

Chapter 5

Multiprocessors and Thread-Level Parallelism

5.1 Introduction

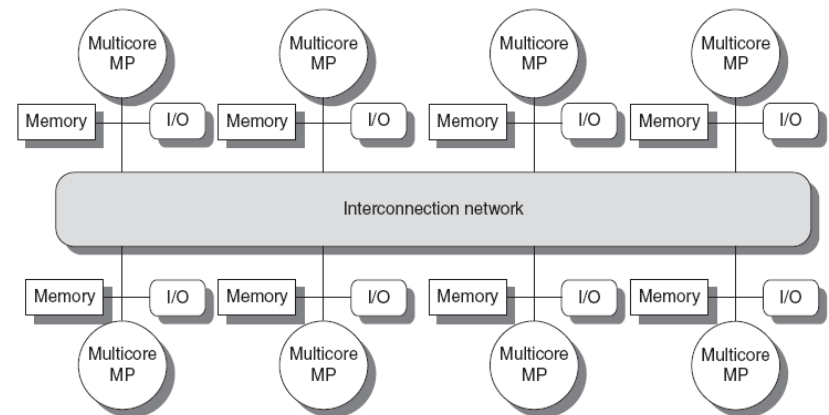
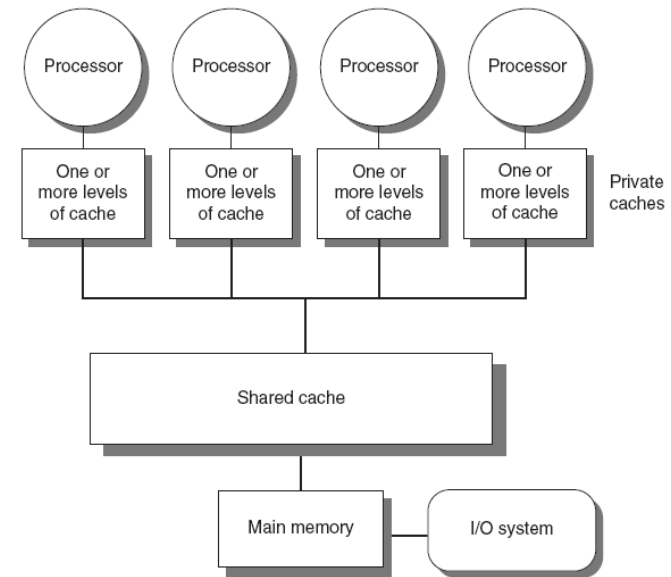
- Importance of multiprocessing
 - Power wall, ILP wall: power and silicon costs grew faster than performance
 - Growing interest in high-end servers, cloud computing, SaaS
 - Growth of data-intensive applications, internet, massive data....
 - Insight: current desktop performance is acceptable, since data-compute intensive applications run in the cloud
 - Improved understanding of how to use multiprocessors effectively: servers, natural parallelism in large data sets or large number of independent requests
 - Advantages of replicating a design rather than investing in a unique design

Introduction

- Thread-Level parallelism
 - Have multiple program counters
 - Uses MIMD model
 - Targeted for tightly-coupled shared-memory multiprocessors
- For n processors, need n threads
- Amount of computation assigned to each thread = grain size
 - Threads can be used for data-level parallelism, but the overheads may outweigh the benefit

Types

- Symmetric multiprocessors (SMP)
 - Small number of cores
 - Share single memory with uniform memory latency
- 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



Challenges of Parallel Processing

- Two main problems
 - Limited parallelism
 - example: to achieve a speedup of 80 with 100 processors we need to have 99.75% of code able to run in parallel !! (see exmpl p349)
 - Communication costs: 30-50 cycles between separate cores, 100-500 cycle between separate chips (next slide)
- Solutions
 - Limited parallelism
 - better algorithms
 - software systems should maximize hardware occupancy
 - Communication costs; reducing frequency of remote data access
 - HW: caching shared data
 - SW: restructuring data to make more accesses local

Example

- We want a speedup of 80 with 100 processors

Answer Recall from Chapter 1 that Amdahl's law is

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

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:

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

Simplifying this equation yields:

$$0.8 \times \text{Fraction}_{\text{parallel}} + 80 \times (1 - \text{Fraction}_{\text{parallel}}) = 1$$

$$80 - 79.2 \times \text{Fraction}_{\text{parallel}} = 1$$

$$\text{Fraction}_{\text{parallel}} = \frac{80 - 1}{79.2}$$

$$\text{Fraction}_{\text{parallel}} = 0.9975$$

Example - Communication

Suppose we have an application running on a 32-processor multiprocessor, which has a 200 ns time to handle reference to a remote memory. For this application, assume that all the references except those involving communication hit in the local memory hierarchy, which is slightly optimistic. Processors are stalled on a remote request, and the processor clock rate is 3.3 GHz. If the base CPI (assuming that all references hit in the cache) is 0.5, how much faster is the multiprocessor if there is no communication versus if 0.2% of the instructions involve a remote communication reference?

Example - Communication

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 practice, the performance analysis is much more complex, since some fraction of the noncommunication references will miss in the local hierarchy and the remote access time does not have a single constant value. For example, the cost of a remote reference could be quite a bit worse, since contention caused by many references trying to use the global interconnect can lead to increased delays.

