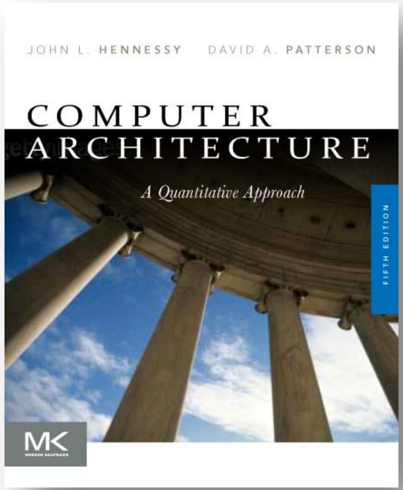


Chapter 5

Multiprocessors and Thread-Level Parallelism

Contents

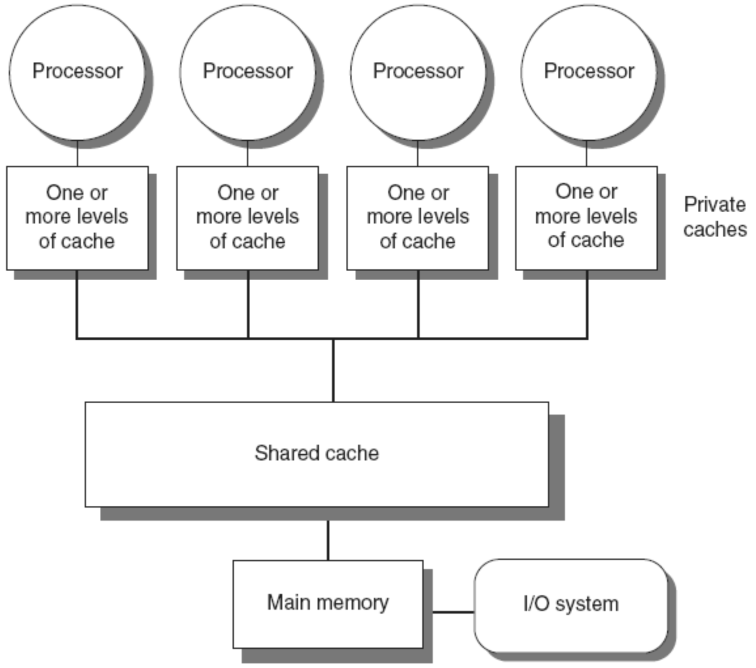
1. . Introduction. Why multiprocessors?

- Need for more computing power
  - Data intensive applications
  - Utility computing requires powerful processors
- Several ways to increase processor performance
  - Increased clock rate → limited ability
  - Architectural → ILP, CPI – increasingly more difficult
  - Multi-processor, multi-core systems → more feasible based on current technologies
- Advantages of multiprocessors and multi-core → Replication rather than unique design.

Multiprocessor types

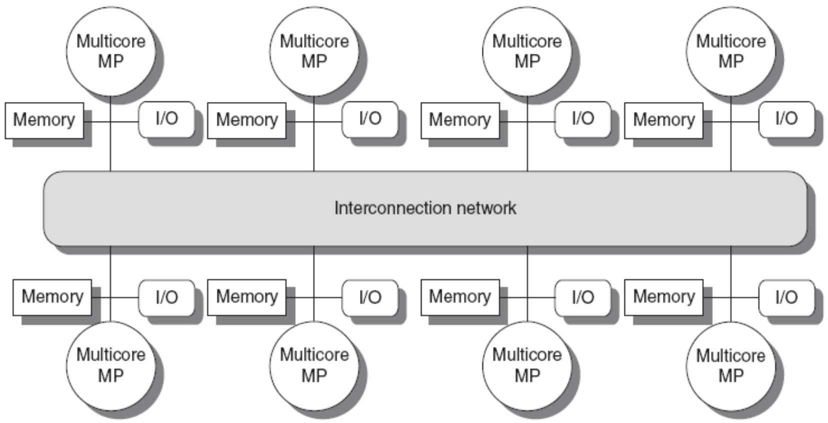
Symmetric multiprocessors (SMP)

- Share single memory with uniform memory access/latency (UMA)
- Small number of cores


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

Distributed shared memory (DSM)

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



Important ideas

- Technology drives the solutions.
  - Multi-cores have altered the game!!
  - Thread-level parallelism (TLP) vs ILP.
- Computing and communication deeply intertwined.
  - Write serialization exploits broadcast communication on the interconnection network or the bus connecting L1, L2, and L3 caches for cache coherence.
- Access to data located at the fastest memory level greatly improves the performance.
- Caches are critical for performance but create new problems
  - Cache coherence protocols:
    1. Cache snooping → traditional multiprocessor
    2. Directory based → multi-core processors

Review of basic concepts

- *Cache* → smaller, faster memory which stores copies of the data from frequently used main memory locations.
- Cache writing policies
  - *write-through* → every write to the cache causes a write to main memory.
  - *write-back* → writes are not immediately mirrored to main memory. Locations written are marked dirty and written back to the main memory only when that data is evicted from the cache. A read miss may require two memory accesses: write the dirty location to memory and read new location from memory.
- Caches are organized in *blocks* or *cache lines.*
- Cache blocks consist of
  - Tag → contains (part of) address of actual data fetched from main memory
  - Data block
  - Flags → dirty bit, shared bit,
- Broadcast networks → all nodes share a communication media and hear all messages transmitted, e.g., bus.

Cache coherence; consistency

- Coherence
  - **Reads** by any processor must return the most recently written value
  - **Writes** to the same location by any two processors are seen in the same order by all processors
- Consistency
  - A **read** returns the last value written
  - If a processor **writes** location A followed by location B, any processor that sees the new value of B must also see the new value of A

Thread-level parallelism (TLP)

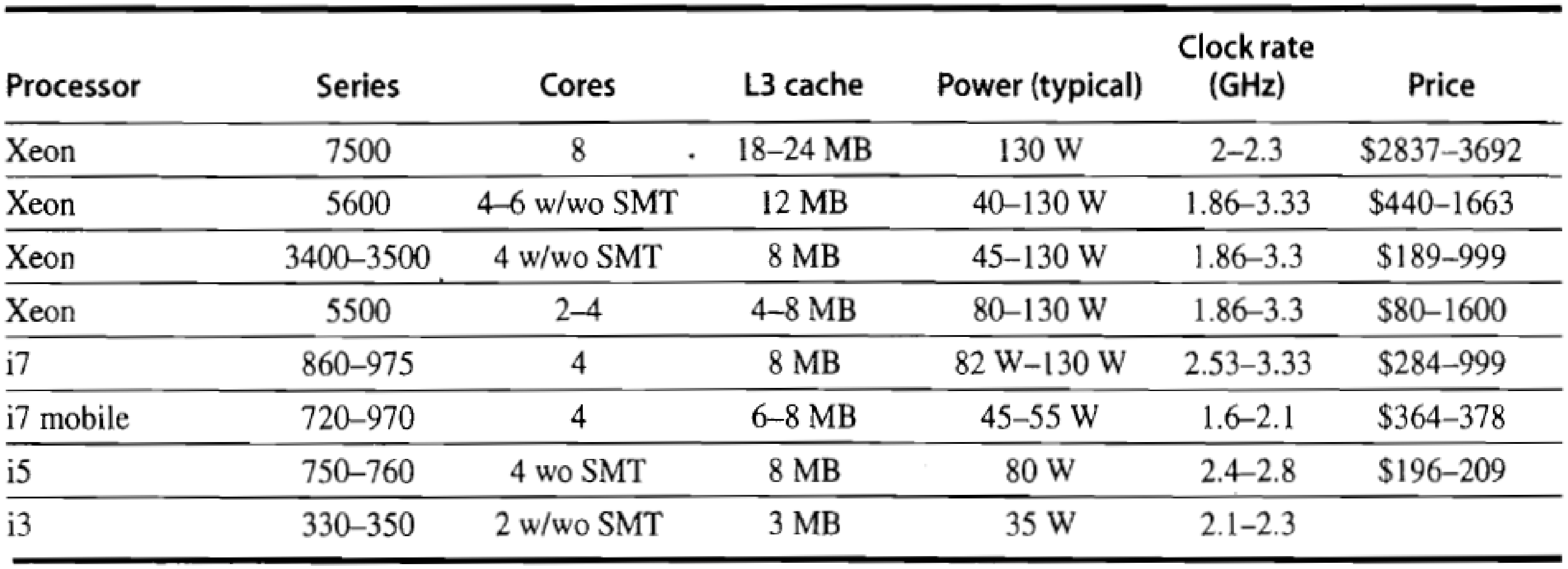
- Distribute the workload among a set of concurrently running threads.
- Uses MIMD model → multiple program counters
- Targeted for tightly-coupled shared-memory multiprocessors
- To be effective need *n* threads for *n* processors.
- Amount of computation assigned to each thread = grain size
  - Threads can be used for data-level parallelism, but the overheads may outweigh the benefit
- Speedup
  - Maximum speedup with n processors is n; embarrassingly parallel
  - The actual speedup depends on the ratio of parallel versus sequential portion of a program according to Amdahl's law.

TLP and ILP

- The costs for exploiting ILP are prohibitive in terms of silicon area and of power consumption.
- Multicore processor have altered the game
  - Shifted the burden for keeping the processor busy from the hardware and architects to application developers and programmers.
  - Shift from ILP to TLP
- Large-scale multiprocessors are not a large market, they have been replaced by clusters of multicore systems.
- *Given the slow progress in parallel software development in the past 30 years it is reasonable to assume that TLP will continue to be very challenging.*

Multi-core processors

- Cores are now the building blocks of chips.
- Intel offers a family of processors based on the Nehalem architecture with a different number of cores and L3 caches



Processor	Series	Cores	L3 cache	Power (typical)	Clock rate (GHz)	Price
Xeon	7500	8	18–24 MB	130 W	2–2.3	$2837–3692
Xeon	5600	4–6 w/wo SMT	12 MB	40–130 W	1.86–3.33	$440–1663
Xeon	3400–3500	4 w/wo SMT	8 MB	45–130 W	1.86–3.3	$189–999
Xeon	5500	2–4	4–8 MB	80–130 W	1.86–3.3	$80–1600
i7	860–975	4	8 MB	82 W–130 W	2.53–3.33	$284–999
i7 mobile	720–970	4	6–8 MB	45–55 W	1.6–2.1	$364–378
i5	750–760	4 wo SMT	8 MB	80 W	2.4–2.8	$196–209
i3	330–350	2 w/wo SMT	3 MB	35 W	2.1–2.3	

Figure 5.34 The characteristics for a range of Intel parts based on the Nehalem microarchitecture. This chart still collapses a variety of entries in each row (from 2 to 8!). The price is for an order of 1000 units.

TLC - exploiting parallelism

- Speed-up = Execution time with one thread / Execution time with N threads.
- Amdahl’s insight:
  - Depends on the ratio of parallel to sequential execution blocks.
  - It is not sufficient to reduce the parallel execution time e.g., by increasing the number of threads. It is critical to reduce the sequential execution time!!

Problem

- What fraction of a computation can be sequential if we wish to achieve a speedup of 80 with 100 processors?

Solution

- According to Amdahl's law

$$\text{Speedup} = \frac{1}{\frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}} + (1 - \text{Fraction}_{\text{enhanced}})}$$

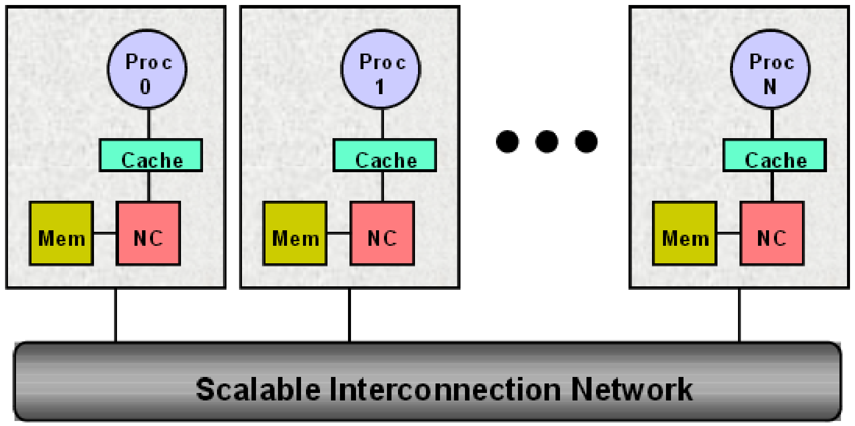
For simplicity in this example, assume that the program operates in only two modes: parallel with all processors fully used, which is the enhanced mode, or serial with only one processor in use. With this simplification, the speedup in enhanced mode is simply the number of processors, while the fraction of enhanced mode is the time spent in parallel mode. Substituting into the previous equation:

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

- The parallel fraction is 0.9975. This implies that only 0.25% of the computation can be sequential!!!

DSM

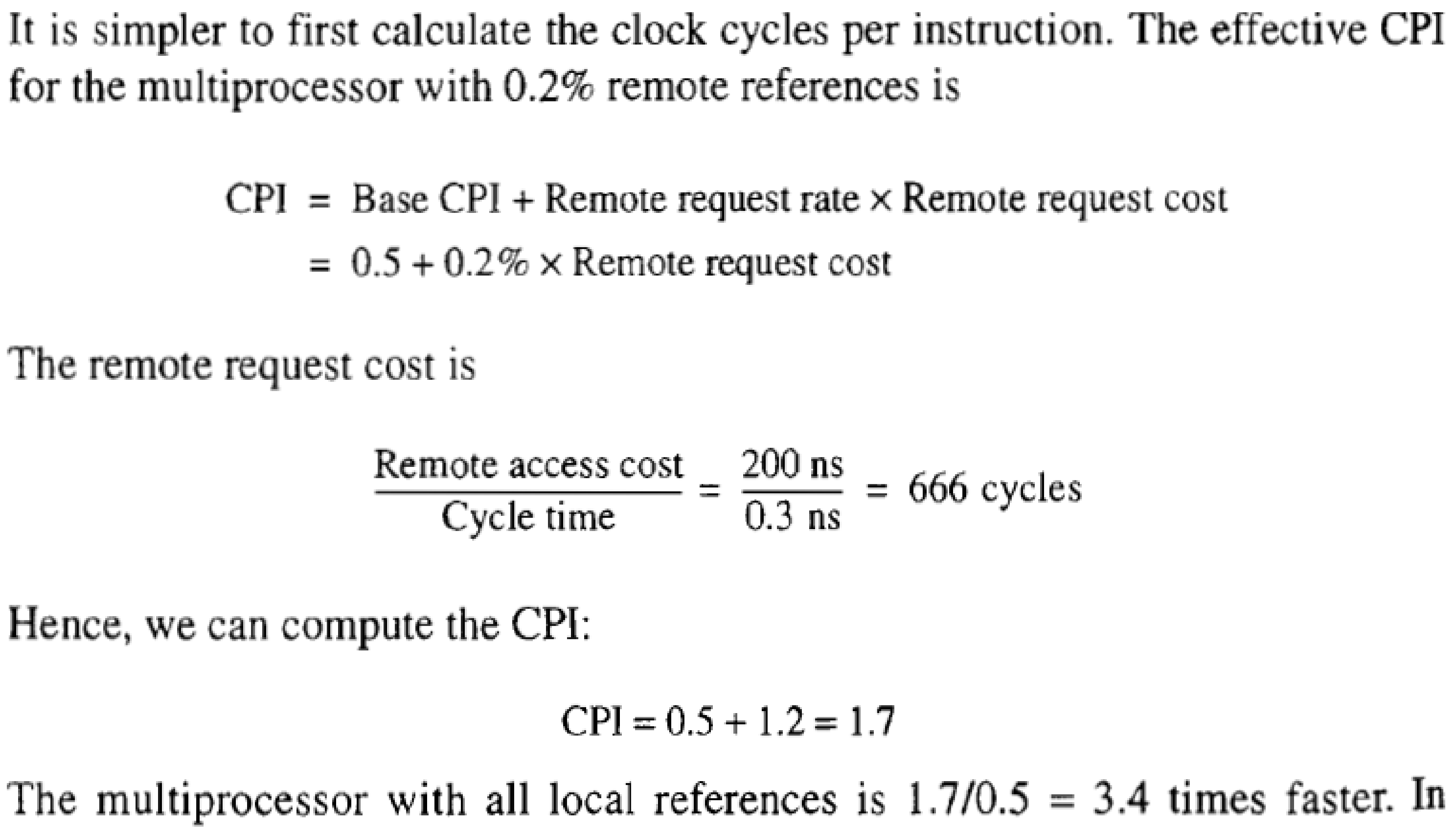
- Pros:
  1. Cost-effective way to scale memory bandwidth if most accesses are to local memory
  2. Reduced latency of local memory accesses
- Cons
  1. Communicating data between processors more complex
  2. Must change software to take advantage of increased memory bandwidth



An example distributed shared-memory architecture. Rather than being connected by a single shared bus, the nodes are connected by a scalable interconnection network. The node controller (NC) manages communication between processing nodes.

Slowdown due to remote access

A multiprocessor has a 3.3 GHz clock (0.3 nsec) and CPI = 0.5 when references are satisfied by the local cache. A processor stalls for a remote access which requires a 200nsec. How much faster is an application which uses only local references versus when 0.2% of the references are remote?



It is simpler to first calculate the clock cycles per instruction. The effective CPI for the multiprocessor with 0.2% remote references is

$$\begin{aligned}\text{CPI} &= \text{Base CPI} + \text{Remote request rate} \times \text{Remote request cost}\\ &= 0.5 + 0.2\% \times \text{Remote request cost}\end{aligned}$$

The remote request cost is

$$\frac{\text{Remote access cost}}{\text{Cycle time}} = \frac{200 \text{ ns}}{0.3 \text{ ns}} = 666 \text{ cycles}$$

Hence, we can compute the CPI:

$$\text{CPI} = 0.5 + 1.2 = 1.7$$

The multiprocessor with all local references is 1.7/0.5 = 3.4 times faster. In

2. Data access in SMA

- Caching data
  - reduces the access time but demands cache coherence
- Two distinct data states
  - Global state → defined by the data in main memory
  - Local state → defined by the data in local caches
- In multi-core L3 cache is shared; L1 and L2 caches are private
- *Cache coherence* → defines the behavior of reads and writes to the *same memory location*. The value that should be returned by a read is the most recent value of the data item.
- *Cache consistency* → defines the behavior of reads and writes with respect to *different memory locations*. It determines when a written value will be returned by a read .

