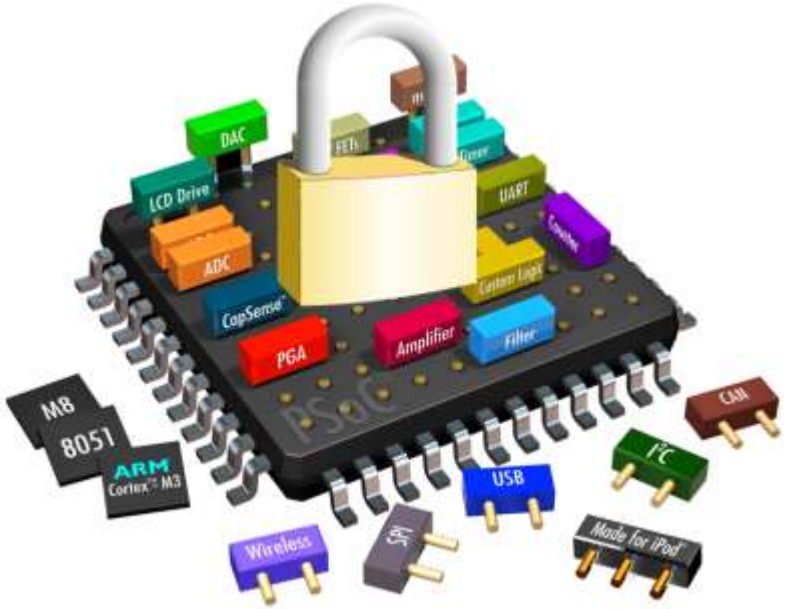
Advanced Computer Architecture

Thread-Level Parallelism



Shift to Multiprocessing

> "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, Intel Developer Forum (2005)

Historical Context

While some researchers predicted the end of uniprocessor advances as early as the 1960s, the period of 1986-2003 actually saw the highest performance growth since the first transistorized computers.

The Inflection Point

By 2005, the industry recognized that multiprocessors would play a major role from low-end to high-end computing, marking a clear shift in architectural focus.

This transition wasn't just a technical choice—it represented a fundamental rethinking of how to advance computing performance in the face of physical and practical limitations.

Why Multiprocessing Became Essential

Diminishing Returns

Attempts to extract more instruction-level parallelism (ILP) led to dramatically lower efficiencies in silicon and energy use between 2000-2005, as power and silicon costs grew faster than performance.

Cloud Computing Growth

Increasing interest in high-end servers as cloud computing and software-as-a-service became more important to the computing landscape.

Data-Intensive Applications

Growth in applications driven by the availability of massive amounts of data on the Internet requiring parallel processing capabilities.

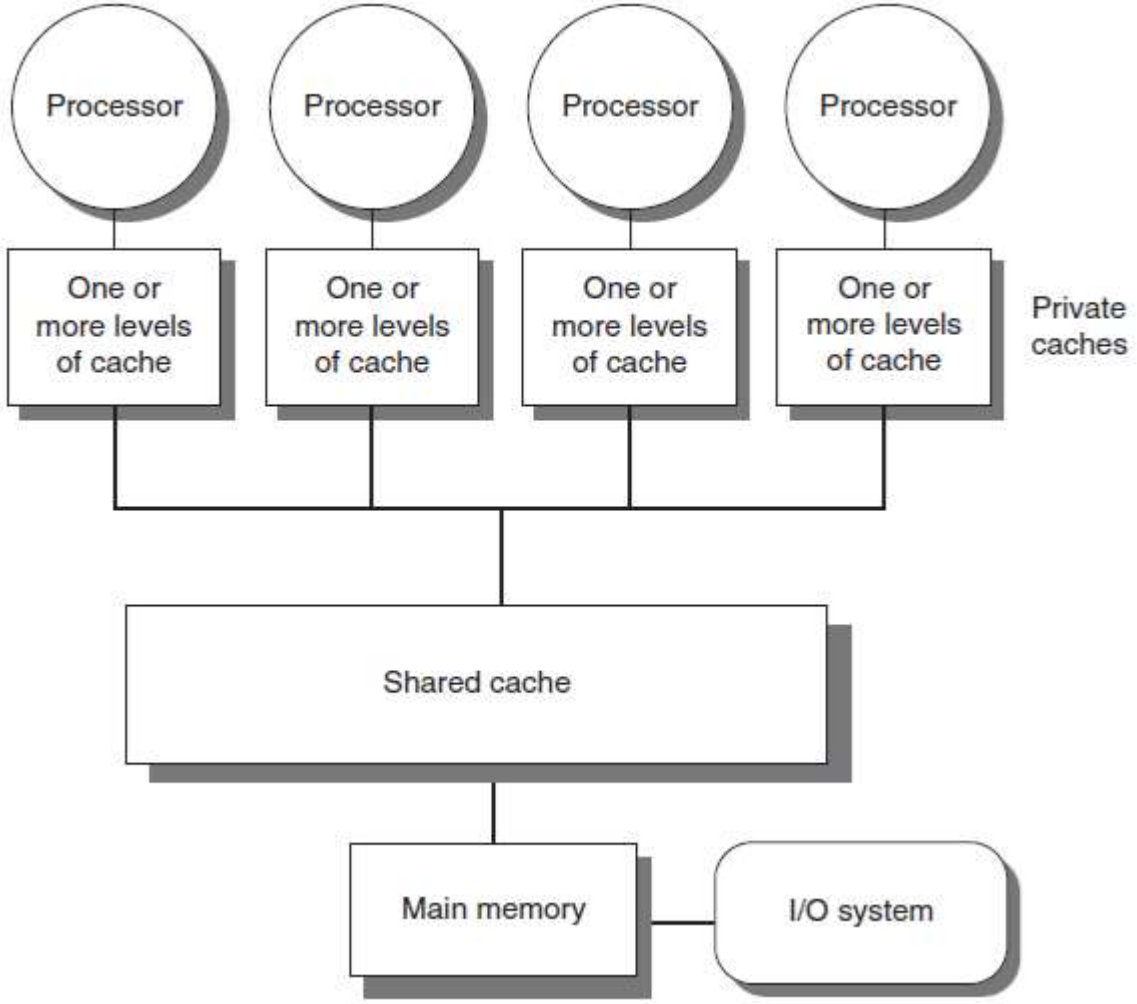
Desktop Performance Plateau

Recognition that increasing desktop performance was less important (outside of graphics), as current performance became acceptable or compute-intensive tasks moved to the cloud.

Additional Drivers of Multiprocessing

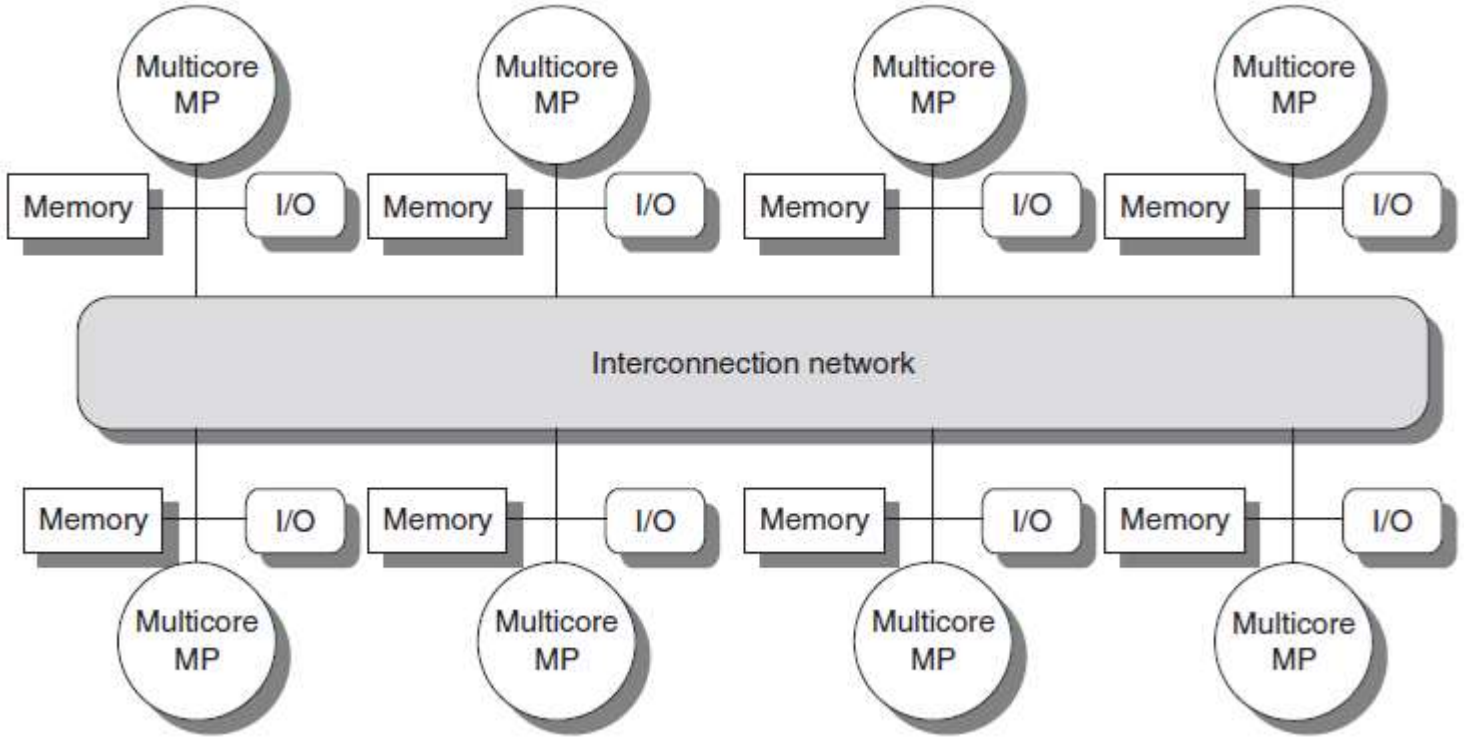
- Improved Understanding
  - Better knowledge of how to use multiprocessors effectively, especially in server environments with significant natural parallelism from large datasets or independent requests.
- Design Leverage
  - The advantages of leveraging a design investment by replication rather than unique design; all multiprocessor designs provide such leverage.
- These factors collectively pushed the industry toward thread-level parallelism as the primary means of performance scaling, fundamentally changing how processors are designed and programmed.

Symmetric Multiprocessors (SMPs)



- Small number of cores (typically ≤8)
- Centralized shared memory with equal access
- Also called Uniform Memory Access (UMA)
- All existing multicores are SMPs

Distributed Shared Memory (DSM)



- Physically distributed memory among processors
- Also called Non-Uniform Memory Access (NUMA)
- Required for larger processor counts
- Multi-chip multiprocessors use distributed memory

Challenge #1: Limited Parallelism

80x	0.25%	99.75%
Target Speedup	**Sequential Limit**	**Parallel Requirement**
With 100 processors	Maximum sequential portion allowed	Portion that must be parallel

- Amdahl's Law shows that even a small sequential portion of a program severely limits potential speedup. To achieve 80x speedup with 100 processors, only 0.25% of the original computation can be sequential.
- This fundamental limitation makes it difficult to achieve good speedups in any parallel processor, regardless of architecture.

