Review of Memory Hierarchy (Week#15b)

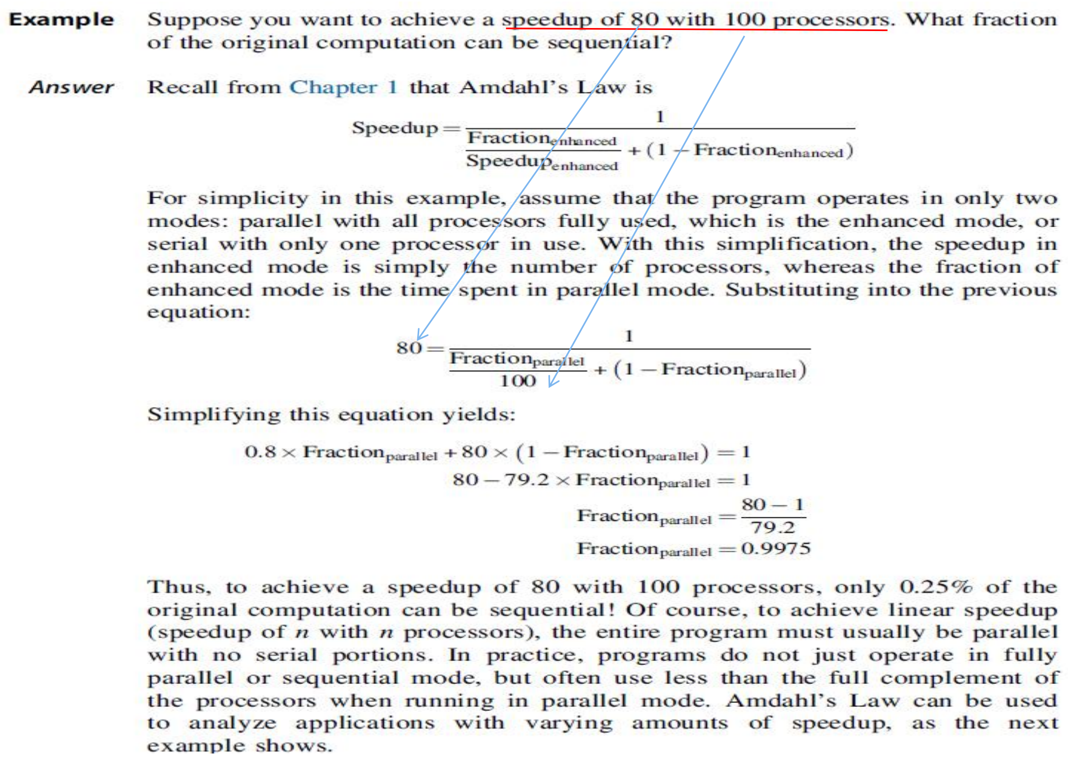
Multiprocessor Architecture: Issues and Approach

- ✓ **Distributed shared memory (DSM):** consists of multiprocessors with physically distributed memory, called distributed shared memory (DSM).
- ✓ To support larger processor counts, memory must be distributed among the processors rather than centralized;
- ✓ larger number of processors also raises the need for a high-bandwidth interconnect. Both directed networks (i.e., switches) and indirect networks (typically multidimensional meshes) are used.
- ✓ Distributing the memory among the nodes increases the bandwidth and reduces the latency to local memory.
- ✓ **NUMA Processor:** DSM multiprocessor is also called a NUMA (nonuniform memory access) because access time depends on location of a data word in memory.
- ✓ disadvantage of DSM: communicating data among processors becomes complex and requires more effort to take advantage of increased memory bandwidth
- ✓ Multicore-based multiprocessors with few processor chips use distributed memory,
- ✓ In both SMP and DSM architectures, communication among threads occurs through a shared address space

Challenges of Parallel Proceesing

- ✓ Application of multiprocessors ranges from running independent tasks with essentially no communication to running parallel programs where threads must communicate to complete the task.
- ✓ key bottleneck when scaling to a larger processor counts (approaching 100 or more), often all aspects of the software and hardware need attention.
  1. first hurdle has to do with the limited parallelism available in programs,
  2. second arises from the relatively high cost of communications.

Challenges of Parallel Proceesing



Example Suppose you want to achieve a speedup of 80 with 100 processors. What fraction of the original computation can be sequential?

Answer Recall from Chapter 1 that Amdahl's Law is

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

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

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

Simplifying this equation yields:

$$0.8 \times \text{Fraction}_{\text{parallel}} + 80 \times (1 - \text{Fraction}_{\text{parallel}}) = 1$$
$$80 - 79.2 \times \text{Fraction}_{\text{parallel}} = 1$$
$$\text{Fraction}_{\text{parallel}} = \frac{80 - 1}{79.2}$$
$$\text{Fraction}_{\text{parallel}} = 0.9975$$

Thus, to achieve a speedup of 80 with 100 processors, only 0.25% of the original computation can be sequential! Of course, to achieve linear speedup (speedup of n with n processors), the entire program must usually be parallel with no serial portions. In practice, programs do not just operate in fully parallel or sequential mode, but often use less than the full complement of the processors when running in parallel mode. Amdahl's Law can be used to analyze applications with varying amounts of speedup, as the next example shows.

Challenges of Parallel Processing

Example Suppose we have an application running on a 100-processor multiprocessor, and assume that application can use 1, 50, or 100 processors. If we assume that 95% of the time we can use all 100 processors, how much of the remaining 5% of the execution time must employ 50 processors if we want a speedup of 80?

Answer We use Amdahl's Law with more terms:

$$\text{Speedup} = \frac{1}{\frac{\text{Fraction}_{100}}{\text{Speedup}_{100}} + \frac{\text{Fraction}_{50}}{\text{Speedup}_{50}} + (1 - \text{Fraction}_{100} - \text{Fraction}_{50})}$$

Substituting in:

$$80 = \frac{1}{\frac{0.95}{100} + \frac{\text{Fraction}_{50}}{50} + (1 - 0.95 - \text{Fraction}_{80})}$$

Simplifying:

$$0.76 + 1.6 \times \text{Fraction}_{50} + 4.0 - 80 \times \text{Fraction}_{50} = 1$$
$$4.76 - 78.4 \times \text{Fraction}_{50} = 1$$
$$\text{Fraction}_{50} = 0.048$$

If 95% of an application can use 100 processors perfectly, to get a speedup of 80, 4.8% of the remaining time must be spent using 50 processors and only 0.2% can be serial!

Challenges of Parallel Processing

Example Suppose we have an application running on a 32-processor multiprocessor that has a 100 ns delay to handle a 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 obviously optimistic. Processors are stalled on a remote request, and the processor clock rate is 4 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

$$\frac{\text{Remote access cost}}{\text{Cycle time}} = \frac{100\,\text{ns}}{0.25\,\text{ns}} = 400\,\text{cycles}$$

Therefore we can compute the CPI:

$$\begin{aligned}\text{CPI} &= 0.5 + 0.20\% \times 400 \\ &= 1.3\end{aligned}$$

The multiprocessor with all local references is 1.3/0.5 = 2.6 times faster. In practice, the performance analysis is much more complex because 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 worse because contention caused by many references trying to use the global interconnect can lead to increased delays, or the access time might be better if memory were distributed and the access was to the local memory.

