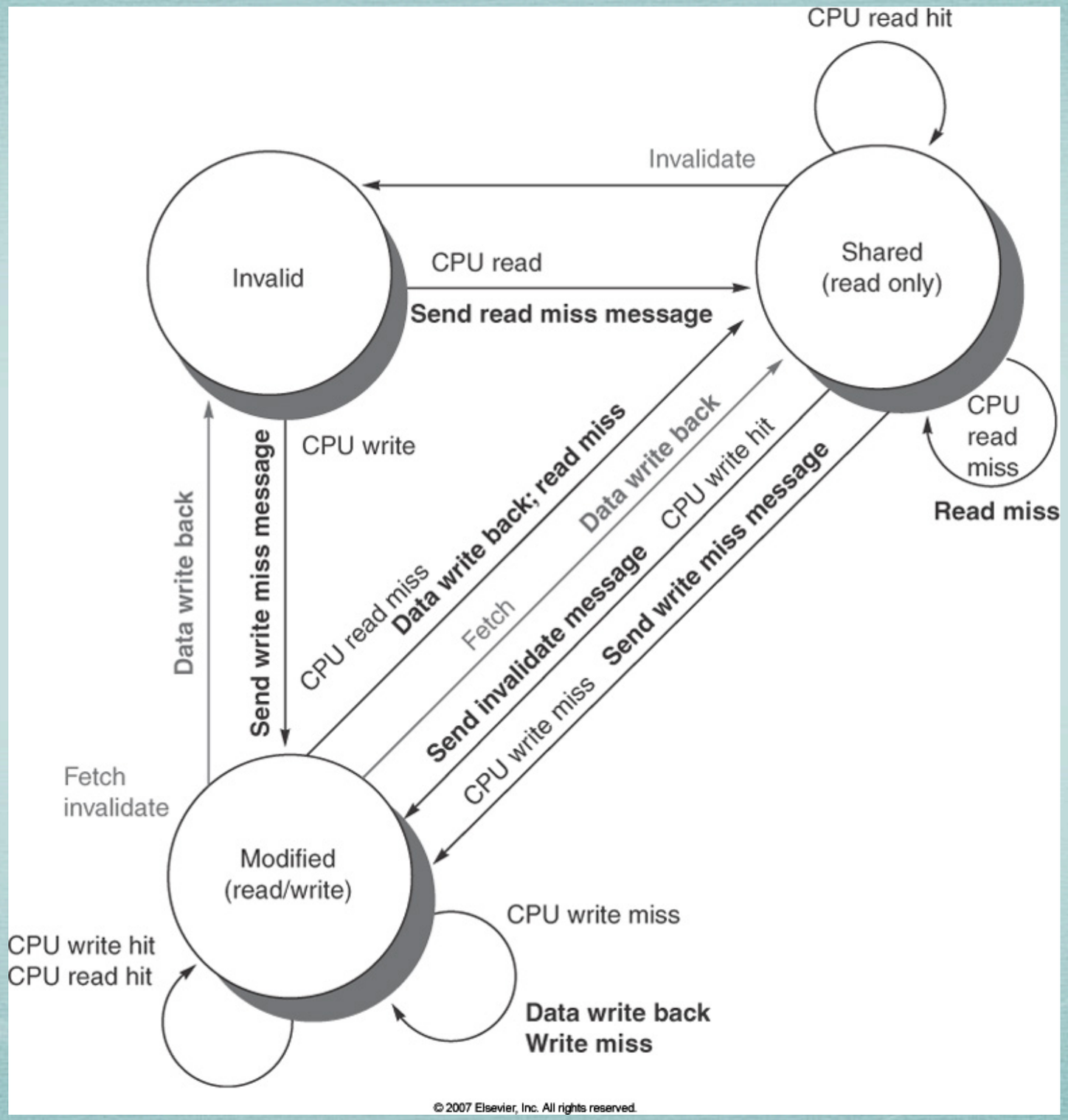
Directory-Based Protocol States

- Shared
  - ⋆ one or more processors have the block cached
  - ⋆ memory has up to date value
- Uncached
  - ⋆ no processor has a copy of the cache block
- Modified
  - ⋆ exactly one processor has a copy of the cache block
  - ⋆ memory copy is out of date
  - ⋆ the processor with the copy is the block's **owner**