Challenge #2: Communication Latency

1

Latency Costs

Communication between cores may cost 35-50 clock cycles, and between chips 100-500+ cycles depending on the mechanism and scale.

2

Performance Impact

Even with just 0.2% of instructions requiring remote communication, performance can drop by 3.4x compared to all-local references.

3

Contention Effects

In practice, contention for the interconnect can make remote access latency even worse than the base values.

These long-latency remote communications represent one of the biggest performance challenges in multiprocessor systems, requiring both architectural and software solutions.

Cache Coherence

- Cache coherence ensures that multiple processors have a consistent view of memory despite having private caches. The problem arises because each processor's view of memory is through its own cache.

Time	Event	Cache contents for processor A on X	Cache contents for processor B on X	Memory contents on X
0	Processor A write 1 to X	1		1
1	Processor A reads 1 from X	1		1
2	Processor B reads 1 from X	1	1	1
3	Processor B writes 0 to X	1	0	0
4	Processor A writes 1 to X	1	0	1

Cache Coherence Requirements

- Read-after-Write Consistency
  - A read by a processor to a location it previously wrote must return the value it wrote, assuming no other processor wrote to that location in between.
- Write Propagation
  - A read to a location that follows a write by another processor must return the written value if sufficient time has passed and no other writes occurred.
- Write Serialization
  - Writes to the same location must be seen in the same order by all processors. This prevents processors from seeing different sequences of values.

Coherence vs. Consistency

Coherence

Defines what values can be returned by a read operation to a specific memory location.

- Focuses on a single memory location
- Ensures all processors see the same sequence of values
- Maintains the illusion of a single memory location despite multiple copies

Consistency

Determines when a written value will be visible to other processors.

- Concerns relationships between different memory locations
- Defines ordering rules for memory operations
- Affects how programmers reason about parallel code

Both properties are essential for writing correct shared-memory programs. Coherence is necessary but not sufficient for parallel correctness.

Cache Coherence Protocol

- Directory based
  - The sharing status of a particular block of physical memory is kept in one location, called the directory
  - Two types
    - One centralized directory in SMP (Symmetric Multiprocessors)
    - Distributed directories in DSM (Distributed Shared Memory)
- Snoop based
  - Every cache that has a copy of the data from a block of physical memory could track the sharing status of the block

Write Invalidate Protocol

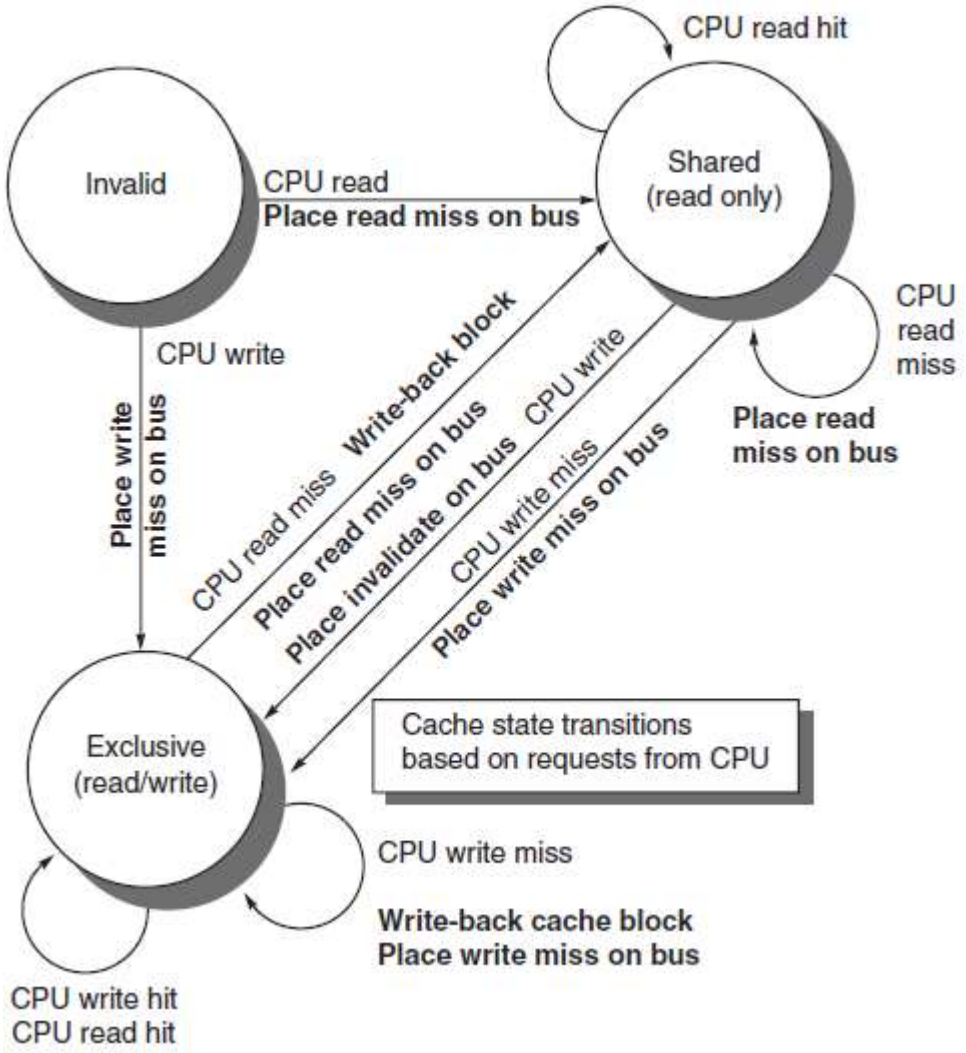
Processor activity	Bus activity	Processor A cache	Processor B cache	Memory location X
				0
Processor A reads X	Cache miss for X	0		0
Processor B reads X	Cache miss for X	0	0	0
Processor A writes 1 to X	Invalidation for X	1		0
Processor B reads X	Cache miss for X	1	1	1

- Alternative
  - Write update (or write broadcast)
  - Offers lower cache miss rate, but consumes considerably more bandwidth

MSI Protocol States

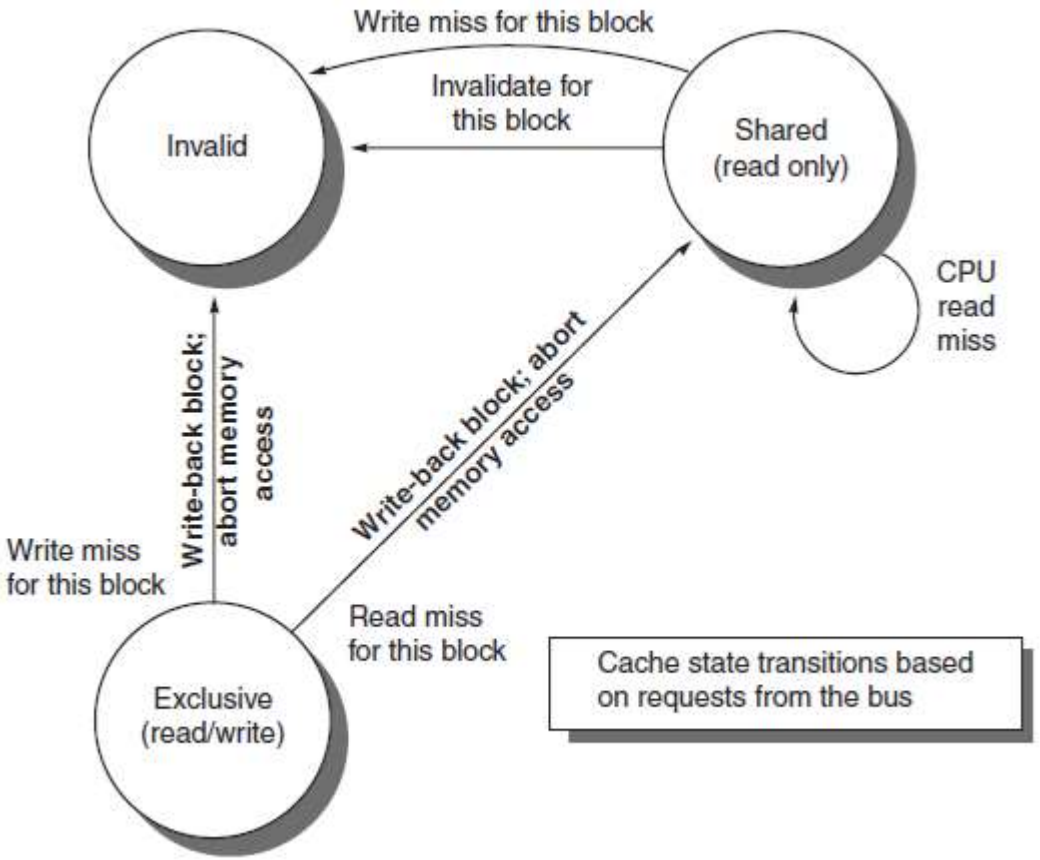
- Modified (M)
  - Block has been updated in this cache and is inconsistent with memory.
  - This cache has exclusive ownership and must supply data on a read miss.
- Shared (S)
  - Block is potentially shared with other caches.
  - The copy is clean (matches memory).
  - A write requires gaining exclusive access first.
- Invalid (I)
  - Block is not present in the cache or has been invalidated.
  - A read or write will cause a cache miss.

MSI Transitions Requested by CPU



Request	State	Cache action	Explanation
Read hit	S or M	Normal hit	Read data in local cache.
Read miss	I	Normal miss	Place read miss on bus.
Read miss	S	Replacement	Address conflict miss: place read miss on bus.
Read miss	M	Replacement	Address conflict miss: place read miss on bus.
Write hit	S	Coherence	Place invalidate on bus.
Write miss	I	Normal miss	Place write miss on bus.
Write miss	S	Replacement	Address conflict miss: place write miss on bus.
Write miss	M	Replacement	Address conflict miss: write-back block, then place write miss on bus.

MSI Transitions Requested by the Bus



Request	State	Cache action	Explanation
Read miss	S	No action	Allow shared cache or memory to service read miss.
Read miss	M	Coherence	Attempt to share data: place cache block on bus and change state to shared.
Invalidate	S	Coherence	Attempt to write shared block; invalidate the block.
Write miss	S	Coherence	Attempt to write shared block; invalidate the block.
Write miss	M	Coherence	Attempt to write block that is exclusive elsewhere; write-back the cache block and make its state invalid in the local cache.

Extensions

- MESI (additional Exclusive state)
  - If a block is in the E state, it can be written without generating any invalidates, which optimizes the case where a block is read by a single cache before being written by that same cache.
- MOESI (additional Owner state)
  - In MSI and MESI protocols, when there is an attempt to share a block in the Modified state, the state is changed to Shared (in both the original and newly sharing cache), and the block must be written back to memory.
  - In a MOESI protocol, the block can be changed from the Modified to Owned state in the original cache without writing it to memory.

Limitations of Snooping Protocols

Bandwidth Bottleneck

Shared bus becomes a bottleneck as processor count or speed increases

Limited Scalability

Practical limit of 4-8 high-performance cores with symmetric access

Broadcast Overhead

Every cache must process every coherence request

Directory Solution

Moving to directory-based protocols eliminates broadcast requirement

These limitations have led designers to adopt directory protocols for larger systems and hybrid approaches for intermediate-sized systems.

