IC-UNICAMP

MO401

IC/Unicamp
2013s1
Prof Mario Côrtes

Capítulo 5

Multiprocessors and Thread-Level Parallelism

IC-UNICAMP

Tópicos

- Centralized shared-memory architectures
- Performance of symmetric shared-memory architectures
- Distributed shared-memory and directory-based coherence
- Synchronization
- Memory consistency

5.1 Introduction

- Importance of multiprocessing (from low to high end)
  - Power wall, ILP wall: power and silicon costs growed 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

5.1 Introduction

- Thread-Level parallelism
  - Have multiple program counters
  - Uses MIMD model (use of TLP is relatively recent)
  - Targeted for tightly-coupled shared-memory multiprocessors
  - Explit TLP in two ways
    - tightly-coupled threads in single task → parallel processing
    - execution of independent tasks or processes → request-level parallelism (multiprogramming is one form)
- In this chapter: 2-32 processors + shared-memory (multicore + multithread)
  - next chapter: warehouse-scale computers
  - not covered: large-scale multicomputer (Culler)

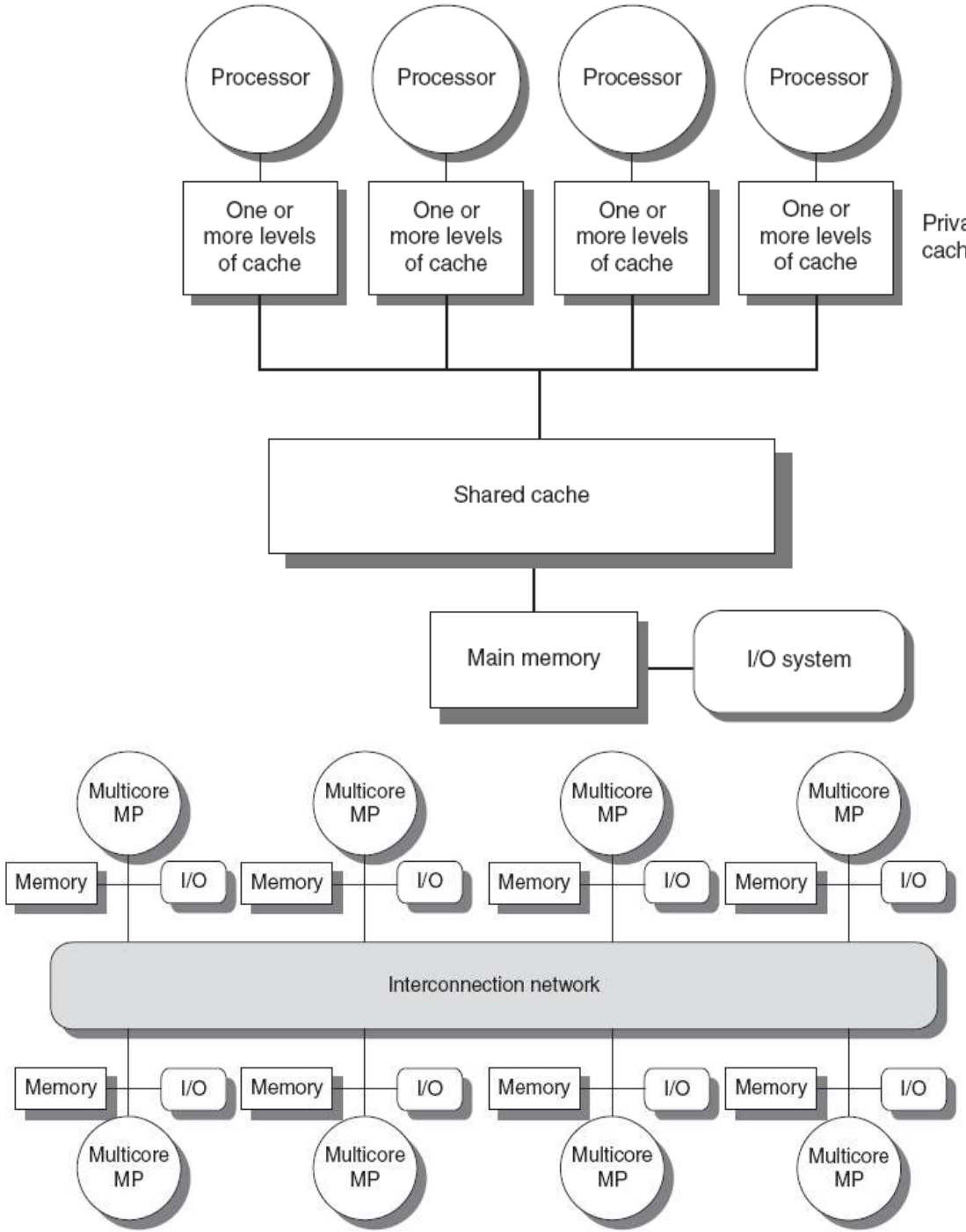
IC-UNICAMP

Multiprocessor architecture: issues/approach

- To use MIMD, *n* processors, at least *n* threads are needed
- Threads typically identified by programmer or created by OS (request-level)
- Could be many iterations of a single loop, generated by compiler
- Amount of computation assigned to each thread = grain size
  - Threads can be used for data-level parallelism, but the overheads may outweigh the benefit
  - Grain size must be sufficiently large to exploit parallelism
    - a GPU could be able to parallelize operations on short vectors, but in a MIMD the overhead could be too large

Types

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



IC-UNICAMP

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 !!
  - 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

IC-UNICAMP

Exmpl p350: communication costs

Example 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?

Answer 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.

5.2 Centralized Shared-Memory Architectures

IC-UNICAMP

- Motivation: large multilevel caches reduce memory BW needs
- Originallly: processors were single core, one board, memory on a shared bus
- Recently: bus capacity not enough; μp directly connected to memory chip; accessing remote data goes through remote μp memory owner → asymmetric access
  - two multicore chips: latency to local memory ≠ remote memory
- Processors cache private and shared data
  - private data: ok, as usual
  - shared data: new problem → cache coherence

Cache Coherence

- Processors may see different values through their caches
- Example p352

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

- Informal definition: a memory system is coherent if any read of a data item returns the most recently written value

Cache Coherence

- A memory system is coherent if
  1. A read by processor P to location X that follows a write by P to X, with no writes of X by another processor occurring between the write and the read by P, always returns the value written by P
     - Preserves program order
  2. A read by a processor to location X that follows a write by another processor to X returns written value if the read and write are sufficiently separated in time and no other writes to X occur between the two accesses
     - if a processor could continuously read old value → incoherent memory
  3. Writes to the same location are serialized. Two writes to the same location by any two processors are seen in the same order by all processors.

- Three properties: sufficient conditions for coherence
- But, what if two processors have “simultaneous” accesses to memory location X, P1 reads X and P2 writes X? What is P1 supposed to read?
  - when a written value must be seen by a reader is defined by a memory consistency model

IC-UNICAMP

Memory Consistency

- Coherence and consistency are complementary
  - Cache coherence defines the behavior of reads and writes to the same memory location
  - Memory consistency defines the behavior of reads and writes with respect to accesses to other memory locations
- Consistency model in section 5.6
- For now
  - a write does not complete (does not allow next write to start) until all processors have seen the effect of that write (write propagation)
  - the processor does not change the order of any write with respect to any other memory access.
- Example
  - if one processor writes location A and then location B
  - any processor that sees new value of B must also see new value of A
- Writes must be completed in program order

Enforcing Coherence

- Coherent caches provide:
  - *Migration*: movement of data to local storage → reduced latency
  - *Replication*: multiple copies of data → reduced latency and contention
- Cache coherence protocols
  - Directory based
    - Sharing status of each block kept in one location, the directory
    - In SMP: centralized directory in memory or outermost cache in a multicore
    - In DSM: distributed directory (sec 5.4)
  - Snooping
    - Each core broadcast its memory operations, via bus or other structure
    - Each core monitors (snoops) the broadcasting media and tracks sharing status of each block
- Snooping popular with bus-based multiprocessing
  - Multicore architecture changed the picture → all multicores share some level of cache on chip → some designers switched to directory based coherence

IC-UNICAMP

Snoopy Coherence Protocols

- Write invalidate
  - On write, invalidate all other copies
  - Use bus itself to serialize
    - Write cannot complete until bus access is obtained
- Write update
  - On write, update all copies
  - Consumes more BW
- Which is better? Depends on memory access pattern
  - After I write, what is more likely? Others read? I write again?
- Coherence protocols are orthogonal to cache write policies
  - Invalidate
    - write through?
    - write back?
  - Update
    - write through?
    - write back?

Exmpl: Invalidate and Write Back

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

Figure 5.4 An example of an invalidation protocol working on a snooping bus for a single cache block (X) with write-back caches. We assume that neither cache initially holds X and that the value of X in memory is 0. The processor and memory contents show the value after the processor and bus activity have both completed. A blank indicates no activity or no copy cached. When the second miss by B occurs, processor A responds with the value canceling the response from memory. In addition, both the contents of B's cache and the memory contents of X are updated. This update of memory, which occurs when a block becomes shared, simplifies the protocol, but it is possible to track the ownership and force the write-back only if the block is replaced. This requires the introduction of an additional state called "owner," which indicates that a block may be shared, but the owning processor is responsible for updating any other processors and memory when it changes the block or replaces it. If a multicore uses a shared cache (e.g., L3), then all memory is seen through the shared cache; L3 acts like the memory in this example, and coherency must be handled for the private L1 and L2 for each core. It is this observation that led some designers to opt for a directory protocol within the multicore. To make this work the L3 cache must be inclusive (see page 397).

IC-UNICAMP

Snoopy Coherence Protocols

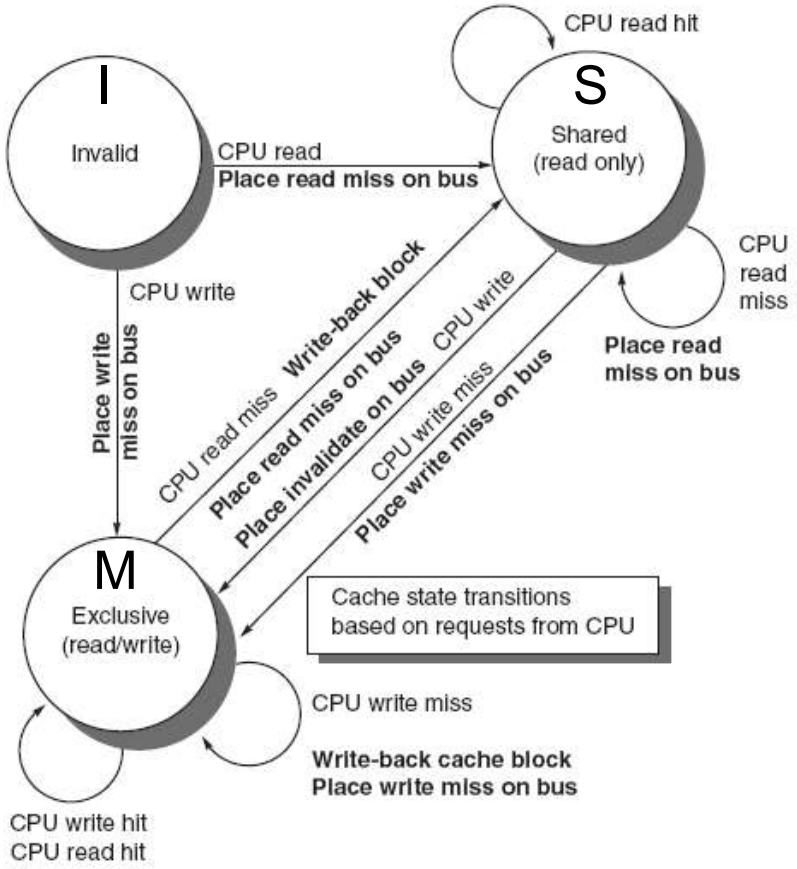
- Bus or broadcasting media acts as write serialization mechanism: writes to a memory location are in bus order
- How to locate an item when a read miss occurs?
  - In write through cache, all copies are valid (updated)
  - In write-back cache, if a cache has data in dirty state, it sends the updated value to the requesting processor (bus transaction)
- 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
- Há diferentes protocolos de coerência
  - Para write invalidate: MSI (prox slide), MESI, MOESI
- Snoopy requer adição de tags de estado a cada bloco da cache: estado do protocolo usado → shared, modified, exclusive, invalid
  - Como tanto o processador como o snoopy controller devem acessar os cache tags, normalmente os tags são duplicados