Cache Coherence

- 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

- Coherence
 - All 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
 - When a written value will be returned by a read
 - 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 one location
 - Snooping
 - Each core tracks sharing status of each block

Snoopy Coherence Protocols

- Write invalidate
 - On write, invalidate all other copies
 - Use bus itself to serialize
 - Write cannot complete until bus access is obtained

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

- Write update
 - On write, update all copies

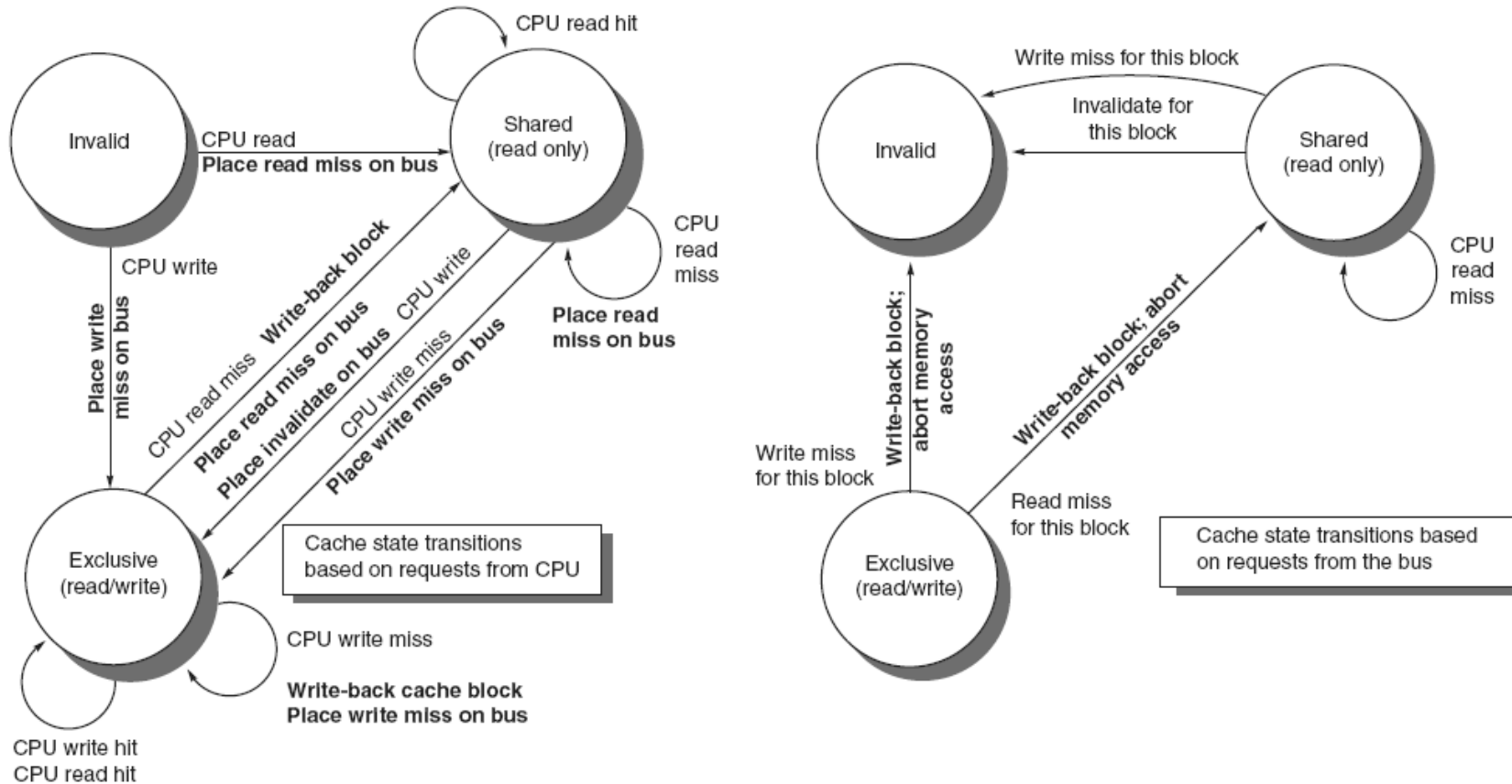
Snoopy Coherence Protocols

- Locating an item when a read miss occurs
 - In write-back cache, the updated value must be sent to the requesting processor
- Cache lines marked as shared or exclusive/modified
 - Only writes to shared lines need an invalidate broadcast
 - After this, the line is marked as exclusive