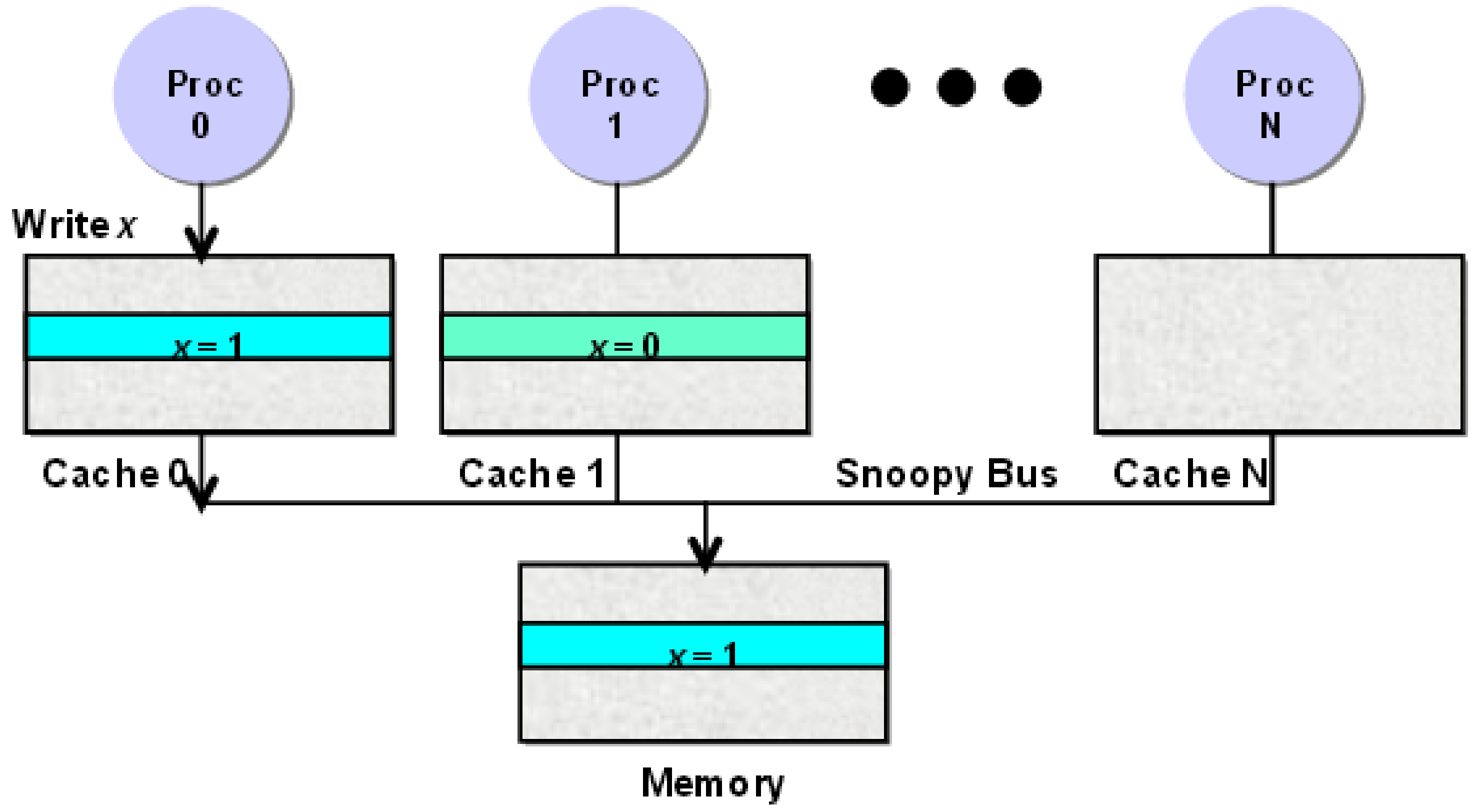
Conditions for coherence

A read of processor P to location X

1. That follows a write by P to X, with no other processor writing to X between the write and read executed by P, should return the value written by P.
2. That follows a write by another processor Q to X, should return the value written by Q provided that:
   - there is sufficient time between the write and the read operations
   - There is no other write to X

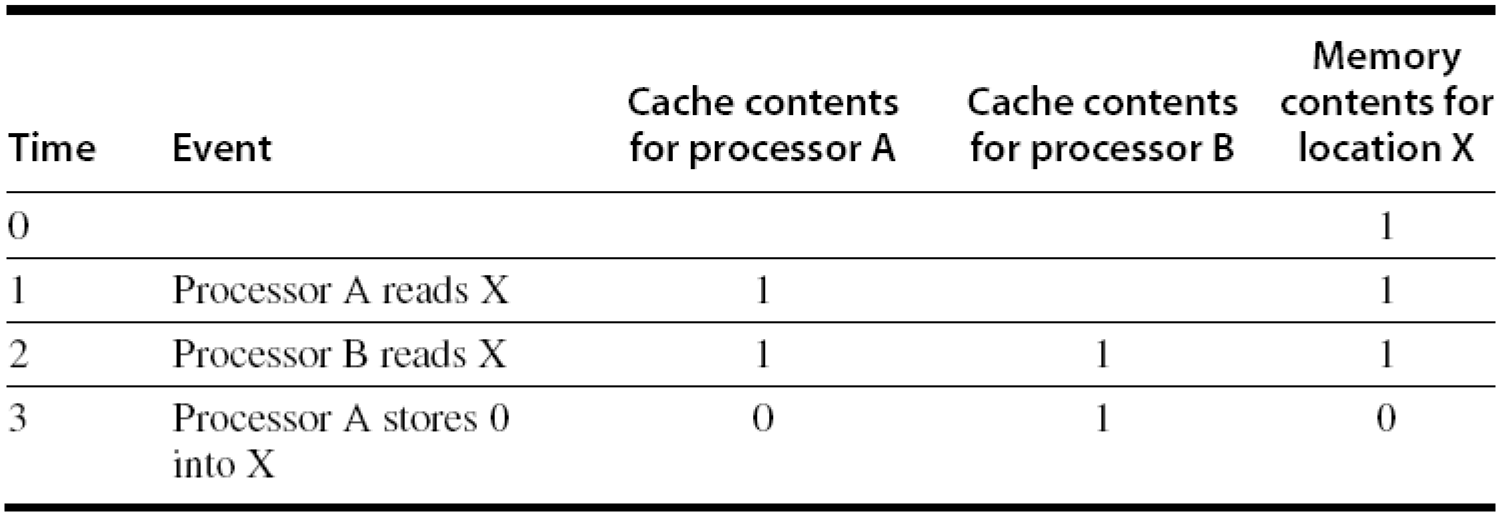
Writes to the same location are serialized.

- If P and Q write to X in this order some processors may see the value written by Q before they see the value written by P.



. The cache coherence problem. Initially processors 0 and 1 both read location *x*, initially containing the value 0, into their caches. When processor 0 writes the value 1 to location *x*, the stale value 0 for location x is still in processor 1's cache.

Processors may see different values through their caches



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

Cache coherence; consistency

- Coherence
  - **Reads** by any processor must return the most recently written value
  - **Writes** to the same location by any two processors are seen in the same order by all processors
- Consistency
  - A **read** returns the last value written
  - If a processor **writes** location A followed by location B, any processor that sees the new value of B must also see the new value of A

Enforcing coherence

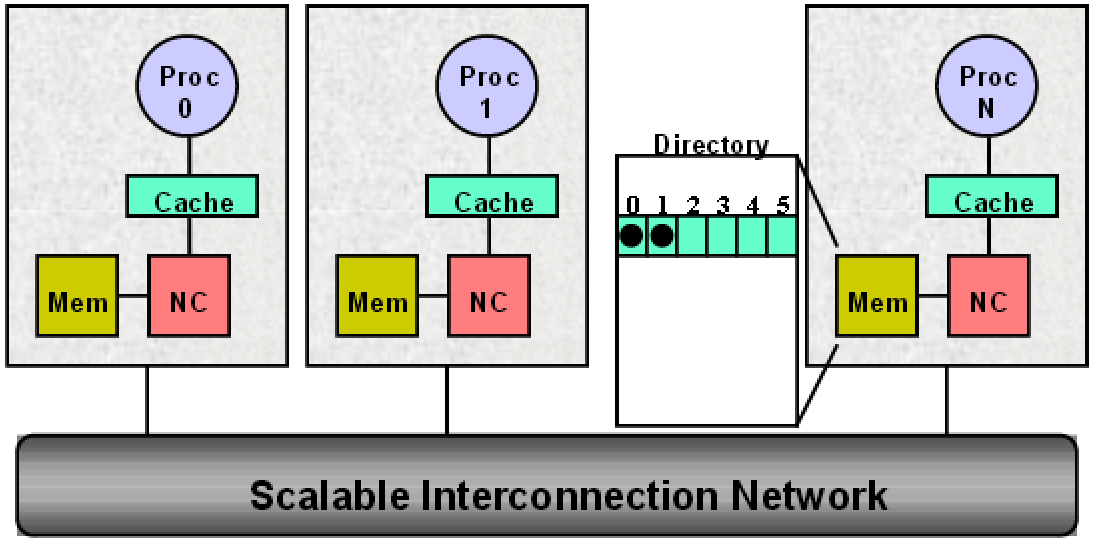
- Coherent caches provide:
  - *Migration*: movement of data
  - *Replication*: multiple copies of data
- Cache coherence protocols
  - Directory-based
    - Sharing status of each block kept in the directory
  - Snooping
    - Each core tracks sharing status of each block

Directory-based cache coherence protocols

- All information about the blocks is kept in the directory.
- SMP multiprocessors: one centralized directory

  Directory is located

  1. In the outmost cache for multi-core systems.
  2. In main memory



DSM multiprocessors: distributed directory. More complex → each node maintains a directory which tracks the sharing information of every cache line in the node

Communication between private and shared caches

- *Multi-core processor* ➔ a bus connects private L1 and L2 instruction (I) and data (D) caches to the shared L3 cache.
- To invalidate a cached item the processor changing the value must first acquire the bus and then place the address of the item to be invalidated on the bus.
- *DSM* ➔Locating the value of an item is harder for write-back caches because the current value of the item can be in the local caches of another processor.

Snoopy coherence protocols

- Two strategies:
  1. Write invalidate ➔ on **write**, invalidate all other copies.
     - Used in modern microprocessors
     - Example: a write-back cache during read misses of item X, processors A and B. Once A writes X it invalidates the B's cache copy of X

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

  2. Write update or write broadcast ➔ update all cached copies of a data item when the item is written
     - Consumes more bandwidth thus not used in recent multiprocessors

Implementation of cache invalidate

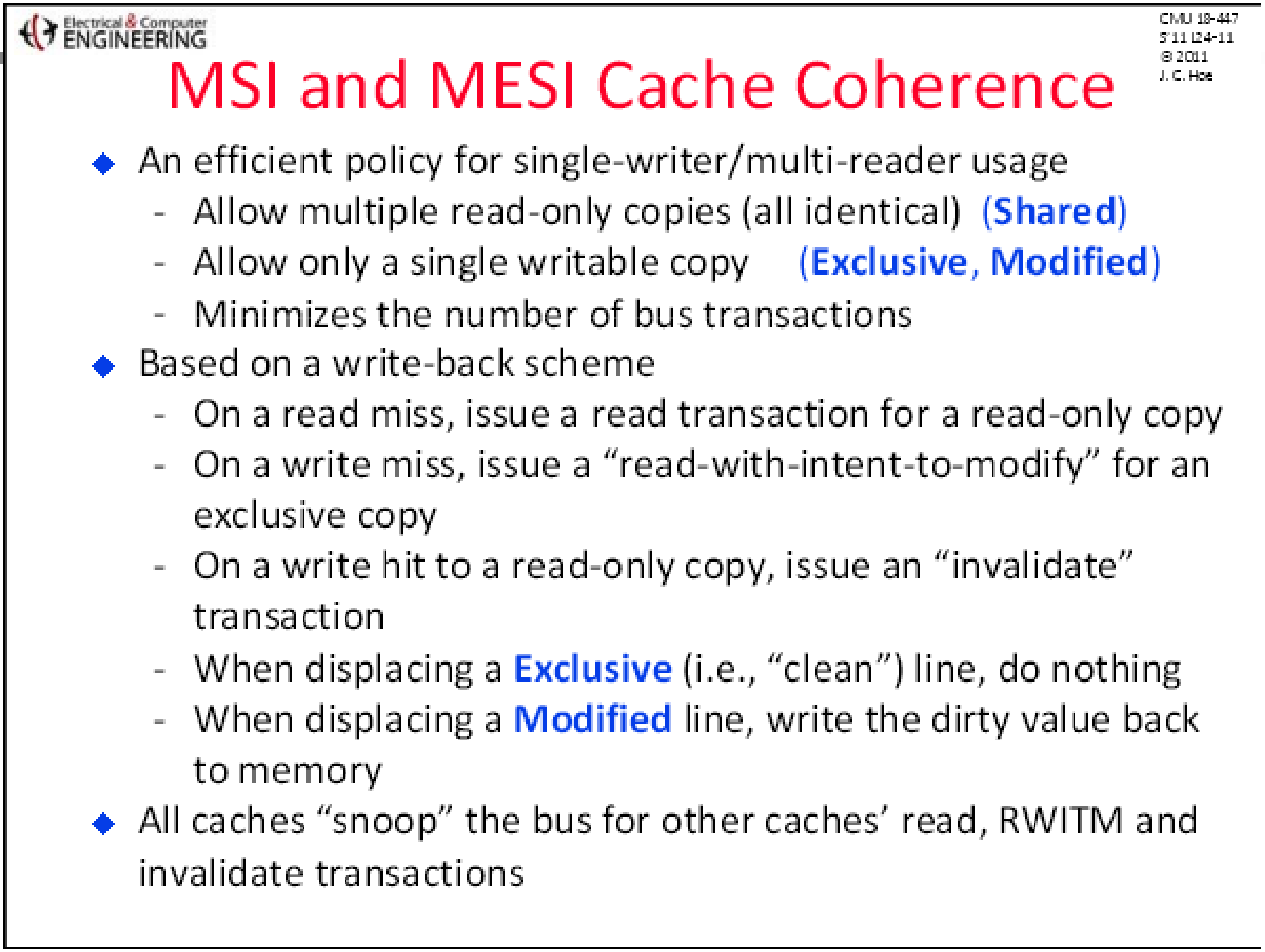
- All processors snoop on the bus.
- To invalidate the processor changing an item acquires the bus and broadcasts the address of the item.
- If two processors attempt to change at the same time the bus arbitrator allows access to only one of them.
- How to find the most recent value of a data item
  - Write-through cache → the value is in memory but write buffers could complicate the scenario.
  - Write-back cache → harder problem, the item could be in the private cache of another processor.
- A block of cache has extra state bits
  - Valid bit – indicates if the block is valid or not
  - Dirty bit - indicates if the block has been modified
  - Shared bit – cache block is shared with other processors

MSI – Modified, Shared, Invalid protocol

- Each core of a multi-core or each CPU runs a cache controller
  - Responds to requests coming from two sources
    1. The local core
    2. The bus (or other broadcast network connecting caches and memory)
  - Implements a finite-state machine

- The states of a cache block
  1. Invalid → another core has modified the block
  2. Shared → the block is shared with other cores
  3. Modified → the block in private cache has been updated by the local core

- When an item in a block is referenced (read or write):
  - Hit → The block is available
  - Miss → The bloc is not available

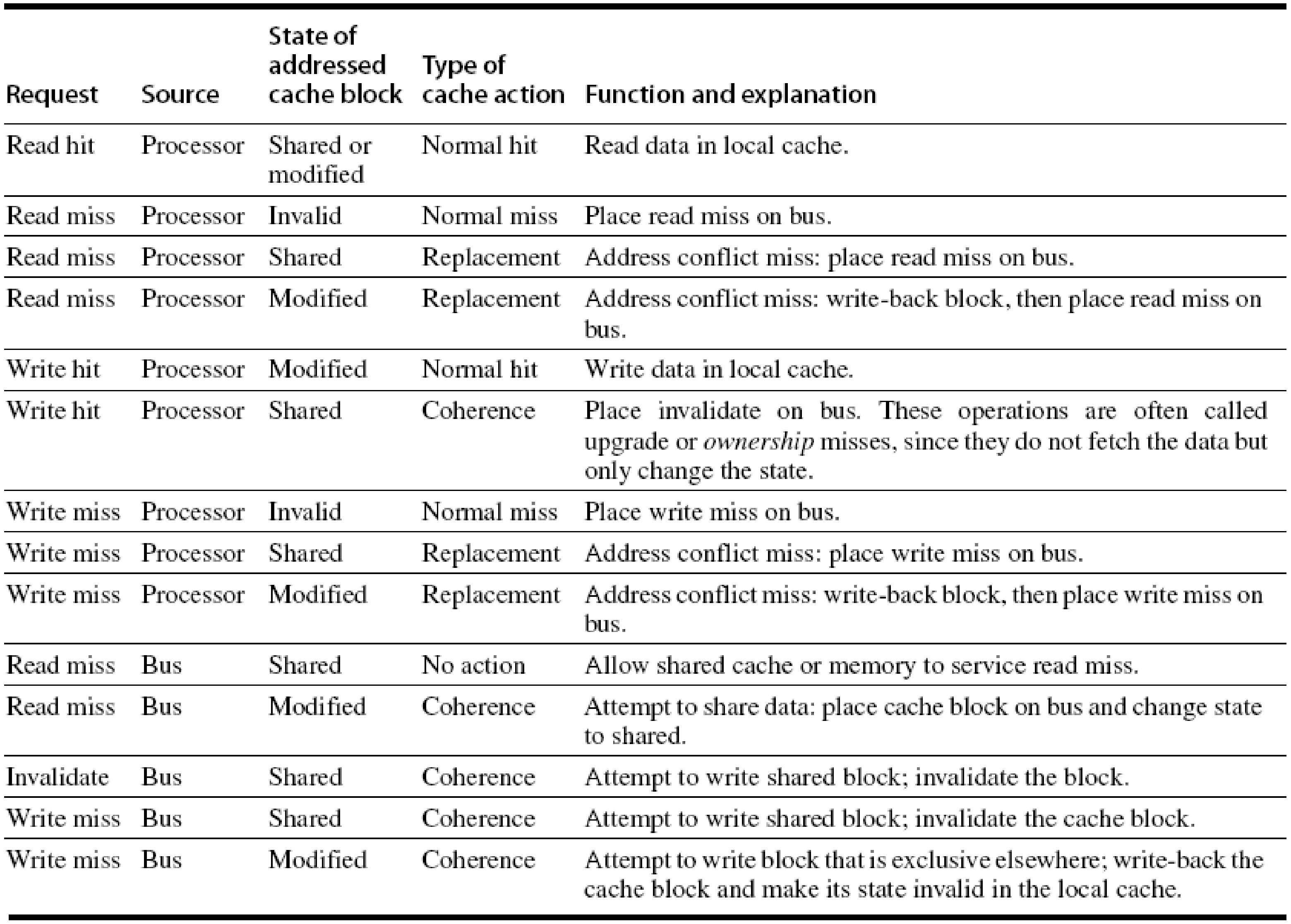

Electrical & Computer ENGINEERING

CMU 18-447
S'11 L24-11
© 2011
J. C. Hoe

MSI and MESI Cache Coherence

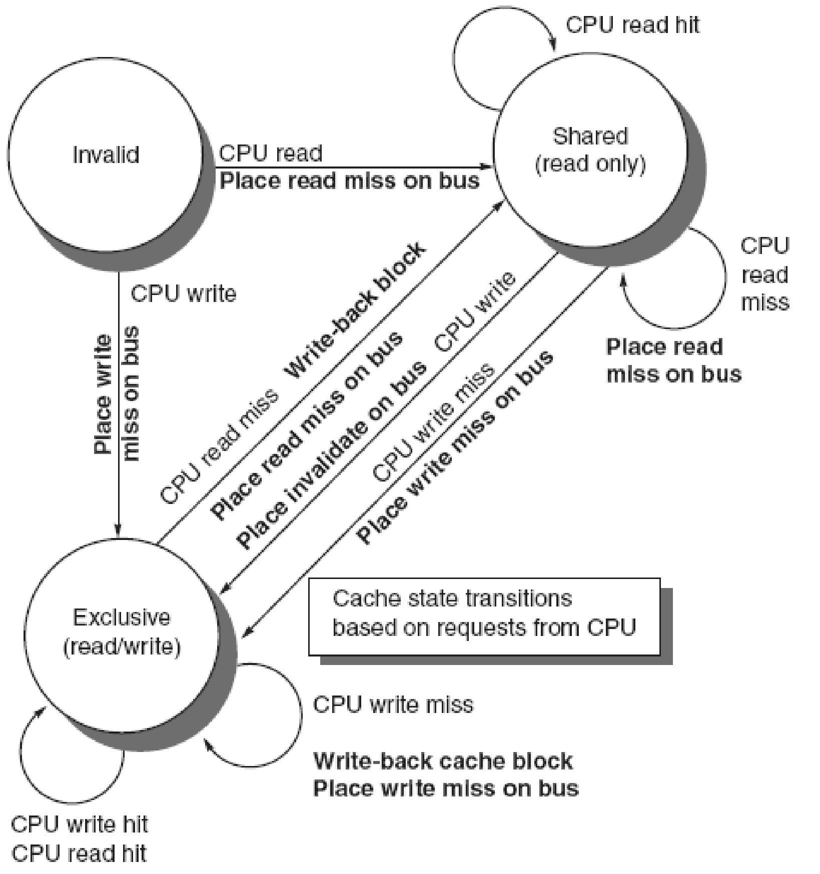
- An efficient policy for single-writer/multi-reader usage
  - Allow multiple read-only copies (all identical) **(Shared)**
  - Allow only a single writable copy **(Exclusive, Modified)**
  - Minimizes the number of bus transactions
- Based on a write-back scheme
  - On a read miss, issue a read transaction for a read-only copy
  - On a write miss, issue a "read-with-intent-to-modify" for an exclusive copy
  - On a write hit to a read-only copy, issue an "invalidate" transaction
  - When displacing a **Exclusive** (i.e., "clean") line, do nothing
  - When displacing a **Modified** line, write the dirty value back to memory
- All caches "snoop" the bus for other caches' read, RWITM and invalidate transactions