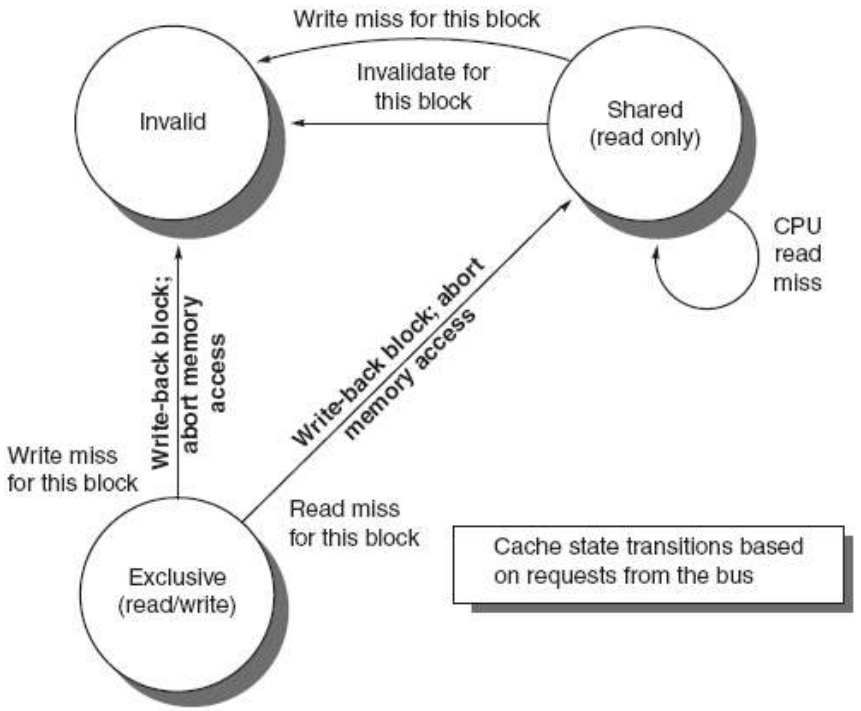
Fig 5.5 Snoopy Coherence Protocols: MSI

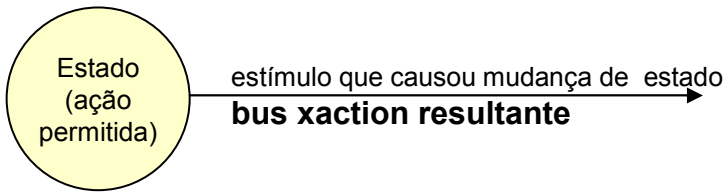
Request	Source	State of addressed cache block	Type of cache action	Function and explanation
Read hit	Processor	Shared or modified	Normal hit	Read data in local cache.
Read miss	Processor	Invalid	Normal miss	Place read miss on bus.
Read miss	Processor	Shared	Replacement	Address conflict miss: place read 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.

Snoopy Coherence Protocols: MSI

IC.







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

Snoopy Coherence Protocols

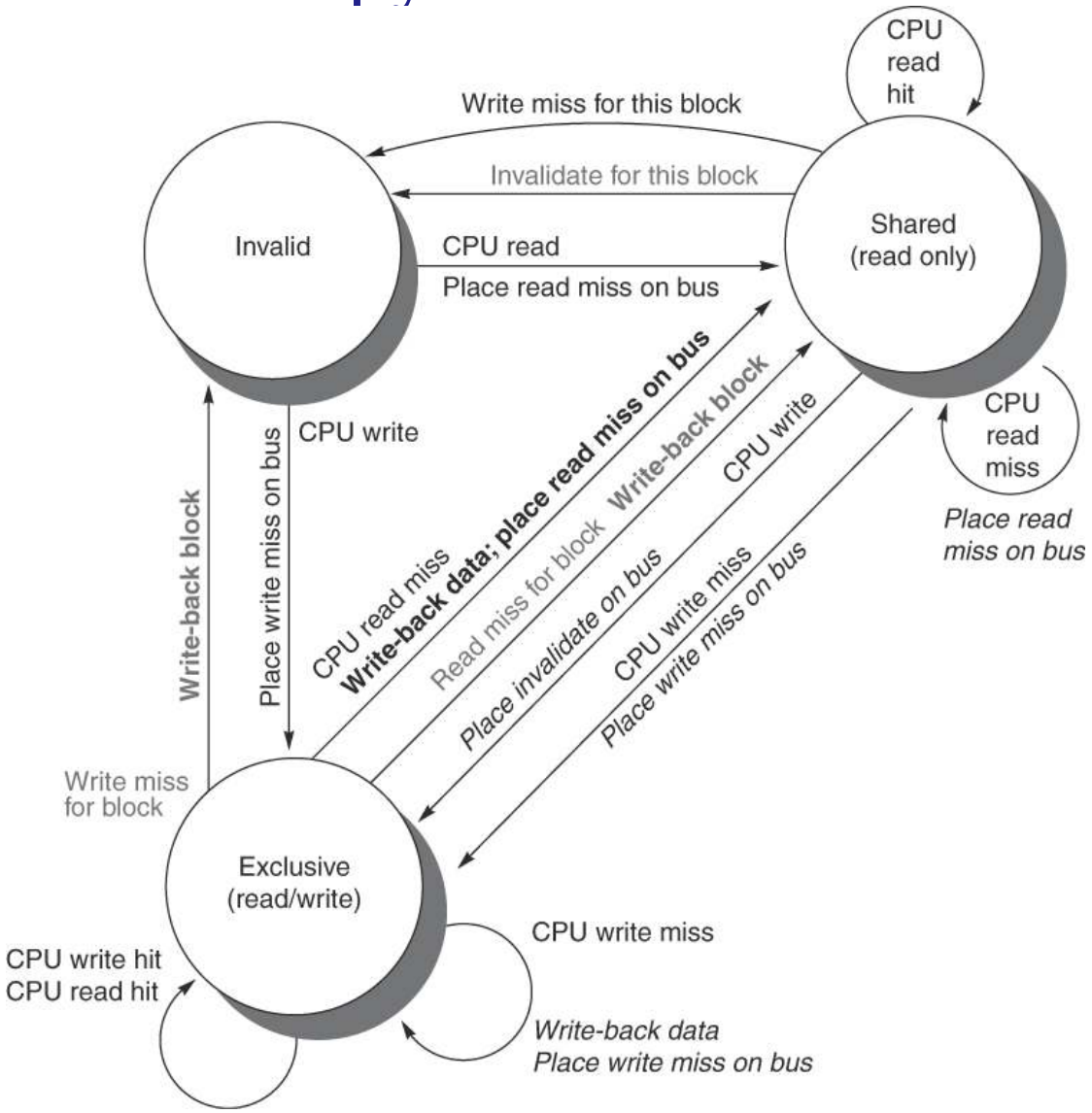


Figure 5.7 Cache coherence state diagram with the state transitions induced by the local processor shown in black and by the bus activities shown in gray. Activities on a transition are shown in bold.

IC-UNICAMP

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 Exclusive-clean
  - Owned state: MOESI
    - solves problem: if block in shared state, who should supply a copy in case a processor misses?
      - Before: everybody + memory abort
      - Now: owner

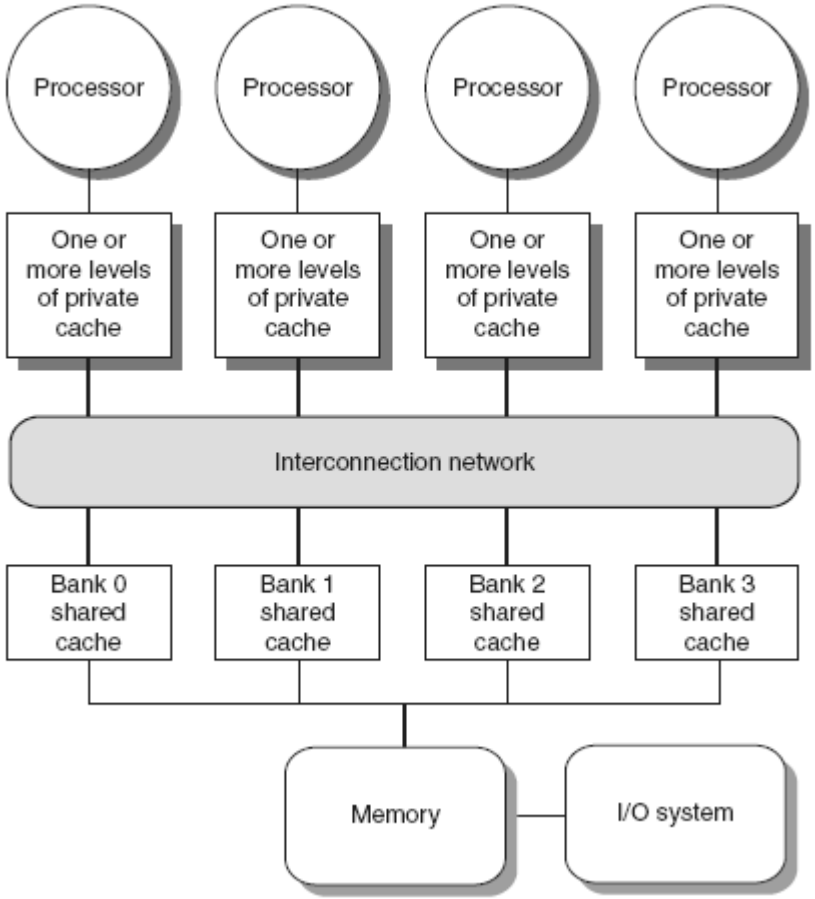
IC-UNICAMP

Limitations of SMP and snooping

- As numbers of processors grow, any centralized resource can become a bottleneck
- In a multicore, private L1/L2 and shared L3 (on chip) → ok up to 8 cores
- Snooping bandwith. Solutions
  - duplicate cache tags
  - directory at the outermost cache level (Intel i7 and Xeon)

Limitations (2)

- To solve bus traffic limitations
- Use interconnection network
  - crossbars or point-to-point networks with banked memory
- Does not scale well



IC-UNICAMP

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

IC-UNICAMP

Evolution

- Bus + snoop + small scale multiprocessing = ok
- As number or 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)

IC-UNICAMP

5.3 Performance of SMP

- Performance depends on many factors
  - overall cache performance = uniprocessor cache miss traffic + communication traffic
  - processor count, cache size, block size
- 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

Exmpl p366: miss identification

Example Assume that words x1 and x2 are in the same cache block, which is in the shared state in the caches of both P1 and P2. Assuming the following sequence of events, identify each miss as a true sharing miss, a false sharing miss, or a hit. Any miss that would occur if the block size were one word is designated a true sharing miss.

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

Answer 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.

Study on a commercial workload

4 processor shared-memory, Alpha, 4 instructions issue, 1998 (but structure similar to modern multicore chips)

Cache 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

Figure 5.9 The characteristics of the cache hierarchy of the Alpha 21164 used in this study and the Intel i7. Although the sizes are larger and the associativity is higher on the i7, the miss penalties are also higher, so the behavior may differ only slightly. For example, from Appendix B, we can estimate the miss rates of the smaller Alpha L1 cache as 4.9% and 3% for the larger i7 L1 cache, so the average L1 miss penalty per reference is 0.34 for the Alpha and 0.30 for the i7. Both systems have a high penalty (125 cycles or more) for a transfer required from a private cache. The i7 also shares its L3 among all the cores.

Study on a commercial workload

- OLTP: Online transaction-processing workload modeled after TPC-B. Requests to an Oracle DB
- DSS: Decision Support System based on TPC-D, also with Oracle
- Alta Vista: web search engine

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

Figure 5.10 The distribution of execution time in the commercial workloads. The OLTP benchmark has the largest fraction of both OS time and processor idle time (which is I/O wait time). The DSS benchmark shows much less OS time, since it does less I/O, but still more than 9% idle time. The extensive tuning of the AltaVista search engine is clear in these measurements. The data for this workload were collected by Barroso, Gharachorloo, and Bugnion [1998] on a four-processor AlphaServer 4100.