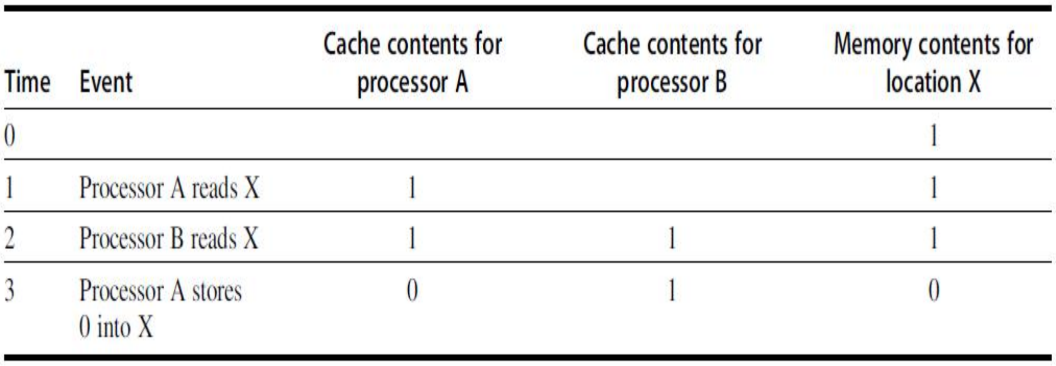
This problem could have also been analyzed using Amdahl's Law, an exercise we leave to the reader.

Centralized Shared-Memory Architectures

- ✓ Use of large, multilevel caches can reduce the memory bandwidth demands of a processor that motivates centralized memory multiprocessors.
- ✓ Accessing a chip's local memory whether for an I/O operation or for an access from another chip requires going through the chip that "owns" that memory.
- ✓ Symmetric shared-memory machines usually support the caching of both shared and private data.
- ✓ Private data are used by a single processor, while shared data are used by multiple processors, essentially providing communication among the processors through reads and writes of the shared data.
- ✓ When a private item is cached, its location is migrated to the cache, reducing the average access time as well as the memory bandwidth required.
- ✓ When shared data are cached, the shared value may be replicated in multiple caches.

What Is Multiprocessor Cache Coherence?

- ✓ Memory held by two different processors is through their individual caches, the processors could end up seeing different values for the same memory location, as Figure 5.3 illustrates.



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

Figure 5.3 The cache coherence problem for a single memory location (X), read and written by two processors (A and B). We initially assume that neither cache contains the variable and that X has the value 1. We also assume a write-through cache; a write-back cache adds some additional but similar complications. After the value of X has been written by A, A's cache and the memory both contain the new value, but B's cache does not, and if B reads the value of X it will receive 1!

What Is Multiprocessor Cache Coherence?

- ✓ A memory system is coherent if any read of a data item returns the most recently written value of that data item.
- ✓ **Coherence:** defines what values can be returned by a read.
- ✓ **Consistency:** determines when a written value will be returned by a read. A memory system is coherent if:

1. 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.

2.Read by a processor to location X that follows a write by another processor to X returns the written value if the read and write are sufficiently separated in time and no other writes to X occur between two accesses.

3. Writes to the same location are serialized;For example, if the values 1 and then 2 are written to a location, processors can never read the value of the location as 2 and then later read it as 1.

Write Serialization: to ensure that all writes to the same location are seen in the same order; this property is called write serialization.

Coherence defines the behavior of reads and writes to the same memory location, while consistency defines the behavior of reads and writes with respect to accesses to other memory locations.

Basic Schemes for Enforcing Coherence

- ✓ A program running on multiple processors have copies of same data in several caches.
- ✓ coherent multiprocessor caches provide both migration and replication of shared data items.

Coherent caches provide migration:

- ✓ a data item can be moved to a local cache . This migration reduces both the latency to access a shared data item that is allocated remotely and the bandwidth demand on the shared memory.
- ✓ Caches make a copy of the data item in the local cache, coherent caches also provide replication for shared data that are being read simultaneously.

Replication :

- ✓ reduces both latency of access and contention for a read shared data item.
- ✓ Supporting this migration and replication is critical to performance in accessing shared data.
- ✓ multiprocessors adopt a hardware solution by introducing a protocol to maintain coherent caches are called cache coherence protocols.

Basic Schemes for Enforcing Coherence

Cache coherence protocol

- ✓ tracking the state of any sharing of a data block.
- ✓ state of any cache block is kept using status bits associated with the block, similar to the valid and dirty bits kept in a uniprocessor cache.

There are two classes of protocols in use, each of which uses different techniques to track the sharing status:

Directory based:

- ✓ sharing status of a particular block of physical memory is kept in one location, called the directory.
- ✓ There are two very different types of directory-based cache coherence.
- ✓ In an SMP, one centralized directory is used associated with memory or some other single serialization point such as outermost cache in a multicore.
- ✓ In a Distributed Shared Memory, it makes no sense to have a single directory because that would create a single point of contention and make it difficult to scale to many multicore chips with eight or more cores.

Basic Schemes for Enforcing Coherence

Snooping:

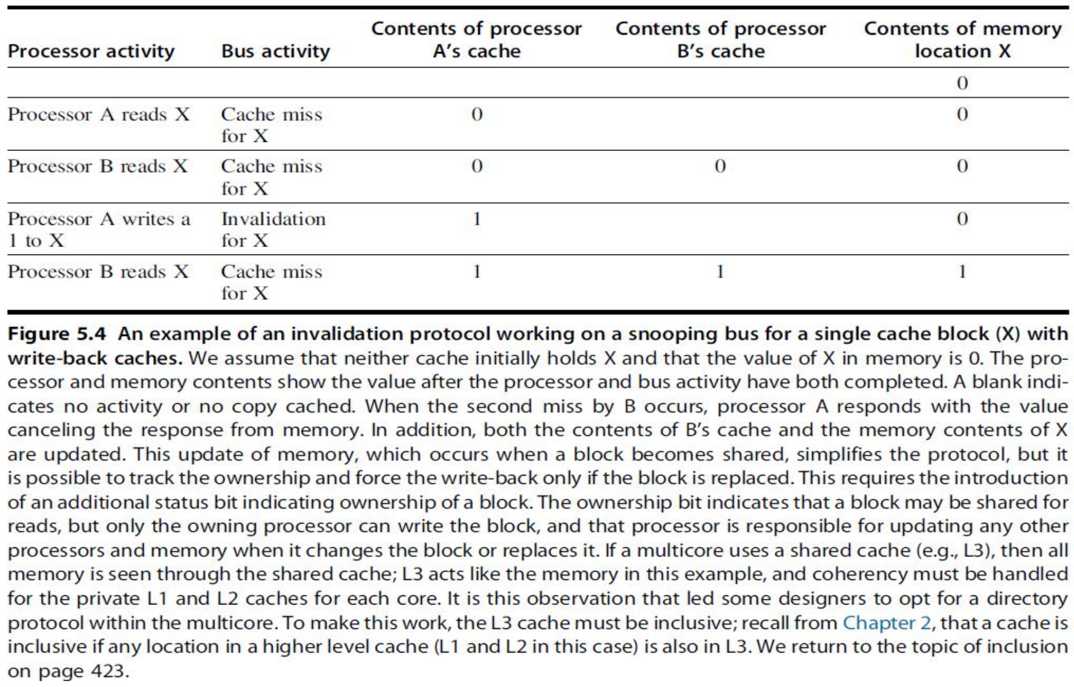
- ✓ every cache that has a copy of the data from a block of physical memory could track the sharing status of the block.
- ✓ In an Symmetric Multiprocessor, caches are typically all accessible via broadcast medium (e.g., a bus) and cache controllers monitor or snoop on the medium to determine whether they have a copy of a block that is requested on a bus or switch access.
- ✓ Snooping can also be used as the coherence protocol for a multichip multiprocessor, and some designs support a snooping protocol on top of a directory protocol within each multicore.