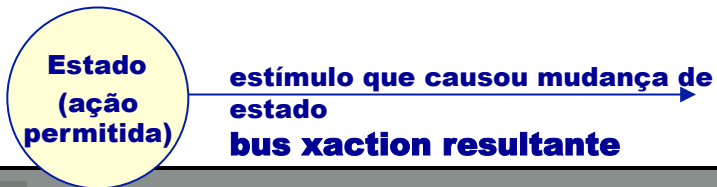
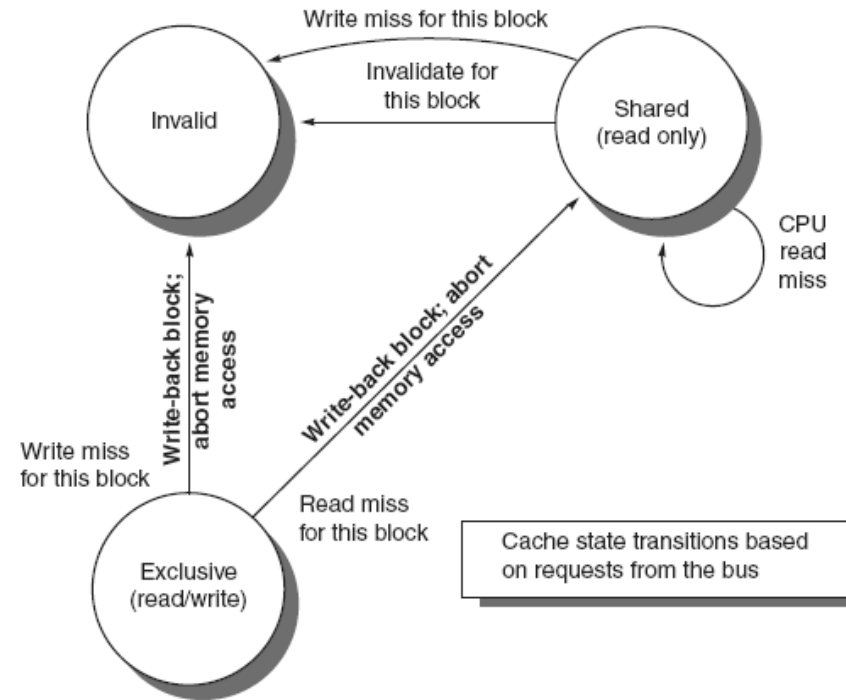
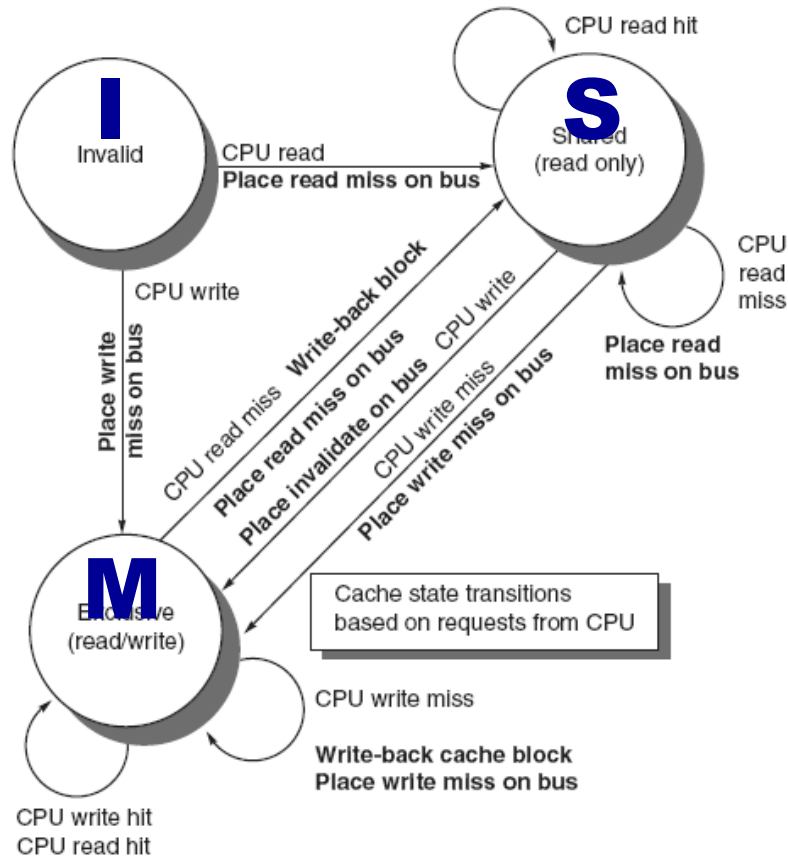
Snoopy Coherence Protocols

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 <i>ownership</i> 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.