Performance Study: Commercial Workload

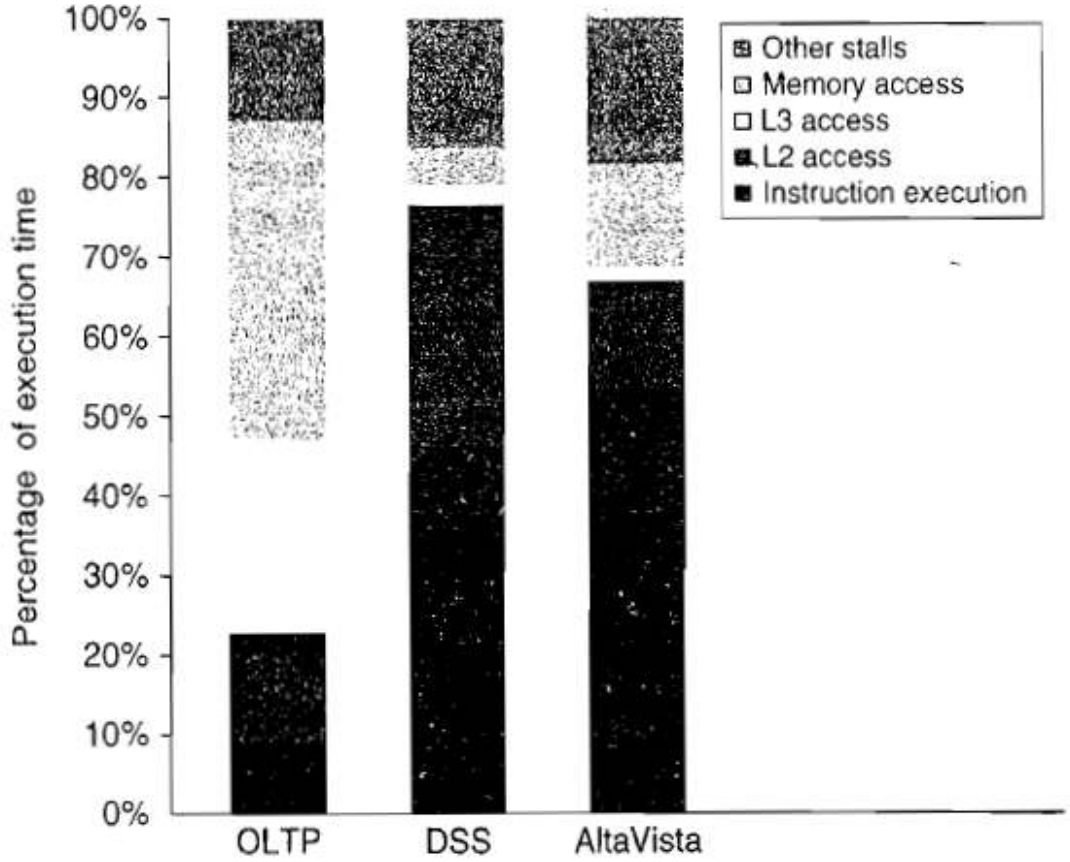


Figure 5.11 The execution time breakdown for the three programs (OLTP, DSS, and AltaVista) in the commercial workload. The DSS numbers are the average across six different queries. The CPI varies widely from a low of 1.3 for AltaVista, to 1.61 for the DSS queries, to 7.0 for OLTP. (Individually, the 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].

Performance Study: Commercial Workload

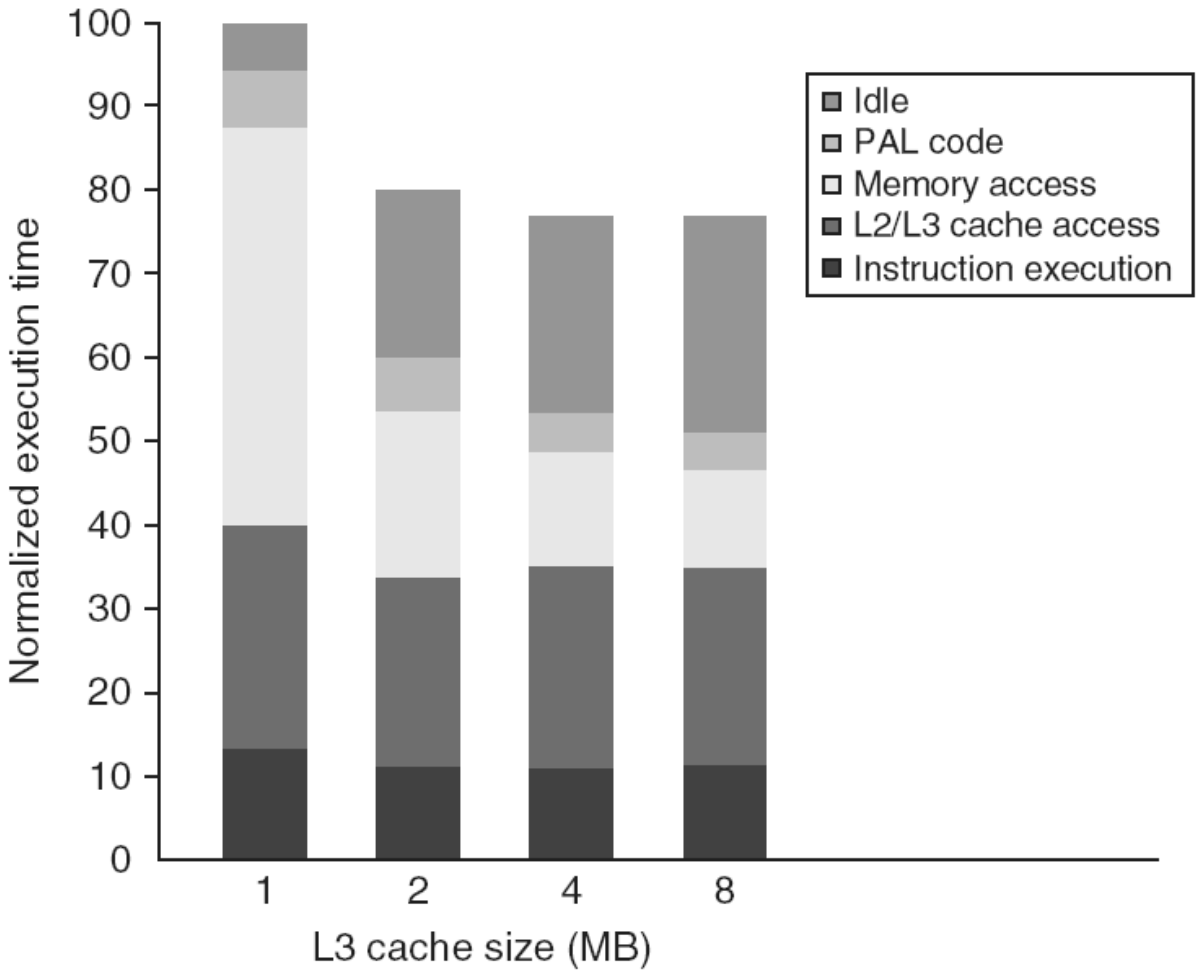


Figure 5.12 The relative performance of the OLTP workload as the size of the L3 cache, which is set as two-way set associative, grows from 1 MB to 8 MB. The idle time also grows as cache size is increased, reducing some of the performance gains. This growth occurs because, with fewer memory system stalls, more server processes are needed to cover the I/O latency. The workload could be retuned to increase the computation/communication balance, holding the idle time in check. The PAL code is a set of sequences of specialized OS-level instructions executed in privileged mode; an example is the TLB miss handler.

Performance Study: Commercial Workload

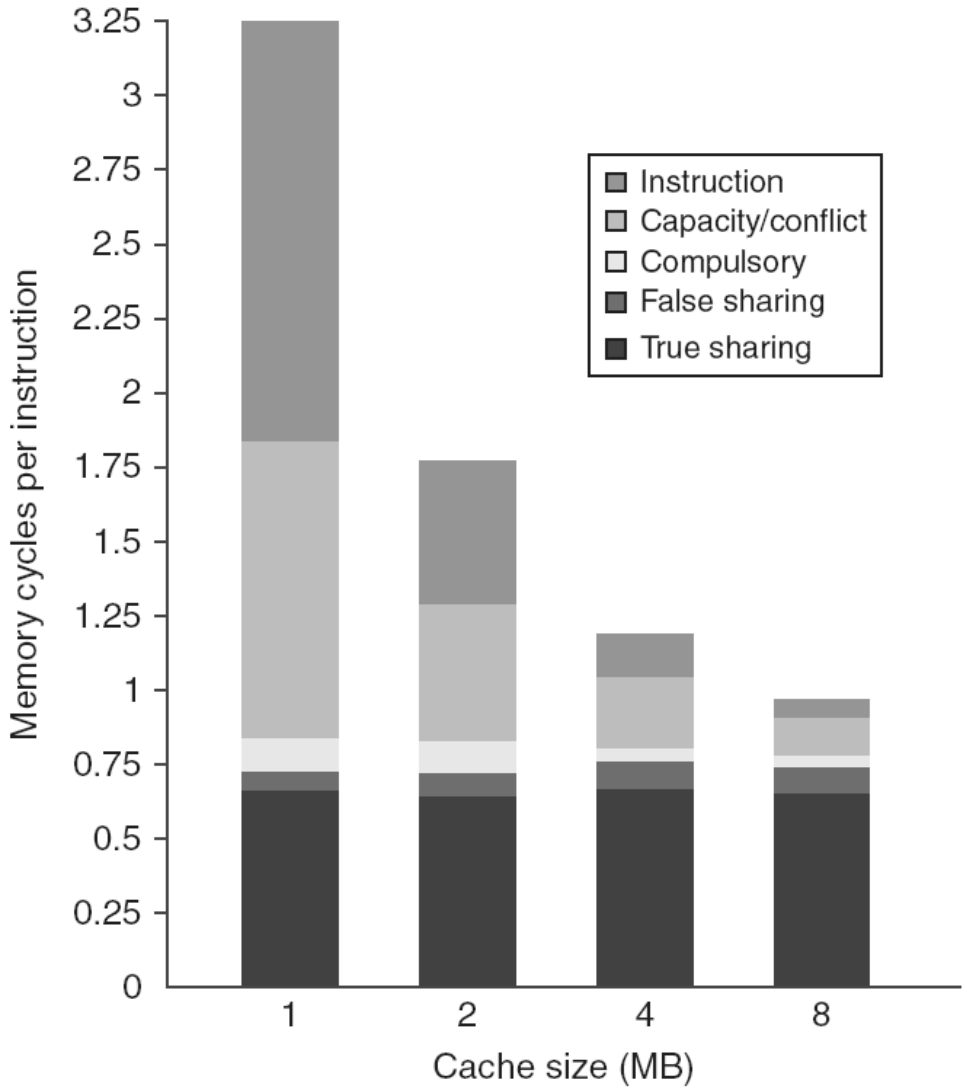


Figure 5.13 The contributing causes of memory access cycle shift as the cache size is increased. The L3 cache is simulated as two-way set associative.

Performance Study: Commercial Workload

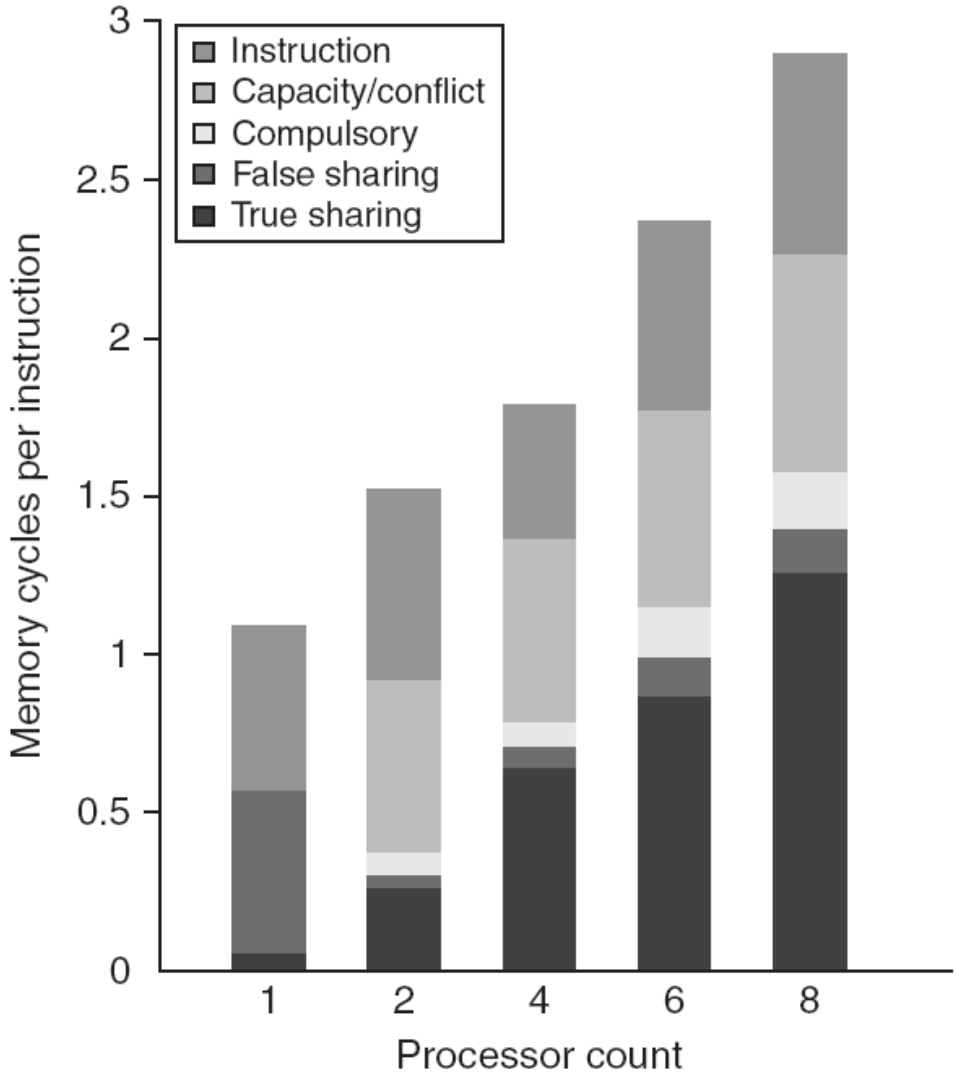


Figure 5.14 The contribution to memory access cycles increases as processor count increases primarily due to increased true sharing. The compulsory misses slightly increase since each processor must now handle more compulsory misses.