Snooping Coherence Protocols:

Write invalidate protocol: to ensure that a processor has exclusive access to a data item before writing that item. This style of protocol is called a write invalidate protocol because it invalidates other copies on a write.

Basic Schemes for Enforcing Coherence

Figure 5.4 shows an example of an invalidation protocol with write-back caches in action.



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 status bit indicating ownership of a block. The ownership bit indicates that a block may be shared for reads, but only the owning processor can write the block, and that 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 caches 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; recall from Chapter 2, that a cache is inclusive if any location in a higher level cache (L1 and L2 in this case) is also in L3. We return to the topic of inclusion on page 423.

Basic Schemes for Enforcing Coherence

Invalidation protocol with write-back :

- ✓ consider a write followed by a read by another processor: write requires exclusive access, any copy held by the reading processor must be invalidated.
- ✓ when the read occurs, it misses in the cache and is forced to fetch a new copy of the data.
- ✓ For a write, require that the writing processor has exclusive access preventing any other processor from being able to write simultaneously.
- ✓ If two processors attempt to write the same data simultaneously, one of them wins the race causing the other processor's copy to be invalidated.
- ✓ For other processor to complete its write, it must obtain a new copy of the data, which must now contain the updated value. This protocol enforces write serialization.
- ✓ Alternative to an invalidate protocol is to update all the cached copies of a data item when that item is written. This type of protocol is called a write update or write broadcast protocol.
- ✓ Write update protocol must broadcast all writes to shared cache lines, it consumes more bandwidth.

Basic Implementation Techniques

- ✓ All coherence schemes require some method of serializing accesses to the same cache block, either by serializing access to the communication medium or to another shared structure.
- ✓ In a write through cache, it is easy to find the recent value of a data item because all written data are always sent to the memory, from which the most recent value of a data item can always be fetched.
- ✓ In a write-back cache, problem to find most recent data value is harder because the most recent value of a data item can be in a private cache rather in shared cache.
- ✓ Write-back caches can use the same snooping scheme both for cache misses and for writes: each processor snoops every address placed on shared bus.
- ✓ If a processor finds a dirty copy of the requested cache block, it provides that cache block in response to the read request and causes the memory (or L3) access to be aborted.
- ✓ Retrieving cache block from another processor's private cache (L1 or L2) take longer than retrieving it from L3.

Basic Implementation Techniques

- ✓ write-back caches generate lower requirements for memory bandwidth and support larger numbers of faster processors.
- ✓ Multicore processors use write-back at outermost levels of cache
- ✓ Cache tags can be used to implement the process of snooping and valid bit for each block makes invalidation easy to implement.
- ✓ Read misses generated by an invalidation or other event rely on snooping capability.
- ✓ For writes, if there are no other cached copies then write does not need to be placed on the bus in a write-back cache.
- ✓ Not sending the write reduces both time to write and required bandwidth.
- ✓ To track whether or not a cache block is shared, extra state bit is associated with each cache block as a valid bit and a dirty bit.

Basic Implementation Techniques

Inclusions:

- ✓ Every bus transaction must check cache-address tags, which interfere with processor cache accesses.
- ✓ to reduce this interference is to duplicate the tags and have snoop accesses directed to the duplicate tags.
- ✓ Another approach is to use a directory at the shared L3 cache;
- ✓ directory indicates whether a given block is shared and possibly which cores have copies.
- ✓ invalidates can be directed only to those caches with copies of cache block.
- ✓ This requires L3 must have a copy of data item in L1 or L2, a property called inclusion.

An Example Protocol

- ✓ A snooping coherence protocol is implemented by incorporating a finite state controller in each core.
- ✓ Controller responds to requests from the processor and from the bus changing the state of the selected cache block using the bus to access data or to invalidate it.
- ✓ Single controller allows multiple operations to distinct blocks to proceed in interleaved fashion.
- ✓ Interconnection network that supports a broadcast to all coherence controllers and their associated private caches can be used to implement snooping.
- ✓ protocol has three states: invalid, shared, and modified.
- ✓ Shared state indicates that the block in the private cache is potentially shared.
- ✓ Modified state indicates that the block has been updated in the private cache.

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, because 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 read shared data: place cache block on bus, write-back block, 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.

An Example Protocol

- ✓ Figure 5.6 shows write invalidate, cache coherence protocol for a private write-back cache showing the states and state transitions for each block in the cache.
- ✓ cache states are shown in circles, with any access permitted by the local processor without a state transition shown in parentheses under the name of the state.
- ✓ a read miss to a block in the shared state is a miss for that cache block but for a different address.

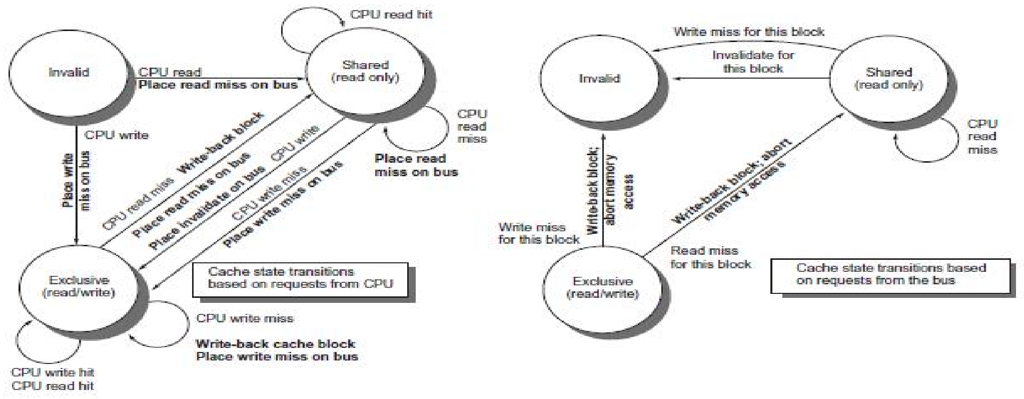


Figure 5.6 A write invalidate, cache coherence protocol for a private write-back cache showing the states and

An Example Protocol

- ✓ left side of diagram shows state transitions based on actions of the processor associated with this cache; the right side shows transitions based on operations on the bus.
- ✓ A read miss and a write miss in exclusive state occur when address requested by processor does not match the address in local cache block.
- ✓ To write a block in the shared state generates an invalidate.
- ✓ Whenever a bus transaction occurs, all private caches that contain the cache block take the action dictated by right half of diagram.