Snoopy Coherence Protocols



Snoopy Coherence Protocols: MSI



Miss para um bloco em estado ≠ inválido → dado está lá mas *wrong tag* → miss

Snoopy Coherence Protocols

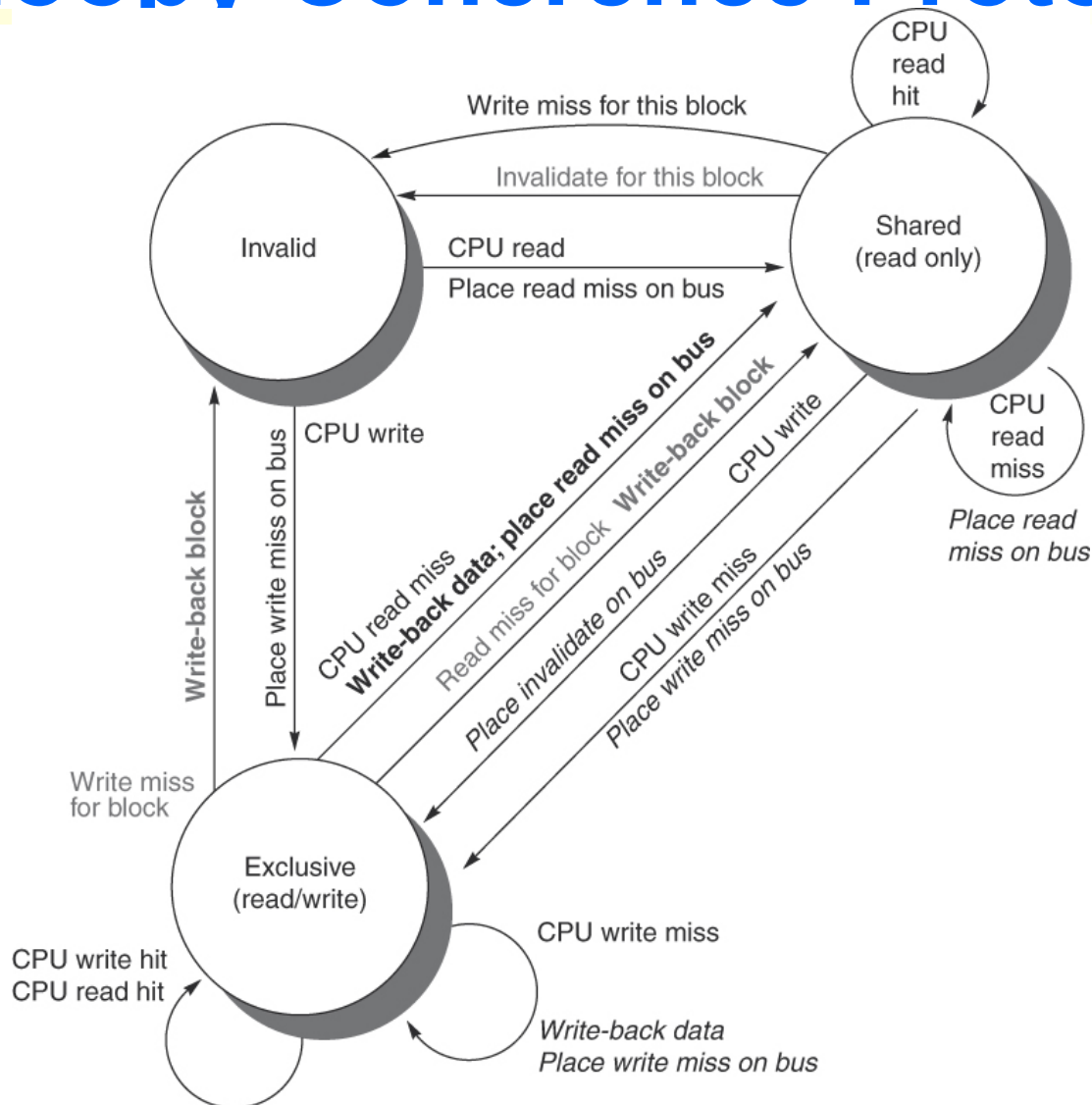


Figure 5.7 Cache coherence state diagram with the state transitions induced by the local processor shown in black and by the bus activities shown in gray. Activities on a transition are shown in bold.

Snoopy Coherence Protocols

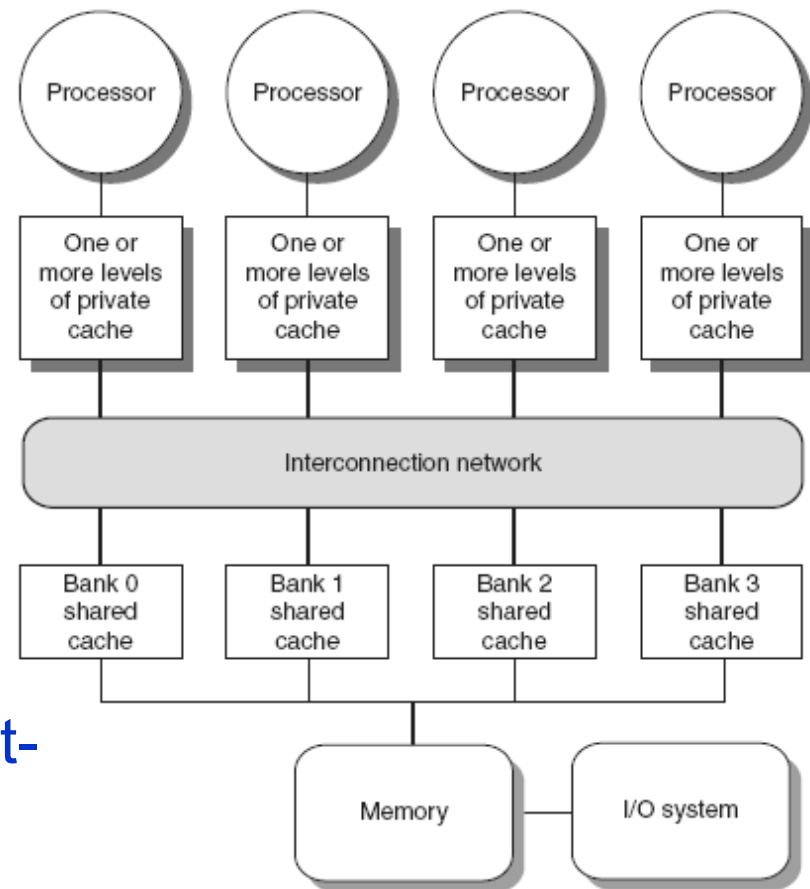
- Complications for the basic MSI protocol:
 - Operations are not atomic
 - E.g. detect miss, acquire bus, receive a response
 - Creates possibility of deadlock and races
 - One solution: processor that sends invalidate can hold bus until other processors receive the invalidate
- Extensions:
 - Add exclusive state to indicate clean block in only one cache (MESI protocol)
 - Prevents needing to write invalidate on a write if a block is read by a single cache then written by the same cache
 - Owned state (MOESI)

Snoopy Coherence Protocols

- Extensions:
 - Owned state (MOESI)
 - Indicates that the block is owned by that cache and it is out of date in memory
 - Owner must supply it on a miss and write-back it in replacement
 - MSI/MESI
 - Share a block in M state forces a write-back