Performance Study: Commercial Workload

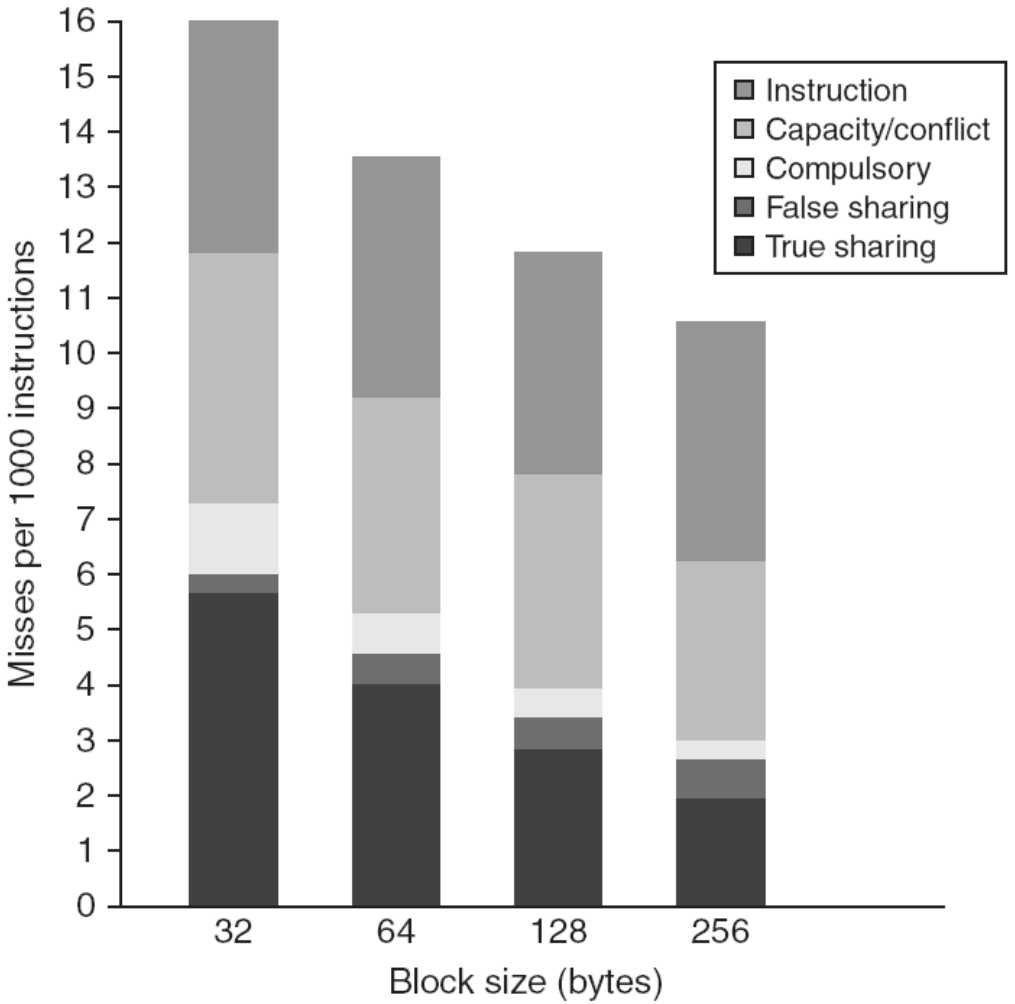


Figure 5.15 The number of misses per 1000 instructions drops steadily as the block size of the L3 cache is increased, making a good case for an L3 block size of at least 128 bytes. The L3 cache is 2 MB, two-way set associative.

Performance Study: Multiprogramming and OS Workload

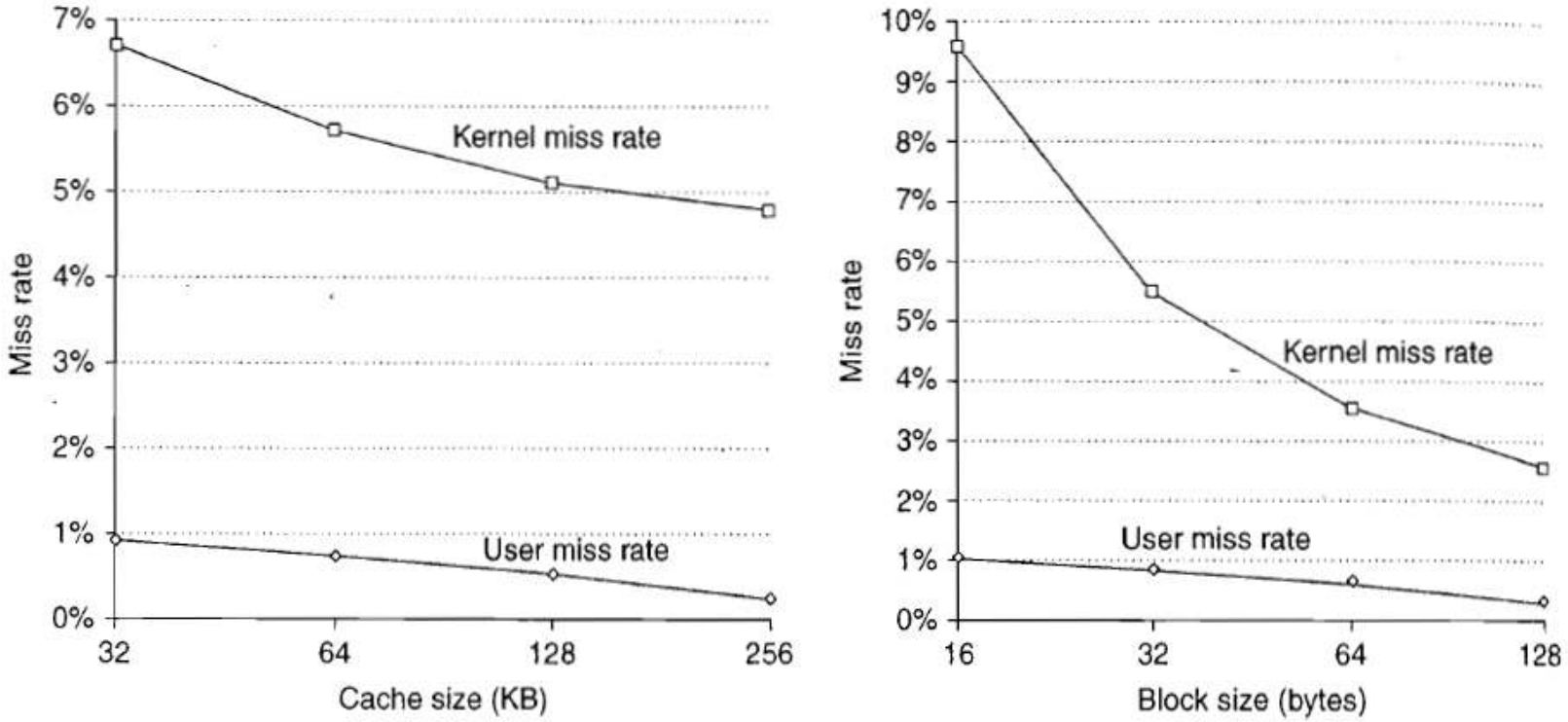


Figure 5.17 The data miss rates for the user and kernel components behave differently for increases in the L1 data cache size (on the left) versus increases in the L1 data cache block size (on the right). Increasing the L1 data cache from 32 KB to 256 KB (with a 32-byte block) causes the user miss rate 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).

Performance Study: Multiprogramming and OS Workload

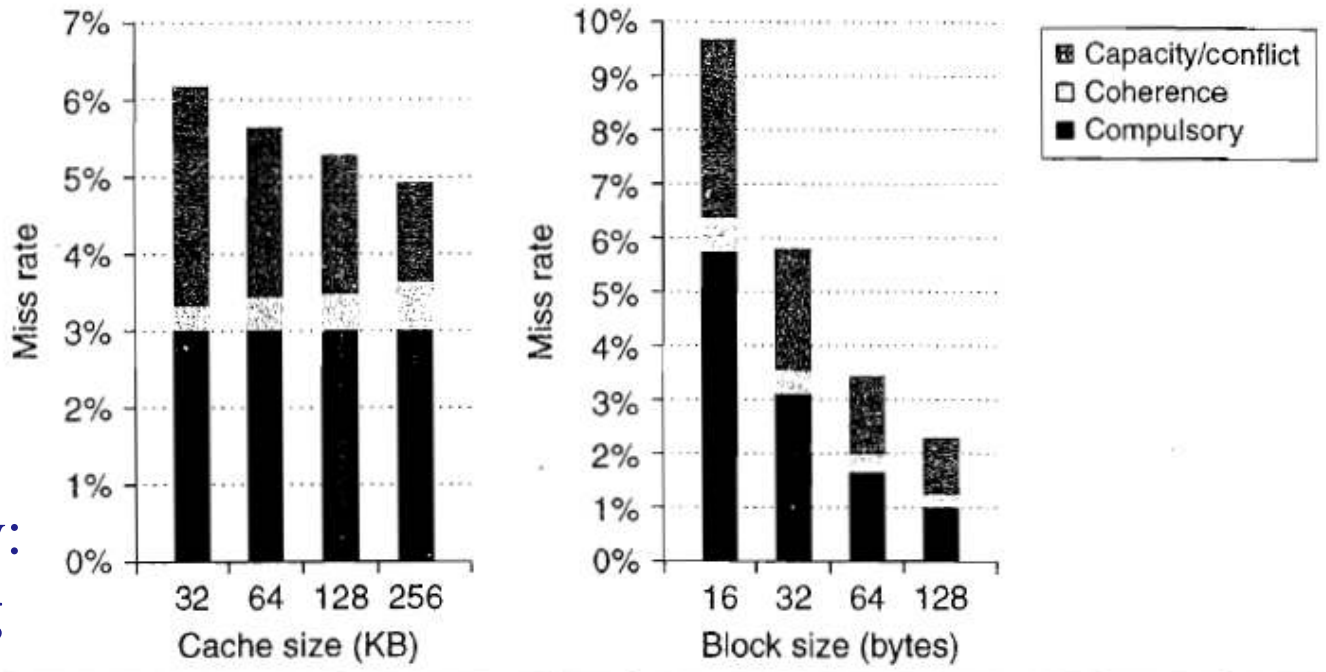


Figure 5.18 The components of the kernel data miss rate change as the L1 data cache size is increased from 32 KB to 256 KB, when the multiprogramming workload is run on eight processors. The compulsory miss rate component stays constant, since it is unaffected by cache size. The capacity component drops 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%.