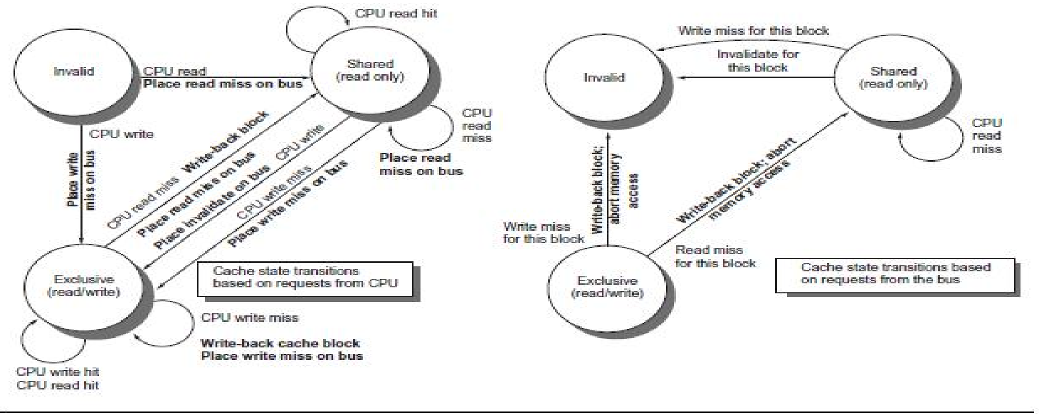


Figure 5.6 A write invalidate, cache coherence protocol for a private write-back cache showing the states and

An Example Protocol

- ✓ protocol assumes that shared cache provides data on a read miss for a block that is clean in all local caches.
- ✓ variations on invalidate protocols, including the introduction of the exclusive unmodified state, as to whether a processor or memory provides data on a miss.
- ✓ In a multicore chip, shared cache act as equal to memory

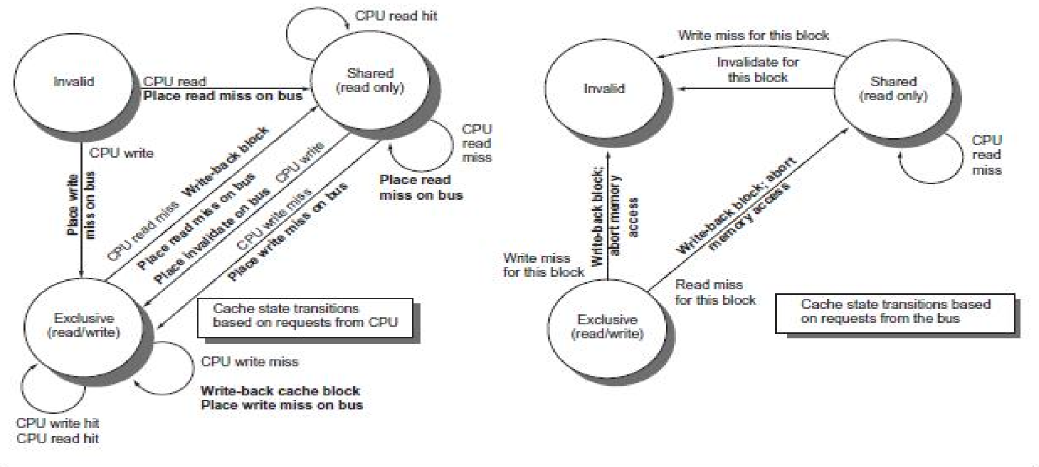


Figure 5.6 A write invalidate, cache coherence protocol for a private write-back cache showing the states and

An Example Protocol

- ✓ Figure 5.7 handle read and write misses on the bus, are snooping component of protocol.
- ✓ Memory block in the shared state is always up to date in the outer shared cache (L2 or L3 or memory)
- ✓ all accesses from the cores go through that level.
- ✓ protocol assumes that operations are atomic—that is, an operation can be done in such a way that no intervening operation can occur.
- ✓ Non atomic actions introduce the possibility that the protocol can reach a deadlock state.

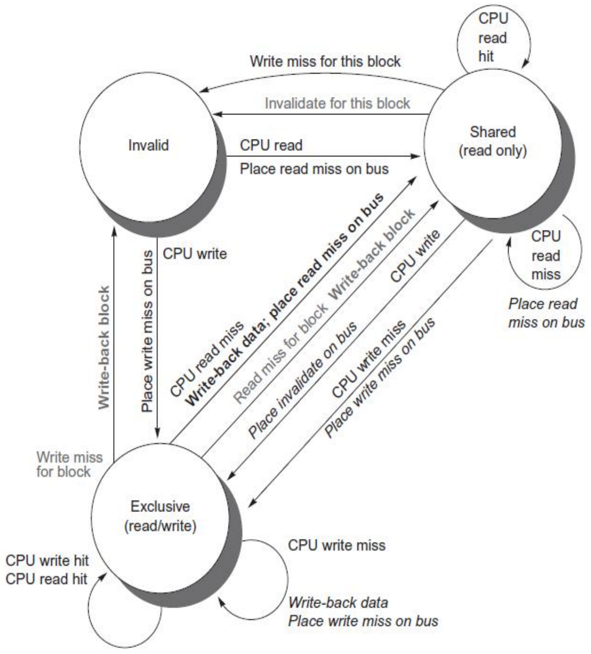


Figure 5.7 Cache coherence state diagram with the state transitions induced by the local processor shown in *black* and by the bus activities shown in *gray*. As in Figure 5.6, the activities on a transition are shown in *bold*.

Extensions to the Basic Coherence Protocol

First Extension:

- ✓ MESI (Modified, Exclusive, Shared, and Invalid) adds the state Exclusive to the basic MSI (Modified, Shared, and Invalid) protocol, yielding four states.
- ✓ exclusive state indicates that a cache block is resident in only a single cache but is clean.
- ✓ 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.
- ✓ when a read miss to a block in the E state occurs, the block must be changed to the S state to maintain coherence.
- ✓ if another processor issues a read miss, the state is changed from exclusive to shared.
- ✓ advantage : a subsequent write to a block in exclusive state by same core need not acquire bus access or generate invalidate, since block is exclusively in local cache;processor changes state to modified.
- ✓ This state is added by using dirty bit to indicate that a bock is modified.
- ✓ Intel i7 uses a variant of MESI protocol called MESIF, which adds a state (Forward) to designate sharing processor should respond to a request. It is designed to enhance performance in distributed memory organizations.