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



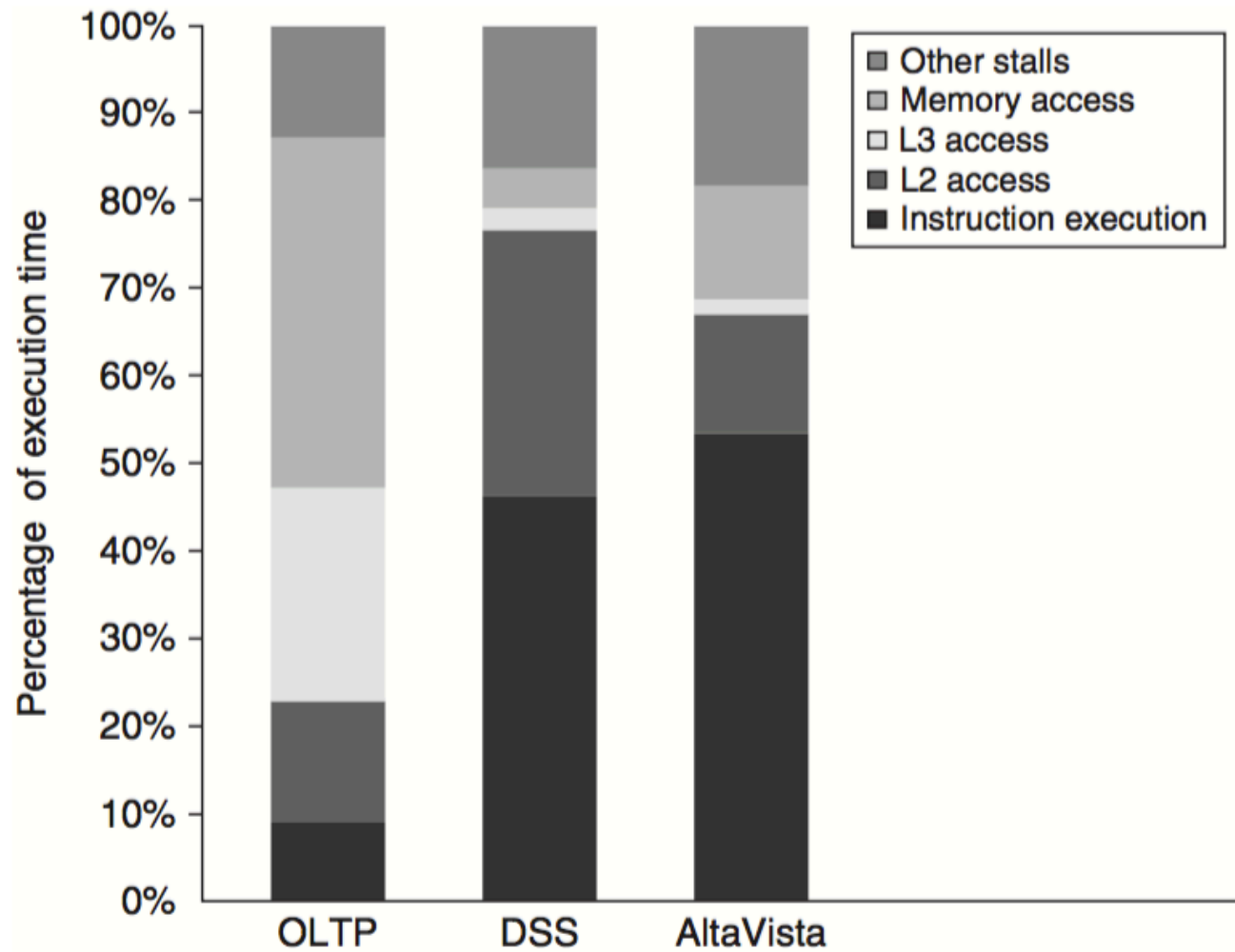
Evolution

- Bus + snoop + small scale multiprocessing = ok
- As number of processors increase
 - multibus: snoopy?
 - interconnection network: snoopy?
- Snoopy demands broadcast, ok with bus
 - also possible in interconnection network → traffic, latency, write serialization
- All solutions but single bus lack its easy “bus order” → write serialization
- Races?
- Directory is more appropriate for implementing cache coherence protocols in large scale multiprocessors
- (see history, devil in details, textbook)