Performance Study: Multiprogramming and OS Workload

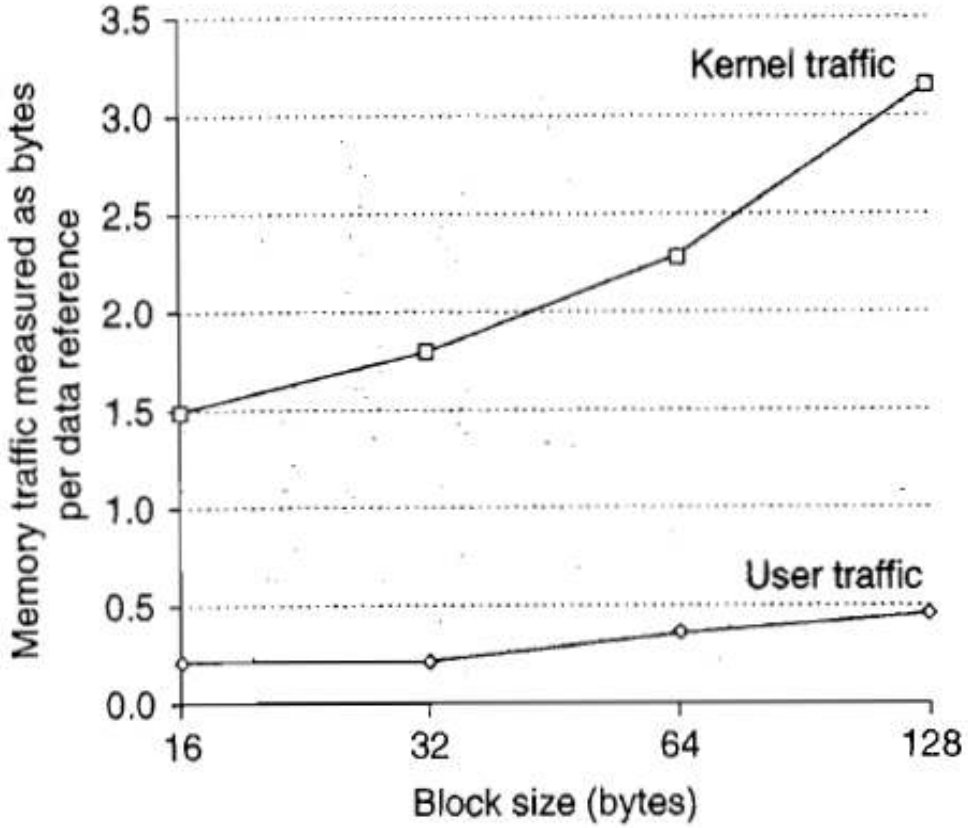
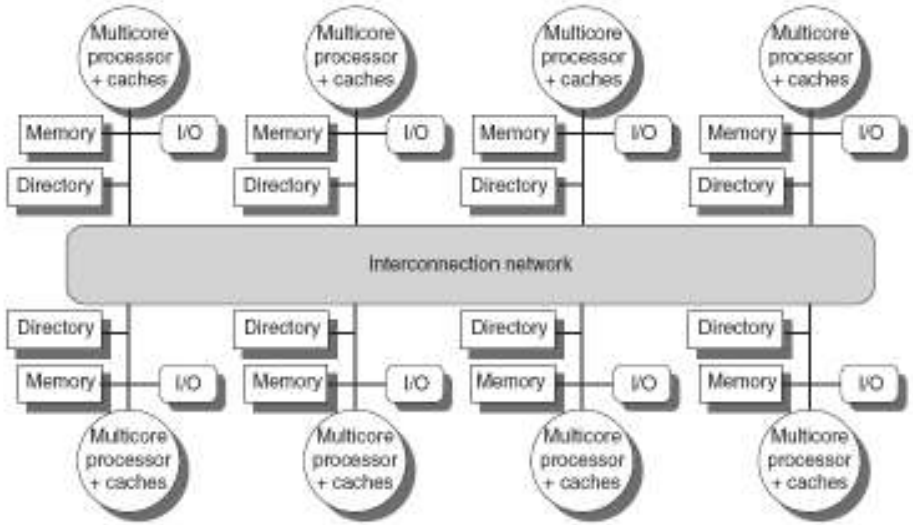


Figure 5.19 The number of bytes needed per data reference grows as block size is increased for both the kernel and user components. It is interesting to compare this chart against the data on scientific programs shown in Appendix I.

IC-UNICAMP

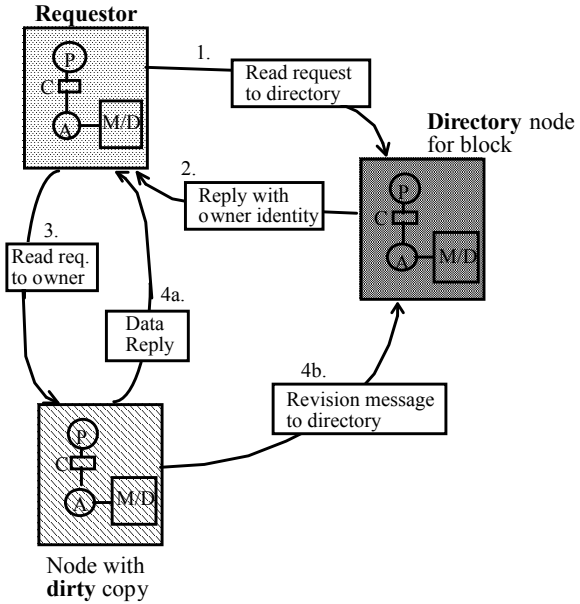
5.4 Directory Protocols

- 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
    - indicates which cores may have copies; inval → only to these
    - ok if inclusive
  - Not scalable beyond shared L3 (centralized directory)
- Implement in a distributed fashion (next to memory)
  - each memory block has bit vector; total overhead = # memory blocks * # nodes

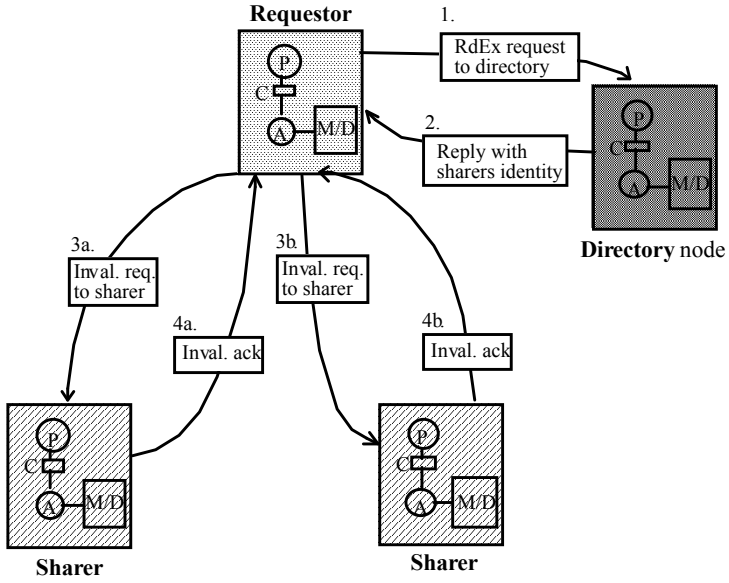


Protocolos de cache e de diretório

- São coisas diferentes.
- Em um bus, bus transactions fazem a comunicação (única) necessária para o snooping e a integridade do protocolo
- Em rede, não há broadcasting, podem ser necessárias múltiplas network transactions para completar uma operação.



(a) Read miss to a block in dirty state



(b) Write miss to a block with two sharers

IC-UNICAMP

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
- Nodes
  - Local = Requestor
  - Home = node with directory

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.

Figure 5.21 The possible messages sent among nodes to maintain coherence, along with the source and destination node, the contents (where P = requesting node number, A = requested address, and D = data contents), and the function of the message. The first three messages are requests sent by the local node to the home. The fourth through sixth messages are messages sent to a remote node by the home when the home needs the data to satisfy a read or write miss request. Data value replies are used to send a value from the home node back to the requesting node. Data value write-backs occur for two reasons: when a block is replaced in a cache and must be written back to its home memory, and also in reply to fetch or fetch/invalidate messages from the home. Writing back the data value whenever the block becomes shared simplifies the number of states in the protocol, since any dirty block must be exclusive and any shared block is always available in the home memory.

Individual cache block in a directory-based system

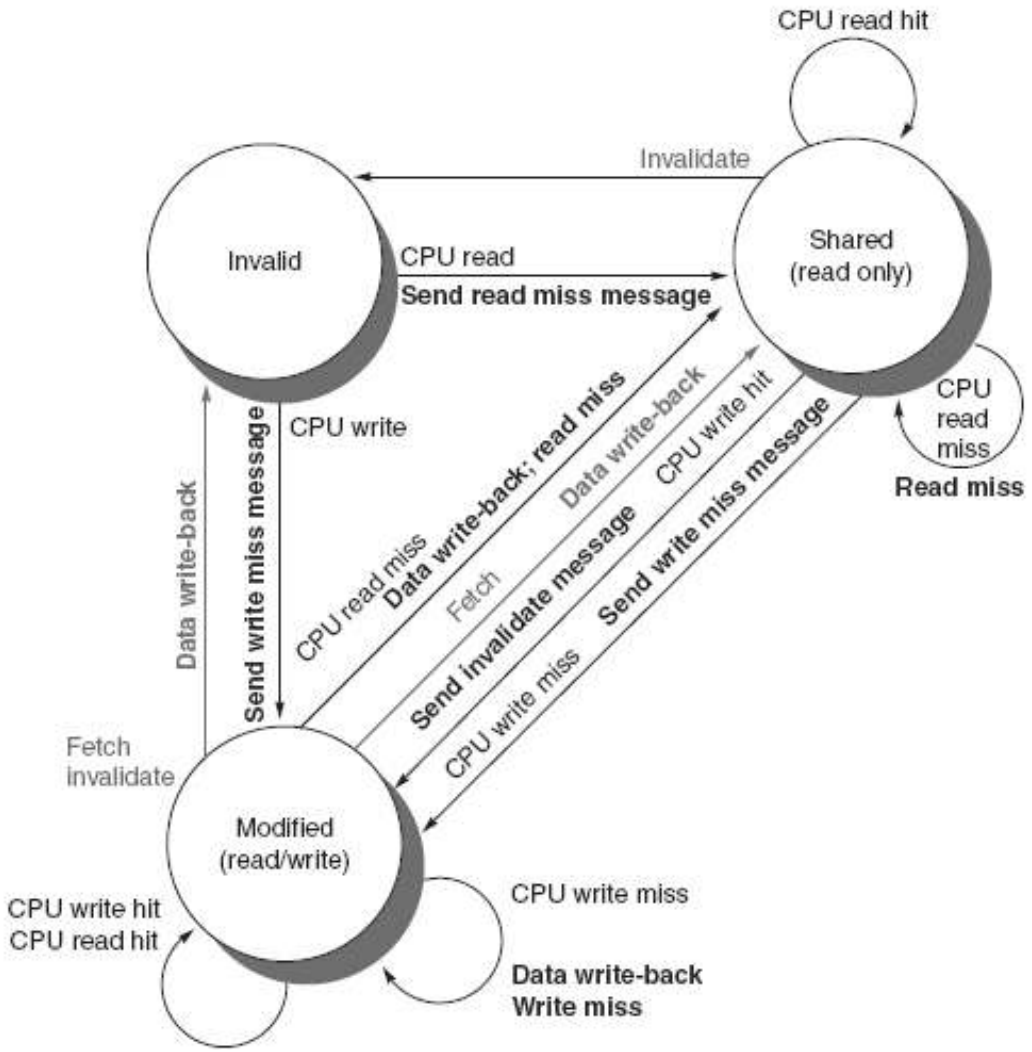


Figure 5.22 State transition diagram for an individual cache block in a directory-based system. Requests by the local processor are shown in black, and those from the home directory are shown in gray. The states are identical to those in the snooping case, and the transactions are very similar, with explicit invalidate and write-back requests replacing the write misses that were formerly broadcast on the bus. As we did for the snooping controller, we assume that an attempt to write a shared cache block is treated as a miss; in practice, such a transaction can be treated as an ownership request or upgrade request and can deliver ownership without requiring that the cache block be fetched.