Extensions to the Basic Coherence Protocol

Second Extension:

- ✓ MOESI adds state Owned to MESI protocol to indicate that the associated block is owned by that cache.
- ✓ In MESI protocols, when there is an attempt to share a block in the Modified state, state is changed to Shared and block must be written back to memory.
- ✓ In MOESI protocol, block can be changed from the Modified to Owned state in the original cache without writing it to memory.
- ✓ Other caches sharing the block, keep the block in the Shared state;
- ✓ O state is hold by original cache indicates that the main memory copy is out of date and designated cache is the owner.
- ✓ owner of the block must supply it on a miss since memory is not up to date and must write the block back to memory if it is replaced.
- ✓ AMD Opteron processor family uses the MOESI protocol

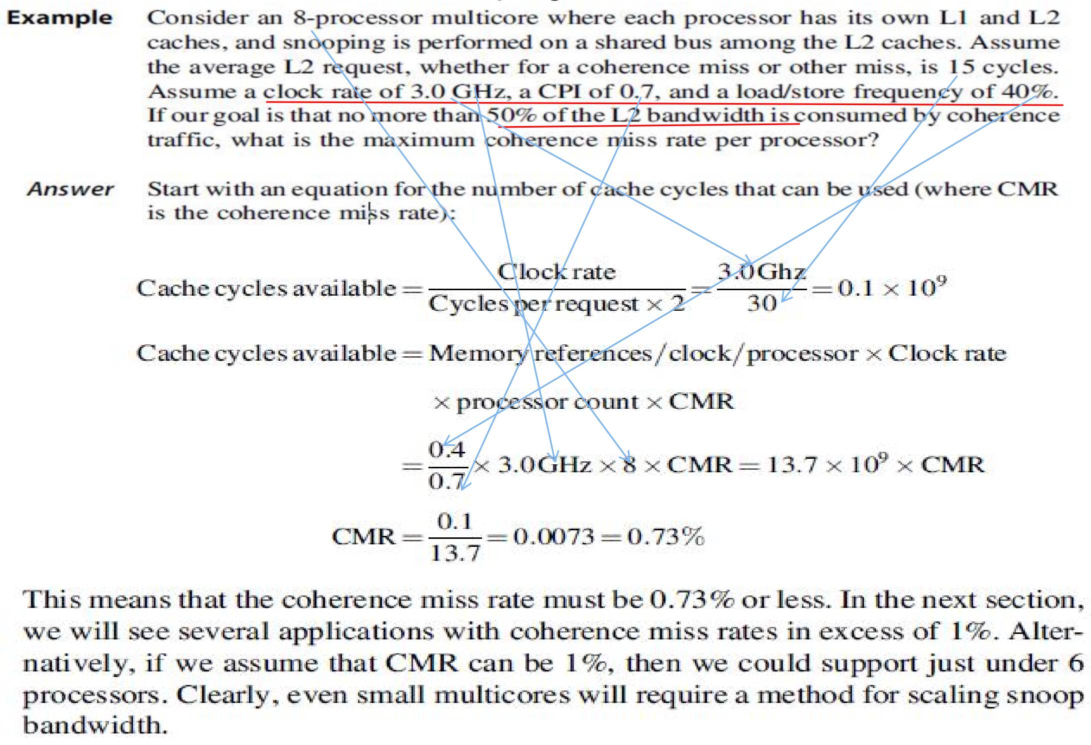
Limitations in Symmetric Shared-Memory Multiprocessors and Snooping Protocols

Multicore chips use three different approaches:

1. IBM Power8 has up to 12 processors in a single multicore, uses 8 parallel buses that connect distributed L3 caches and up to 8 separate memory channels.
2. Xeon E7 uses three rings to connect up to 32 processors, a distributed L3 cache, and two or four memory channels.
3. Fujitsu SPARC64 X+ uses a crossbar to connect a shared L2 to up to 16 cores and multiple memory channels.

- ✓ SPARC64 X+ is a symmetric organization with uniform access time.
- ✓ Power8 has nonuniform access time for both L3 and memory.
- ✓ access time differences among memory addresses within a single Power8 multicore are not large
- ✓ Xeon E7 can operate as if access times were uniform;
- ✓ Snooping bandwidth at the caches can also become a problem because every cache must examine every miss, and having additional interconnection bandwidth only pushes the problem to the cache.

Limitations in Symmetric Shared-Memory Multiprocessors and Snooping Protocols



Example Consider an 8-processor multicore where each processor has its own L1 and L2 caches, and snooping is performed on a shared bus among the L2 caches. Assume the average L2 request, whether for a coherence miss or other miss, is 15 cycles. Assume a clock rate of 3.0 GHz, a CPI of 0.7, and a load/store frequency of 40%. If our goal is that no more than 50% of the L2 bandwidth is consumed by coherence traffic, what is the maximum coherence miss rate per processor?

Answer Start with an equation for the number of cache cycles that can be used (where CMR is the coherence miss rate):

$$\text{Cache cycles available} = \frac{\text{Clock rate}}{\text{Cycles per request} \times 2} = \frac{3.0\,\text{Ghz}}{30} = 0.1 \times 10^9$$

$$\text{Cache cycles available} = \text{Memory references/clock/processor} \times \text{Clock rate}$$
$$\times \text{ processor count} \times \text{CMR}$$
$$= \frac{0.4}{0.7} \times 3.0\,\text{GHz} \times 8 \times \text{CMR} = 13.7 \times 10^9 \times \text{CMR}$$
$$\text{CMR} = \frac{0.1}{13.7} = 0.0073 = 0.73\%$$

This means that the coherence miss rate must be 0.73% or less. In the next section, we will see several applications with coherence miss rates in excess of 1%. Alternatively, if we assume that CMR can be 1%, then we could support just under 6 processors. Clearly, even small multicores will require a method for scaling snoop bandwidth.

Limitations in Symmetric Shared-Memory Multiprocessors and Snooping Protocols

There are several techniques for increasing snoop bandwidth:

1. tags can be duplicated. This doubles the effective cache-level snoop bandwidth. If we assume that half coherence requests do not hit on a snoop request and cost of snoop request is only 10 cycles then average cost of a CMR to 12.5 cycles.

2. If outermost cache L3 is shared, distribute that cache so that each processor has a portion of the memory and handles snoops for that portion of the address space. This approach used by IBM 12-core Power8, leads to a NUCA design, but scales snoop bandwidth at L3. If there is a snoop hit in L3, then still broadcast to all L2 caches

3. place a directory at the level of shared cache (L3). L3 acts as a filter on snoop requests and must be inclusive. use of a directory at L3 means that we need not snoop or broadcast to all L2 caches, but only those that the directory indicates may have a copy of block. L3 may be distributed, associated directory entries may also be distributed. This approach is used in Intel Xeon E7 series, which supports from 8 to 32 cores.

Limitations in Symmetric Shared-Memory Multiprocessors and Snooping Protocols

Figure 5.8 shows how a multicore with a distributed cache system.