MSI Protocol Example

- P0: read 120
- P0: write 120 <-- 80
- P2: write 120 <-- 80
- P1: read 110
- P0: write 108 <-- 48
- P0: write 130 <-- 78
- P2: write 130 <-- 78

P0

	State	Addr	Data
B0	I	100	00 10
B1	S	108	00 08
B2	M	110	00 30
B3	I	118	00 10

P1

	State	Addr	Data
B0	I	100	00 10
B1	M	128	00 68
B2	I	110	00 10
B3	S	118	00 10

P2

	State	Addr	Data
B0	S	120	00 20
B1	S	108	00 08
B2	I	110	00 10
B3	I	118	00 10

Memory

Addr	Data
100	00 10
108	00 08
110	00 10
118	00 18
120	00 20
128	00 28
130	00 30

MSI Protocol Example

- P0: read 120
- P0: write 120 <-- 80
- P2: write 120 <-- 80
- P1: read 110
- P0: write 108 <-- 48
- P0: write 130 <-- 78
- P2: write 130 <-- 78

P0

	State	Addr	Data
B0	I→S	120	00 20
B1	S	108	00 08
B2	M	110	00 30
B3	I	118	00 10

P1

	State	Addr	Data
B0	I	100	00 10
B1	M	128	00 68
B2	I	110	00 10
B3	S	118	00 10

P2

	State	Addr	Data
B0	S	120	00 20
B1	S	108	00 08
B2	I	110	00 10
B3	I	118	00 10

Memory

Addr	Data
100	00 10
108	00 08
110	00 10
118	00 18
120	00 20
128	00 28
130	00 30

Place read miss on bus

MSI Protocol Example

- P0: read 120
- P0: write 120 <-- 80
- P2: write 120 <-- 80
- P1: read 110
- P0: write 108 <-- 48
- P0: write 130 <-- 78
- P2: write 130 <-- 78

P0

	State	Addr	Data
B0	S→M	120	00 80
B1	S	108	00 08
B2	M	110	00 30
B3	I	118	00 10

P1

	State	Addr	Data
B0	I	100	00 10
B1	M	128	00 68
B2	I	110	00 10
B3	S	118	00 10

P2

	State	Addr	Data
B0	S→I	120	00 20
B1	S	108	00 08
B2	I	110	00 10
B3	I	118	00 10

Place invalidate on bus

Memory

Addr	Data
100	00 10
108	00 08
110	00 10
118	00 18
120	00 20
128	00 28
130	00 30

MSI Protocol Example

- P0: read 120
- P0: write 120 <-- 80
- P2: write 120 <-- 80
- P1: read 110
- P0: write 108 <-- 48
- P0: write 130 <-- 78
- P2: write 130 <-- 78

P0

	State	Addr	Data
B0	M→I	120	00 80
B1	S	108	00 08
B2	M	110	00 30
B3	I	118	00 10

P1

	State	Addr	Data
B0	I	100	00 10
B1	M	128	00 68
B2	I	110	00 10
B3	S	118	00 10

P2

	State	Addr	Data
B0	I→M	120	00 80
B1	S	108	00 08
B2	I	110	00 10
B3	I	118	00 10

Memory

Addr	Data
100	00 10
108	00 08
110	00 10
118	00 18
120	00 80
128	00 28
130	00 30

Write-back block

Place write miss on bus

MSI Protocol Example

- P0: read 120
- P0: write 120 <-- 80
- P2: write 120 <-- 80
- P1: read 110
- P0: write 108 <-- 48
- P0: write 130 <-- 78
- P2: write 130 <-- 78

P0

	State	Addr	Data
B0	I	120	00 80
B1	S	108	00 08
B2	M→S	110	00 30
B3	I	118	00 10

P1

	State	Addr	Data
B0	I	100	00 10
B1	M	128	00 68
B2	I→S	110	00 30
B3	S	118	00 10

P2

	State	Addr	Data
B0	M	120	00 80
B1	S	108	00 08
B2	I	110	00 10
B3	I	118	00 10

Memory

Addr	Data
100	00 10
108	00 08
110	00 30
118	00 18
120	00 80
128	00 28
130	00 30

Write-back block

Place read miss on bus

MSI Protocol Example

- P0: read 120
- P0: write 120 <-- 80
- P2: write 120 <-- 80
- P1: read 110
- P0: write 108 <-- 48
- P0: write 130 <-- 78
- P2: write 130 <-- 78

P0

	State	Addr	Data
B0	I	120	00 80
B1	S→M	108	00 48
B2	S	110	00 30
B3	I	118	00 10

P1

	State	Addr	Data
B0	I	100	00 10
B1	M	128	00 68
B2	S	110	00 30
B3	S	118	00 10

P2

	State	Addr	Data
B0	M	120	00 80
B1	S→I	108	00 08
B2	I	110	00 10
B3	I	118	00 10

Place invalidate on bus

Memory

Addr	Data
100	00 10
108	00 08
110	00 30
118	00 18
120	00 80
128	00 28
130	00 30

MSI Protocol Example

- P0: read 120
- P0: write 120 <-- 80
- P2: write 120 <-- 80
- P1: read 110
- P0: write 108 <-- 48
- P0: write 130 <-- 78
- P2: write 130 <-- 78

P0

	State	Addr	Data
B0	I→M	130	00 78
B1	M	108	00 48
B2	S	110	00 30
B3	I	118	00 10

P1

	State	Addr	Data
B0	I	100	00 10
B1	M	128	00 68
B2	S	110	00 30
B3	S	118	00 10

P2

	State	Addr	Data
B0	M	120	00 80
B1	I	108	00 08
B2	I	110	00 10
B3	I	118	00 10

Place write miss on bus

Memory

Addr	Data
100	00 10
108	00 08
110	00 30
118	00 18
120	00 80
128	00 28
130	00 30

MSI Protocol Example

- P0: read 120
- P0: write 120 <-- 80
- P2: write 120 <-- 80
- P1: read 110
- P0: write 108 <-- 48
- P0: write 130 <-- 78
- P2: write 130 <-- 78

P0

	State	Addr	Data
B0	M→I	130	00 78
B1	M	108	00 48
B2	S	110	00 30
B3	I	118	00 10

P1

	State	Addr	Data
B0	I	100	00 10
B1	M	128	00 68
B2	S	110	00 30
B3	S	118	00 10

P2

	State	Addr	Data
B0	M	120	00 80
B1	I→M	130	00 78
B2	I	110	00 10
B3	I	118	00 10

Memory

Addr	Data
100	00 10
108	00 08
110	00 30
118	00 18
120	00 80
128	00 28
130	00 78

Write-back block

Place write miss on bus

Understanding Cache Coherence Misses

True Sharing Misses

Arise from actual communication of data through cache coherence

- First write by a processor to a shared block causes invalidation
- When another processor reads a modified word, a miss occurs

False Sharing Misses

Arise from invalidation-based coherence with single valid bit per block

- Occurs when a block is invalidated because some word in the block, other than the one being read, is written
- Would not occur if block size were a single word

These two types of coherence misses combine with the traditional "three C's" (compulsory, capacity, and conflict) to determine overall multiprocessor cache performance.

Example: Identifying Sharing Misses

- Assume words x1 and x2 are in the same cache block, which is in the shared state in the caches of both P1 and P2.

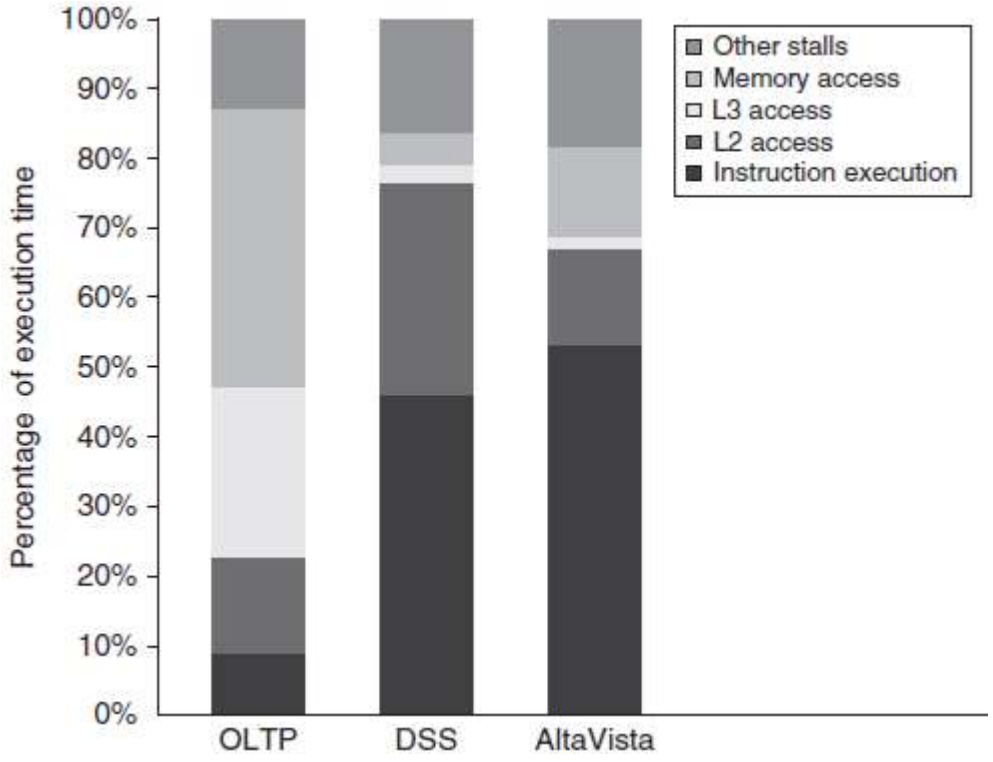
Time	P1	P2	Classification
1	Write x1		True sharing miss (x1 was read by P2)
2		Read x2	False sharing miss (x2 invalidated by write of x1)
3	Write x1		False sharing miss (block shared due to read in P2)
4		Write x2	False sharing miss (same reason as step 3)
5	Read x2		True sharing miss (value being read was written by P2)

Commercial Workload Study

Benchmark	% Time user mode	% Time kernel	% Time idle
OLTP (online transaction-processing)	71	18	11
DSS (decision support system)	87	4	9
AltaVista (web search)	>98	<1	<1