Directory

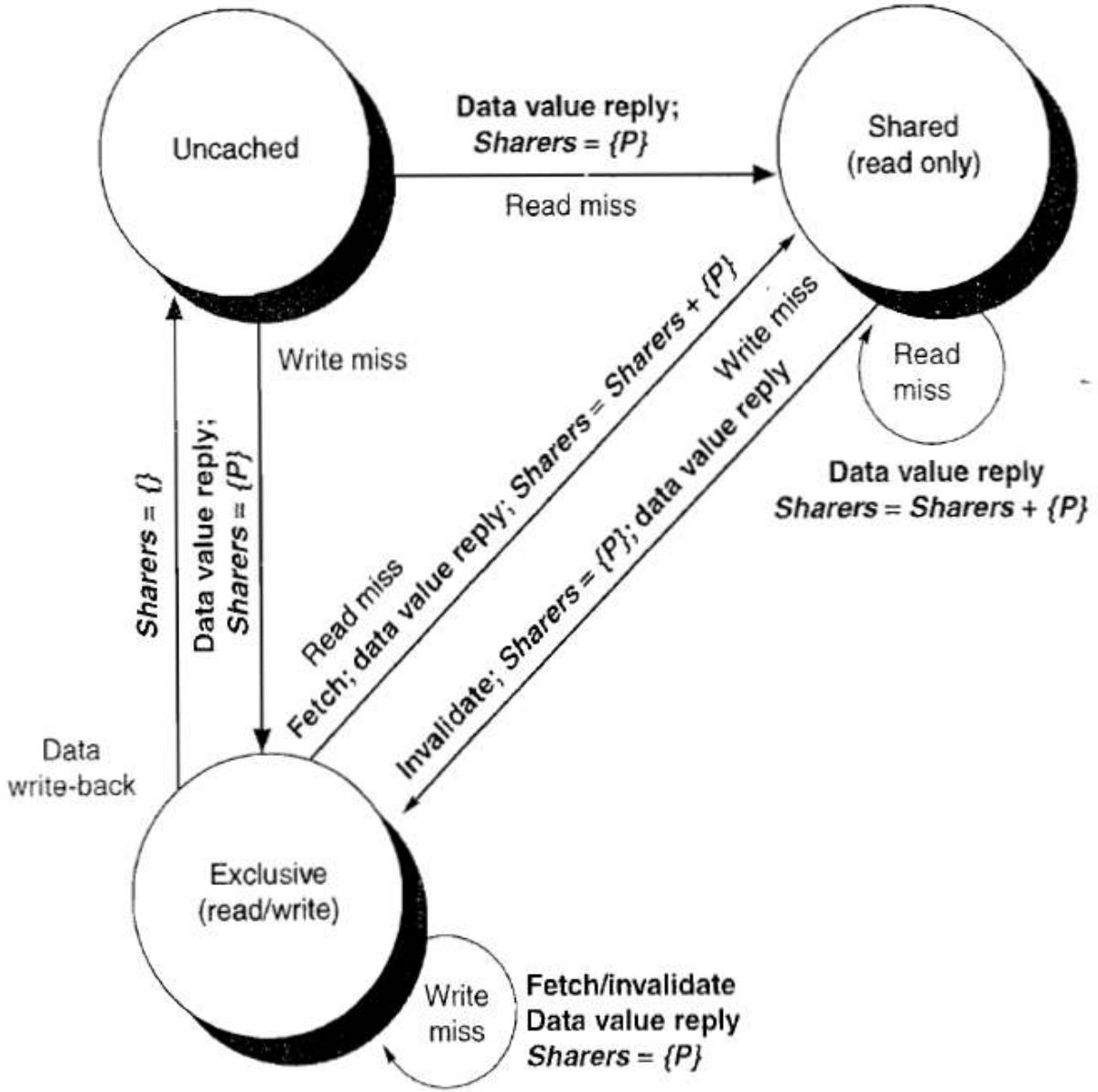


Figure 5.23 The state transition diagram for the directory has the same states and structure as the transition diagram for an individual cache. All actions are in gray because they are all externally caused. Bold indicates the action taken by the directory in response to the request.

IC-UNICAMP

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

IC-UNICAMP

Directory Protocols

- 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 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

5.5 Synchronization

- Basic building blocks: atomic read-modify-write
  - 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

Example LL-SC

Atomic exchange in memory location specified by R1:

```
try:    MOV     R3,R4       ;move 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
```

LL-SC implementing an atomic fetch-and-increment:

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

IC-UNICAMP

Implementing Locks

- Spin lock: a processor continuously tries to acquire

If no coherence, lock kept in memory:

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

If coherence, cached lock:

```
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
```

Cached Spin Locks: bus 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

Figure 5.24 Cache coherence steps and bus traffic for three processors, P0, P1, and P2. This figure assumes write invalidate coherence. P0 starts with the lock (step 1), and the value of the lock is 1 (i.e., locked); it is initially exclusive and owned by P0 before step 1 begins. P0 exits and unlocks the lock (step 2). P1 and P2 race to see which reads the unlocked value during the swap (steps 3 to 5). P2 wins and enters the critical section (steps 6 and 7), while P1's attempt fails so it starts spin waiting (steps 7 and 8). In a real system, these events will take many more than 8 clock ticks, since acquiring the bus and replying to misses take much longer. Once step 8 is reached, the process can repeat with P2, eventually getting exclusive access and setting the lock to 0.

5.6 Models of Memory Consistency

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

IC-UNICAMP

Exmpl p393: sequential consistency

Example Suppose we have a processor where a write miss takes 50 cycles to establish ownership, 10 cycles to issue each invalidate after ownership is established, and 80 cycles for an invalidate to complete and be acknowledged once it is issued. Assuming that four other processors share a cache block, how long does a write miss stall the writing processor if the processor is sequentially consistent? Assume that the invalidates must be explicitly acknowledged before the 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?

Answer 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.

IC-UNICAMP

The programmer´s view

- To implement, delay completion of all memory accesses until all invalidations caused by the access are completed
  - Reduces performance!
- Alternatives: synchronization
  - Exmpl: a variable is read and updated by two different processors
    - Each processor surrounds the memory operation with lock/unlock
      - “Unlock” after write
      - “Lock” after read
    - Data races
    - Programs with synchronization are “data-race free”
- In general, behavior of unsynchronized programs is unpredictable

IC-UNICAMP

Relaxed Consistency Models

- Idea: performance → allow writes out-of-order, but with synchronization
- Rules:
  - X → Y
    - Operation X must complete before operation Y is done
    - Sequential consistency requires:
      - R → W, R → R, W → R, W → W
  - Relax W → R
    - "Total store ordering" or "processor consistency"
  - Relax W → W
    - "Partial store order"
  - Relax R → W and R → R
    - "Weak ordering" and "release consistency"

IC-UNICAMP

5.7 Crosscutting issues

- Compiler optimization and the consistency model
  - Unless sync points are clearly identified, the compiler cannot interchange a read and a write → could affect semantics
- Using speculation to hide latency in strict consistency models
  - Use delayed commit
  - If an invalidation arrives for a result that has not been committed, use speculation recovery
    1. gets most of the advantage of relaxed consistency models
    2. implementation has low cost
    3. simple programming model

IC-UNICAMP

Inclusion and its implementation

- All blocks present in a higher level cache are also in lower levels
- Problems: different block sizes, replacement, levels of associativity
- Designers are still split on enforcement of inclusion
  - Intel i7: inclusion for L3 (directory in L3, no need to snoop in L1/L2)
  - AMD Opteron: inclusion on L2 but no inclusion on L3

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

Answer Assume that L1 and L2 are direct mapped and that the block size of L1 is b bytes and the block size of L2 is $4b$ bytes. Suppose L1 contains two blocks with starting addresses x and $x + b$ and that x mod $4b = 0$, meaning that x also is the starting address of a block in L2; then that single block in L2 contains the L1 blocks x, $x+b$, $x + 2b$, and $x + 3b$. Suppose the processor generates a reference to block y that maps to the block containing x in both caches and hence misses. Since L2 missed, it fetches $4b$ bytes and replaces the block containing x, $x + b$, $x + 2b$, and $x + 3b$, while L1 takes b bytes and replaces the block containing x. Since L1 still contains $x + b$, but L2 does not, the inclusion property no longer holds.

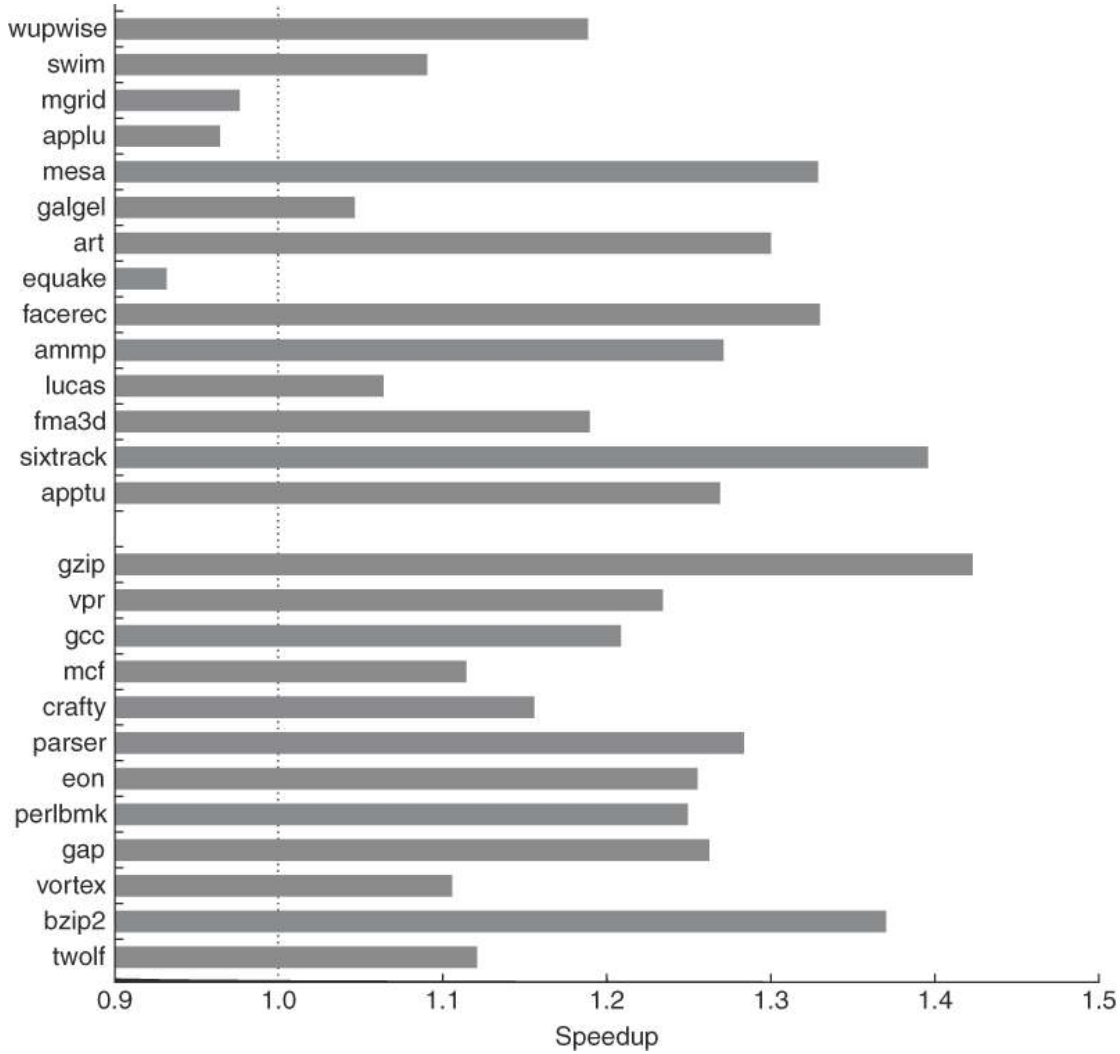
Multiprocessing and multithreading

- Studies
  - Sun T1: 4-8 core, fine grain multithreading
  - IBM Power 5: dual core, simultaneous multithreading

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.

Fig 5.26: SMT vs ST on IBM server



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.