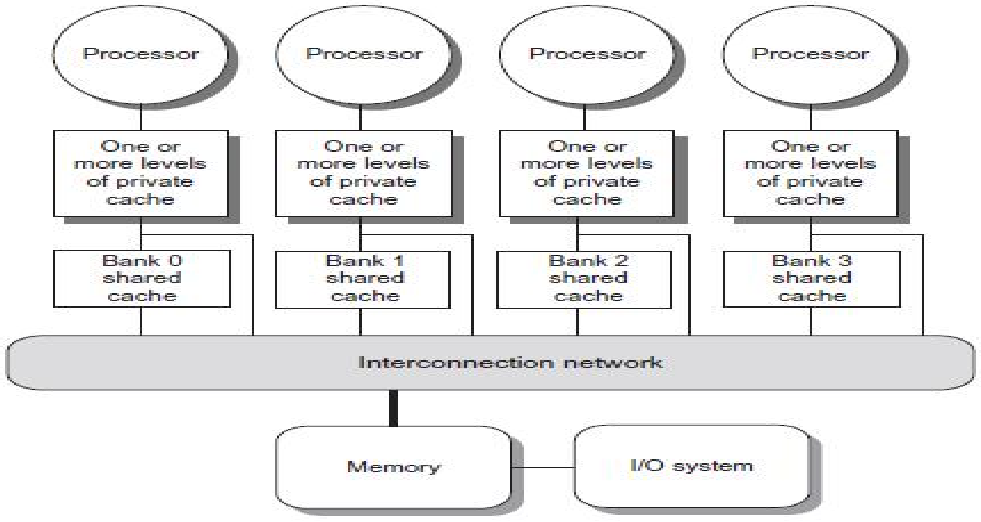


Figure 5.8 A single-chip multicore with a distributed cache. In current designs, the distributed shared cache is usually L3, and levels L1 and L2 are private. There are typically multiple memory channels (2–8 in today's designs). This design is NUCA, since the access time to L3 portions varies with faster access time for the directly attached core. Because it is NUCA, it is also NUMA.

Implementing Snooping Cache Coherence

- ✓ In early designs, a single line was used to signal when all necessary invalidates had been received and were being processed.
- ✓ processor that generated the miss could release the bus, knowing that any required actions would be completed before any activity related to the next miss.
- ✓ ensure that two processors that attempt to write the same block at the same time, a situation which is called a race, are strictly ordered: one write is processed and precedes before the next is begun.
- ✓ In a multicore using multiple buses, races can be eliminated if each block of memory is associated with only a single bus, ensuring that two attempts to access the same block must be serialized.
- ✓ It is possible to combine snooping and directories and several designs use snooping within a multicore and directories among multiple chips or a combination of directories at one cache level and snooping at another level.

Performance of Symmetric Shared-Memory Multiprocessors

- ✓ Misses that arise from interprocessor communication are called coherence misses, can be broken into two separate sources.

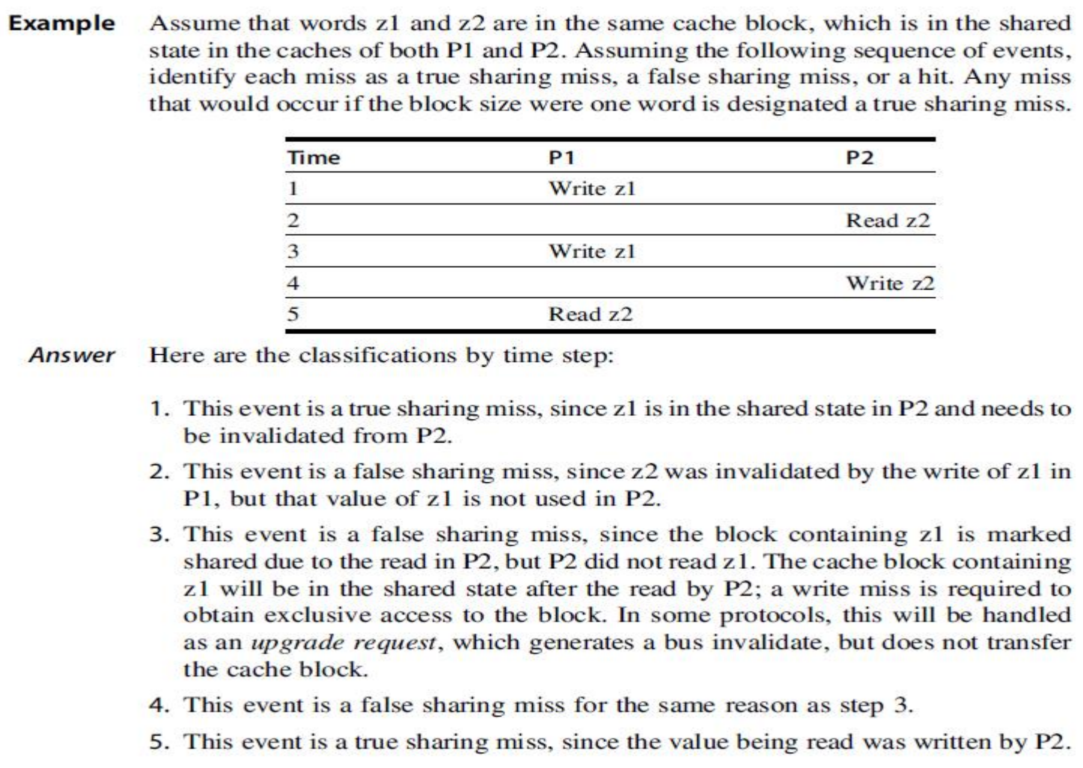
First Source:

- ✓ In an invalidation-based protocol, first write by a processor to a shared cache block causes an invalidation to establish ownership of that block.
- ✓ when another processor attempts to read a modified word in that cache block, a miss occurs and the resultant block is transferred.
- ✓ Both these misses are classified as true sharing misses because they directly arise from sharing of data among processors.

Second Source:

- ✓ **False sharing:** arises from the use of an invalidation based coherence algorithm with a single valid bit per cache block.
- ✓ False sharing occurs when a block is invalidated.
- ✓ If word written into is used by processor that received the invalidate, then reference was a true sharing reference and caused a miss independent of block size.
- ✓ If word write and read are different and invalidation does not cause a new value to be communicated but causes cache miss, then it is a false sharing miss. In false sharing miss, block is shared, but no word in cache is shared.

Performance of Symmetric Shared-Memory Multiprocessors



Example Assume that words z1 and z2 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 z1	
2		Read z2
3	Write z1	
4		Write z2
5	Read z2	

Answer Here are the classifications by time step:

1. This event is a true sharing miss, since z1 is in the shared state in P2 and needs to be invalidated from P2.
2. This event is a false sharing miss, since z2 was invalidated by the write of z1 in P1, but that value of z1 is not used in P2.
3. This event is a false sharing miss, since the block containing z1 is marked shared due to the read in P2, but P2 did not read z1. The cache block containing z1 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.

A Commercial Workload

Cache level	Characteristic	Alpha 21164	Intel i7
L1	Size	8 KB I/8 KB D	32 KB I/32 KB D
	Associativity	Direct-mapped	8-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 MiB (total 8 MiB unshared)	2 MiB per core (8 MiB total shared)
	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. Both systems have a high penalty (125 cycles or more) for a transfer required from a private cache. A key difference is that L3 is shared in the i7 versus four separate, unshared caches in the Alpha server.