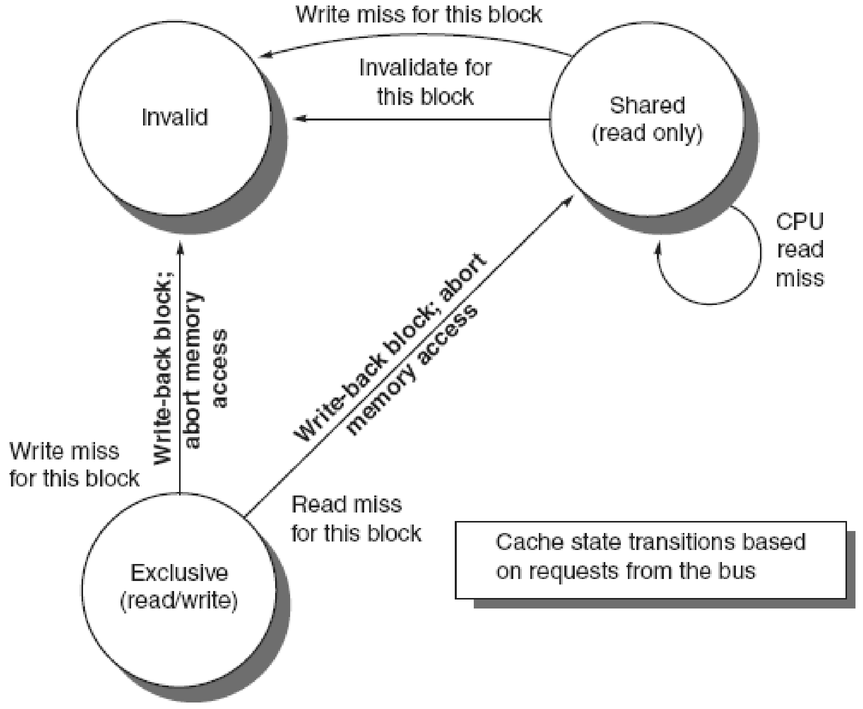
MSI - snoopy coherence protocol



Request	Source	State of addressed cache block	Type of cache action	Function and explanation
Read hit	Processor	Shared or modified	Normal hit	Read data in local cache.
Read miss	Processor	Invalid	Normal miss	Place read miss on bus.
Read miss	Processor	Shared	Replacement	Address conflict miss: place read miss on bus.
Read miss	Processor	Modified	Replacement	Address conflict miss: write-back block, then place read miss on bus.
Write hit	Processor	Modified	Normal hit	Write data in local cache.
Write hit	Processor	Shared	Coherence	Place invalidate on bus. These operations are often called upgrade or *ownership* misses, since they do not fetch the data but only change the state.
Write miss	Processor	Invalid	Normal miss	Place write miss on bus.
Write miss	Processor	Shared	Replacement	Address conflict miss: place write miss on bus.
Write miss	Processor	Modified	Replacement	Address conflict miss: write-back block, then place write miss on bus.
Read miss	Bus	Shared	No action	Allow shared cache or memory to service read miss.
Read miss	Bus	Modified	Coherence	Attempt to share data: place cache block on bus and change state to shared.
Invalidate	Bus	Shared	Coherence	Attempt to write shared block; invalidate the block.
Write miss	Bus	Shared	Coherence	Attempt to write shared block; invalidate the cache block.
Write miss	Bus	Modified	Coherence	Attempt to write block that is exclusive elsewhere; write-back the cache block and make its state invalid in the local cache.

MSI snoopy coherence protocol




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

Problem

How can the snooping protocol with the state diagram at the right can be changed for a write-through cache?

What is the major hardware functionality that is not needed with a write-through cache compared with a write back cache?

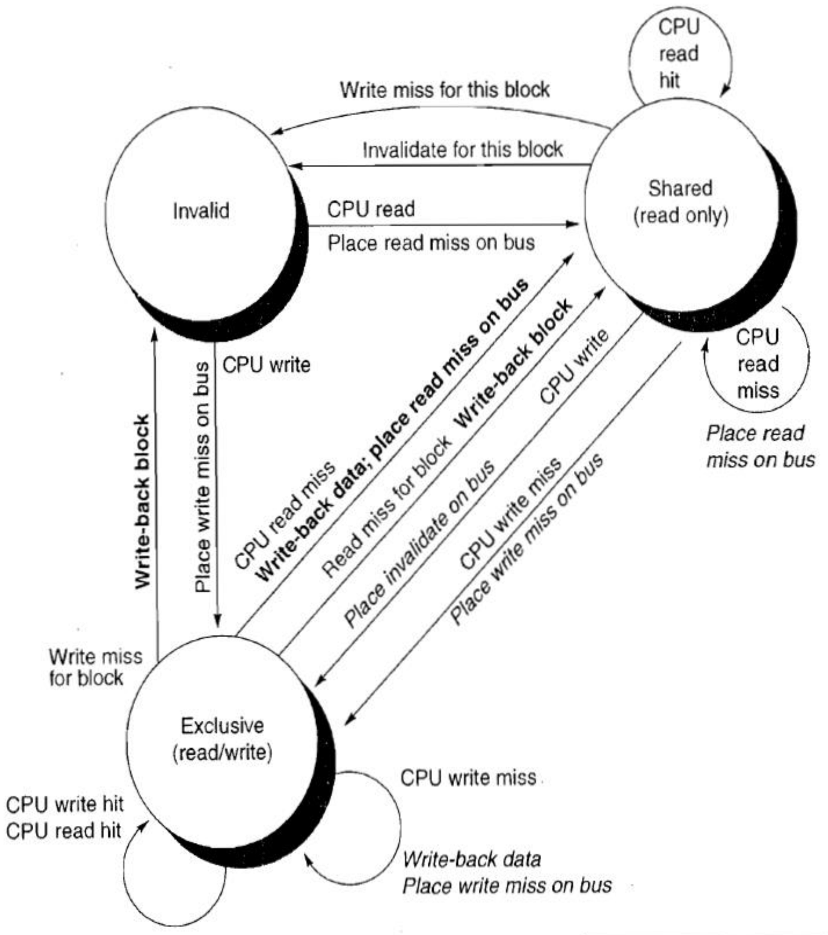


Figure 5.7 Cache coherence state diagram with the state transitions induced by the local processor shown in black and by the bus activities shown in gray. As in Figure 5.6, the activities on a transition are shown in bold.

Solution

A **write** to a block in the valid or the shared state causes a **write-invalidate** broadcast to flush the block from other caches and move to an **exclusive** state.

We leave the **exclusive** state through either an invalidate from another processor or a **read** miss generated by the CPU when a block is displaced from cache by another block

We are moved from the **shared** state only by a **write** from the CPU or an **invalidate** from from another processor

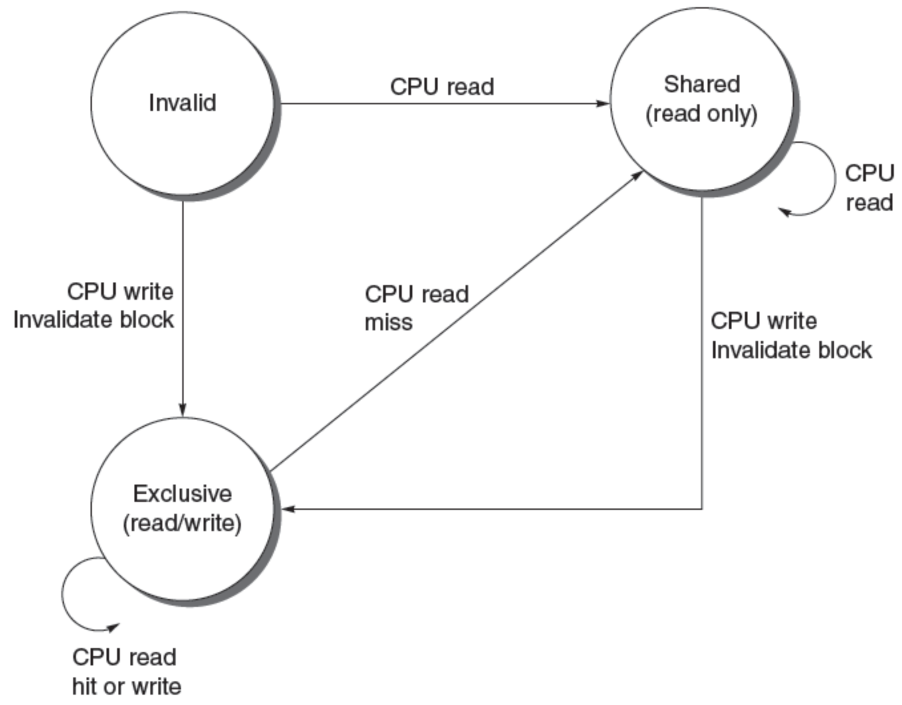


Figure S.34 CPU portion of the simple cache coherency protocol for write-through caches.

Solution (cont'd)

When another processor writes a block that is resident in our cache, we unconditionally invalidate the corresponding block in our cache. This ensures that the next time we read the data, we will load the updated value of the block from memory.

Whenever the bus sees a read miss, it must change the state of an exclusive block to shared as the block is no longer exclusive to a single cache.

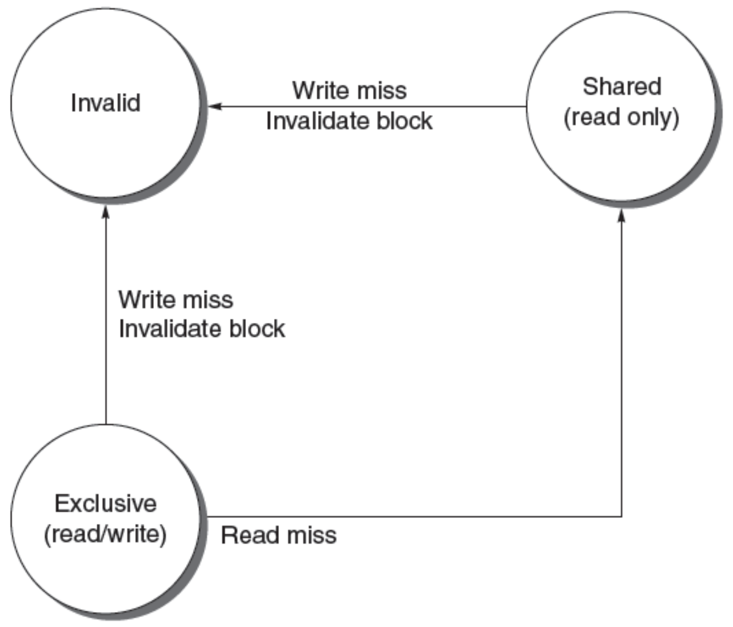


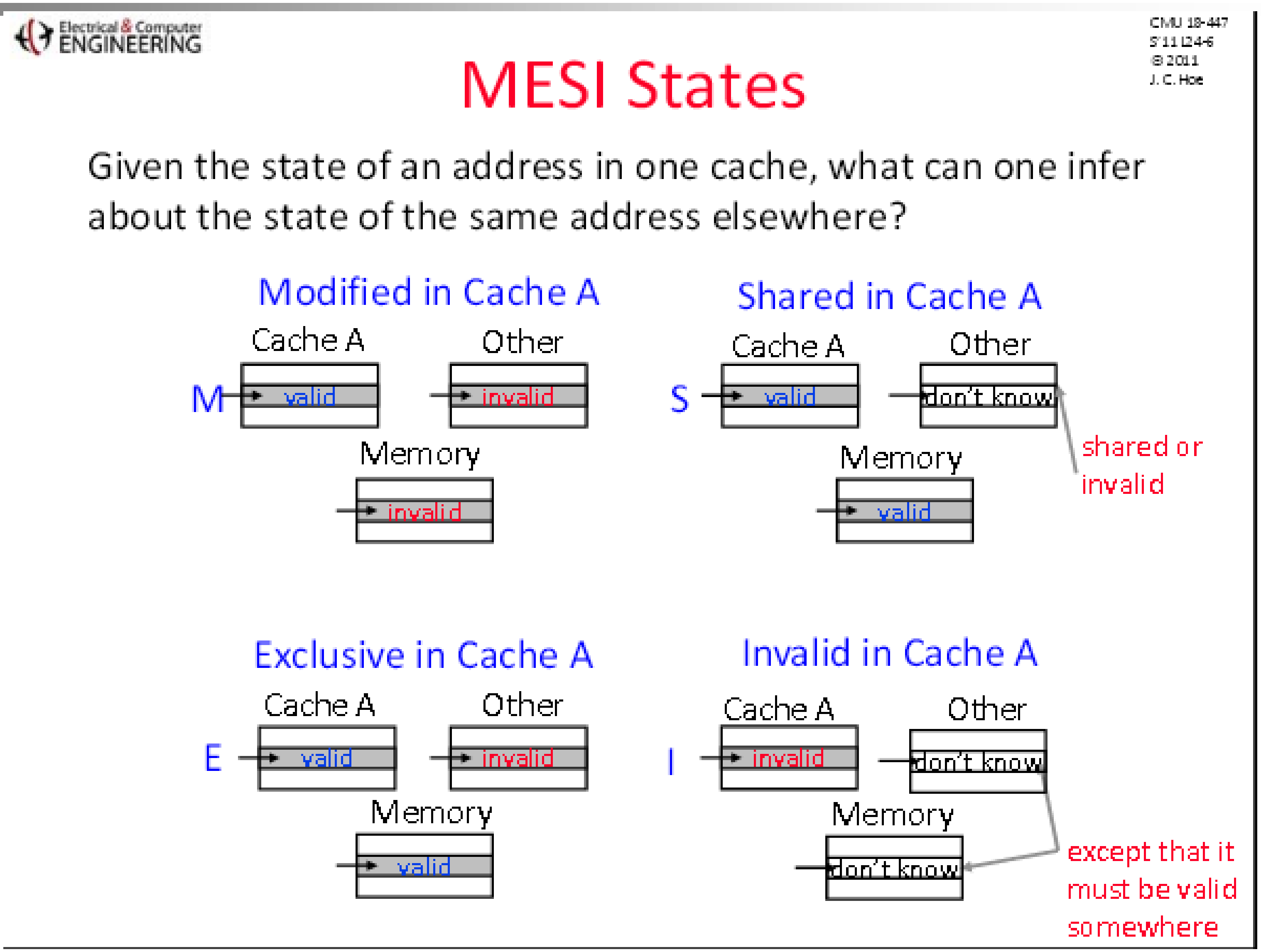
Figure S.35 Bus portion of the simple cache coherency protocol for write-through caches.

Solution (cont'd)

- It is not possible for valid cache blocks to be incoherent with respect to main memory in a system with write-through caches.
- The major change introduced in moving from a write-back to write-through cache is the elimination of the need to access dirty blocks in another processor's caches.
- The write-through protocol it is no longer necessary to provide the hardware to force write back on read accesses or to abort pending memory accesses.
- As memory is updated during any write on a write-through cache, a processor that generates a read miss will always retrieve the correct information from memory.

Extensions to MSI protocol

- Complications for the basic MSI protocol:
  - Operations are not atomic
    - E.g. detect miss, acquire bus, receive a response
    - Creates possibility of deadlock and races
    - One solution: processor that sends invalidate can hold bus until other processors receive the invalidate
- Extensions:
  - MESI protocol ➔ add exclusive state to indicate when a clean block is resident in only one cache.
    - Prevents the need to write invalidate on a write
  - MOESI protocol ➔ Owned state; indicates that the block is owned by that cache and it is out-of-date in memory



Electrical & Computer ENGINEERING

CMU 18-447
S'11 L24-6
© 2011
J. C. Hoe

MESI States

Given the state of an address in one cache, what can one infer about the state of the same address elsewhere?