IC-UNICAMP

5.8 Putting all together: multicores

Feature	AMD Opteron 8439	IBM Power 7	Intel Xenon 7560	Sun T2
Transistors	904 M	1200 M	2300 M	500 M
Power (nominal)	137 W	140 W	130 W	95 W
Max. cores/chip	6	8	8	8
Multithreading	No	SMT	SMT	Fine-grained
Threads/core	1	4	2	8
Instruction issue/clock	3 from onē thread	6 from one thread	4 from one thread	2 from 2 threads
Clock rate	2.8 GHz	4.1 GHz	2.7 GHz	1.6 GHz
Outermost cache	L3; 6 MB; shared	L3; 32 MB (using embedded DRAM); shared or private/core	L3; 24 MB; shared	L2; 4 MB; shared
Inclusion	No, although L2 is superset of L1	Yes, L3 superset	Yes, L3 superset	Yes
Multicore coherence protocol	MOESI	Extended MESI with behavioral and locality hints (13-state protocol)	MESIF	MOESI
Multicore coherence implementation	Snooping	Directory at L3	Directory at L3	Directory at L2
Extended coherence support	Up to 8 processor chips can be connected with HyperTransport in a ring, using directory or snooping. System is NUMA.	Up to 32 processor chips can be connected with the SMP links. Dynamic distributed directory structure. Memory access is symmetric outside of an 8-core chip.	Up to 8 processor cores can be implemented via Quickpath Interconnect. Support for directories with external logic.	Implemented via four coherence links per processor that can be used to snoop. Up to two chips directly connect, and up to four connect using external ASICs.

Figure 5.27 Summary of the characteristics of four recent high-end multicore processors (2010 releases) designed for servers. The table includes the highest core count versions of these processors; there are versions with lower core counts and higher clock rates for several of these processors. The L3 in the IBM Power7 can be all shared or partitioned into faster private regions dedicated to individual cores. We include only single-chip implementations of multicores.

Performance vs # cores: SPECrate

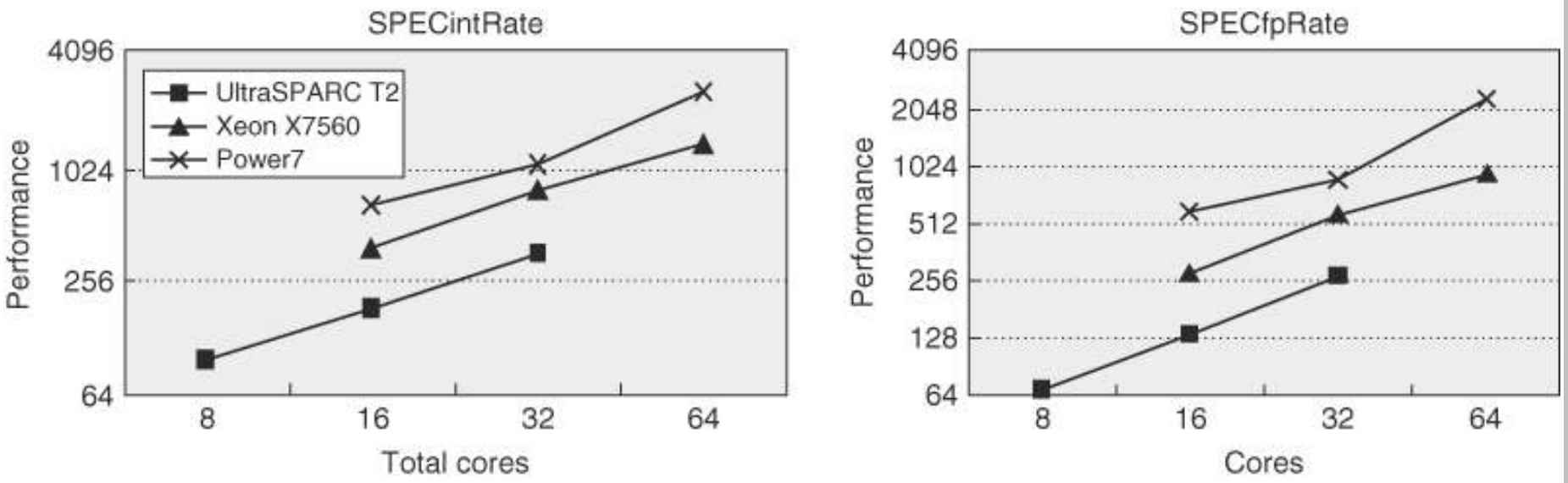


Figure 5.28 The performance on the SPECRate benchmarks for three multicore processors as the number of processor chips is increased. Notice for this highly parallel benchmark, nearly linear speedup is achieved. Both plots are on a log-log scale, so linear speedup is a straight line.

Performance vs # cores: SPECjbb2005

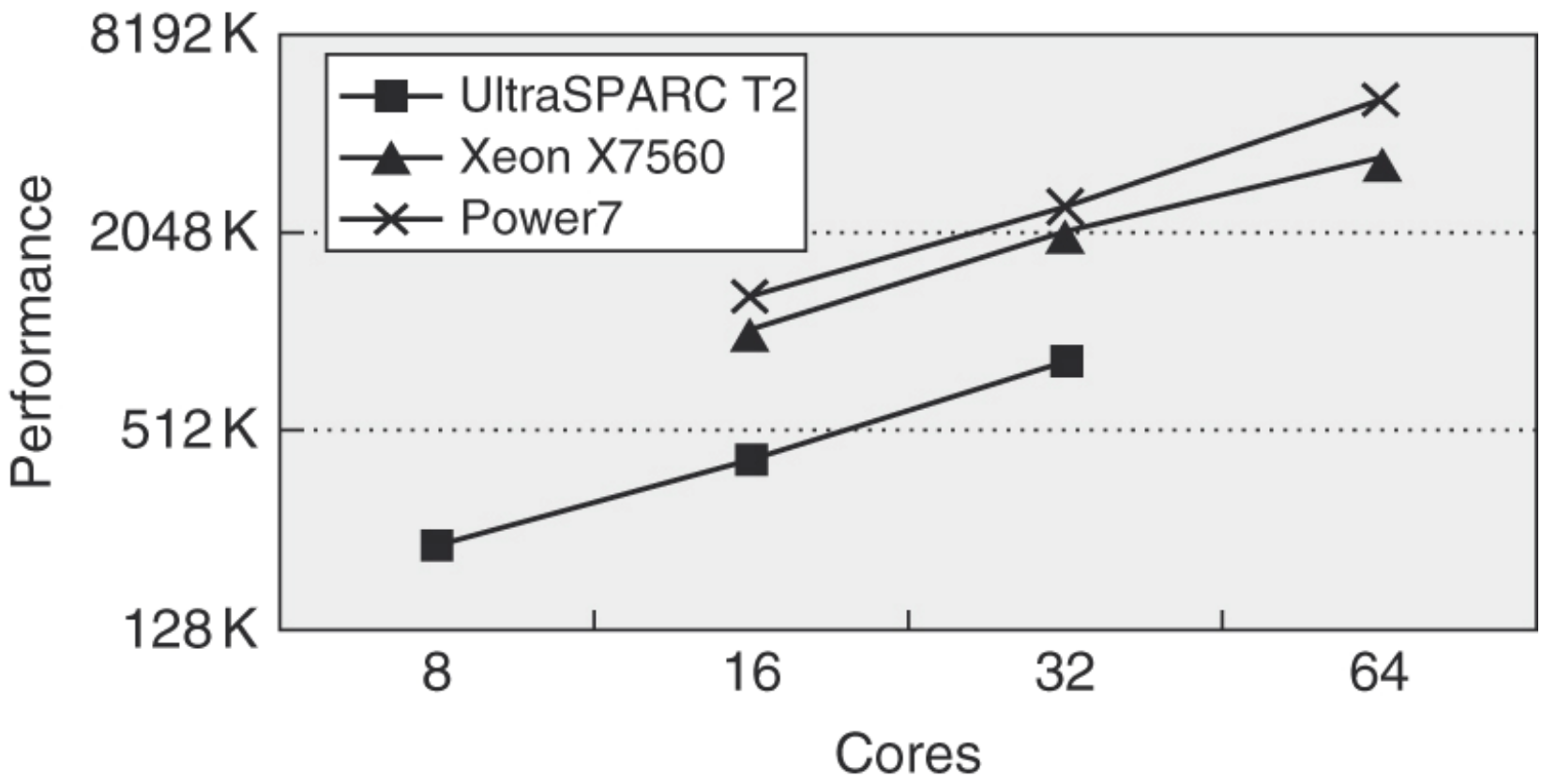


Figure 5.29 The performance on the SPECjbb2005 benchmark for three multicore processors as the number of processor chips is increased. Notice for this parallel benchmark, nearly linear speedup is achieved.

Intel i7: Energy efficiency vs SMT

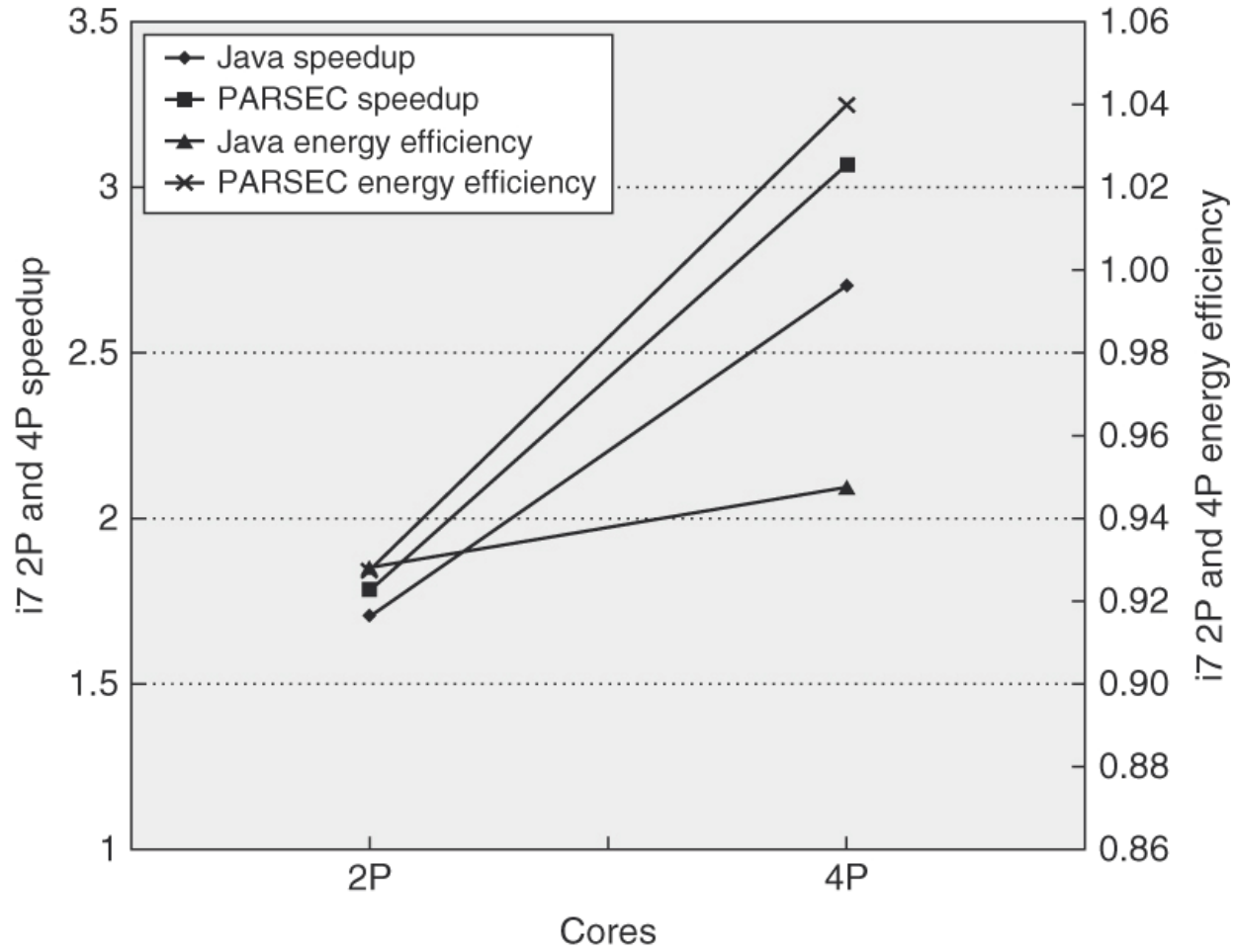


Figure 5.30 This chart shows the speedup for two- and four-core executions of the parallel Java and PARSEC workloads without SMT. These data were collected by Esmaeilzadeh et al. [2011] using the same setup as described in Chapter 3. Turbo Boost is turned off. The speedup and energy efficiency are summarized using harmonic mean, implying a workload where the total time spent running each 2p benchmark is equivalent.