A Commercial Workload

Figure 5.10 shows effect of increasing the cache size, using two-way set associative caches, which reduces large number of conflict misses. Execution time is improved as L3 cache grows because of the reduction in L3 misses

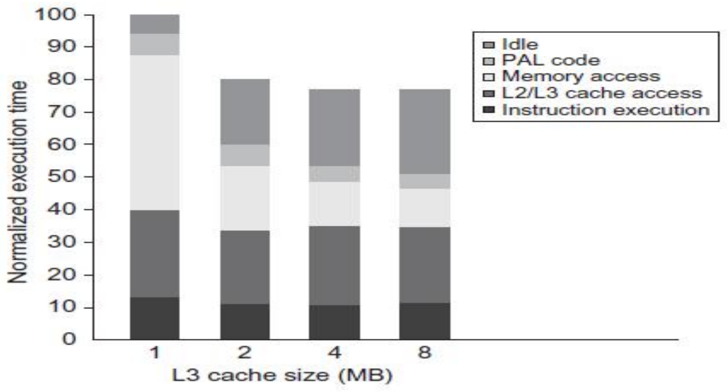


Figure 5.10 The relative performance of the OLTP workload as the size of the L3 cache, which is set as two-way set associative, grows from 1 to 8 MiB. The idle time also grows as cache size is increased, reducing some of the performance gains. This growth occurs because, with fewer memory system stalls, 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.

A Commercial Workload

Figure 5.11 shows these data, displaying the number of memory access cycles contributed per instruction from five sources.

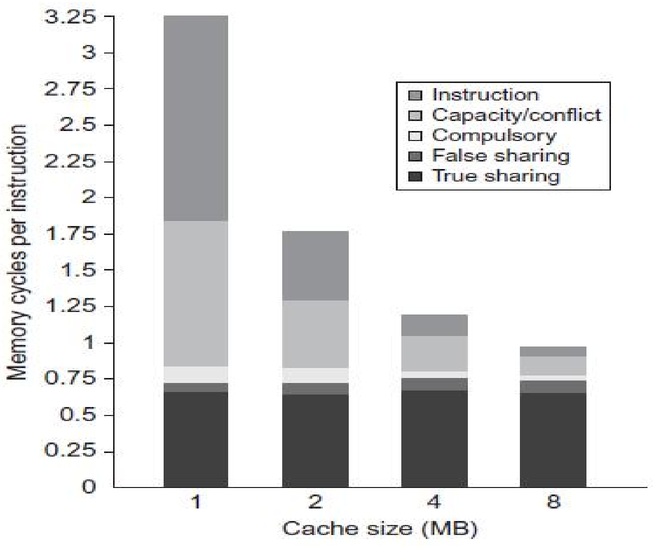


Figure 5.11 The contributing causes of memory access cycle shift as the cache size is increased. The L3 cache is simulated as two-way set associative.

A Commercial Workload

Figure 5.12 shows these data assuming a base configuration with a 2 MiB, two-wav set associative L3 cache.

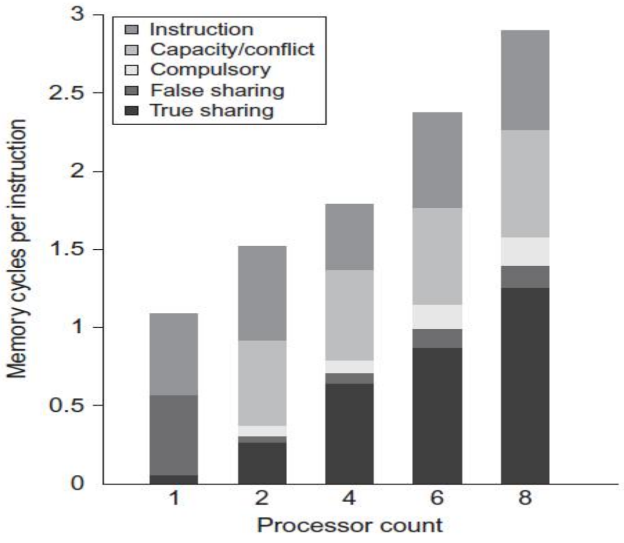


Figure 5.12 The contribution to memory access cycles increases as processor count increases primarily because of increased true sharing. The compulsory misses slightly increase because each processor must now handle more compulsory misses.

A Commercial Workload

Figure 5.13 shows the number of misses per 1000 instructions as block size is increased from 32 to 256 bytes.

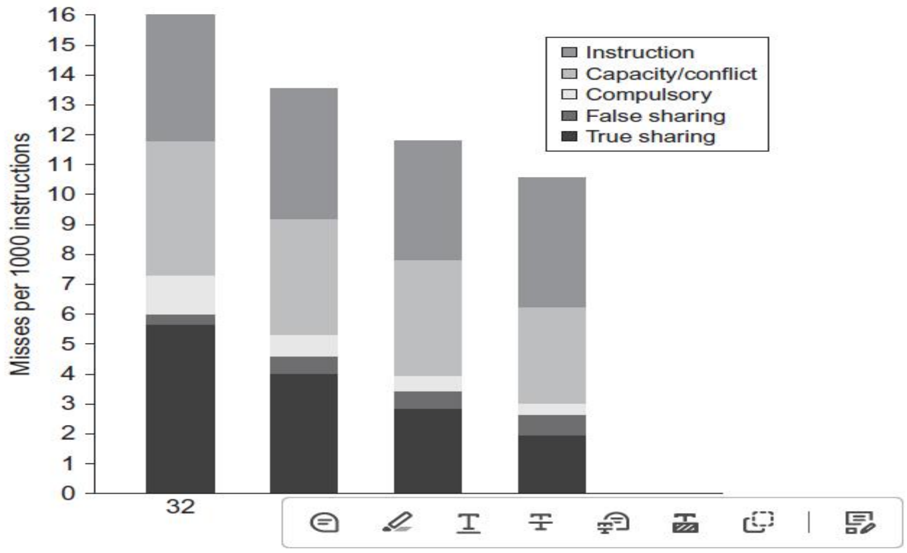


Figure 5.13 The number of misses per 1000 instructions drops steadily as the block size of the L3 cache is increased, making a good case for an L3 block size of at least 128 bytes. The L3 cache is 2 MiB, two-way set associative.

A Commercial Workload

Increasing block size from 32 to 256 bytes affects four of miss rate components:

1) True sharing miss rate decreases by more than a factor of 2, indicating locality in true sharing patterns.
2) Compulsory miss rate significantly decreases.
3) Conflict/capacity misses show a small decrease, indicating that spatial locality is not high in uniprocessor misses that occur with L3 caches larger than 2 MiB.
4) False sharing miss rate although small in absolute terms, nearly doubles.

- ✓ If there were an instruction-only cache that the spatial locality is very poor.
- ✓ In case of mixed L2 and L3 caches such as instruction-data conflicts may also contribute to the high instruction cache miss rate for larger blocks.
- ✓ Miss penalty for a larger block size L3 to perform as well as the 32-byte block size L3 can be expressed as a multiplier on the 32-byte block size penalty.