Modified in Cache A

M
- Cache A: valid
- Other: invalid
- Memory: invalid

Shared in Cache A

S
- Cache A: valid
- Other: don't know
- Memory: valid

shared or invalid

Exclusive in Cache A

E
- Cache A: valid
- Other: invalid
- Memory: valid

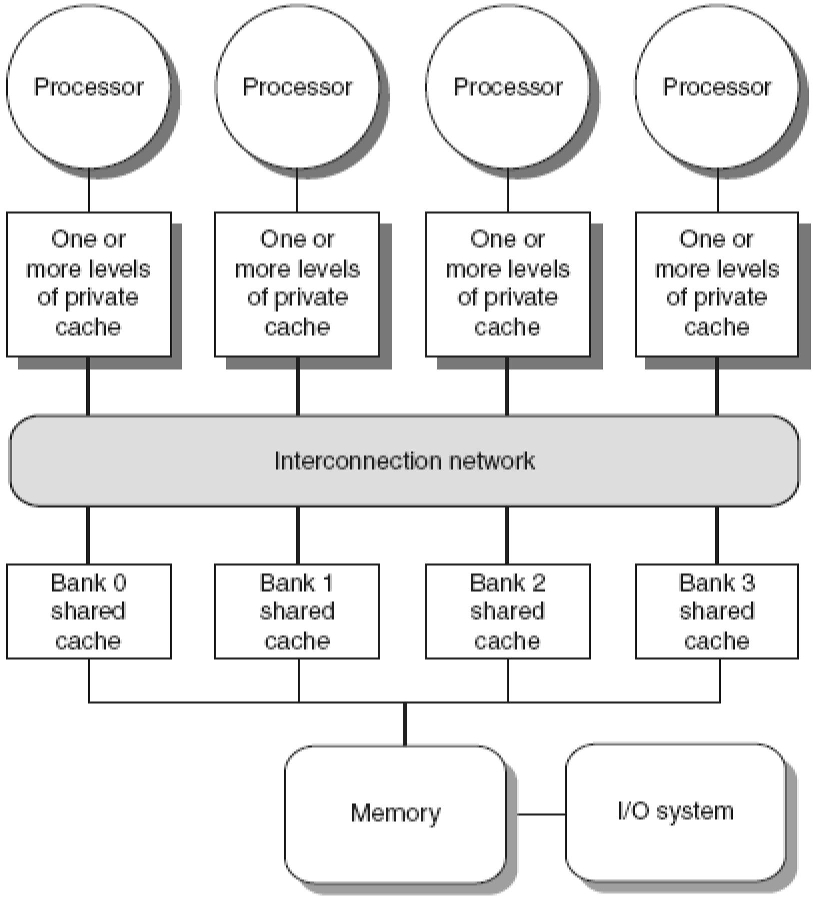
Invalid in Cache A

I
- Cache A: invalid
- Other: don't know
- Memory: don't know

except that it must be valid somewhere

Coherence protocols: extensions

- Shared memory bus and snooping bandwidth is bottleneck for scaling symmetric multiprocessors
  - Duplicating tags
  - Place directory in outermost cache
  - Use crossbars or point-to-point networks with banked memory



Coherence protocols

- AMD Opteron:
  - Memory directly connected to each multicore chip in NUMA-like organization
  - Implement coherence protocol using point-to-point links
  - Use explicit acknowledgements to order operations

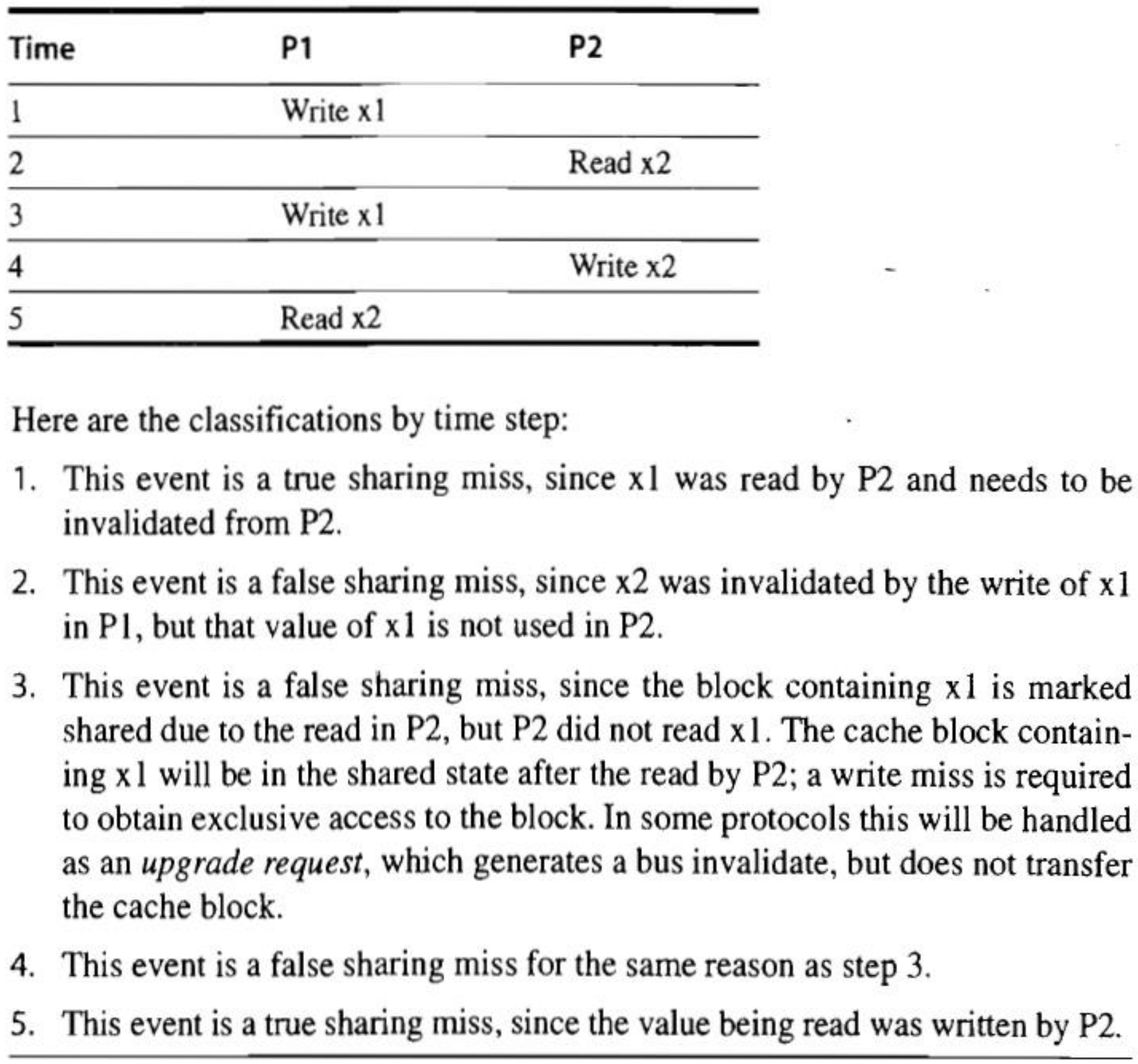
True and false sharing misses

- Coherence influences cache miss rate
- Coherence misses
  - True sharing misses
    - Write to shared block (transmission of invalidation)
    - Read an invalidated block
  - False sharing misses
    - A block is invalidated because some word in the block other than the one read was written into. A subsequent reference to the block causes a miss

x1 and x2 are in the same block in the shared state in the caches of P1 and P2.

Identify the true and false sharing misses in each of the five steps.

Prior to time step 1 , x1 was read by P2.



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

Here are the classifications by time step:

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.
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.

Problem

- How to change the code of an application to avoid false sharing?
- What can be done by a compiler and what requires programmer directives?

Solution

- False sharing occurs when both the data object size is smaller than the granularity of cache block valid bit(s) coverage and more than one data object is stored in the same cache block frame in memory.
- Two ways to prevent false sharing.
  - Changing the cache block size or the amount of the cache block covered by a given valid bit are hardware changes and shall not be discussed.
  - Software solution → allocate data objects so that

1. only one truly shared object occurs per cache block frame in memory and
2. no non-shared objects are located in the same cache block frame as any shared object. If this is done, then even with just a single valid bit per cache block, false sharing is impossible.

- Shared, read-only-access objects could be combined in a single cache block and not contribute to the false sharing problem because such a cache block can be held by many caches and accessed as needed without an invalidations to cause unnecessary cache misses.

Solution (cont'd)

- If shared data objects are explicitly identified in the program source code, then the compiler should, with knowledge of memory hierarchy details, be able to avoid placing more than one such object in a cache block frame in memory. If shared objects are not declared, then programmer directives may need to be added to the program. The remainder of the cache block frame should not contain data that would cause false sharing misses. The sure solution is to pad with block with non-referenced locations.
- Padding a cache block frame containing a shared data object with unused memory locations may lead to rather inefficient use of memory space. A cache block may contain a shared object plus objects that are read-only as a trade-off between memory use efficiency and incurring some false-sharing misses. This optimization almost certainly requires programmer analysis to determine if it would be worthwhile. A careful attention to data distribution with respect to cache lines and partitioning the computation across processors is needed.

Types of cache misses: the three C's

- 1. _Compulsory_ → on the first access to a block; the block must be brought into the cache; also called cold start misses, or first reference misses.
- 2. _Capacity_ → blocks are being discarded from cache because cache cannot contain all blocks needed for program execution (working set is much larger than cache capacity).
- 3. _Conflict_ → also called collision misses or interference misses occur when several blocks are mapped to the same set or block frame in the case of set associative or direct mapped block placement strategies

3. SMP performance – the workload

1. *OLTEP (on-line transaction processing system)* → Client processes generate requests and servers process them. Oracle database. Server processes consume 85% of user time. The server processes block for I/O after about 25,000 instructions.
2. DSS (decision support system) → 6 queries average about 1.5 million instructions before blocking. Oracle database.
3. Alta Vista (a Web search engine). Alta Vista 200 GB database

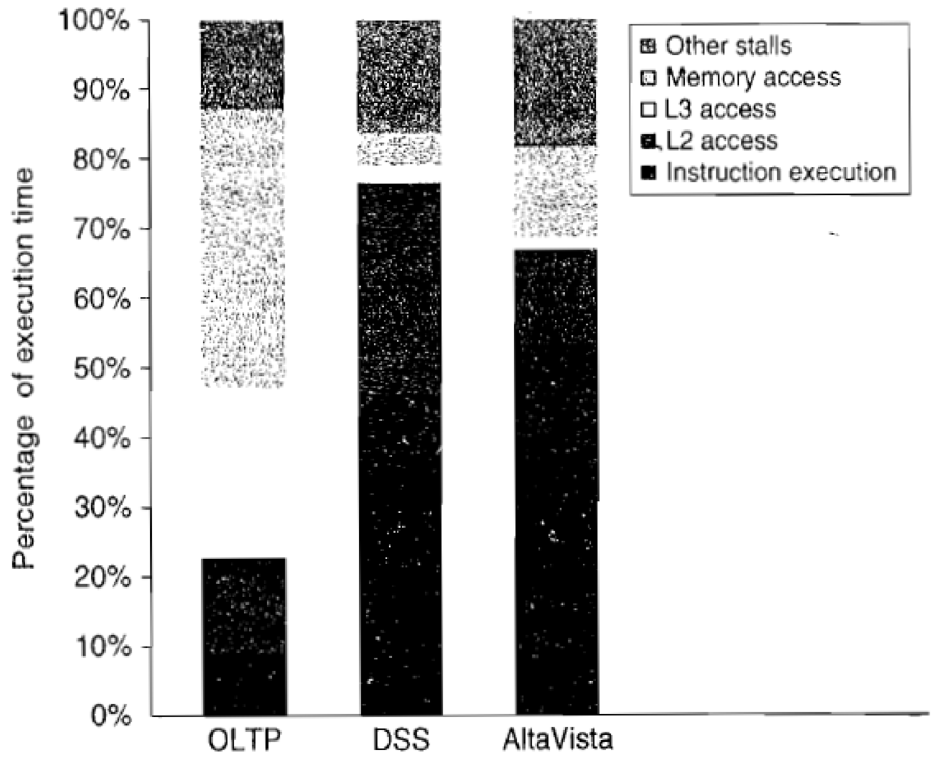


Figure 5.11 The execution time breakdown for the three programs (OLTP, DSS, and AltaVista) in the commercial workload. The DSS numbers are the average across six different queries. The CPI varies widely from a low of 1.3 for AltaVista, to 1.61 for the DSS queries, to 7.0 for OLTP. (Individually, the DSS queries show a CPI range of 1.3 to 1.9.) "Other stalls" includes resource stalls (implemented with replay traps on the 21164), branch mispredict, memory barrier, and TLB misses. For these benchmarks, resource-based pipeline stalls are the dominant factor. These data combine the behavior of user and kernel accesses. Only OLTP has a significant fraction of kernel accesses, and the kernel accesses tend to be better behaved than the user accesses! All the measurements shown in this section were collected by Barroso, Gharachorloo, and Bugnion [1998].

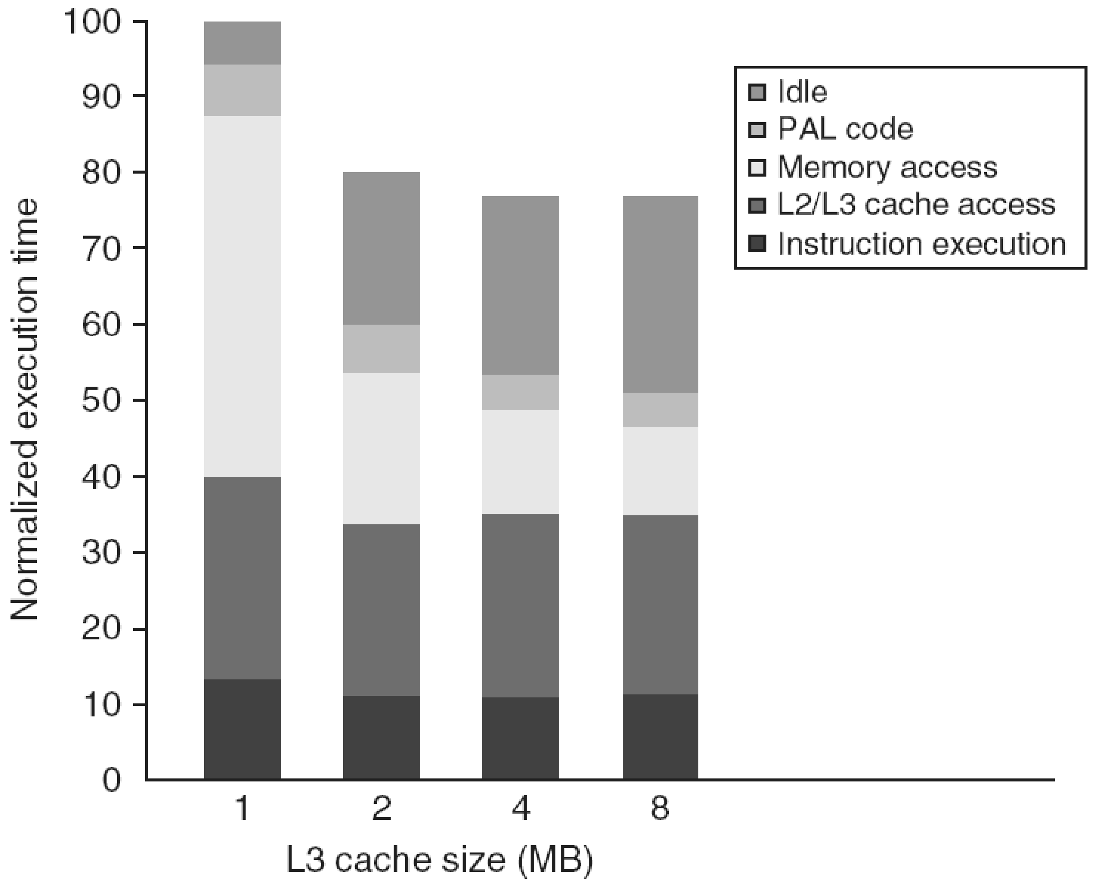
OLTEP - the effect of L3 cache size

Contribution to cache misses

1. Instruction execution
2. L2/L3 cache access
3. Memory access
4. PAL code → instructions executed in kernel mode.

Execution time improves as L3 cache size grows from 1 to 2 MB.

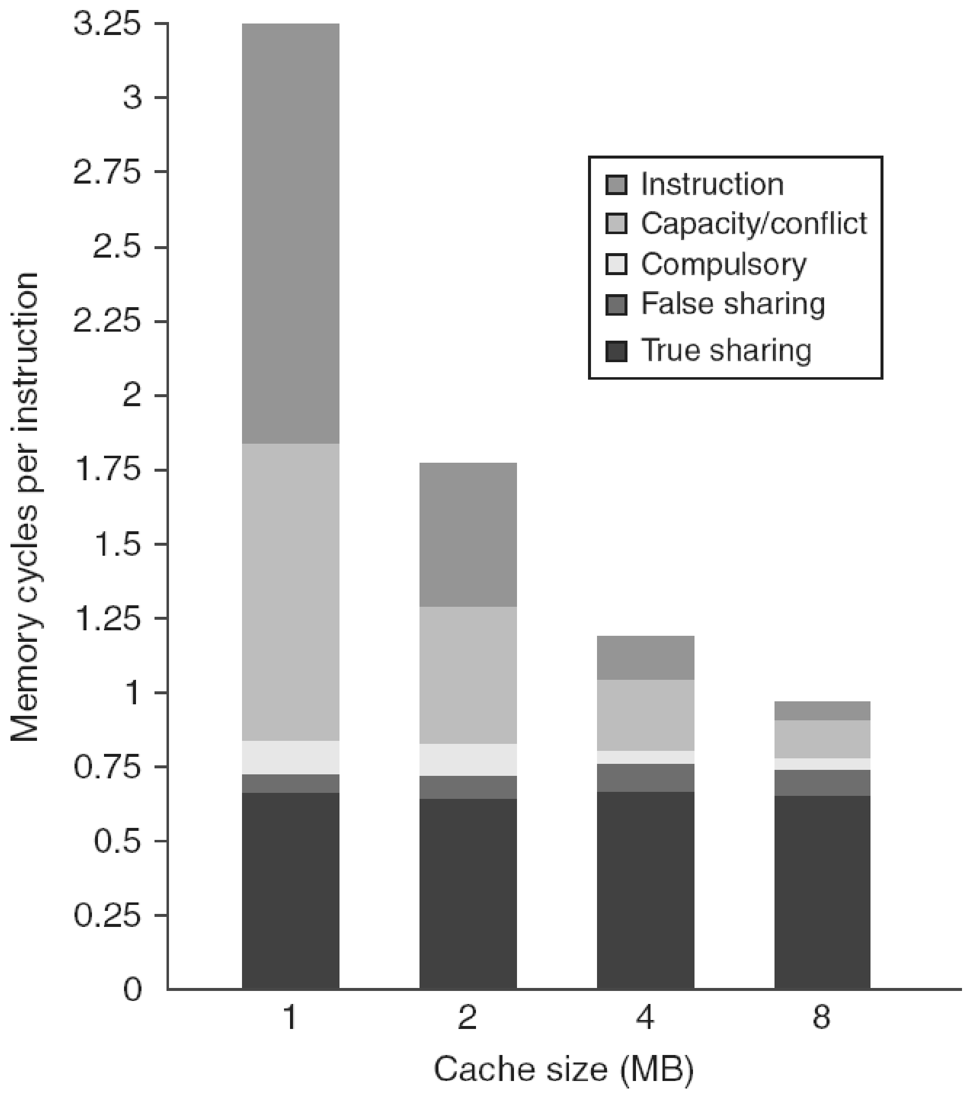
The idle time grows as cache size increases, as fewer memory stalls occur and more processors are needed to cover the I/O bandwidth