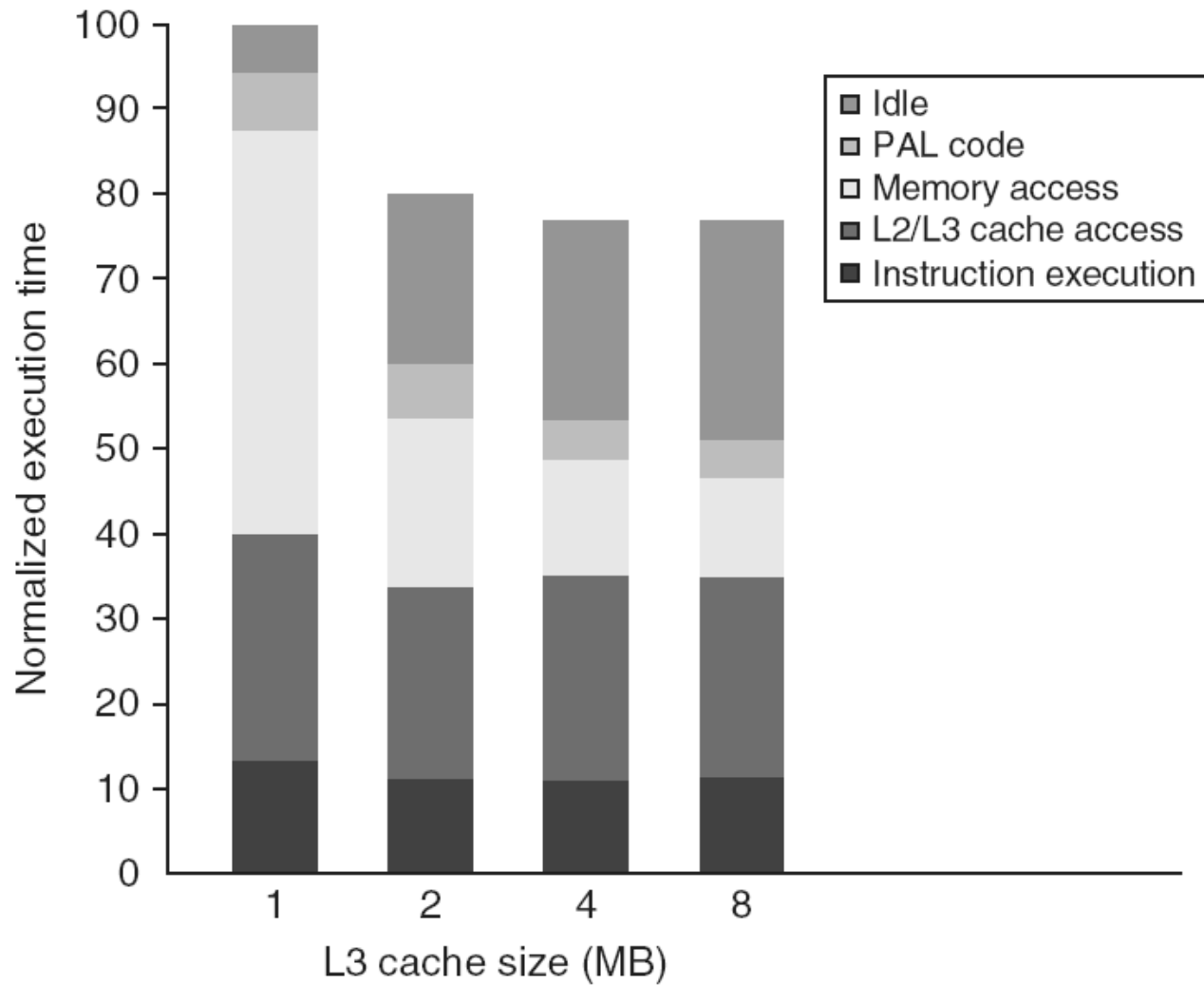
Performance

- Coherence influences cache miss rate
 - Coherence misses
 - True sharing misses
 - Write to shared block (transmission of invalidation)
 - Read an invalidated block
 - False sharing misses
 - Read an unmodified word in an invalidated block