A Multiprogramming and OS Workload

- ✓ Workload used is two independent copies of the compile phase. Compile phase consists of a parallel version of UNIX "make" command executed using eight processors.
- ✓ Workload runs for 5.24 seconds on eight processors, creating 203 processes and performing 787 disk requests on three different file systems.
- ✓ Workload is run with 128 MiB of memory and no paging activity takes place.
- ✓ Workload has three distinct phases: compiling benchmarks, which involves compute activity; installing object files in a library; and removing object files.
- ✓ Last phase is completely dominated by I/O and only two processes are active.
- ✓ In the middle phase, I/O plays a major role and processor is largely idle.
- ✓ Overall workload is much more system- and I/O-intensive than OLTP workload.

A Multiprogramming and OS Workload

For workload measurements, assume the following memory and I/O systems:

- ✓ **Level 1 instruction cache: 32** KB, two-way set associative with a 64-byte block, 1 clock cycle hit time.
- ✓ **Level 1 data cache:** 32 KB, two-way set associative with a 32-byte block, 1 clock cycle hit time. Our focus is on examining the behavior in the Level 1 data cache, in contrast to the OLTP study, which focused on the L3 cache.
- ✓ **Level 2 cache:** 1 MiB unified, two-way set associative with a 128-byte block, 10 clock cycle hit time.
- ✓ **Main memory:** Single memory on a bus with an access time of 100 clock cycles.
- ✓ **Disk system:** Fixed-access latency of 3 ms (less than normal to reduce idle time).

A Multiprogramming and OS Workload

Figure 5.14 shows how the execution time breaks down for the eight processors using the parameters just listed. Execution time is broken down into four components :

1. **Idle:** Execution in the kernel mode idle loop
2. **User:** Execution in user code
3. **Synchronization:** Execution or waiting for synchronization variables
4. **Kernel:** Execution in the OS that is neither idle nor in synchronization access

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

Figure 5.14 The distribution of execution time in the multiprogrammed parallel "make" workload. The high fraction of idle time is due to disk latency when only one of the eight processors is active. These data and the subsequent measurements for this workload were collected with the SimOS system (Rosenblum et al., 1995). The actual runs and data collection were done by M. Rosenblum, S. Herrod, and E. Bugnion of Stanford University.

Performance of Multiprogramming and OS Workload

- ✓ Figure 5.15 shows data miss rate versus data cache size and versus block size for kernel and user components.
- ✓ Increasing data cache size affects user miss rate more than it affects kernel miss rate.
- ✓ Increasing block size has beneficial effects for both miss rates because a larger fraction of misses arise from compulsory and capacity, improved with larger block sizes.

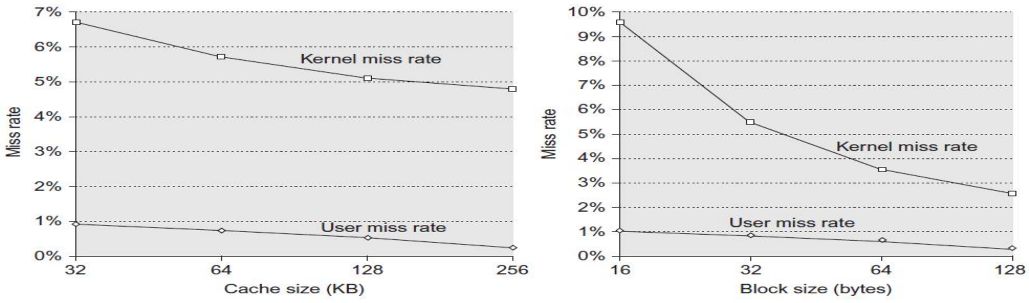


Figure 5.15 The data miss rates for the user and kernel components behave differently for increases in the L1 data cache size (on the left) versus increases in the L1 data cache block size (on the right). Increasing the L1 data cache from 32 to 256 KB (with a 32-byte block) causes the user miss rate to decrease proportionately more than the kernel miss rate: the user-level miss rate drops by almost a factor of 3, whereas the kernel-level miss rate drops by a factor of only 1.3. At the largest size, the L1 is closer to the size of L2 in a modern multicore processors. Thus the data indicates that the kernel miss rate will still be significant in an L2 cache. 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 of Multiprogramming and OS Workload

- ✓ Figure 5.16 shows variation in kernel misses versus increases in cache size and block size.
- ✓ Misses are broken into three classes: compulsory misses, coherence misses, and capacity/conflict misses.

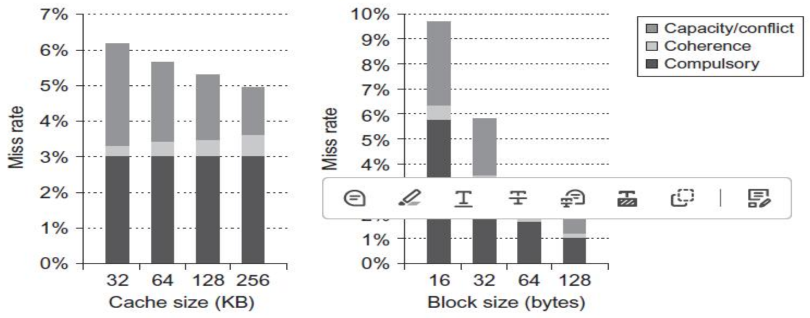


Figure 5.16 The components of the kernel data miss rate change as the L1 data cache size is increased from 32 to 256 KB, when the multiprogramming workload is run on eight processors. The compulsory miss rate component stays constant because it is unaffected by cache size. The capacity component drops by more than a factor of 2, whereas 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 caused by coherence grows from about 7% to about 15%.

Performance of Multiprogramming and OS Workloadj

- ✓ Figure 5.17,kernel has a higher traffic ratio that grows with block size

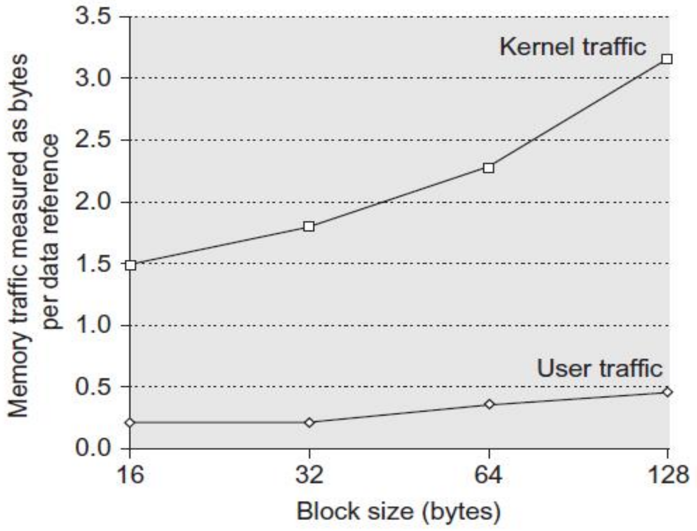


Figure 5.17 The number of bytes needed per data reference grows as block size is increased for both the kernel and user components. It is interesting to compare this chart with the data on scientific programs shown in Appendix I.