OLTEP - factors contributing to L3 miss rate

Memory access cycles contributing to L3 miss rate

1. Instruction – decreases as the L3 cache increases
2. Capacity/conflict – decreases as the L3 cache increases
3. Compulsory – almost constant
4. False sharing – almost constant
5. True sharing – almost constant

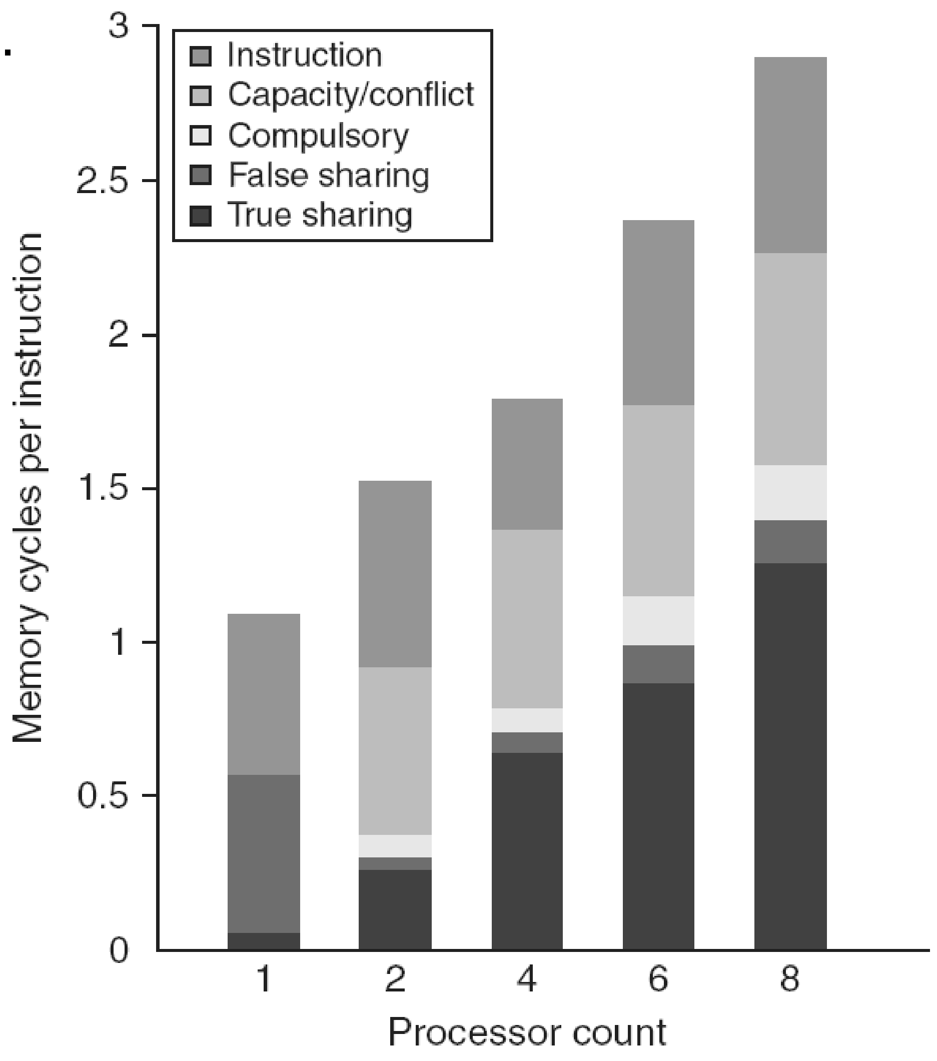


OLTEP – the effect of number of processors

2MB, two-way associative cache.

The memory access cycles per instruction increase with the number of processors.

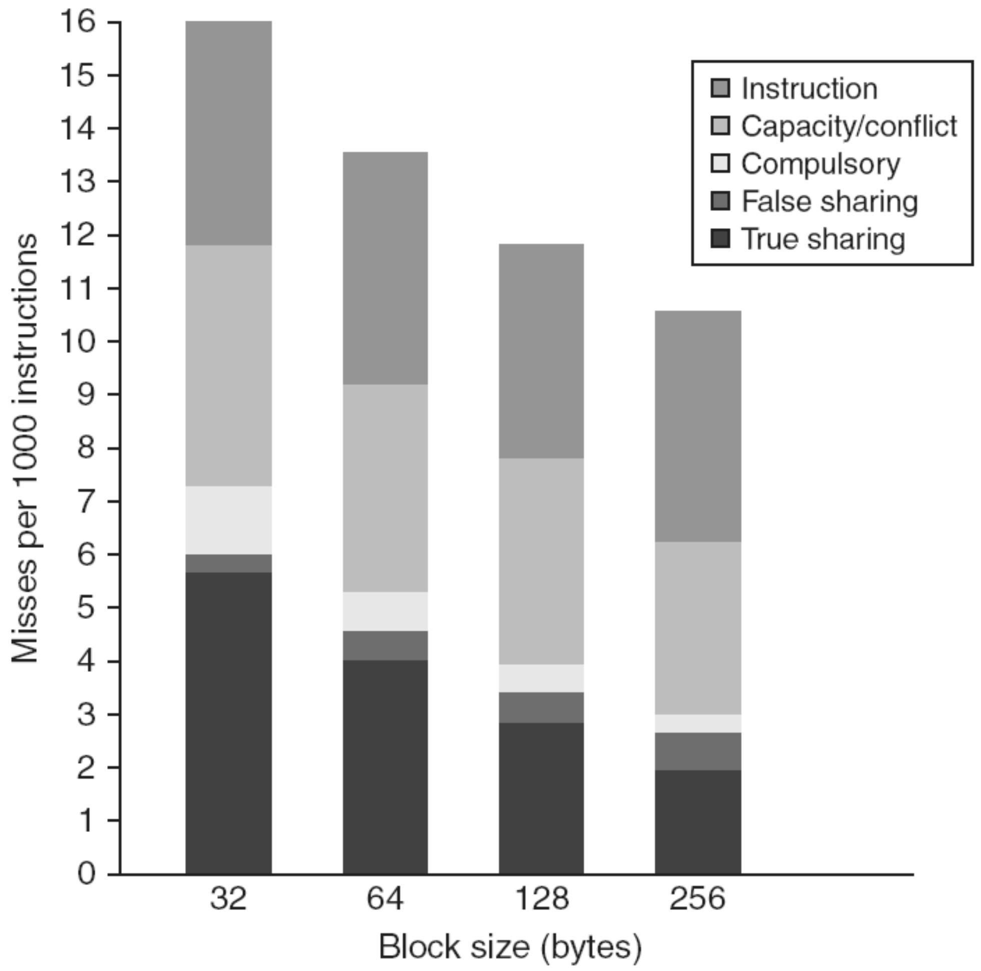
The true sharing miss rate increases



The effect of cache block size

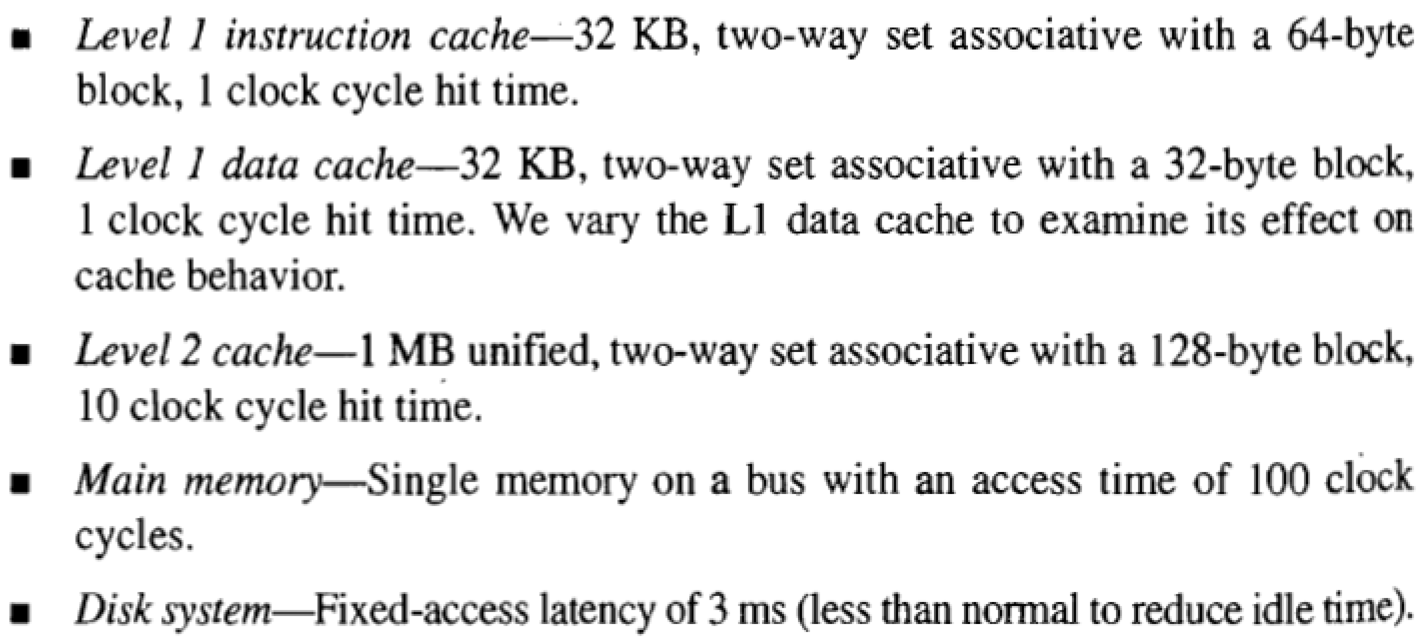
2MB, two-way associative cache.

As the cache block size increases the true sharing misses decrease.

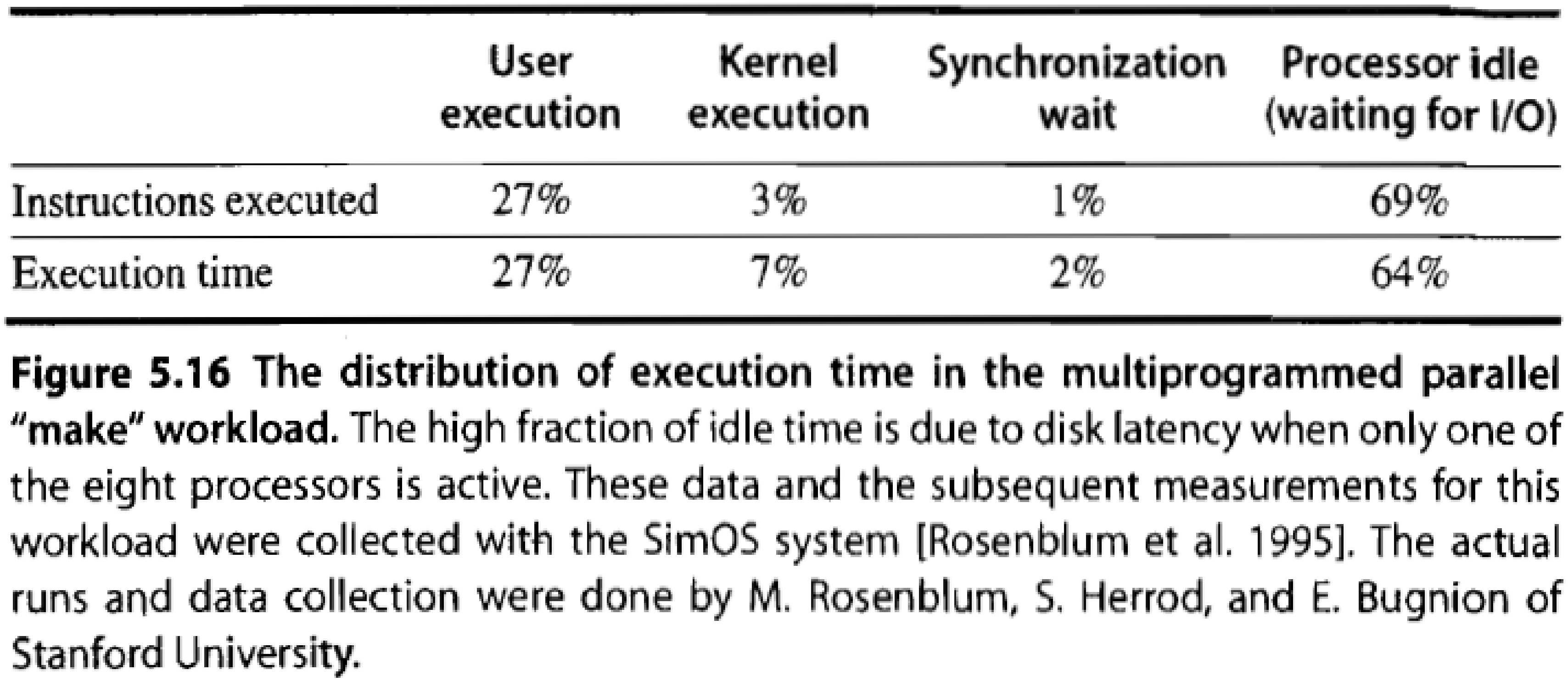


Andrew benchmark

- Emulates a software development environment.
- Parallel version of the **make** Unix command executed on 8 processors.
  - Creates 203 processes
  - 787 disk requests on three different file systems
  - Runs 5.24 seconds on 128 MB of memory, no paging.

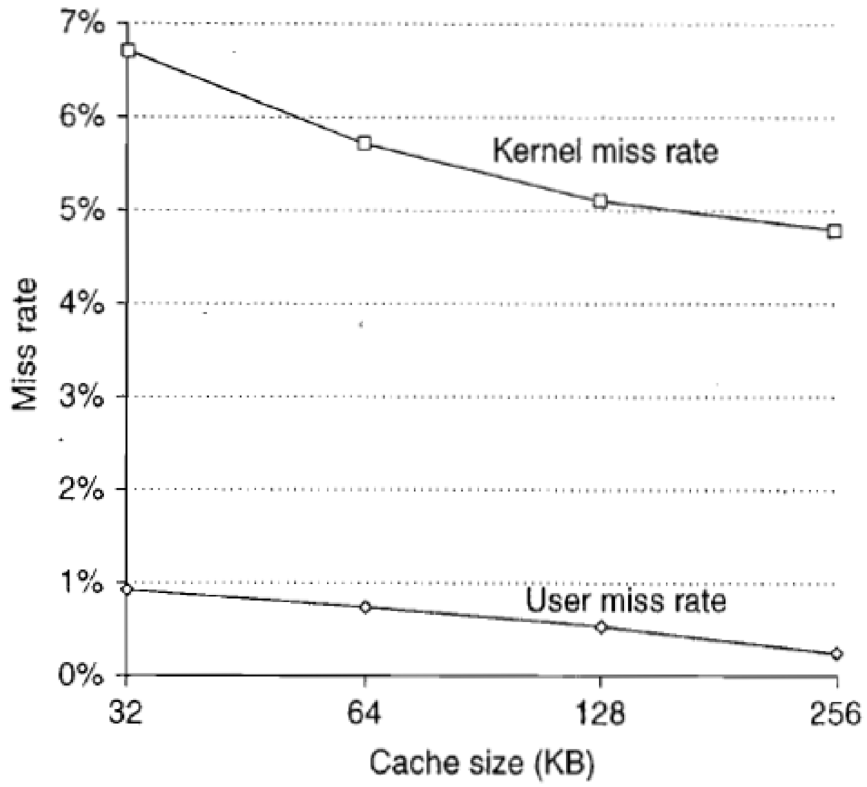


- *Level 1 instruction cache*—32 KB, two-way set associative with a 64-byte block, 1 clock cycle hit time.
- *Level 1 data cache*—32 KB, two-way set associative with a 32-byte block, 1 clock cycle hit time. We vary the L1 data cache to examine its effect on cache behavior.
- *Level 2 cache*—1 MB unified, two-way set associative with a 128-byte block, 10 clock cycle hit time.
- *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	Processor idle (waiting for I/O)
Instructions executed	27%	3%	1%	69%
Execution time	27%	7%	2%	64%

Figure 5.16 The distribution of execution time in the multiprogrammed parallel "make" workload. The high fraction of idle time is due to disk latency when only one of the eight processors is active. These data and the subsequent measurements for this workload were collected with the SimOS system [Rosenblum et al. 1995]. The actual runs and data collection were done by M. Rosenblum, S. Herrod, and E. Bugnion of Stanford University.



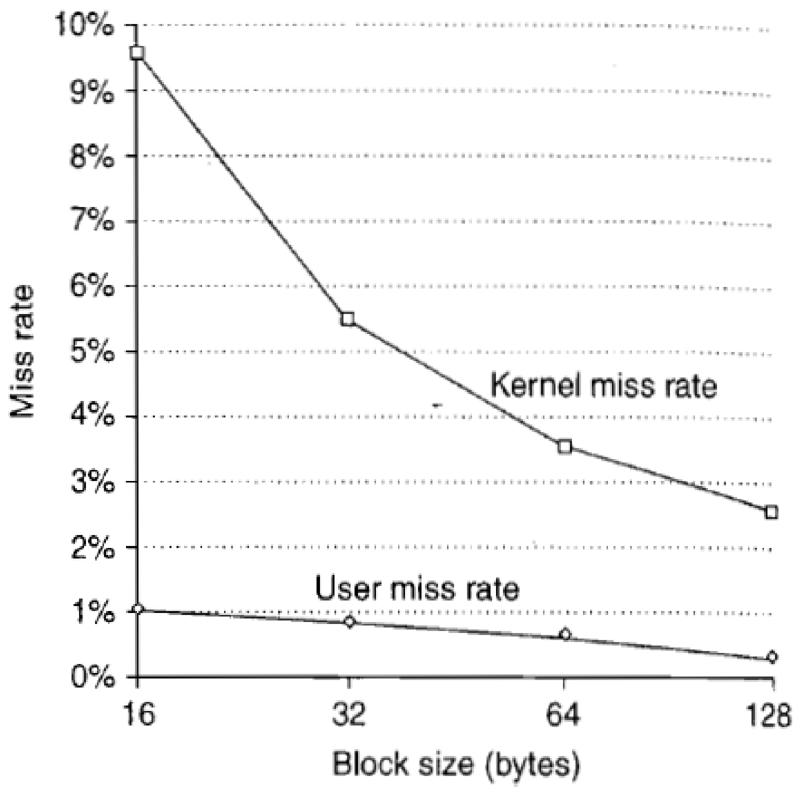


Figure 5.17 The data miss rates for the user and kernel components behave differently for increases in the L1 data cache size (on the left) versus increases in the L1 data cache block size (on the right). Increasing the L1 data cache from 32 KB to 256 KB (with a 32-byte block) causes the user miss rate to decrease proportionately more than the kernel miss rate: the user-level miss rate drops by almost a factor of 3, while the kernel-level miss rate drops only by a factor of 1.3. The miss rate for both user and kernel components drops steadily as the L1 block size is increased (while keeping the L1 cache at 32 KB). In contrast to the effects of increasing the cache size, increasing the block size improves the kernel miss rate more significantly (just under a factor of 4 for the kernel references when going from 16-byte to 128-byte blocks versus just under a factor of 3 for the user references).