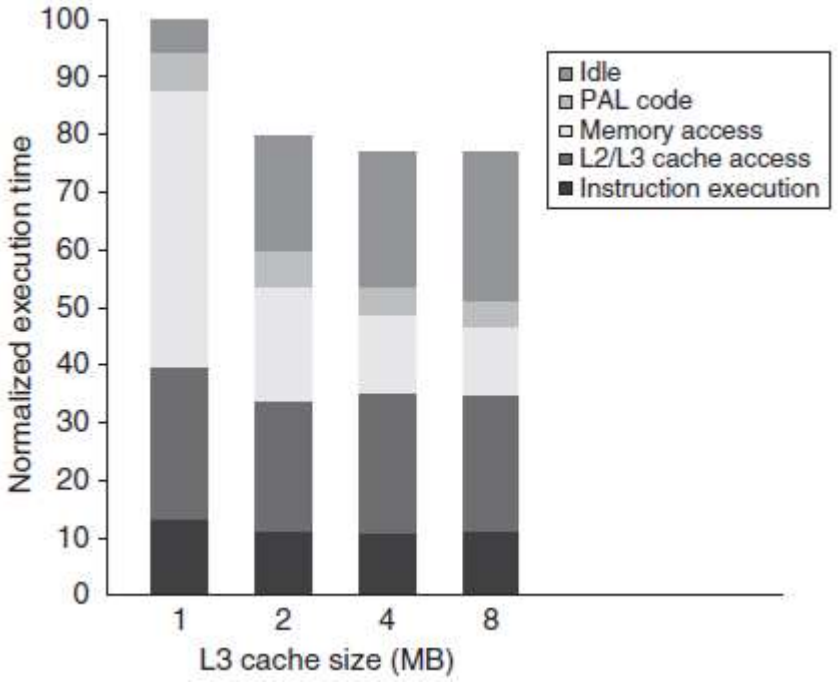
Level	Characteristic	Alpha 21164	Intel i7
L1	Size	8 KB I / 8 KB D	32 KB I / 32 KB D
	Associativity	Direct mapped	4-way I / 8-way D
	Block size	32 B	64 B
	Miss penalty	7	10
L2	Size	96 KB	256 KB
	Associativity	3-way	8-way
	Block size	32 B	64 B
	Miss penalty	21	35
L3	Size	2 MB	2 MB per core
	Associativity	Direct mapped	16-way
	Block size	64 B	64 B
	Miss penalty	80	~100

Execution Time of Workloads

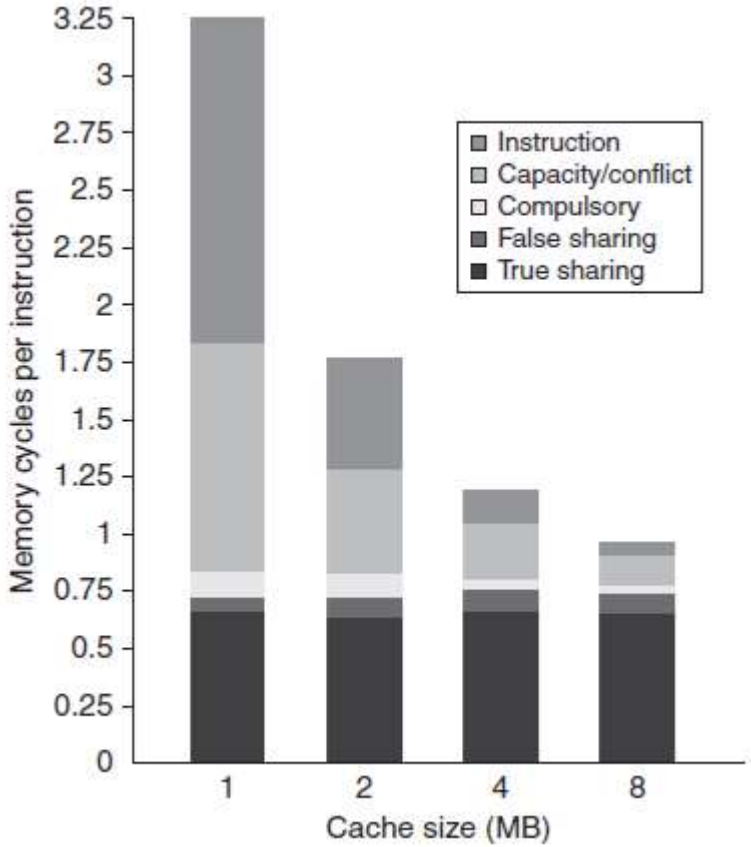


- CPI
  - 7.0 for OLTP
  - 1.6 for DSS
  - 1.3 for AltaVista
- The performance of the OLTP workload is very poor, due to a poor performance of the memory hierarchy
  → High L3 miss penalty

Impact of L3 Cache Size

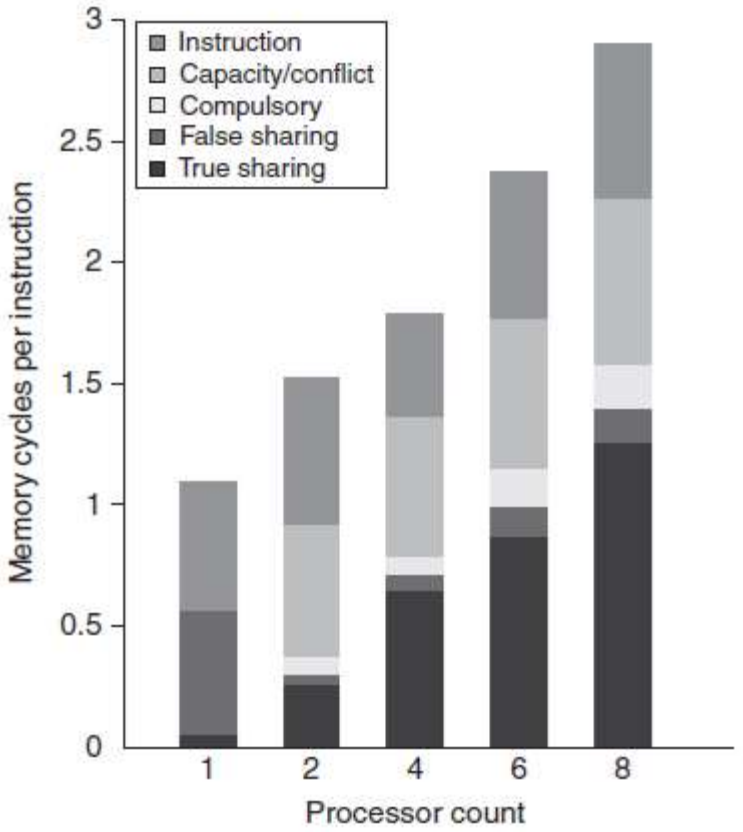


Most performance gain occurs going from 1MB to 2MB L3 cache, with diminishing returns beyond that point.

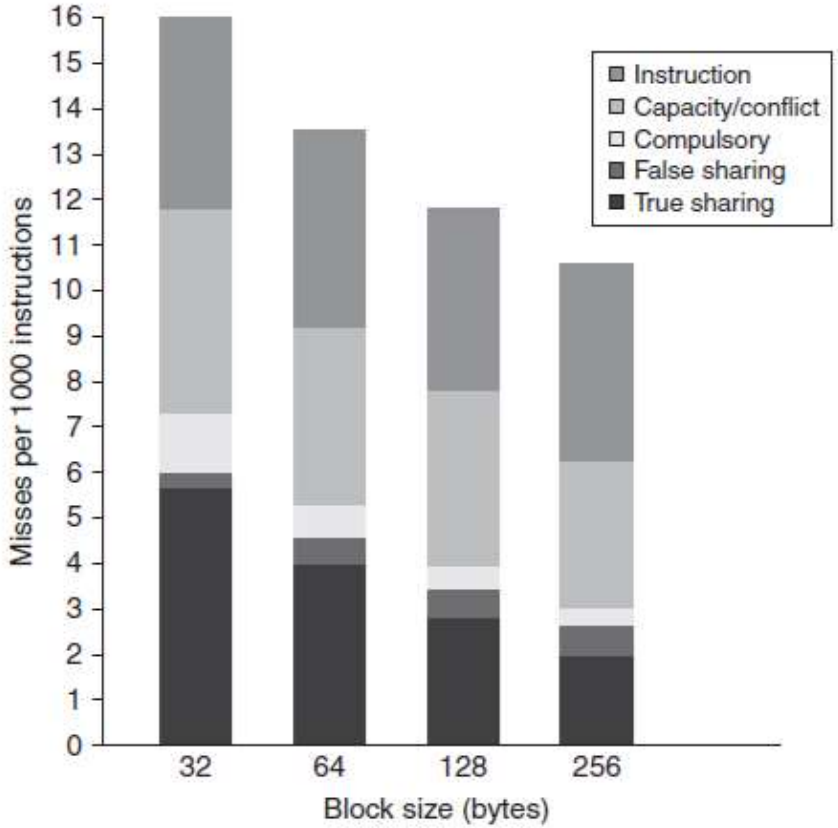


As L3 cache grows, instruction and capacity/conflict misses decrease significantly, but compulsory, false sharing, and true sharing misses remain largely unaffected.

Impact of Processor Count and Block Size



Increasing processor count leads to higher memory access cycles primarily due to increased true sharing misses.



Increasing block size from 32 to 256 bytes significantly reduces true sharing and compulsory misses, with modest impact on capacity/conflict misses.

Multiprogramming and OS Workload

- Multi-programmed workload
  - Two independent copies of the compile phases of the Andrew benchmark, a benchmark that emulates a software development environment
- Memory system

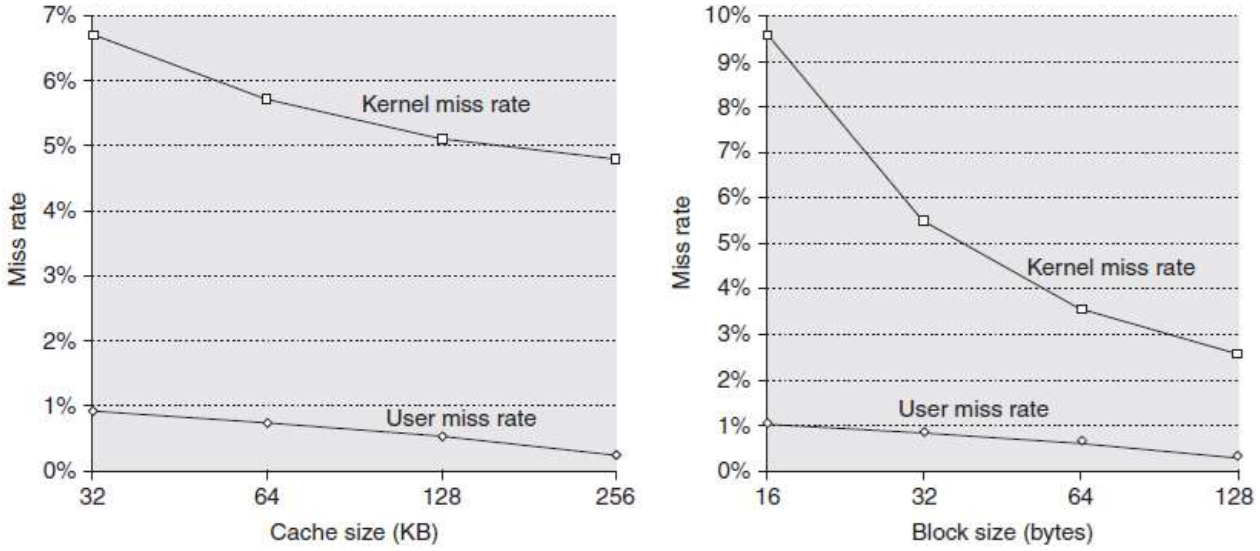
Component	Hit time (cycle)	Size
L1 instruction cache	1	32 KB, 2-way, 64-B block size
L1 data cache	1	32 KB, 2-way, 32-B block size
L2 cache	10	1 MB, unified, 2-way, 128-B block size
Main memory	100	
Disk	3 ms	

Workload Profile

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

- This multiprogramming workload has a significant instruction cache performance loss, at least for the OS
- The instruction cache miss rate in the OS 1.7% to 0.2%
- User-level instruction cache misses are roughly one-sixth of the OS rate

L1 Data Cache Miss Rate



Increasing cache size affects user miss rate more than kernel miss rate. Increasing block size benefits both, but improves kernel miss rate more significantly.