Intel i7: processor count and SMT

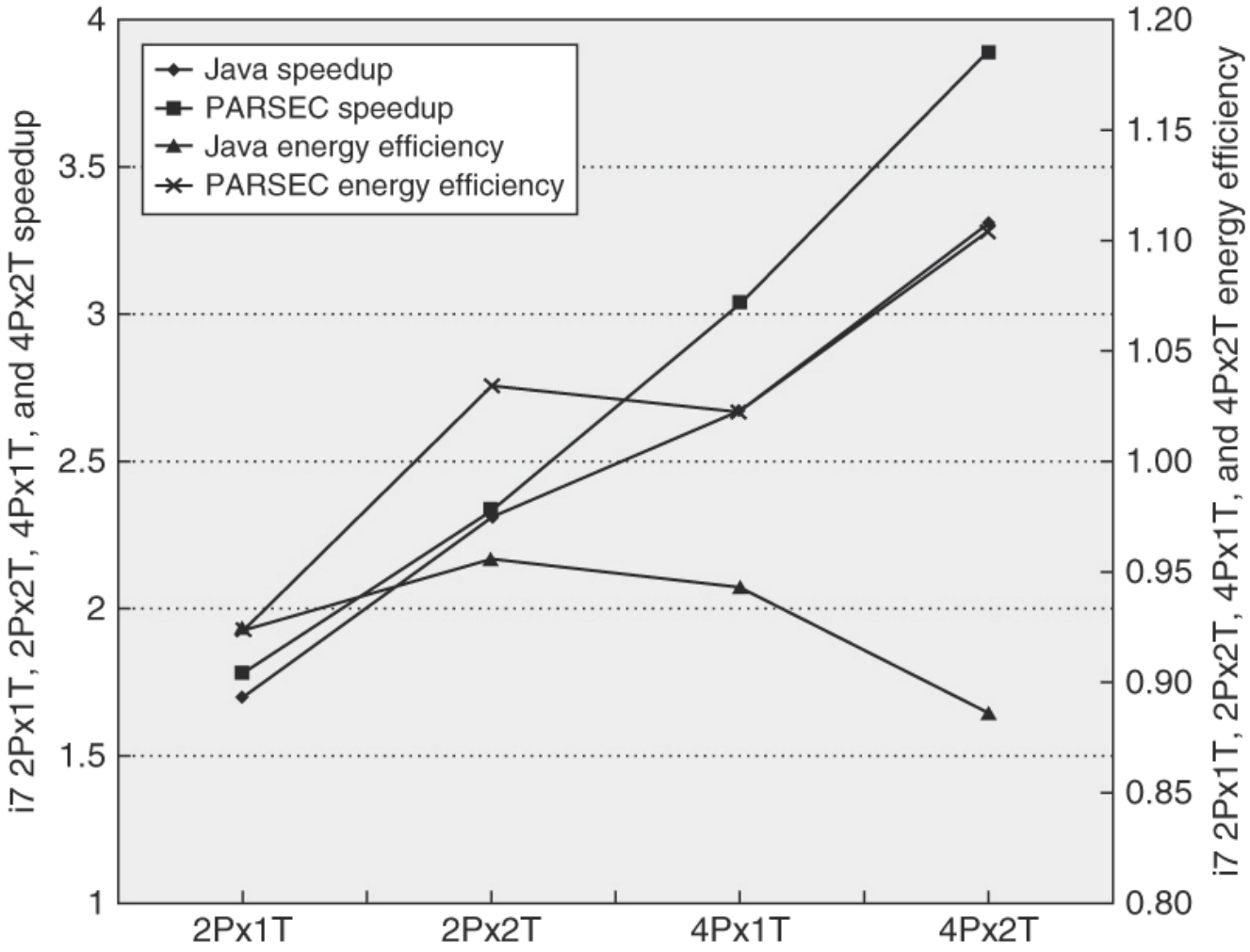


Figure 5.31 This chart shows the speedup for two- and four-core executions of the parallel Java and PARSEC workloads both with and without SMT. Remember that the results above vary in the number of threads from two to eight, and reflect both architectural effects and application characteristics. Harmonic mean is used to summarize results, as discussed in the caption of Figure 5.30.

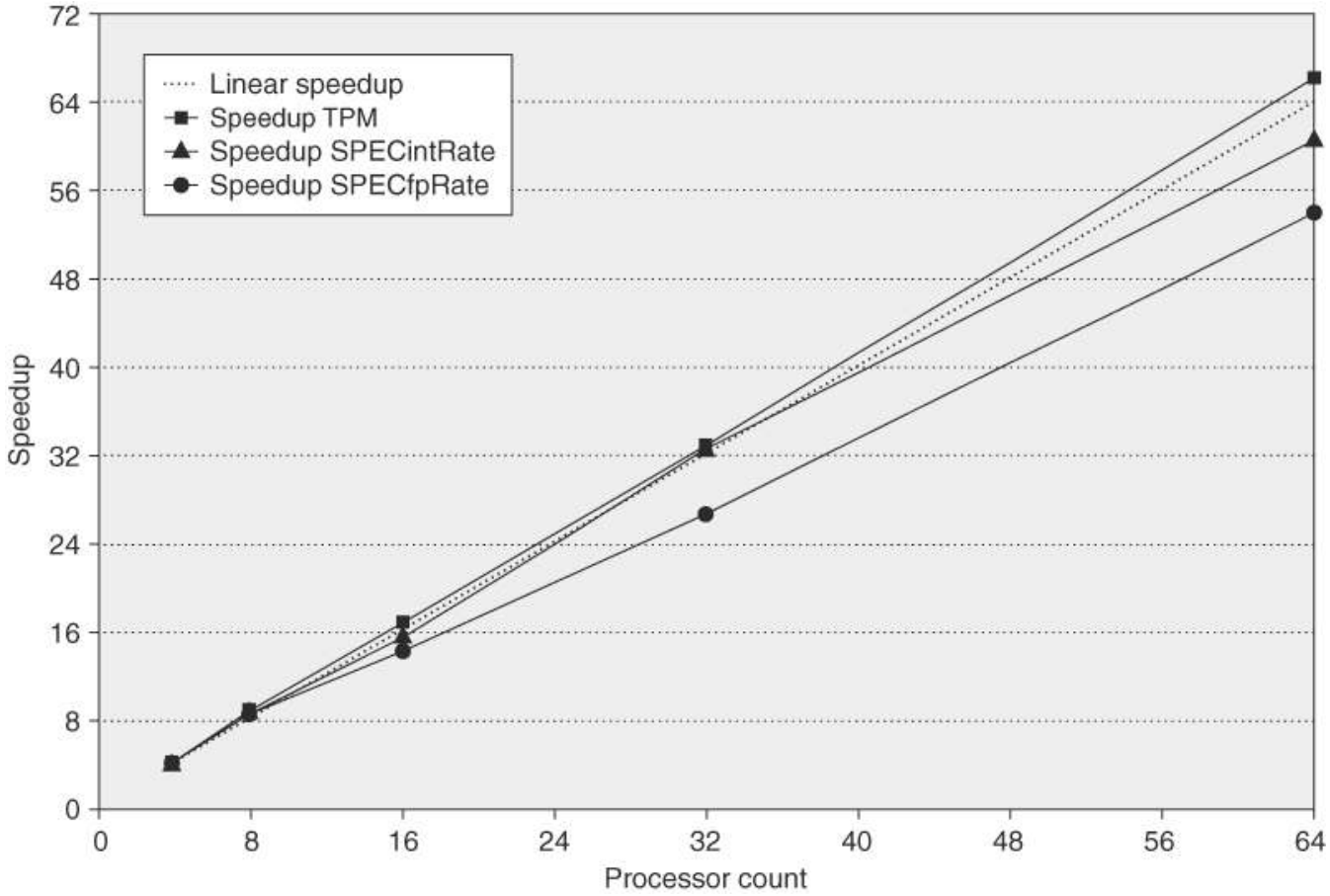


Figure 5.32 Speedup for three benchmarks on an IBM eServer p5 multiprocessor when configured with 4, 8, 16, 32, and 64 processors. The dashed line shows linear speedup.

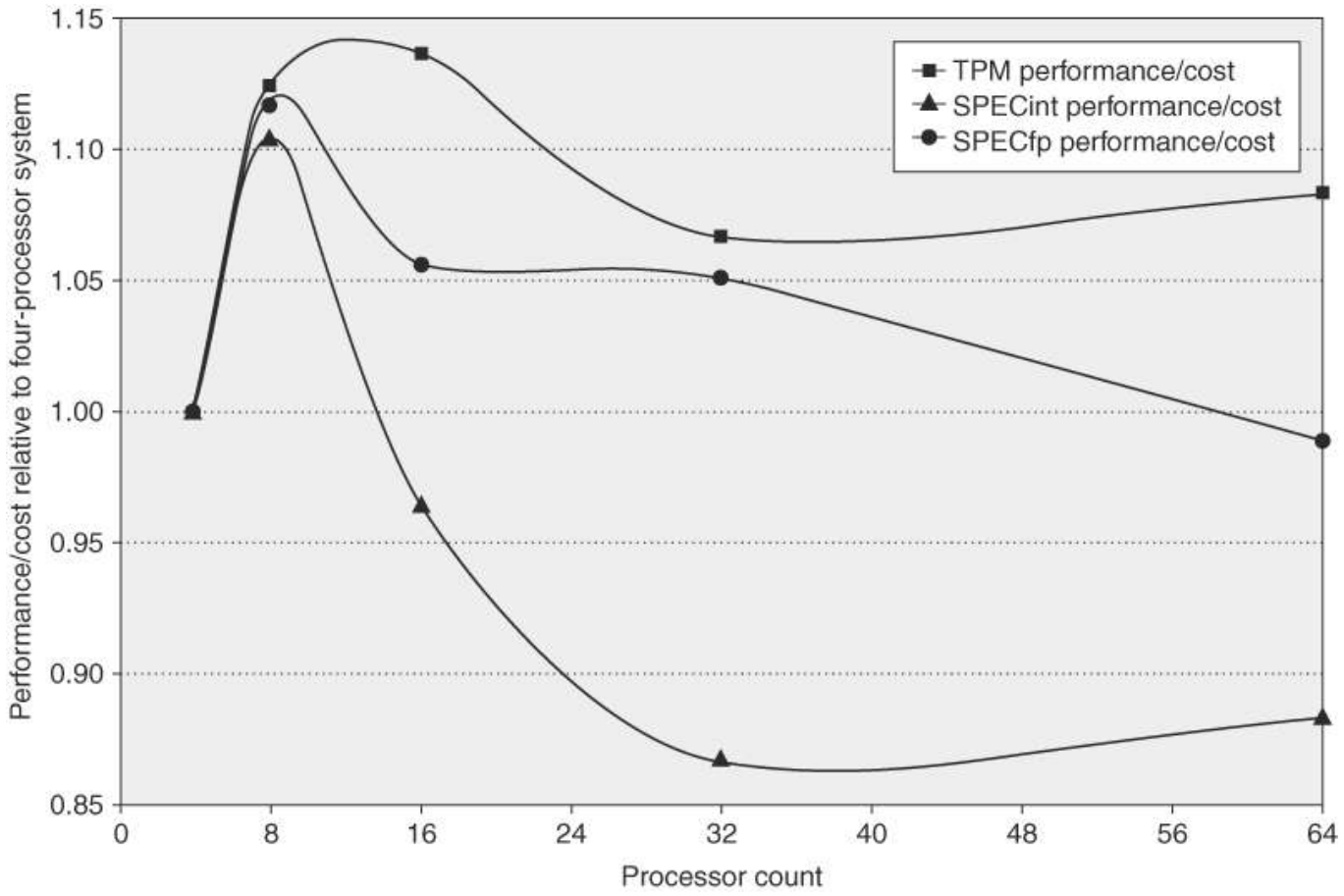


Figure 5.33 The performance/cost relative to a 4-processor system for three benchmarks run on an IBM eServer p5 multiprocessor containing from 4 to 64 processors shows that the larger processor counts can be as cost effective as the 4-processor configuration. For TPC-C the configurations are those used in the official runs, which means that disk and memory scale nearly linearly with processor count, and a 64-processor machine is approximately twice as expensive as a 32-processor version. In contrast, the disk and memory are scaled more slowly (although still faster than necessary to achieve the best SPECRate at 64 processors). In particular, the disk configurations go from one drive for the 4-processor version to four drives (140 GB) for the 64-processor version. Memory is scaled from 8 GB for the 4-processor system to 20 GB for the 64-p-rocessor system.

IC-UNICAMP

XXXXXXXXXX

IC-UNICAMP

XXXXXXXXXX

IC-UNICAMP

XXXXXXXXXX

IC-UNICAMP

XXXXXXXXXX