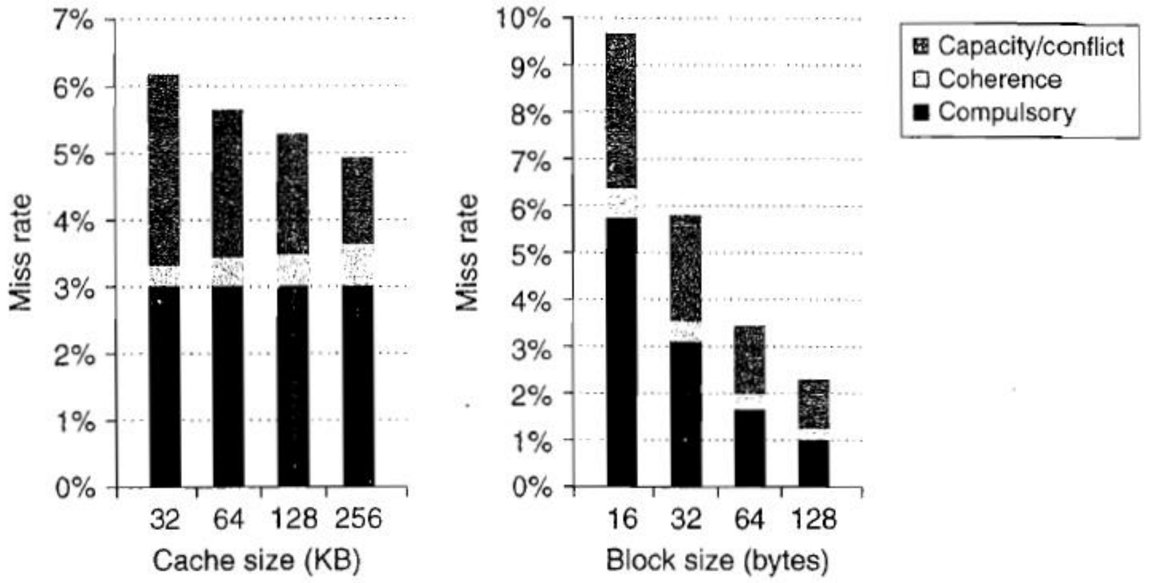


Figure 5.18 The components of the kernel data miss rate change as the L1 data cache size is increased from 32 KB to 256 KB, when the multiprogramming workload is run on eight processors. The compulsory miss rate component stays constant, since it is unaffected by cache size. The capacity component drops by more than a factor of 2, while the coherence component nearly doubles. The increase in coherence misses occurs because the probability of a miss being caused by an invalidation increases with cache size, since fewer entries are bumped due to capacity. As we would expect, the increasing block size of the L1 data cache substantially reduces the compulsory miss rate in the kernel references. It also has a significant impact on the capacity miss rate, decreasing it by a factor of 2.4 over the range of block sizes. The increased block size has a small reduction in coherence traffic, which appears to stabilize at 64 bytes, with no change in the coherence miss rate in going to 128-byte lines. Because there are no significant reductions in the coherence miss rate as the block size increases, the fraction of the miss rate due to coherence grows from about 7% to about 15%.

Problem

- Compare the three approaches for performance evaluation of multiprocessor systems:
  1. Analytical modelling – use mathematical expressions to model the behavior of the systems
  2. Trace-driven simulation – run applications on a real machine and generate a file of relevant events. The traces are then replayed using cache simulators when parameters of the system are changed.
  3. Execution-driven simulators simulate the entire execution maintaining an equivalent structure for the processor state.

Solution

Analytical models

- Can be used to derive high-level insight on the behavior of the system in a very short time.
- The biggest challenge is in determining the values of the parameters.
- While the results from an analytical model can give a good approximation of the relative trends to expect, there may be significant errors in the absolute predictions.

Solution (cont'd)

Trace-driven simulations

- Typically have better accuracy than analytical models. This approach can be fairly accurate when focusing on specific components of the system (e.g., cache system, memory system, etc.).
- Need more time to produce results.
- Does not model the impact of aggressive processors (mispredicted path) and may not model the actual order of accesses with reordering.
- Traces can also be very large, often taking gigabytes of storage, and determining sufficient trace length for trustworthy results is important.
- Hard to generate representative traces from one class of machines that will be valid for all the classes of simulated machines.
- Hard to model synchronization without abstracting the synchronization in the traces to their high-level primitives.

Solution (cont’d)

Execution-driven simulation

1. models all the system components in detail and is consequently the most accurate of the three approaches.
2. The speed of simulation is much slower than that of the other models.
3. In some cases, the extra detail may not be necessary for the particular design parameter of interest.

Problem

- Devise a multiprocessor/cluster benchmark whose performance gets worse as processors are added.

Solution

- Create the benchmark such that all processors update the same variable or small group of variables continually after very little computation.
- For a multiprocessor, the miss rate and the continuous invalidates in between the accesses may contribute more to the execution time than the actual computation and adding more CPU's could slow the overall execution time.
- For a cluster organized as a ring communication costs needed to update the common variables could lead to inverse linear speedup behavior as more processors are added.

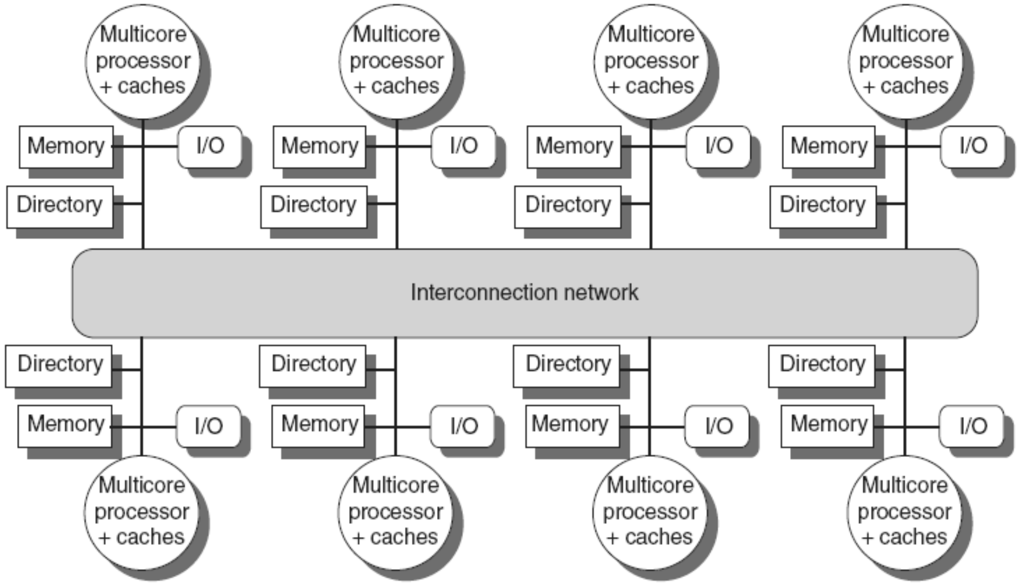
4. DSM – distributed shared memory

- Cache snooping not scalable → the bandwidth of the interconnection network not sufficient when the number of processors increases
- Example:
  - Four 4-core processors running at 4 GHz
  - Able to sustain one reference per clock cycle
  - Most bus traffic due to cache coherence traffic → increasing cache size does not help!!
  - The bus bandwidth required 170 GB/sec far beyond the 4 GB/sec a modern bus could accommodate.
- Distributed directory
  - One entry per memory block
  - Amount of information – (number of memory blocks) x (number of nodes)
- A directory-based coherence protocol must handle
  1. a **read** miss
  2. a **write** to a shared clean cache block

  A **write** miss to a shared block is a combination of (1) and (2)

Directory protocols

- Directory keeps track of every block
  - Which caches have each block
  - Dirty status of each block
- Implemented in shared L3 cache
  - Keep bit vector of size = # cores for each block in L3
  - Not scalable beyond shared L3
- Implemented in a distributed fashion



Directory protocols

- For each block, maintain state:
  1. Shared → One or more nodes have the block cached, value in memory is up-to-date
  2. Uncached → No node has a copy of the cached block
  3. Modified → Exactly one node has a copy of the cache block, value in memory is out-of-date. Need Owner ID
- Need to track which node has a copy of every block
- Directory maintains block states and sends invalidation messages
- To keep track which nodes have copies of a block of cache
  - Bit vector with one bit per memory block maintained by every node
  - Can also identify the owner of each block

Messages

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

Directory protocols

State transition for an individual cache block. Requests from

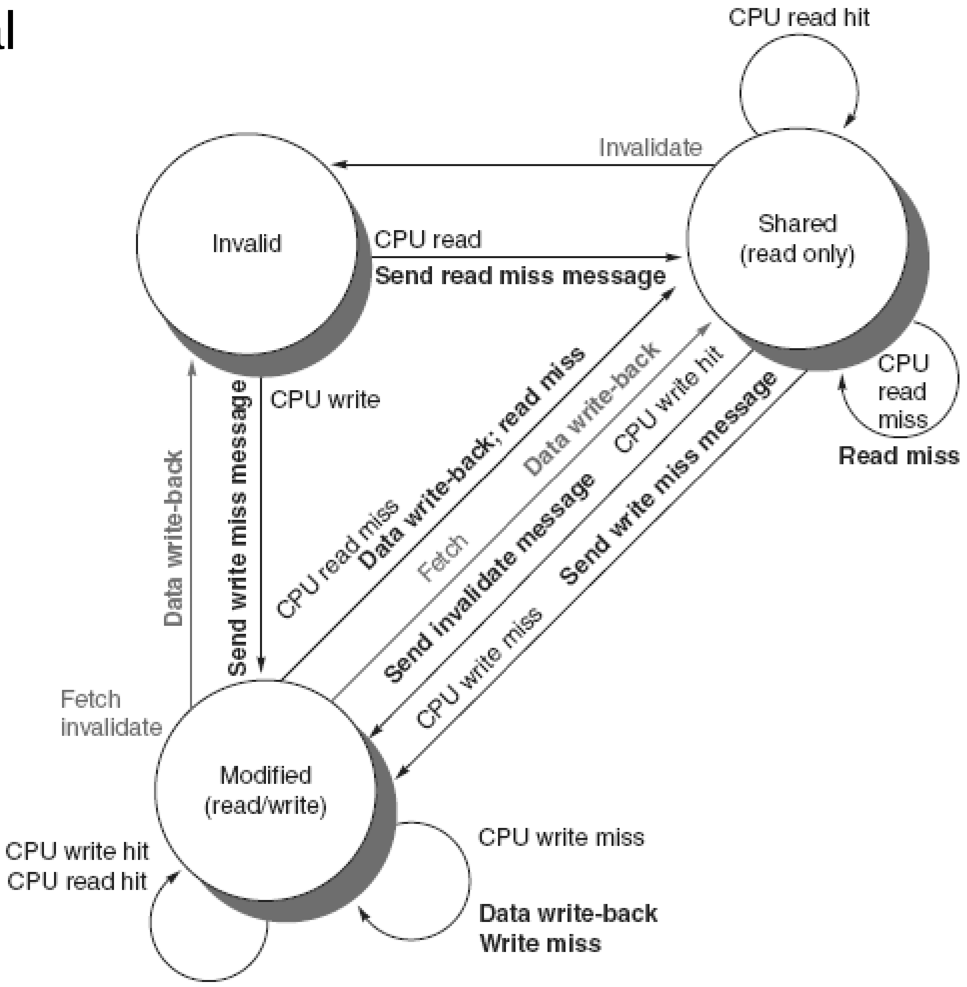
1. Local processor → black
2. Home directory → gray

States similar to those in the snoopy case but

1. Explicit-invalidate
2. Write-back

replace write misses.

An attempt to write a shared cache block is treated as a miss.



Directory protocols

- For uncached block:
  - *Read miss* → Requesting node is sent the requested data and is made the only sharing node, block is now shared.
  - *Write miss* → The requesting node is sent the requested data and becomes the sharing node, block is now exclusive
- For shared block:
  - *Read miss* → The requesting node is sent the requested data from memory, node is added to sharing set
  - Write miss → The requesting node is sent the value, all nodes in the sharing set are sent invalidate messages, sharing set only contains requesting node, block is now exclusive
- For exclusive block:
  - *Read miss* → the owner is sent a data fetch message, block becomes shared, owner sends data to the directory, data written back to memory, sharers set contains old owner and requestor
  - *Data write back* → block becomes un-cached, sharer set is empty
  - *Write miss* → message is sent to old owner to invalidate and send the value to the directory, requestor becomes new owner, block remains exclusive

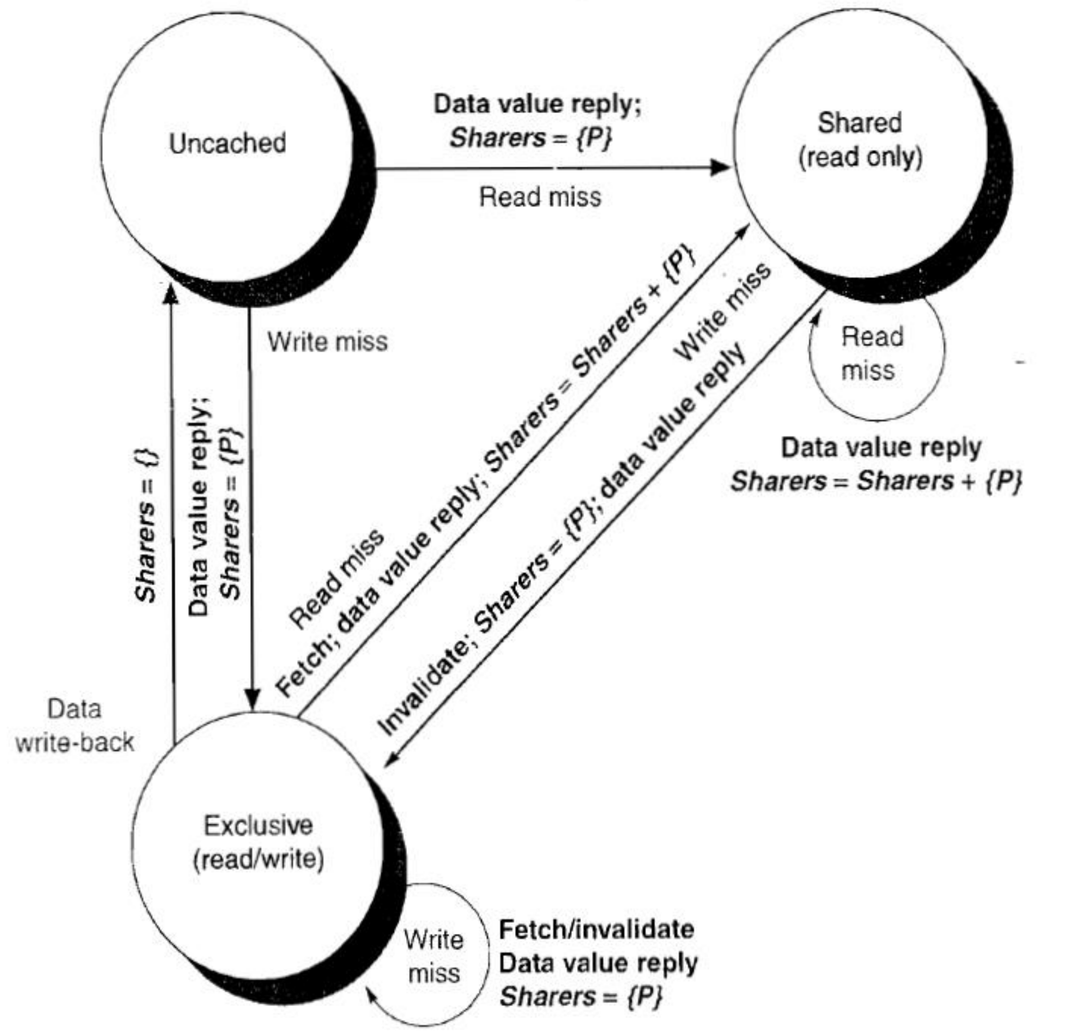

Uncached
Shared (read only)
Exclusive (read/write)
Data value reply; *Sharers* = {*P*}
Read miss
Write miss
Read miss
Data value reply *Sharers* = *Sharers* + {*P*}
Read miss
Fetch; data value reply; *Sharers* = *Sharers* + {*P*}
Write miss
Invalidate; *Sharers* = {*P*}; data value reply
Sharers = {}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

Figure 5.23 The state transition diagram for the directory has the same states and structure as the transition diagram for an individual cache. All actions are in gray because they are all externally caused. Bold indicates the action taken by the directory in response to the request.

5. Synchronization

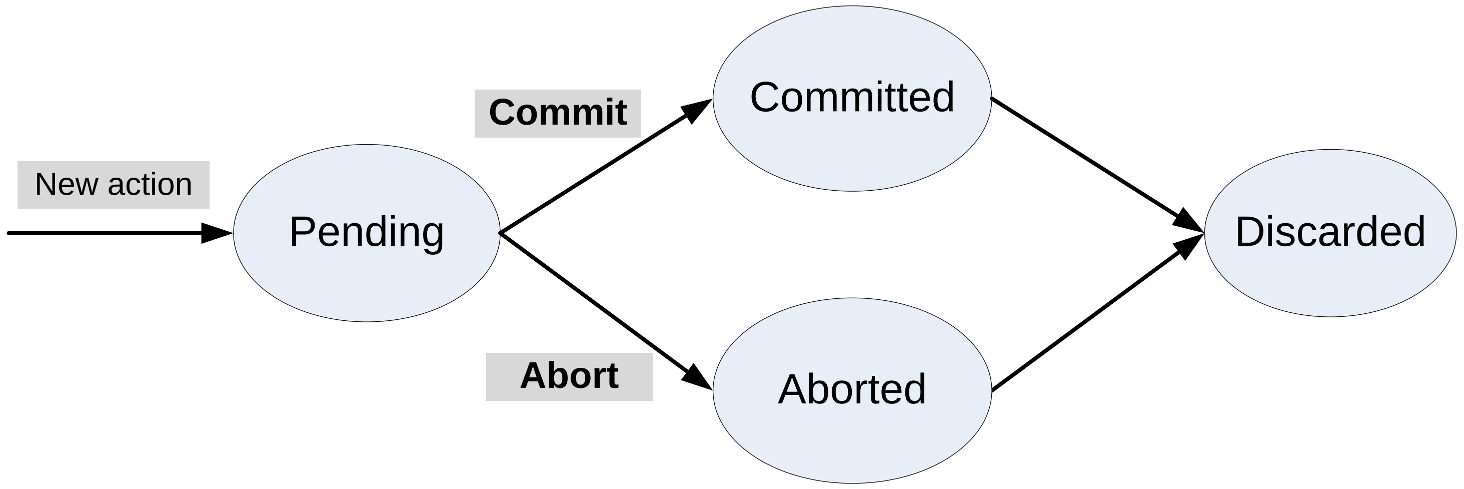
- Synchronization → necessary for coordination of complex activities and multi-threading.
- Atomic actions → actions that cannot be interrupted
- Locks → mechanisms to protect a critical section, code that can be executed by only one thread at a time
- Hardware support for locks
- Thread coordination → multiple threads of a thread group need to act in concert
- Mutual exclusion → only one thread at a time should be allowed to perform an action
- Deadlocks
- Priority inversion

Atomic actions

- Must take special precautions for handling shared resources.
- *Atomic operation* → a multi-step operation should be allowed to proceed to completion without any interruptions and should not expose the state of the system until the action is completed.
- Hiding the internal state of an atomic action reduces the number of states a system can be in thus, it simplifies the design and maintenance of the system.
- Atomicity requires hardware support:
  - Test-and-Set → instruction which writes to a memory location and returns the old content of that memory cell as non-interruptible.
  - Compare-and-Swap → instruction which compares the contents of a memory location to a given value and, only if the two values are the same, modifies the contents of that memory location to a given new value.