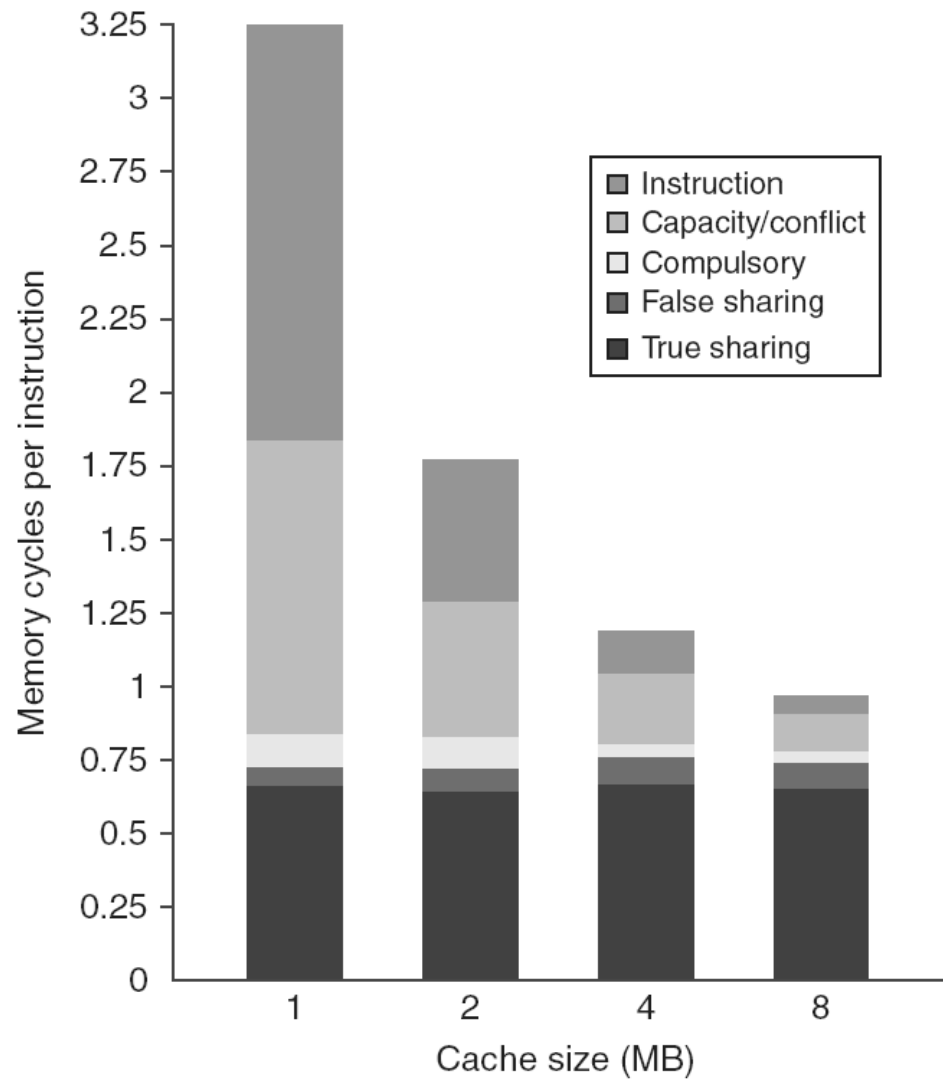
Performance Study: Commercial Workload



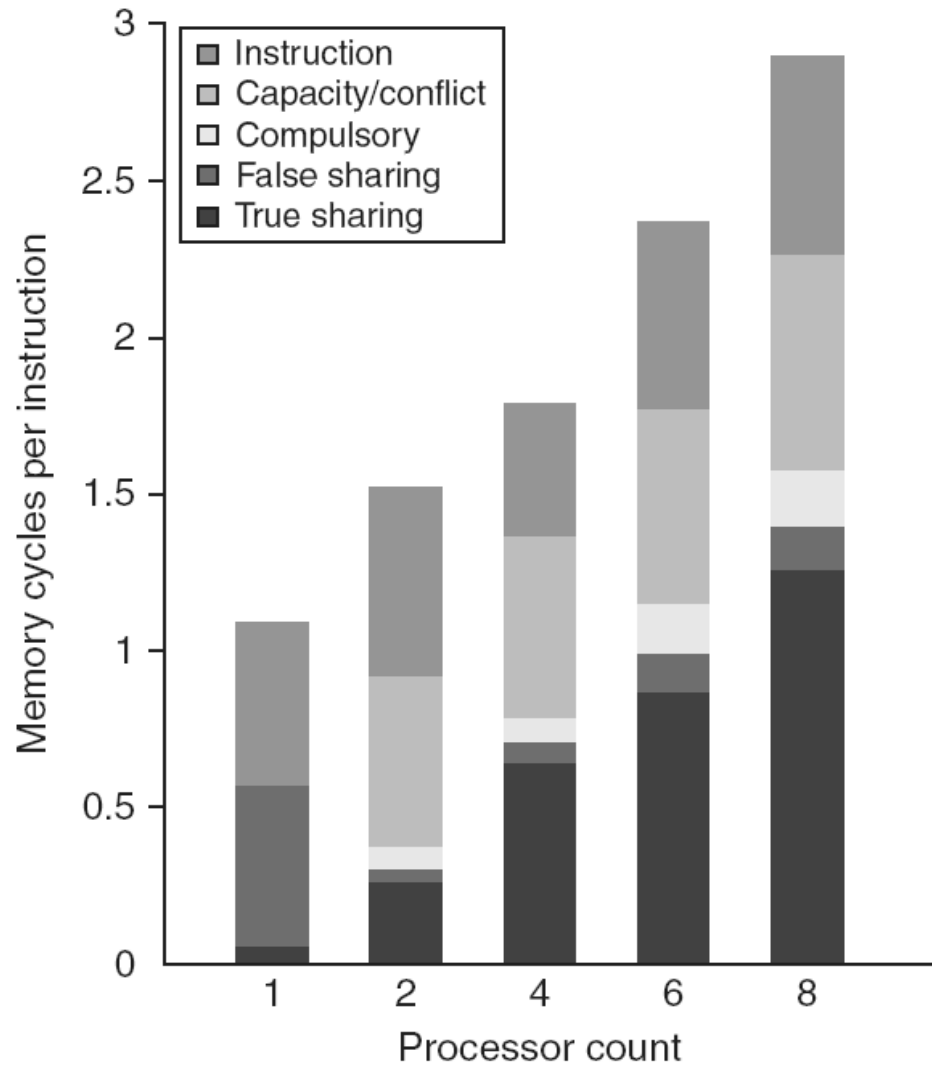
Performance Study: OLTP L3



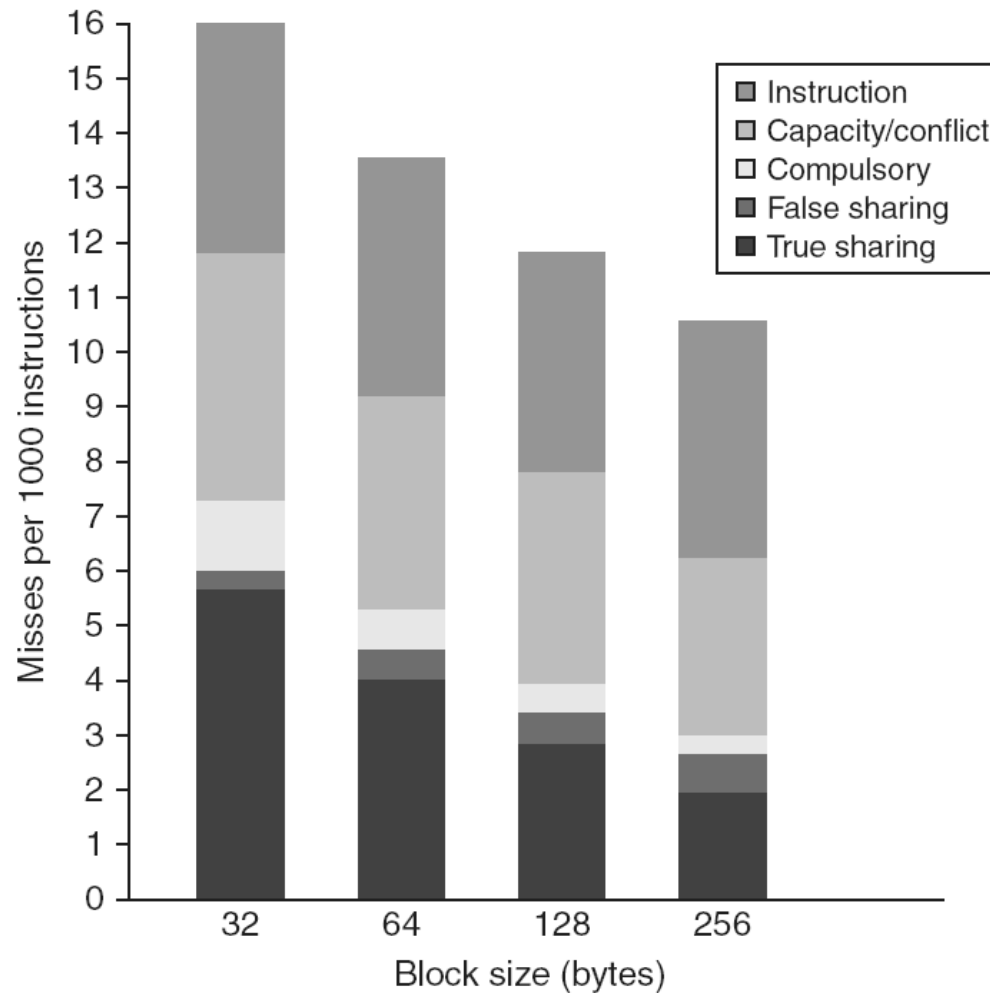
Performance Study: Commercial Workload



Performance Study: Commercial Workload

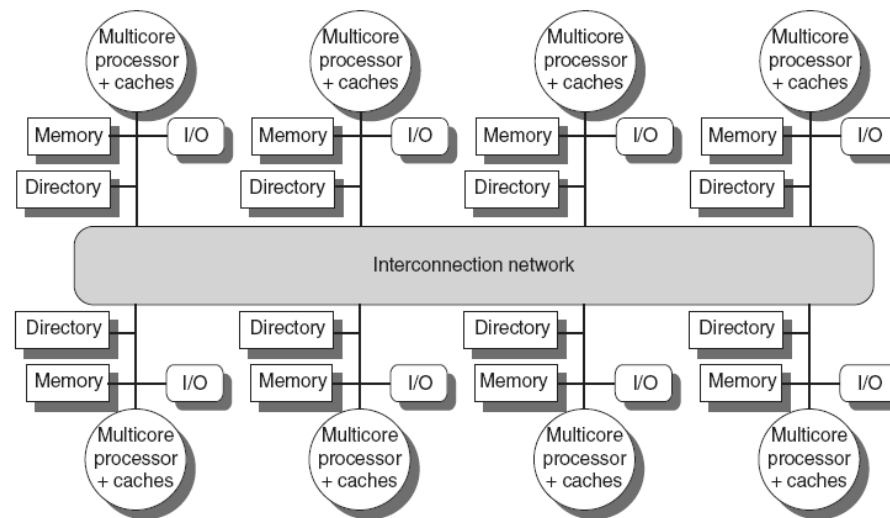


Performance Study: Commercial Workload



Directory Protocols

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



Directory Protocols

- For each block, maintain state:
 - Shared
 - One or more nodes have the block cached, value in memory is up-to-date
 - Set of node IDs
 - Uncached
 - Modified
 - Exactly one node has a copy of the cache block, value in memory is out-of-date
 - Owner node ID
- Directory maintains block states and sends invalidation messages

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

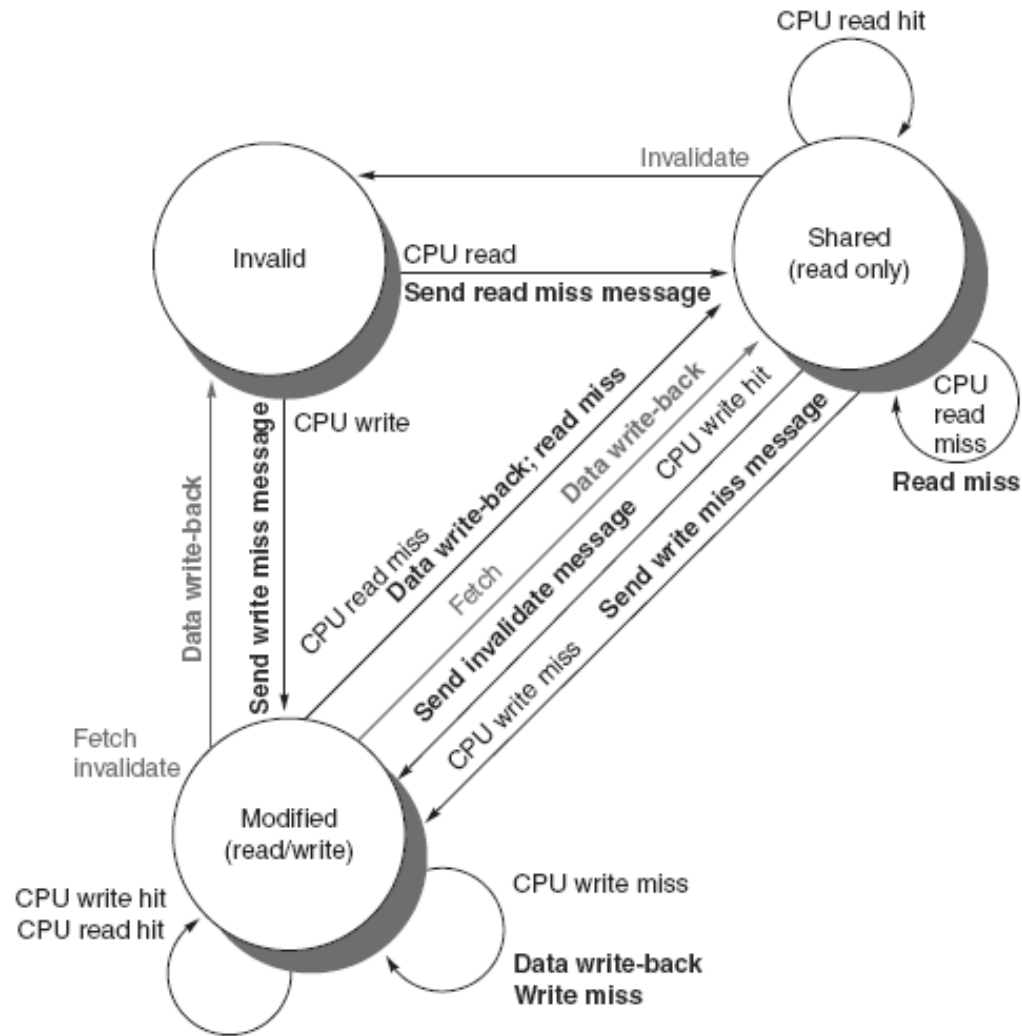
Directory Protocols

- For modified 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 uncached, 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

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



Synchronization

- Basic building blocks:
 - 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

Synchronization

- Basic building blocks:
 - Atomic exchange
 - Swaps register with memory location

```
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
```

Synchronization

- Basic building blocks:
 - Fetch-and-increment
 - Reads original value from memory and increments it in memory

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

Implementing Locks

- Spin lock

- If no coherence:

```

lockit:      DADDUI      R2,R0,#1
             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

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

Implementing Locks

- In summary:
 - Step 1: P0 starts with the lock
 - Step 2: P0 unlocks the lock
 - Steps 3 to 5: P1 and P2 race to see which reads the unlocked value
 - Steps 6,7: P2 wins and enters the critical section
 - Steps 7,8: P1's attempt fails, spins again

Models of Memory Consistency

<u>Processor 1:</u>	<u>Processor 2:</u>
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

Models of Memory Consistency

- Sequential consistency (Lamport 1979):
 - *the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program*

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:
 - $R \rightarrow W, R \rightarrow R, W \rightarrow R, W \rightarrow W$
 - Relax $W \rightarrow R$
 - “Total store ordering”
 - Relax $W \rightarrow W$
 - “Partial store order”
 - 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