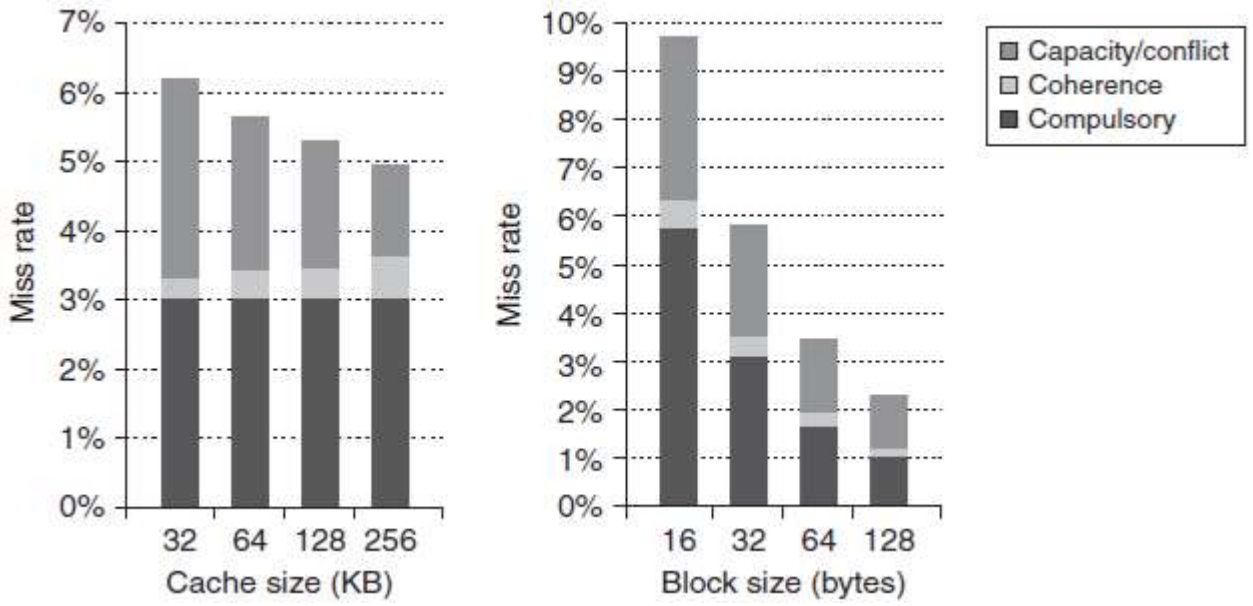
Why OS Behaves Differently

- Kernel initializes all pages before allocation, increasing compulsory misses
- Kernel actually shares data, causing coherence misses

- User processes cause coherence misses only when scheduled on different processors

The OS is a much more demanding user of the memory system than user applications.

Kernel Data Miss Rate Breakdown



- For the kernel references, increasing the cache size reduces only the uniprocessor capacity/conflict miss rate
- In contrast, increasing the block size causes a reduction in the compulsory miss rate
- The absence of large increases in the coherence miss rate as block size is increased means that false sharing effects are probably insignificant, although such misses may be offsetting some of the gains from reducing the true sharing misses.

Challenges

- For the multi-programmed workload, the OS is a much more demanding user of the memory system
- It will become very difficult to build a sufficiently capable memory system
- One possible route to improving performance is to make the OS more cache aware, through either better programming environments or through programmer assistance
- OS and commercial workloads are less amenable to algorithmic or compiler restructuring
- As the number of cores increases predicting the behavior of such applications is likely to get more difficult

Scalability Challenge of Snooping Protocols

Snooping protocols require communication with all caches on every cache miss, including writes of potentially shared data. While simple and inexpensive, this becomes their Achilles' heel for scalability.

Bandwidth Requirements

A system of four 4-core multicores at 4 GHz could require between 4 GB/sec to 170 GB/sec of bus bandwidth - far beyond the capability of any bus-based system.

Multicore Impact

The development of multicore processors has forced designers to shift to distributed memory systems to support the bandwidth demands of individual processors.

Even with distributed memory (which separates local from remote traffic), we gain little unless we eliminate the need for the coherence protocol to broadcast on every cache miss.

Directory-Based Coherence

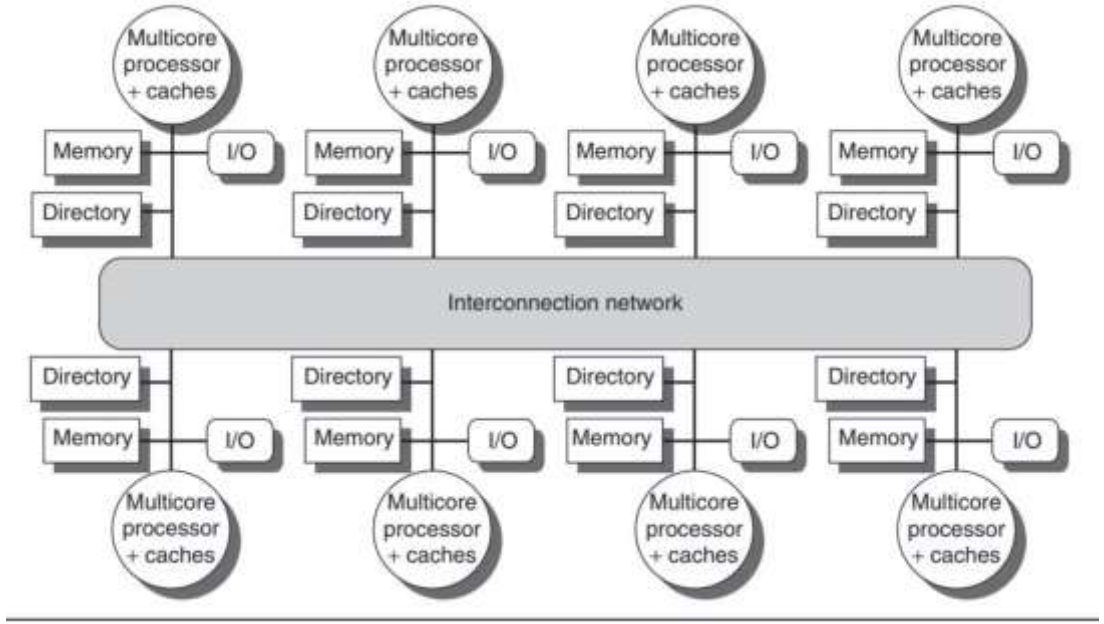


Figure: A directory is added to each node to implement cache coherence in a distributed-memory multiprocessor.

A directory keeps the state of every block that may be cached, including which caches have copies and whether blocks are dirty. This eliminates the need for broadcast messages.

Within a multicore with shared outermost cache (L3), implementation is simple: keep a bit vector equal to the number of cores for each L3 block, indicating which private caches may have copies. Invalidations are only sent to those caches.

For scalability across multiple chips, the directory must be distributed along with memory, allowing different coherence requests to go to different directories.

Directory Structure and Tracking

Directory Entry States

- Shared - One or more nodes have the block cached, memory is up to date
- Uncached - No node has a copy of the cache block
- Modified - Exactly one node has a copy and has written to it; memory is out of date

Tracking Mechanism

The simplest approach uses a bit vector for each memory block. When shared, each bit indicates whether the corresponding processor has a copy. When exclusive, it identifies the owner.

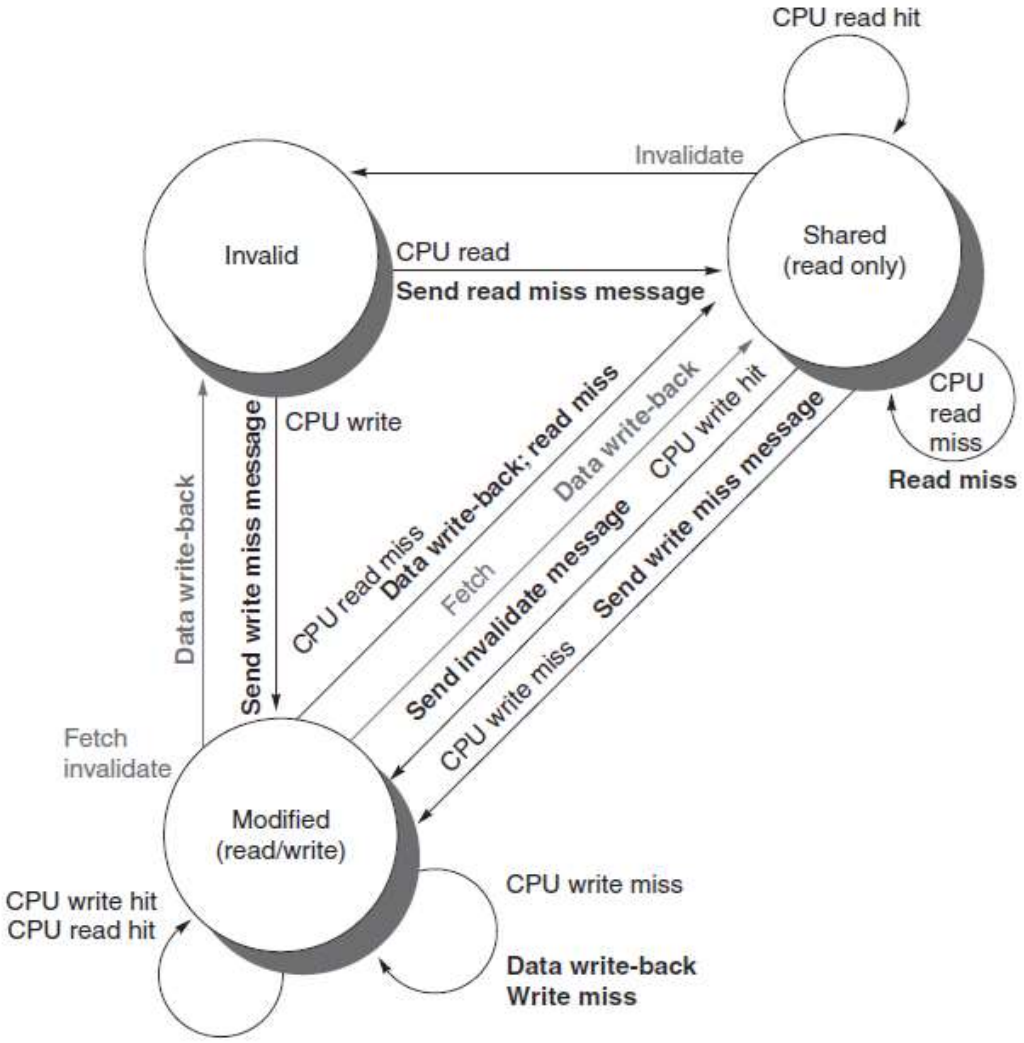
The directory size is proportional to the product of memory blocks times the number of nodes.

This overhead is tolerable for multiprocessors with less than a few hundred processors. Larger systems require more efficient scaling methods.