All-or-nothing atomicity

- Either the entire atomic action is carried out, or the system is left in the same state it was before the atomic action was attempted;
  a transaction is either carried out successfully, or the record targeted by the transaction is returned to its original state.
- Two phases:
  - Pre-commit → during this phase it should be possible to back up from it without leaving any trace. Commit point - the transition from the first to the second phase. During the pre-commit phase all steps necessary to prepare the post-commit phase, e.g., check permissions, swap in main memory all pages that may be needed, mount removable media, and allocate stack space must be carried out; during this phase no results should be exposed and no actions that are irreversible should be carried out.
  - Post-commit phase → should be able to run to completion. Shared resources allocated during the pre-commit cannot be released until after the commit point.



The states of an all-or-nothing action.

Before-or-after atomicity

- The effect of multiple actions is as if these actions have occurred one after another, in some order.

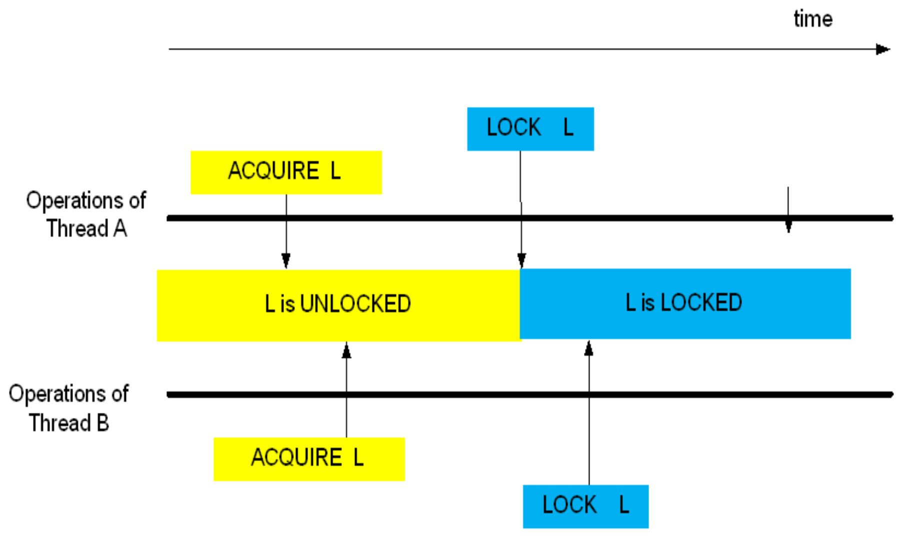
Locks

- *Locks*→ shared variables which acts as a flag to coordinate access to a shared data. Manipulated with two primitives
  - ACQUIRE
  - RELEASE
- Support implementation of before-or-after actions; only one thread can acquire the lock, the others have to wait.
- All threads must obey the convention regarding the locks.
- The two operations ACQUIRE and RELEASE must be atomic.
- Hardware support for implementation of locks
  - RSM – Read and Set Memory
  - CMP –Compare and Swap
- RSM (mem)
  - If *mem=LOCKED* then RSM returns *r=LOCKED* and sets *mem=LOCKED*
  - If *mem=UNLOCKED* the RSM returns *r=LOCKED* and *sets mem=LOCKED*

```
structure lock
  integer state

procedure FAULTY_ACQUIRE (lock reference L)
  while L.state ← LOCKED do nothing
 l.state ← LOCKED

procedure RELEASE (lock reference L)
  L.state ←UNLOCKED
```



```
/*          The hardware instruction RSM  –        Read and Set Memory )
  procedure   RSM (   reference   mem )
    do atomic
  r      mem
  mem       LOCKED
 return          r       /*             If mem was UNLOCKED then RSM returns r  =        UNLOCKED and sets mem  =   LOCKED
                         /*             If mem was LOCKED      then RSM returns r  =        LOCKED       and sets mem  =   LOCKED


/*                        RSM allows us to implement ACQUIRE and RELEASE as critical sections



  procedure     ACQUIRE (  lock    reference  L)
  R 1       RSM(L.  state      )                  /*       Read and Set lock L
   while  R1 =    LOCKED            do         /*          If already LOCKED keep checking
    R1      RSM(L.  state)

  procedure     RELEASE (  lock    reference  L)
 L.  state          UNLOCKED
```

Hardware support for atomic actions:

- Atomic actions
  - Atomic exchange
    - Swaps register with memory location
  - Test-and-set
    - Sets under condition
  - Fetch-and-increment
    - Reads original value from memory and increments it in memory
  - Requires memory read and write in uninterruptable instruction

  - load linked/store conditional
    - If the contents of the memory location specified by the load linked are changed before the store conditional to the same address, the store conditional fails

Cache coherence and spin locks

- *Spin lock* → a lock the system keep trying to acquire
- The system supports cache coherence → cache the locks
  - When a thread tries to acquire a lock it will use the local copy rather than requiring a global memory access.
  - Locality of lock access – a thread that used a lock is likely to need it again.

Implementing locks

- Spin lock
  - If no coherence:

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

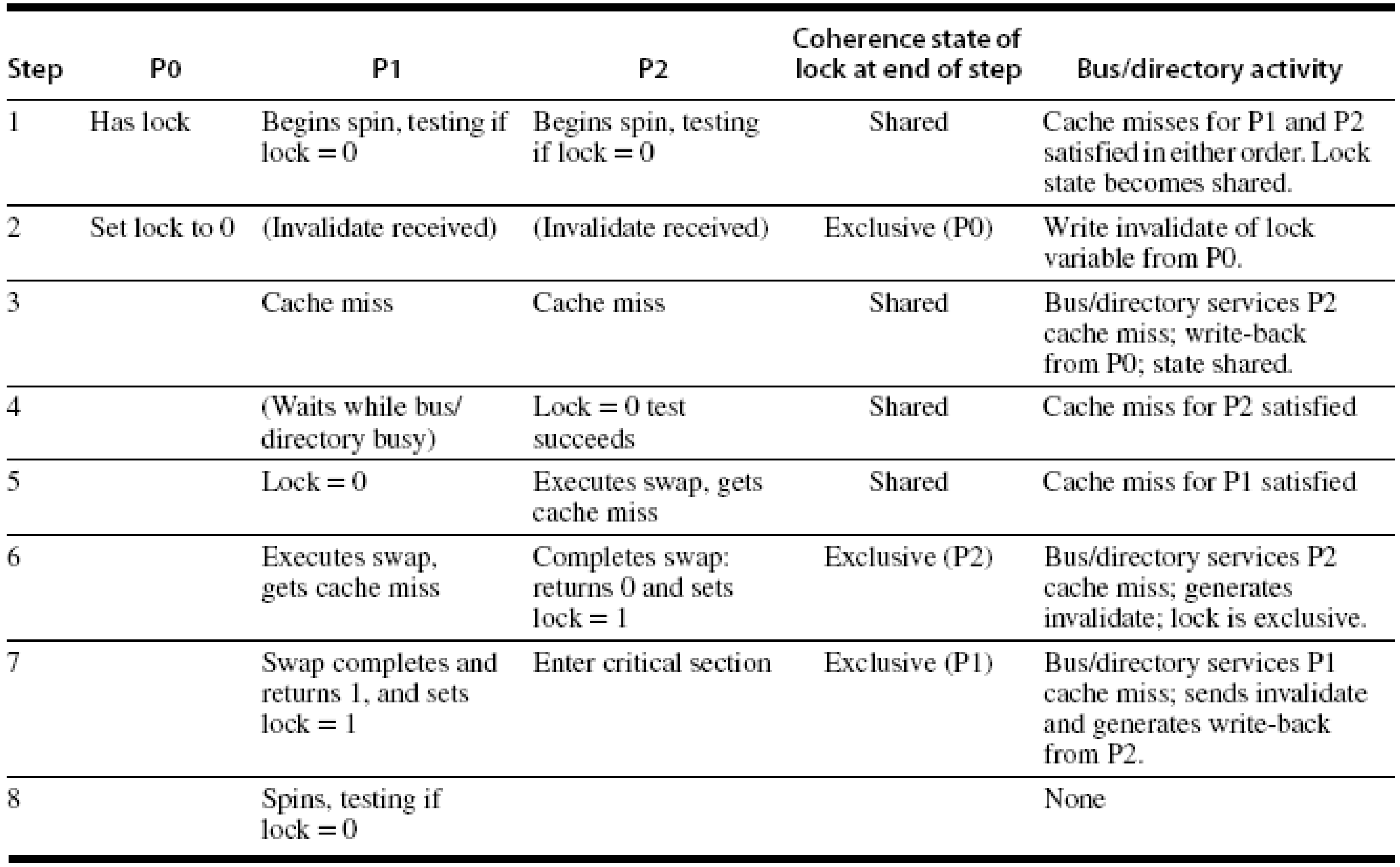
  - If coherence:

```
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
```

Implementing locks

- Advantage of this scheme: reduces memory traffic

lock=0 ➔unlocked lock=1 ➔locked



Step	P0	P1	P2	Coherence state of lock at end of step	Bus/directory activity
1	Has lock	Begins spin, testing if lock = 0	Begins spin, testing if lock = 0	Shared	Cache misses for P1 and P2 satisfied in either order. Lock state becomes shared.
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; state shared.
4		(Waits while bus/ directory busy)	Lock = 0 test succeeds	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; lock is exclusive.
7		Swap completes and returns 1, and sets lock = 1	Enter critical section	Exclusive (P1)	Bus/directory services P1 cache miss; sends invalidate and generates write-back from P2.
8		Spins, testing if lock = 0			None

Thread coordination

- *Critical section* ➔ code that accesses a shared resource
- *Race conditions* ➔ two or more threads access shared data and the result depends on the order in which the threads access the shared data.
- *Mutual exclusion* ➔ only one thread should execute a critical section at any one time.
- *Scheduling algorithms* → decide which thread to choose when multiple threads are in a RUNNABLE state
  - FIFO – first in first out
  - LIFO – last in first out
  - Priority scheduling
  - EDF – earliest deadline first
- *Preemption* → ability to stop a running activity and start another one with a higher priority.

Conditions for thread coordination

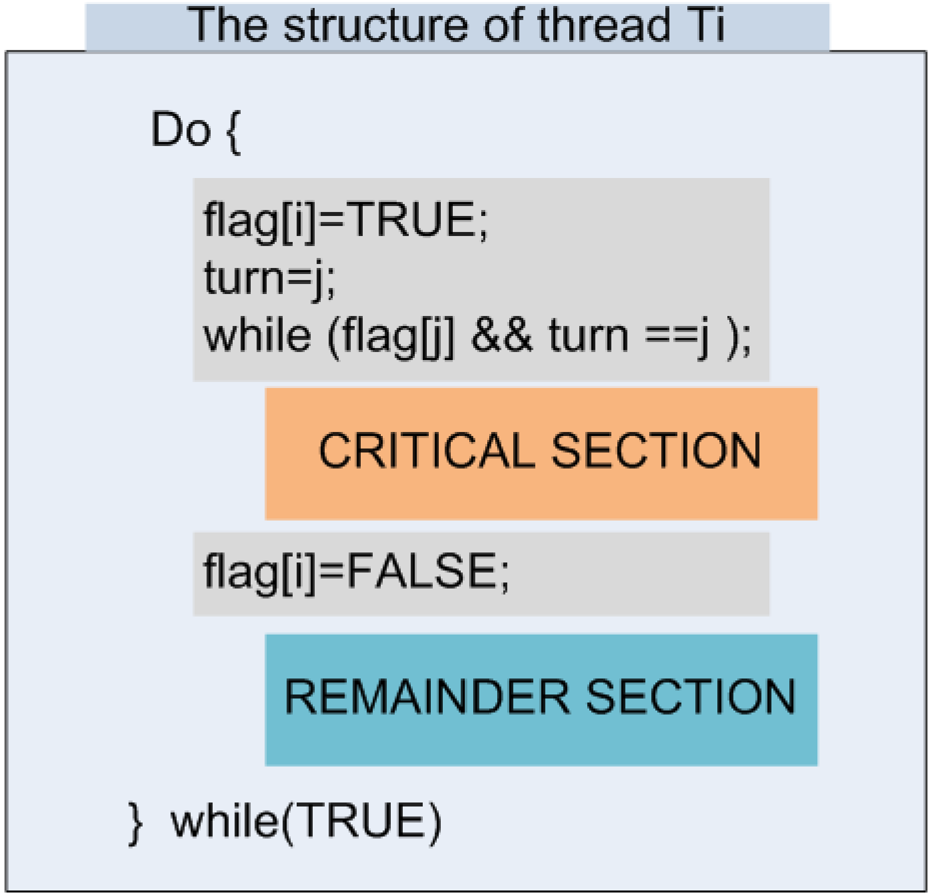
1. *Safety*: The required condition will never be violated.
2. *Liveness:* The system should eventually progress irrespective of contention.
3. *Freedom From Starvation:* No process should be denied progress for ever. That is, every process should make progress in a finite time.
4. *Bounded Wait:* Every process is assured of not more than a fixed number of overtakes by other processes in the system before it makes progress.
5. *Fairness:* dependent on the scheduling algorithm
   - FIFO: No process will ever overtake another process.
   - LRU: The process which received the service least recently gets the service next.

Conditions for mutual exclusion

- A solution for mutual exclusion problem should guarantee:
  - Safety → the mutual exclusion property is never violated
  - Liveness → a thread will access the shared resource in a finite time
  - Freedom for starvation → a thread will access the shared resource in a finite time
  - Bounded wait → a thread will access the shared resource at least after a given number of accesses by other threads.

A solution to critical section problem

- Applies only to two threads T_i and T_j with i,j ={0,1} which share
  - *integer turn* → if *turn=i* then it is the turn of T_i to enter the critical section
  - *boolean flag[2]* → if *flag[i]= TRUE* then T_i is ready to enter the critical section
- To enter the critical section thread T_i
  - sets *flag[i]= TRUE*
  - sets *turn=j*
- If both threads want to enter then *turn* will end up with a value of either *i* or *j* and the corresponding thread will enter the critical section.
- T_i enters the critical section only if either *flag[j]= FALSE* or *turn=i*
- The solution is correct
  - Mutual exclusion is guaranteed
  - The liveliness is ensured
  - The bounded-waiting is met
- But this solution may not work as load and store instructions can be interrupted on modern computer architectures



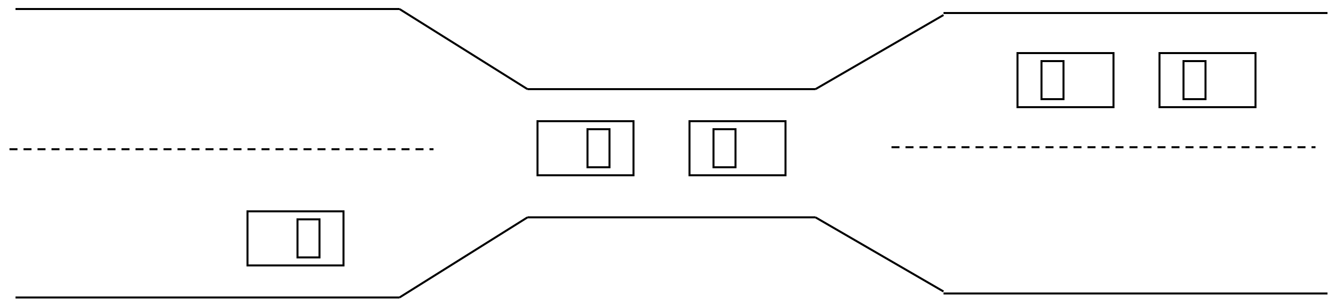
The structure of thread Ti

```
Do {
    flag[i]=TRUE;
    turn=j;
    while (flag[j] && turn ==j );
        CRITICAL SECTION
    flag[i]=FALSE;
        REMAINDER SECTION
} while(TRUE)
```

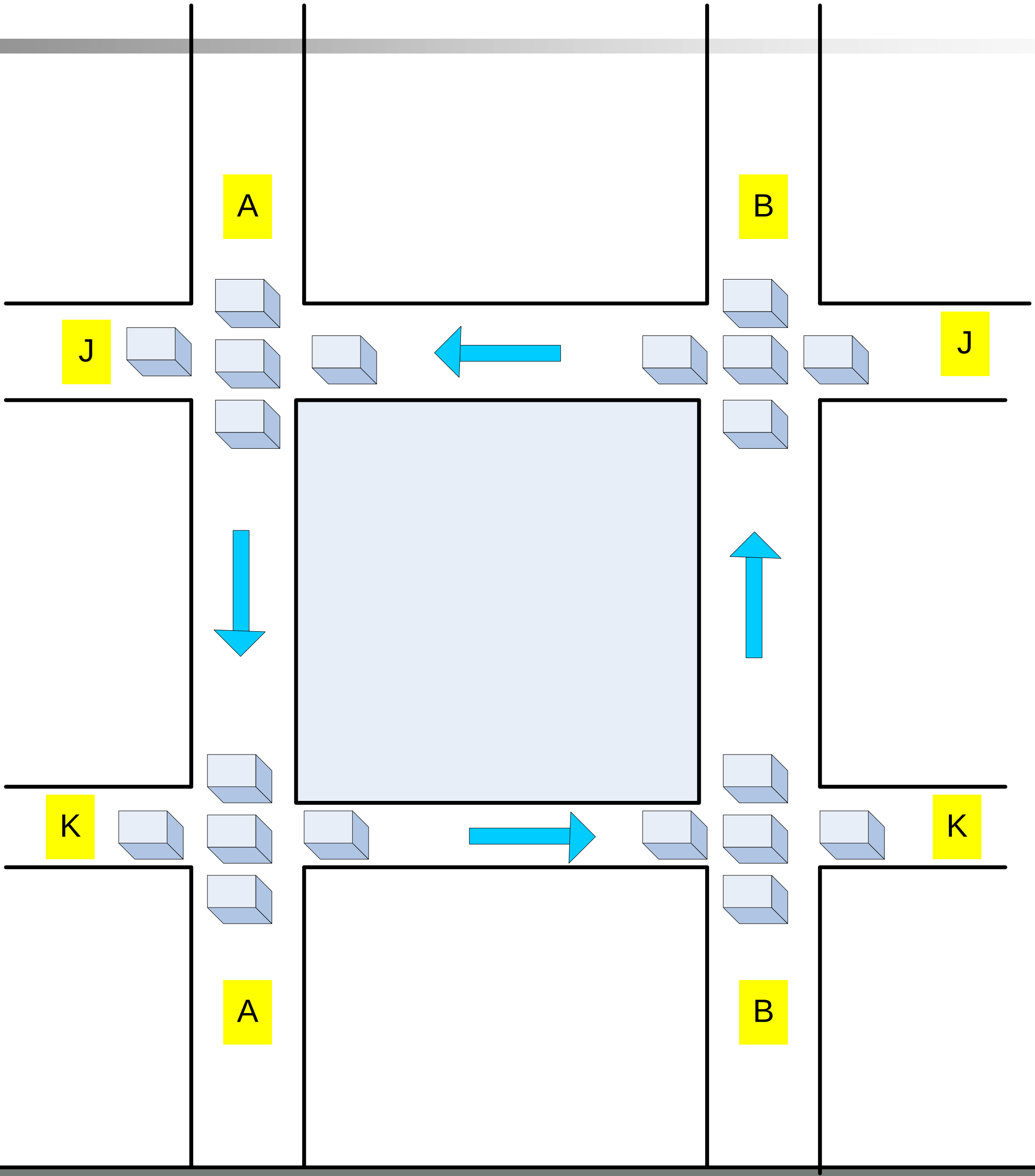
Side effects of thread coordination

- *Priority inversion* → a lower priority activity is allowed to run before one with a higher priority
- *Deadlocks*
  - Happen quite often in real life and the proposed solutions are not always logical: “When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone.” a pearl from Kansas legislation.
  - Examples
    - Deadlock jury.
    - Deadlock legislative body.