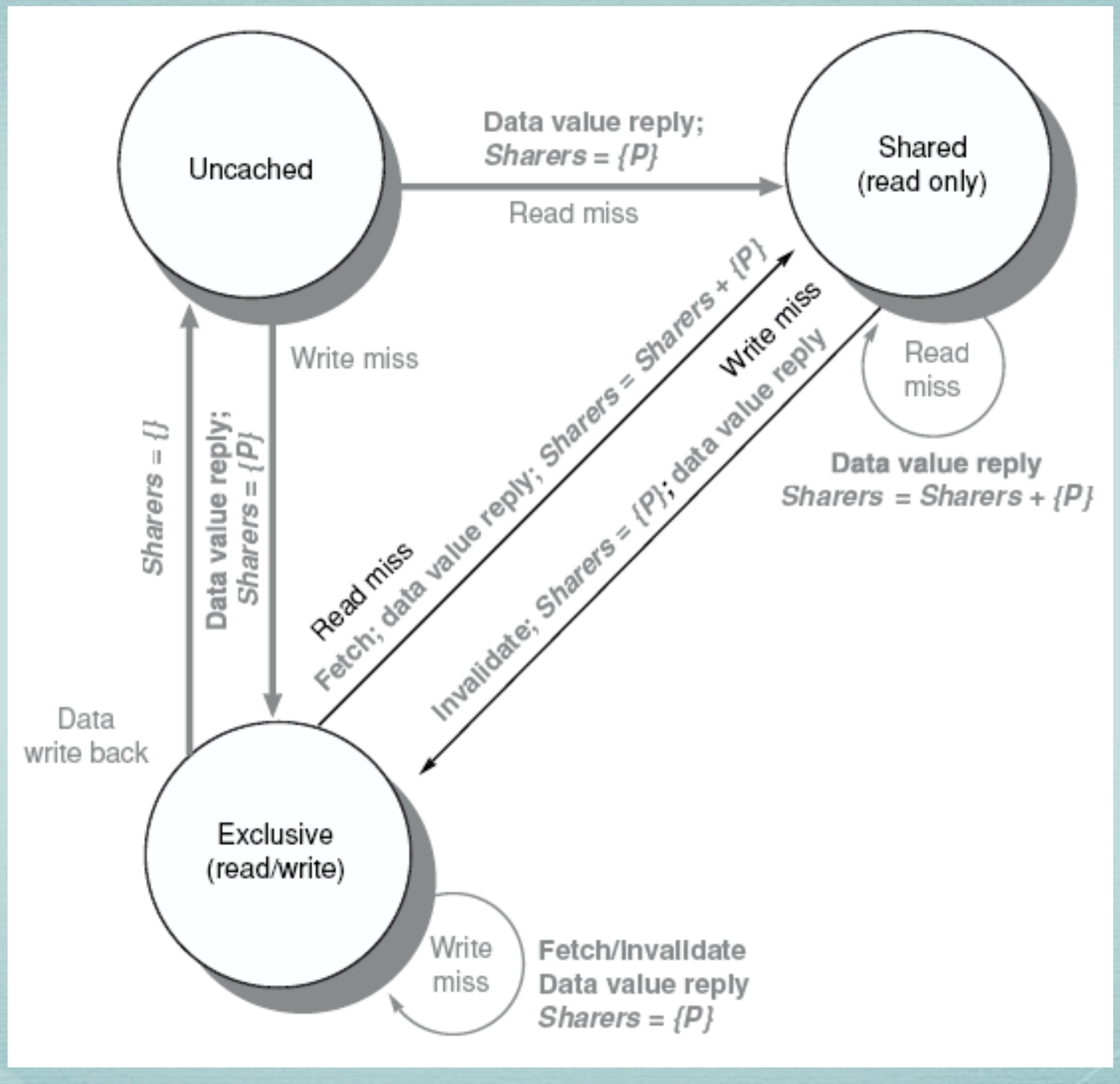
Possible Messages

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.

States for Individual Caches



States for Directories



SYNCHRONIZATION

Basic Idea

- Hardware support for atomically reading and writing a memory location
  - ⋆ enables software to implement locks
  - ⋆ a variety of synchronizations possible with locks
- Multiple (equivalent) approaches possible
- Synchronization libraries on top of hardware primitives

Some Examples

- Atomic exchange
  - ⋆ exchange a register and memory value atomically
  - ⋆ return the register value if failed, memory value if succeeded
- Test-and-set
  - ⋆ test a memory location and set its value if the test passed
- Fetch-and-increment

Paired Instructions

- Problems with single atomic operations
  - ⋆ complicates coherence
- Alternative: pair of special load and store instructions
  - ⋆ *load linked* or *load locked*
  - ⋆ *store conditional*
  - ⋆ can implement atomic exchange with the pair

```
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
```

Other Primitives can also be built

- Atomic exchange

```
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
```

- 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
```

- Implemented with a *link register* to track the address of LL instruction

Implementing Spin Locks: Uncached

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

Implementing Spin Locks: Cached

```
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

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

Implementing Spin Locks: Linked Load/Store

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

MEMORY CONSISTENCY

Memory Consistency

- Related to accesses to **multiple** shared memory locations
  - ⋆ coherence deals with accesses to **a** shared location

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

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

Memory Consistency

- Related to accesses to **multiple** shared memory locations
  - ⋆ coherence deals with accesses to **a** shared location

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

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

Sequential Consistency

Programmer's View

- Sequential consistency "easy" to reason about
- Most real programs use synchronization
  - ⋆ synchronization primitives usually implemented in libraries
  - ⋆ not using synchronization ⇒ **data races**
- Allowing sequential consistency for synchronization variables ensure correctness
  - ⋆ can implement relaxed consistency for the rest

Relaxed Consistency Models

- Idea: allow reads and writes to finish out of order, but use synchronization to enforce ordering
- Ways to relax consistency (weak consistency)
  - ⋆ Relaxing W→R: processor consistency
  - ⋆ Relaxing W→W: partial store order
  - ⋆ Relaxing R→W and R→R: weak ordering, PowerPC consistency, release consistency
- Two approaches to optimize performance
  - ⋆ use weak consistency, rely on synchronization for correctness
  - ⋆ use sequential or processor consistency with speculative execution

FALLACIES AND PITFALLS

Fallacies and Pitfalls

- Pitfall: Measuring performance of multiprocessors by linear speedup versus execution time
  - ⋆ sequential vs parallel algorithms
  - ⋆ superlinear speedups and cache effects
  - ⋆ strong vs weak scaling
- Pitfall: Not developing software to take advantage of, or optimize for, a multiprocessor architecture