Message Types in Directory Protocols

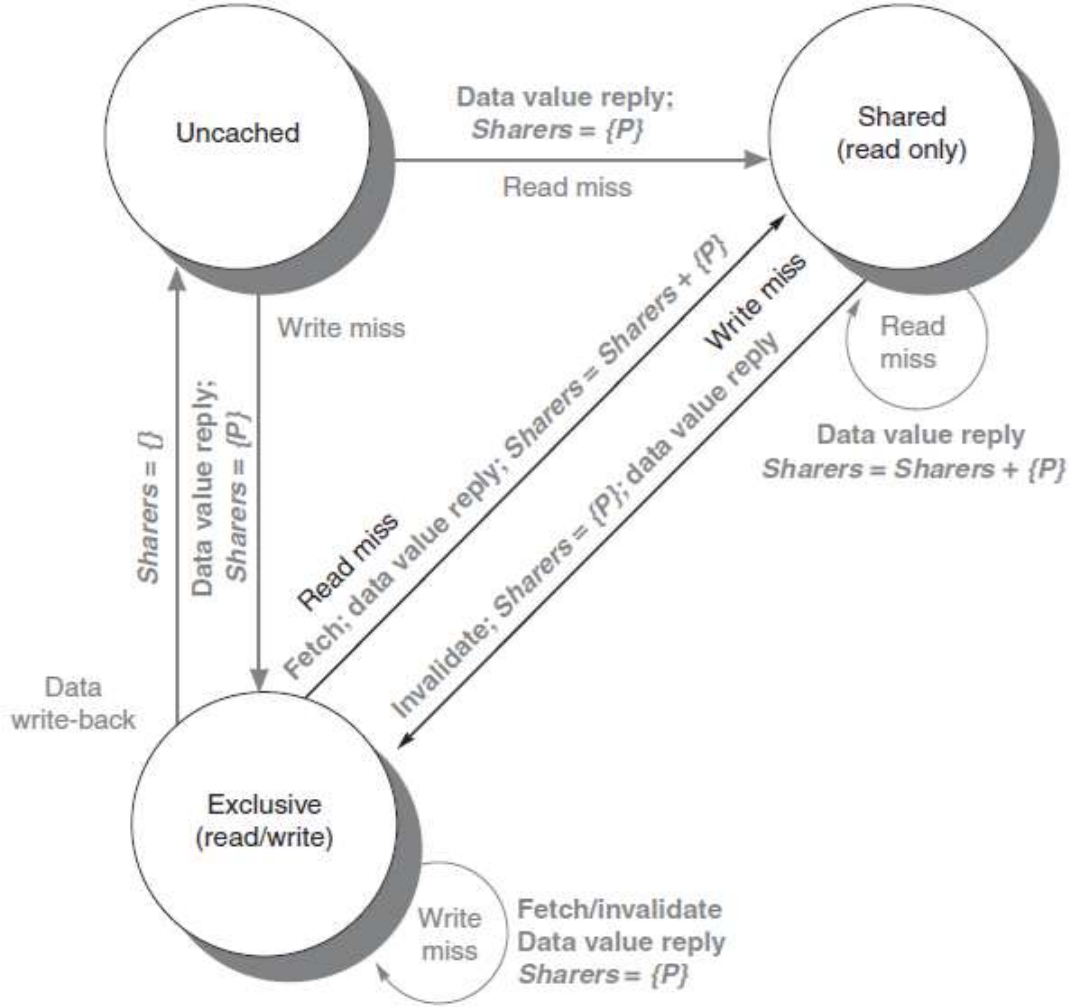
Message type	Source	Destination	Contents	Function
Read miss	Local cache	Home directory	P, A	Node P has a read miss at address A; request data and make P a read sharer
Write miss	Local cache	Home directory	P, A	Node P has a write miss at address A; request data and make P the exclusive owner
Invalidate	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

Cache State Transitions



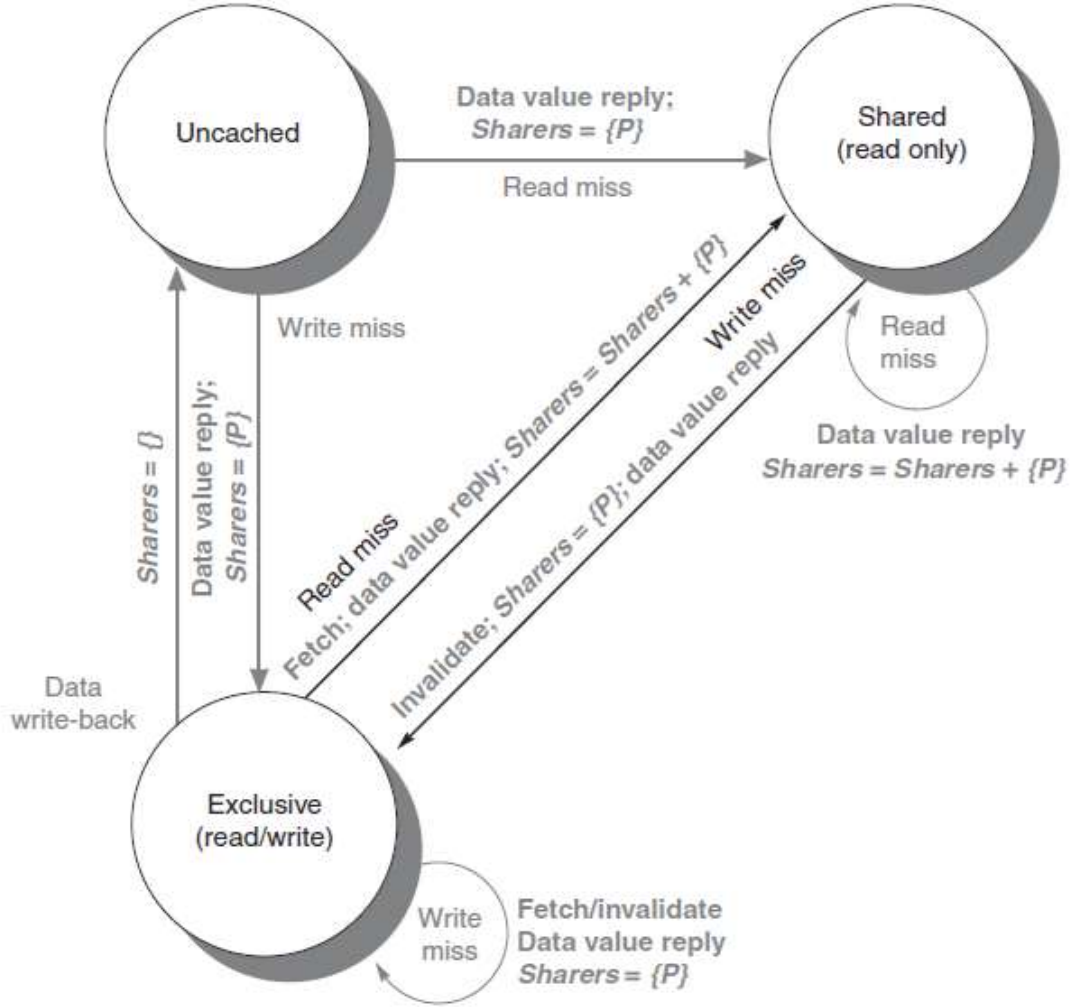
- The states for individual cache blocks are identical to those in snooping protocols: Invalid, Shared, and Exclusive/Modified. However, the transactions differ:
  - Local processor requests are shown in black
  - Directory requests are shown in gray
  - Broadcast write misses are replaced by targeted invalidate and fetch operations
- As with snooping, any cache block must be in exclusive state when written, and any shared block must be up to date in memory.

Directory State Transitions



- In a directory-based protocol, the directory implements the other half of the coherence protocol
- The directory receives three different requests: read miss, write miss, and data write-back

Directory Protocol Operations

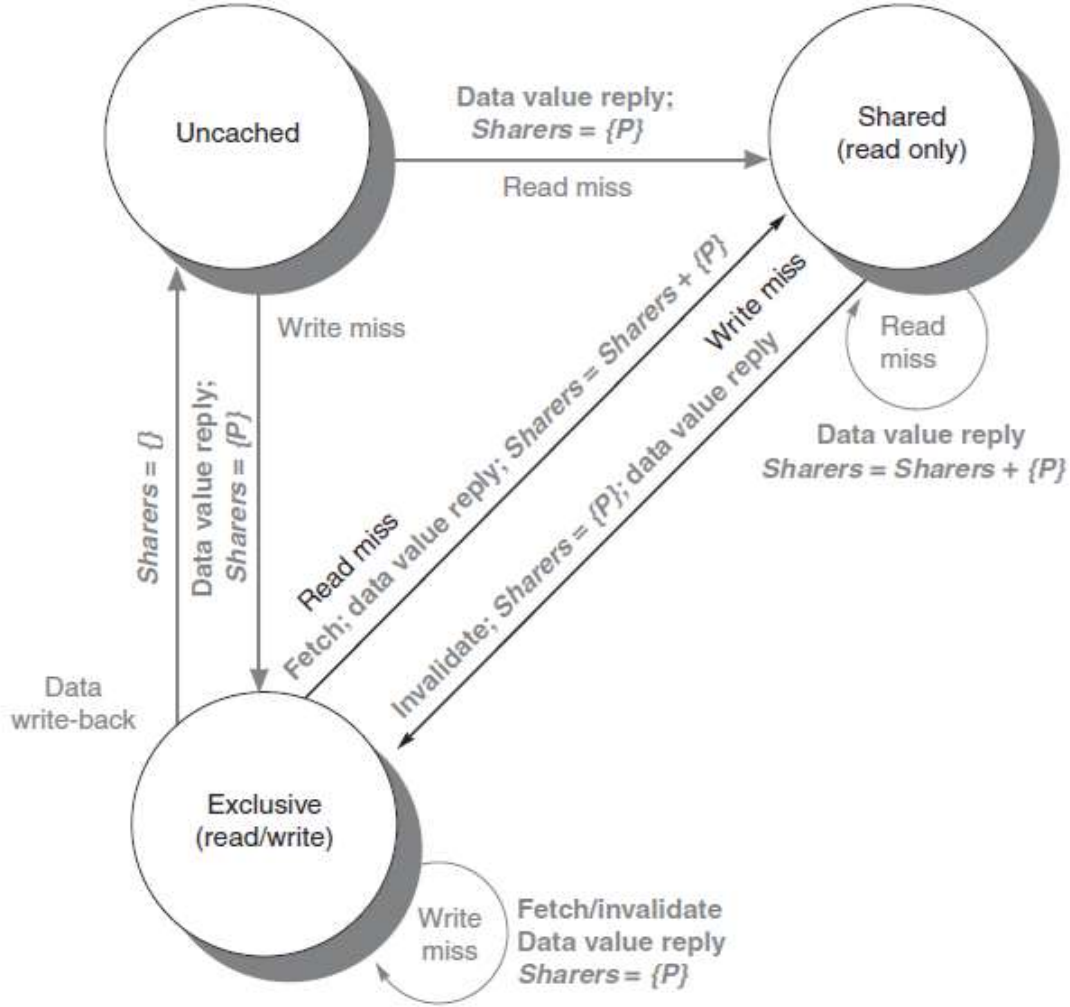


Uncached State

Read miss: Send data to requestor, add to sharers, change to Shared

Write miss: Send data to requestor, make requestor the owner, change to Exclusive