Examples of deadlock



- Traffic only in one direction.
- Solution → one car backs up (preempt resources and rollback). Several cars may have to be backed up .
- Starvation is possible.


A
B
J
J
K
K
A
B

Thread deadlock

- Deadlocks → prevent sets of concurrent threads/processes from completing their tasks.
- How does a deadlock occur → a set of blocked threads each holding a resource and waiting to acquire a resource held by another thread in the set.
- Example
  - locks *A* and *B*, initialized to 1

```
    P0               P1
wait (A);         wait(B)
wait (B);         wait(A)
```

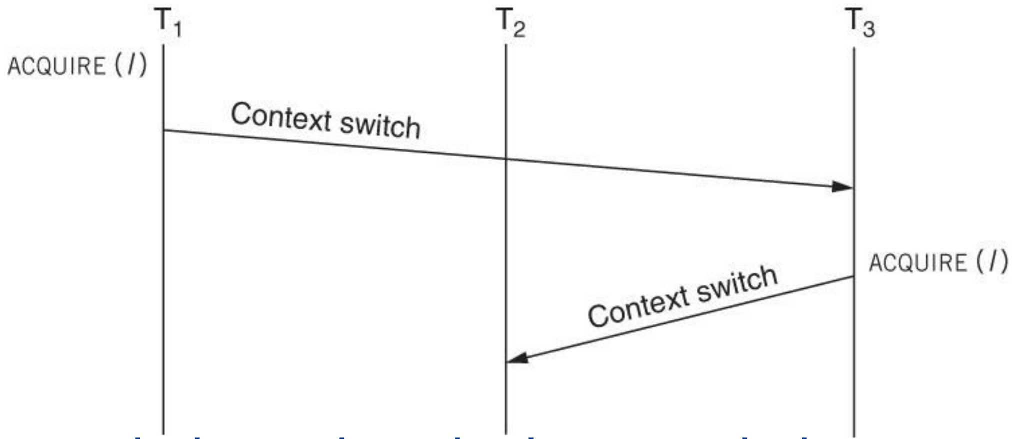
- Aim→ prevent or avoid deadlocks

Priority scheduling

- Each thread/process has a priority and the one with the highest priority (smallest integer ≡ highest priority) is scheduled next.
  - Preemptive
  - Non-preemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time
- Problem ➔ Starvation – low priority threads/processes may never execute
- Solution to starvation ➔ Aging – as time progresses increase the priority of the thread/process
- Priority my be computed dynamically

Priority inversion

A lower priority thread/process prevents a higher priority one from running.



T_3 has the highest priority, T_1 has the lowest priority; T_1 and T_3 share a lock.

1. T_1 acquires the lock, then it is suspended when T_3 starts.
2. Eventually T_3 requests the lock and it is suspended waiting for T_1 to release the lock.
3. T_2 has higher priority than T_1 and runs; neither T_3 nor T_1 can run; T_1 due to its low priority, T_3 because it needs the lock help by T_1.

Solution→ Allow a low priority thread holding a lock to run with the higher priority of the thread which requests the lock

6. Models of memory consistency

- Multiple processors/cores should see a consistent view of memory.
- What properties should be enforced among **reads** and **writes** to different location in a multiprocessor?
- Hard problem!!
- *Sequential consistency* →the result of any execution be the same as if the memory accesses by each processor be kept in order and the accesses were interleaved
    - To enforce it require a processor to delay any memory access until all invalidations caused by that access are completed
    - Simplicity from programmer's point of view
    - Performance disadvanatage

Example

Two threads are running on different processors and A and B are cached by each processor and the initial values are 0.

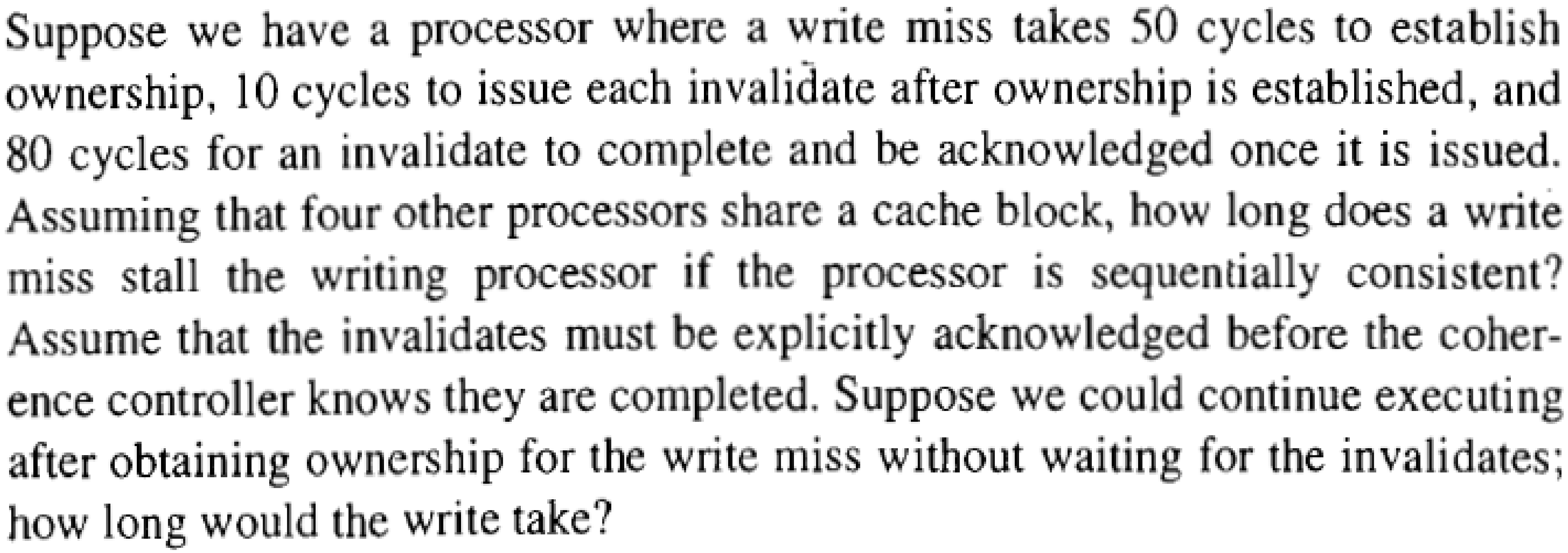
Processor 1:	Processor 2:
A=0	B=0
…	…
A=1	B=1
if (B==0) …	if (A==0) …

Should be impossible for both if-statements to be evaluated as true

- Delayed write invalidate?

- Sequential consistency:
  - Result of execution should be the same as long as:
    - Accesses on each processor were kept in order
    - Accesses on different processors were arbitrarily interleaved

Problem



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 coherence 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?

Solution

When we wait for invalidates, each write takes the sum of the ownership time plus the 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.

Implementing locks

- To implement, delay completion of all memory accesses until all invalidations caused by the access are completed
  - Reduces performance!

- Alternatives:
  - Program-enforced synchronization to force write on processor to occur before read on the other processor
    - Requires synchronization object for A and another for B
      - “Unlock” after write
      - “Lock” after read

Relaxed consistency models

- Rules:
  - $X \rightarrow Y$
    - Operation X must complete before operation Y is done
    - Sequential consistency requires:
      - $R \rightarrow W$, $R \rightarrow R$, $W \rightarrow R$, $W \rightarrow W$
- Relaxing
  1. Relax $W \rightarrow R$

     "Total store ordering"
  2. Relax $W \rightarrow W$

     "Partial store order"
  3. Relax $R \rightarrow W$ and $R \rightarrow R$

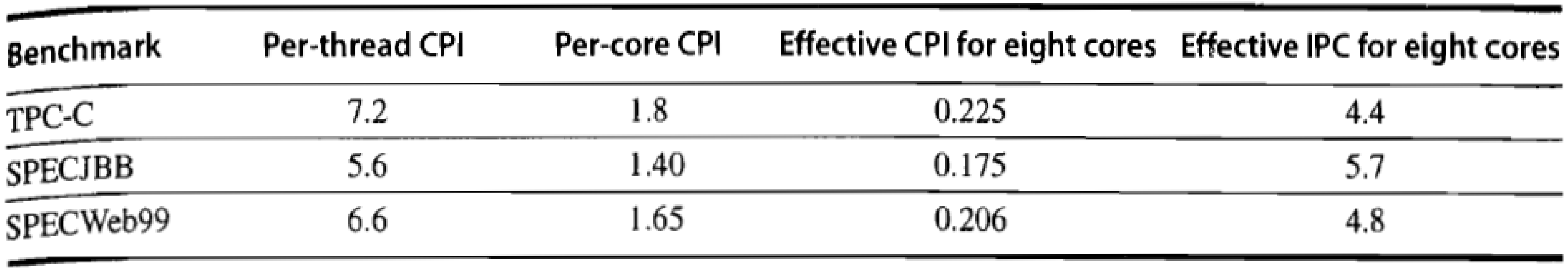
     "Weak ordering" and "release consistency"

Relaxed consistency models

- Consistency model is multiprocessor specific
- Programmers will often implement explicit synchronization
- Speculation gives much of the performance advantage of relaxed models with sequential consistency
  - Basic idea: if an invalidation arrives for a result that has not been committed, use speculation recovery

Multiprocessing and multithreading

- Performance gains on an Intel i7
- Three benchmarks
  - TPC – C (transaction processing)
  - SPECJBB (SPEC Java Business Benchmark)
  - SPECWEb99



Benchmark	Per-thread CPI	Per-core CPI	Effective CPI for eight cores	Effective IPC for eight cores
TPC-C	7.2	1.8	0.225	4.4
SPECJBB	5.6	1.40	0.175	5.7
SPECWeb99	6.6	1.65	0.206	4.8

Figure 5.25 The per-thread CPI, the per-core CPI, the effective eight-core CPI, and the effective IPC (inverse of CPI) for the eight-core Sun T1 processor.

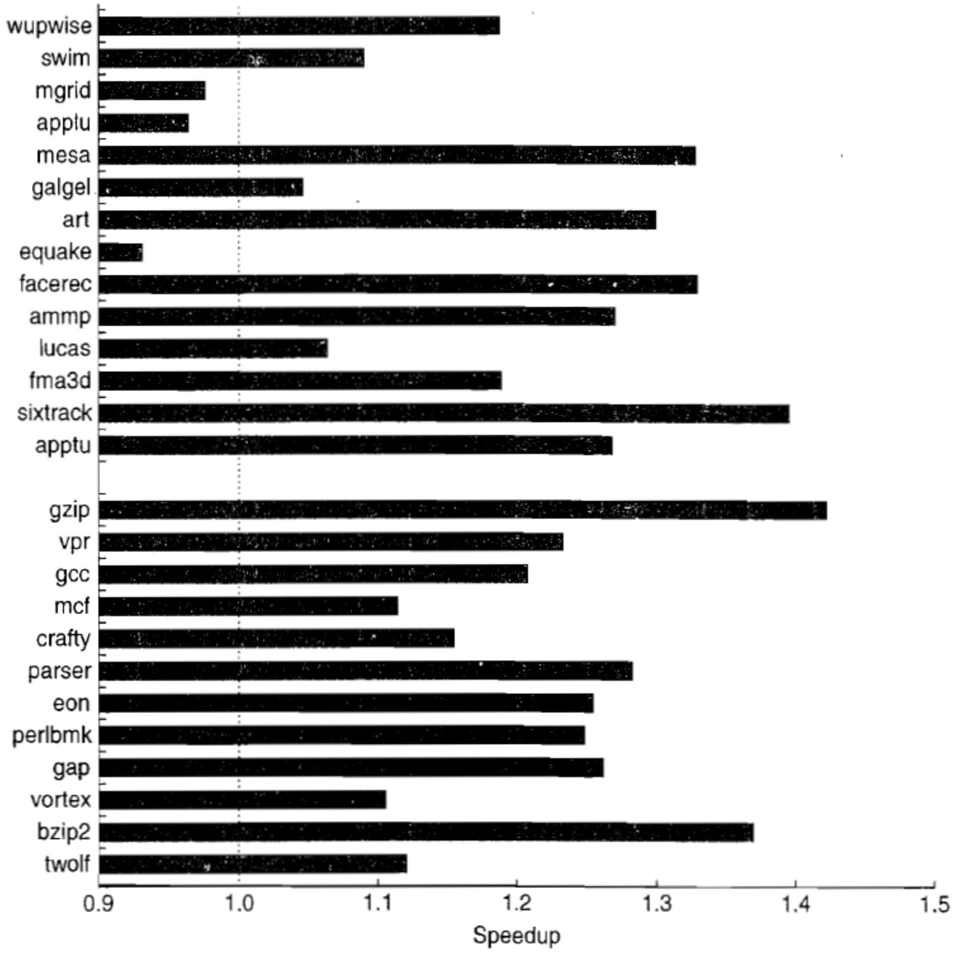


Figure 5.26 A comparison of SMT and single-thread (ST) performance on the eight-processor IBM eServer p5 575. Note that the *y*-axis starts at a speedup of 0.9, a performance loss. Only one processor in each Power5 core is active, which should slightly improve the results from SMT by decreasing destructive interference in the memory system. The SMT results are obtained by creating 16 user threads, while the ST results use only eight threads; with only one thread per processor, the Power5 is switched to single-threaded mode by the OS. These results were collected by John McCalpin of IBM. As we can see from the data, the standard deviation of the results for the SPECfpRate is higher than for SPECintRate (0.13 versus 0.07), indicating that the SMT improvement for FP programs is likely to vary widely.

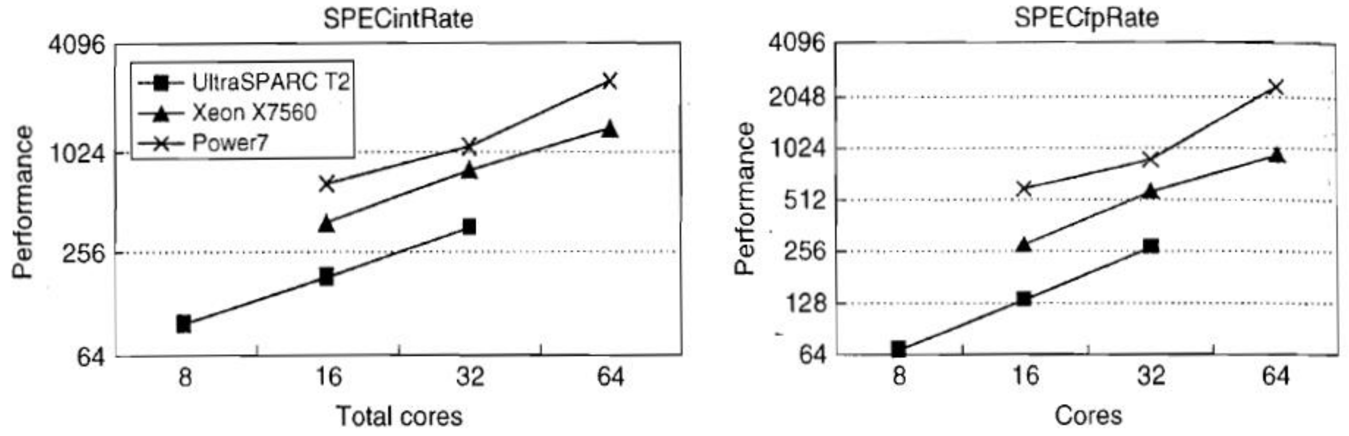


Figure 5.28 The performance on the SPECRate benchmarks for three multicore processors as the number of processor chips is increased. Notice for this highly parallel benchmark, nearly linear speedup is achieved. Both plots are on a log-log scale, so linear speedup is a straight line.

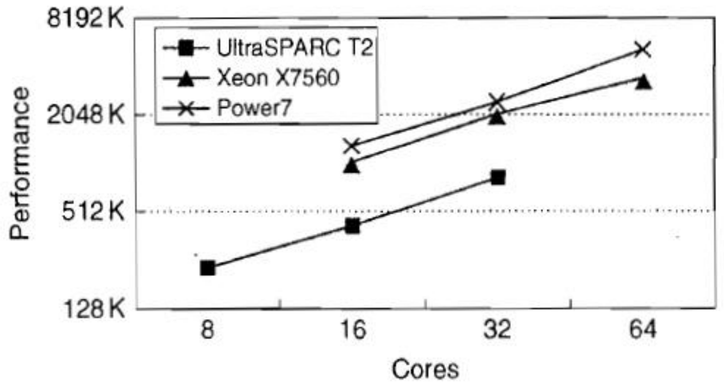


Figure 5.29 The performance on the SPECjbb2005 benchmark for three multicore processors as the number of processor chips is increased. Notice for this parallel benchmark, nearly linear speedup is achieved.