Directory Protocol Operations

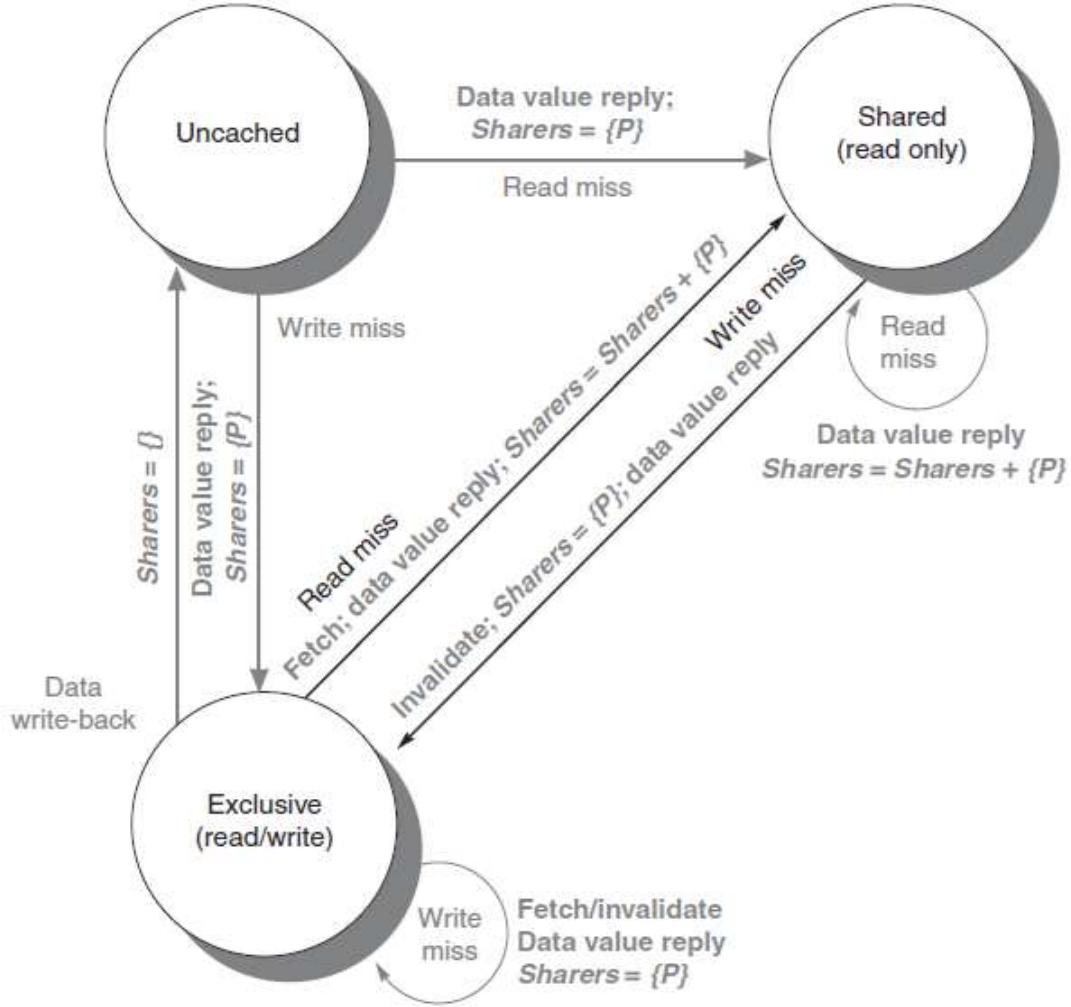


Shared State

Read miss: Send data to requestor, add to sharers

Write miss: Send data to requestor, invalidate all sharers, make requestor the owner, change to Exclusive

Directory Protocol Operations



Exclusive State

Read miss: Fetch data from owner, send to requestor, add both to sharers, change to Shared

Write miss: Fetch/invalidate from owner, send to requestor, make requestor the owner

Data write-back: Update memory, change to Uncached, clear sharers

Protocol Optimizations and Challenges

- Common optimizations
  - Direct forwarding from owner to requestor (bypassing home directory)
  - Reducing latency by avoiding memory updates on clean replacements
  - Specialized handling of upgrade requests (shared to exclusive)
- Implementation challenges
  - Handling non-atomic memory transactions
  - Avoiding deadlock conditions
  - Managing multiple message types
  - Ensuring forward progress

Directory Protocol Example

- P0: read 108
- P0: write 108 <-- 80
- P2: write 108 <-- 84
- P1: read 108
- P0: write 120 <-- 48
- P0: write 120 <-- 78
- P2: write 120 <-- 70

P0

Block	State	Addr	Data
B0	I	108	00 10
B1	S	120	00 10

P1

Block	State	Addr	Data
B0	I	108	00 20
B1	S	120	00 40

P2

Block	State	Addr	Data
B0	M	108	00 20
B1	S	120	00 10

Cache

Addr	State	Sharers
100	U	
108	M	P2

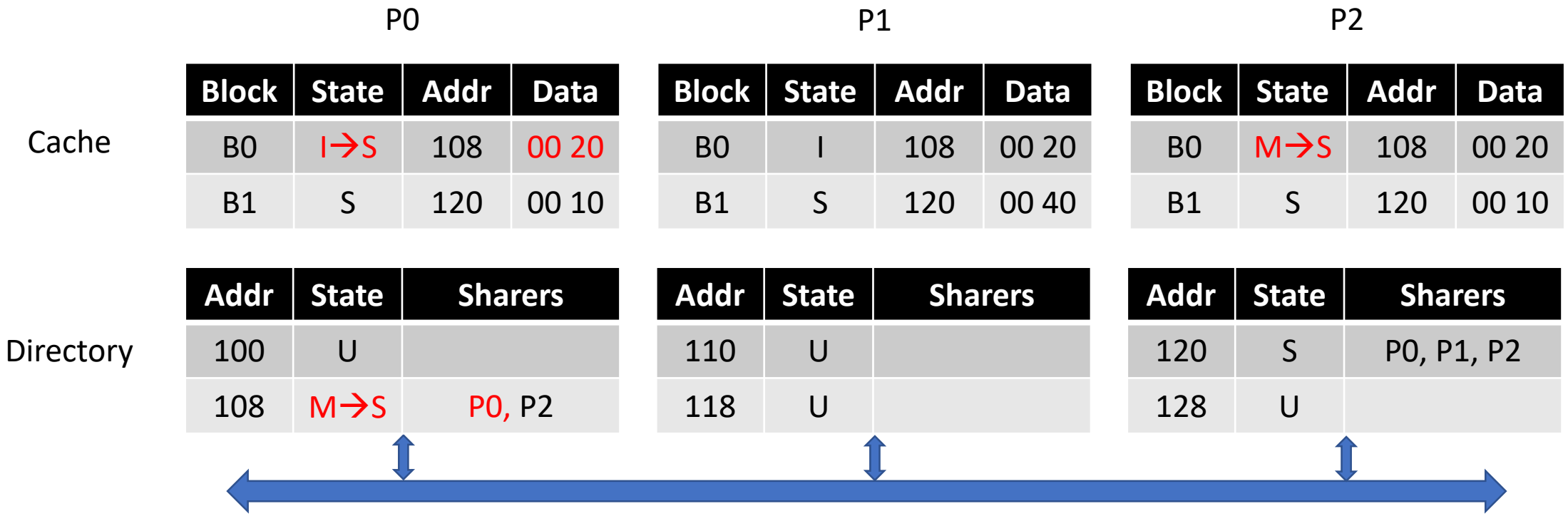
Addr	State	Sharers
110	U	
118	U	

Addr	State	Sharers
120	S	P0, P1, P2
128	U	

Directory

Directory Protocol Example

- P0: read 108
- P0: write 108 <-- 80
- P2: write 108 <-- 84
- P1: read 108
- P0: write 120 <-- 48
- P0: write 120 <-- 78
- P2: write 120 <-- 70



Directory Protocol Example

- P0: read 108
- P0: write 108 <-- 80
- P2: write 108 <-- 84
- P1: read 108
- P0: write 120 <-- 48
- P0: write 120 <-- 78
- P2: write 120 <-- 70

P0

Block	State	Addr	Data
B0	S→M	108	00 80
B1	S	120	00 10

P1

Block	State	Addr	Data
B0	I	108	00 20
B1	S	120	00 40

P2

Block	State	Addr	Data
B0	S→I	108	00 20
B1	S	120	00 10

Cache

Addr	State	Sharers
100	U	
108	S→M	P0

Addr	State	Sharers
110	U	
118	U	

Addr	State	Sharers
120	S	P0, P1, P2
128	U	

Directory

Directory Protocol Example

- P0: read 108
- P0: write 108 <-- 80
- P2: write 108 <-- 84
- P1: read 108
- P0: write 120 <-- 48
- P0: write 120 <-- 78
- P2: write 120 <-- 70

P0

Cache	Block	State	Addr	Data
	B0	M→I	108	00 80
	B1	S	120	00 10

Directory	Addr	State	Sharers
	100	U	
	108	M	P2

P1

Cache	Block	State	Addr	Data
	B0	I	108	00 20
	B1	S	120	00 40

Directory	Addr	State	Sharers
	110	U	
	118	U	

P2

Cache	Block	State	Addr	Data
	B0	I→M	108	00 84
	B1	S	120	00 10

Directory	Addr	State	Sharers
	120	S	P0, P1, P2
	128	U	

Directory Protocol Example

- P0: read 108
- P0: write 108 <-- 80
- P2: write 108 <-- 84
- P1: read 108
- P0: write 120 <-- 48
- P0: write 120 <-- 78
- P2: write 120 <-- 70

P0

Cache	Block	State	Addr	Data
	B0	I	108	00 80
	B1	S	120	00 10

Directory	Addr	State	Sharers
	100	U	
	108	M→S	P1, P2

P1

Cache	Block	State	Addr	Data
	B0	I→S	108	00 84
	B1	S	120	00 40

Directory	Addr	State	Sharers
	110	U	
	118	U	

P2

Cache	Block	State	Addr	Data
	B0	M→S	108	00 84
	B1	S	120	00 10

Directory	Addr	State	Sharers
	120	S	P0, P1, P2
	128	U	

Directory Protocol Example

- P0: read 108
- P0: write 108 <-- 80
- P2: write 108 <-- 84
- P1: read 108
- P0: write 120 <-- 48
- P0: write 120 <-- 78
- P2: write 120 <-- 70

P0

Block	State	Addr	Data
B0	I	108	00 80
B1	S→M	120	00 48

P1

Block	State	Addr	Data
B0	S	108	00 84
B1	S→I	120	00 40

P2

Block	State	Addr	Data
B0	S	108	00 84
B1	S→I	120	00 10

Cache

Addr	State	Sharers
100	U	
108	S	P1, P2

Addr	State	Sharers
110	U	
118	U	

Addr	State	Sharers
120	S→M	P0
128	U	

Directory

Directory Protocol Example

- P0: read 108
- P0: write 108 <-- 80
- P2: write 108 <-- 84
- P1: read 108
- P0: write 120 <-- 48
- P0: write 120 <-- 78
- P2: write 120 <-- 70

P0

Block	State	Addr	Data
B0	I	108	00 80
B1	M	120	00 78

P1

Block	State	Addr	Data
B0	S	108	00 84
B1	I	120	00 40

P2

Block	State	Addr	Data
B0	S	108	00 84
B1	I	120	00 10

Cache

Addr	State	Sharers
100	U	
108	S	P1, P2

Addr	State	Sharers
110	U	
118	U	

Addr	State	Sharers
120	M	P0
128	U	

Directory

Directory Protocol Example

- P0: read 108
- P0: write 108 <-- 80
- P2: write 108 <-- 84
- P1: read 108
- P0: write 120 <-- 48
- P0: write 120 <-- 78
- P2: write 120 <-- 70

P0

Cache

Block	State	Addr	Data
B0	I	108	00 80
B1	M→I	120	00 78

P1

Block	State	Addr	Data
B0	S	108	00 84
B1	I	120	00 40

P2

Block	State	Addr	Data
B0	S	108	00 84
B1	I→M	120	00 70

Directory

Addr	State	Sharers
100	U	
108	S	P1, P2

Addr	State	Sharers
110	U	
118	U	

Addr	State	Sharers
120	M	P2
128	U	

Synchronization

- Synchronization mechanisms are built with user-level software routines that rely on hardware-supplied synchronization instructions
  - For smaller multiprocessors or low-contention situations, the key hardware capability is an uninterruptible instruction or instruction sequence capable of atomically retrieving and changing a value.
  - In high-contention situations, synchronization can become a performance bottleneck because contention introduces additional delays and latency is potentially greater in such multiprocessors.
- Lock and unlock operations form the foundation for creating mutual exclusion and more complex synchronization mechanisms.

Hardware Primitives for Synchronization

Atomic Exchange

Interchanges register value with memory value atomically. Used to implement locks where 0 indicates free and 1 indicates unavailable.

Test-and-Set

Tests a value and sets it if the test passes. Can be used similarly to atomic exchange for synchronization.

Fetch-and-Increment

Returns memory location value and atomically increments it. Using 0 to indicate unclaimed, it functions like exchange.

These primitives provide the ability to atomically read and modify a location, with a way to tell if the operations were performed atomically. They are building blocks for user-level synchronization operations like locks and barriers.

Load Linked and Store Conditional

- An alternative to single atomic operations uses a pair of instructions: load linked (or load locked) and store conditional.
- The store conditional returns 1 if successful and 0 if the memory location was changed between the load linked and store conditional, or if a context switch occurred.

```
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
```

- This sequence implements an atomic exchange on the memory location specified by R1. If any processor modifies the value between LL and SC, the SC returns 0, causing the code to retry.

Atomic Fetch-and-Increment

- An advantage of the load linked/store conditional mechanism is that it can be used to build other synchronization primitives
- Only register-register instructions should be inserted between LL and SC to avoid deadlock and the instruction count should be minimal to reduce SC failure probability.
- An example of the atomic fetch-and-increment implementation:

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

Spin Lock with 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
```

Step	P0	P1	P2	State	Acitivity
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	Un-lock	(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

Implementation with LL/SC

- This example shows another advantage of the load linked/store conditional primitives: The read and write operations are explicitly separated.
- The load linked need not cause any bus traffic.
- This fact allows the following simple code sequence, which has the same characteristics as the optimized version using exchange (R1 has the address of the lock, the LL has replaced the LD, and the SC has replaced the EXCH):

```
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
```

- The first branch forms the spinning loop; the second branch resolves races when two processors see the lock available simultaneously.

Memory Consistency

- Cache coherence ensures multiple processors see a consistent view of memory, but how consistent must this view be?
- The key question: When must a processor see values updated by another processor?
- This determines what properties must be enforced among reads and writes to different locations by different processors.

Challenges

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

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

- Although the question of how consistent memory must be seems simple, it is remarkably complicated
- If writes always take immediate effect and are immediately seen by other processors, it will be impossible for both if statements (labeled L1 and L2) to evaluate their conditions as true.
- But suppose the write invalidate is delayed, and the processor is allowed to continue during this delay. Then, it is possible that both P1 and P2 have not seen the invalidations for B and A (respectively) before they attempt to read the values
- The question now is should this behavior be allowed, and, if so, under what conditions?

Sequential Consistency

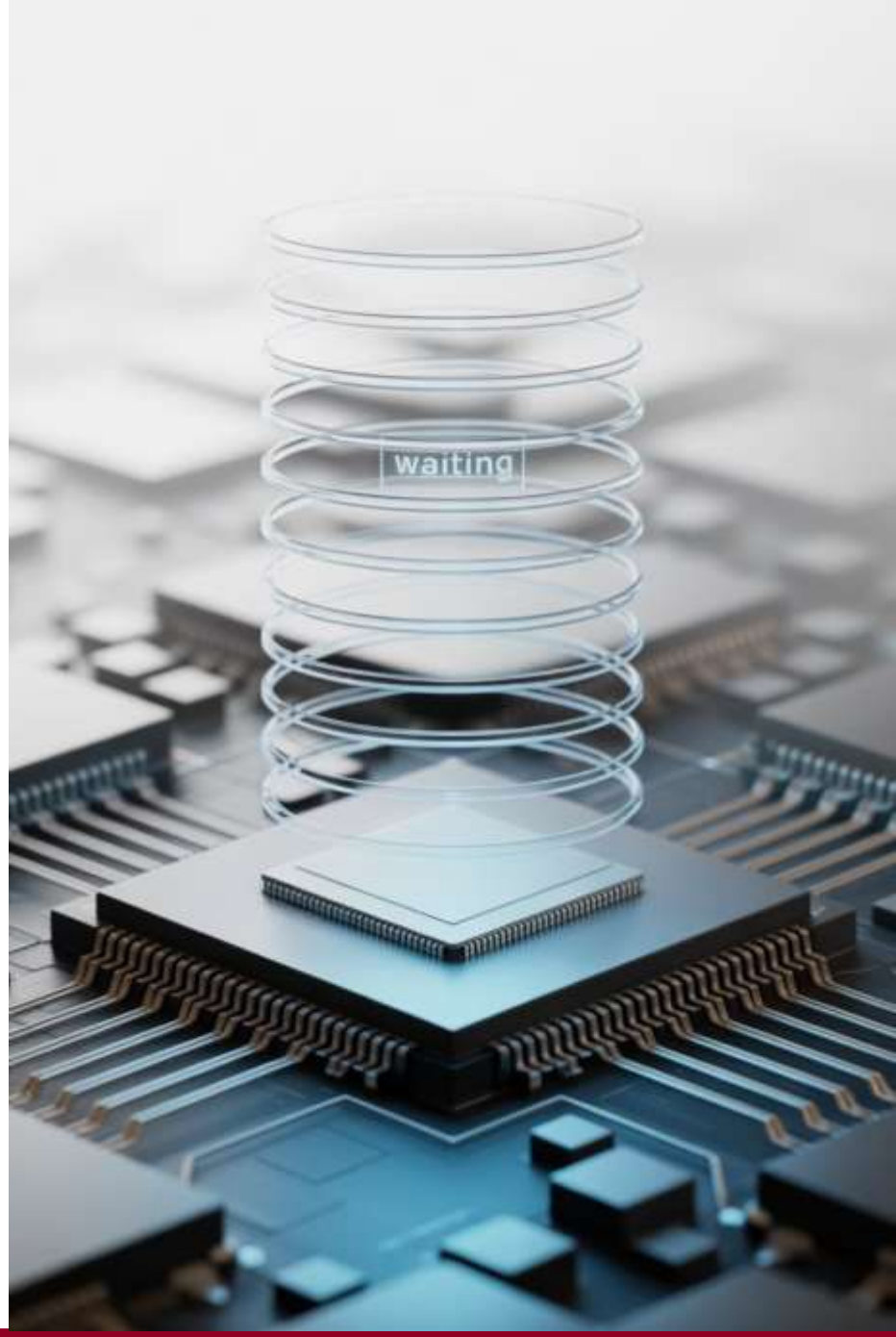
Sequential Consistency

Requires that execution results be the same as if memory accesses by each processor were kept in order and accesses among different processors were arbitrarily interleaved.

Implementation: A processor must delay completion of any memory access until all invalidations caused by that access are completed.

Performance Impact

Example: With 50 cycles for ownership, 10 cycles per invalidate, and 80 cycles for acknowledgment, a write miss with four sharing processors takes 170 cycles under sequential consistency.



Relaxed Consistency

Synchronized Programs

A program is synchronized if all accesses to shared data are ordered by synchronization operations.

Most programs are synchronized because unsynchronized programs have unpredictable behavior dependent on execution speed.

Programmers typically use standard synchronization libraries rather than creating their own mechanisms.

Relaxed Consistency Models

Allow reads and writes to complete out of order, but use synchronization to enforce ordering when needed.

Sequential consistency requires maintaining all four orderings: R→W, R→R, W→R, and W→W.

Relaxed models selectively relax these orderings:

- Total store ordering/processor consistency: Relaxes W→R
- Partial store order: Relaxes W→W
- Weak ordering/release consistency: Relaxes R→W and R→R

Current Trends in Memory Consistency



Modern Implementations

Many current multiprocessors support some form of relaxed consistency model, from processor consistency to release consistency.

Programming Impact

Most programmers use standard synchronization libraries and write synchronized programs, making the choice of consistency model largely invisible.

Performance Considerations

Alternative viewpoint: With speculation, much of the performance advantage of relaxed consistency can be obtained with sequential or processor consistency.

The compiler's ability to optimize memory access to potentially shared variables remains a key consideration in the debate between consistency models.

Compiler Optimization Challenges

- Memory consistency models specify the range of legal compiler optimizations on shared data
- In explicitly parallel programs, compilers cannot freely interchange reads and writes of different shared data items without clear synchronization points, as such transformations might affect program semantics
  - This constraint prevents even simple optimizations like register allocation of shared data
- In implicitly parallelized programs like High Performance FORTRAN (HPF), synchronization points are known, avoiding this issue
- Whether compilers can gain significant advantage from more relaxed consistency models remains an open research question.

Speculation Hides Latency in Strict Models

Dynamic Reordering

Processor uses dynamic scheduling to reorder memory references, executing them potentially out of order.

Violation Detection

If invalidation arrives before commit, processor detects potential sequential consistency violations.

Speculative Recovery

Processor backs out computation and restarts with invalidated memory reference, ensuring correct execution.

- Hill advocated sequential or processor consistency with speculative execution because:
  - Aggressive implementation gains most advantages of relaxed models
  - Adds little to implementation cost of speculative processor
  - Allows programmers to reason using simpler programming models
- The MIPS R10000 used out-of-order capability to support aggressive implementation of sequential consistency

Multilevel Cache Inclusion

- Multilevel inclusion means every cache level is a subset of the level further from the processor
  - This reduces contention between coherence traffic and processor traffic when snoops and cache accesses compete for resources
- Many multiprocessors enforce inclusion, though recent designs with smaller L1 caches and different block sizes sometimes choose not to
  - Intel i7 uses inclusion for L3, while AMD Opteron makes L2 inclusive of L1 but not L3.

Inclusion Challenges

Preserving inclusion becomes complex when L1 and L2 have different block sizes. A block in L2 represents multiple blocks in L1, and a miss in L2 causes replacement of data equivalent to multiple L1 blocks.

01

Problem Setup

L2 block size is four times L1. L1 contains blocks at addresses x and x+b, where x mod 4b = 0.

02

Miss Occurs

Reference to block y maps to same location as x in both caches, causing miss in both L1 and L2.

03

Replacement

L2 fetches 4b bytes and replaces blocks x, x+b, x+2b, x+3b. L1 replaces only block x.

04

Violation

L1 still contains x+b but L2 does not, violating the inclusion property.

To maintain inclusion with multiple block sizes, higher cache levels must be probed during lower-level replacements to invalidate replaced words.