CHAPTER 5 Shared Memory Multiprocessors

Morgan Kaufmann is pleased to present material from a preliminary draft of Parallel Computer Architecture; the material is (c) Copyright 1996 Morgan Kaufmann Publishers. This material may not be used or distributed for any commercial purpose without the express written consent of Morgan Kaufmann Publishers. Please note that this material is a draft of forthcoming publication, and as such neither Morgan Kaufmann nor the authors can be held liable for changes or alterations in the final edition.

5.1 Introduction

The most prevalent form of parallel architecture is the multiprocessor of moderate scale that provides a global physical address space and symmetric access to all of main memory from any processor, often called a Symmetric Multiprocessor or SMP. Every processor has its own cache adn all the processors and memory modules attach to the same interconnect, usually a shared bus. SMPs dominate the server market and are becoming more common on the desktop. They are also important building blocks for larger scale systems. The efficient sharing of resources, such as memory and processors, makes these machines attractive as "throughput engines" for multiple sequential jobs with varying memory and CPU requirements, and the ability to access all shared data efficiently using ordinary loads and stores from any of the processors makes them attractive for parallel programming. The automatic movement and replication of shared data in the local caches can significantly ease the programming task. These features are also very useful for the

operating system, whose different processes share data structures and can easily run on different processors.

From the viewpoint of the layers of the communication architecture in Figure 5-1, the hardware directly supports the shared address space programming model. User processes can read and write shared virtual addresses, and these operations are realized by individual loads and stores of shared physical addresses that are uniformly and efficiently accessible from any processor. In fact, the relationship between the programming model and the hardware operation is so close, that they both are often referred to simply as "shared memory." A message passing programming model can be supported by an intervening software layer—typically a runtime library—that manages portions of the shared address space explicitly as message buffers per process. A send-receive operation pair is realized by copying data between buffers. The operating system need not be involved, since address translation and protection on these shared buffers is provided by the hardware. For portability, most message passing programming interfaces have been implemented on popular SMPs, as well as traditional message passing machines. In fact, such implementations often deliver higher message passing performance than traditional message passing systems—as long as contention for the shared bus and memory does not become a bottleneck—largely because of the lack of operating system involvement in communication. The operating system is still there for input/output and multiprogramming support.

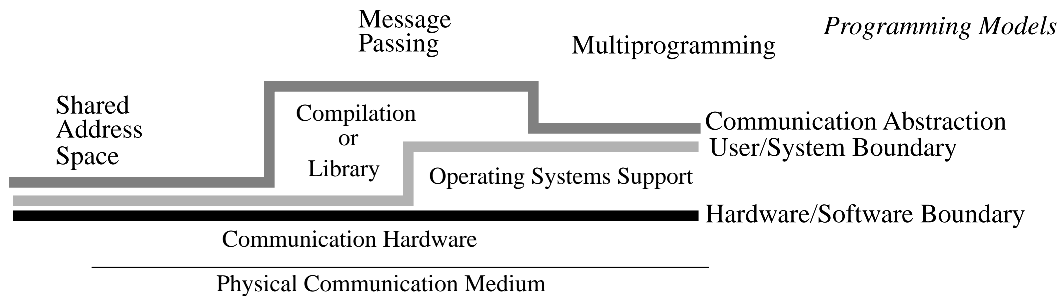


Figure 5-1 Layers of abstraction of the communication architecture for bus-based SMPs.

Since all communication and local computation generates memory accesses in a shared address space, from a system architect's perspective the key high-level design issue is the organization of the extended memory hierarchy. In general, memory hierarchies in multiprocessors fall primarily into four categories, as shown in Figure 5-2, which correspond loosely to the scale of the multiprocessor being considered. The first three are symmetric multiprocessors (all of main memory is equally far away from all processors), while the fourth is not.

In the first category, the "shared cache" approach (Figure 5-2a), the interconnect is located between the processors and a shared first-level cache, which in turn connects to a shared main-memory subsystem. Both the cache and the main-memory system may be interleaved to increase available bandwidth. This approach has been used for connecting very small numbers (2-8) of processors. In the mid 80s it was a common technique to connect a couple of processors on a board; today, it is a possible strategy for a multiprocessor-on-a-chip, where a small number of processors on the same chip share an on-chip first-level cache. However, it applies only at a very small scale, both because the interconnect between the processors and the shared first level cache

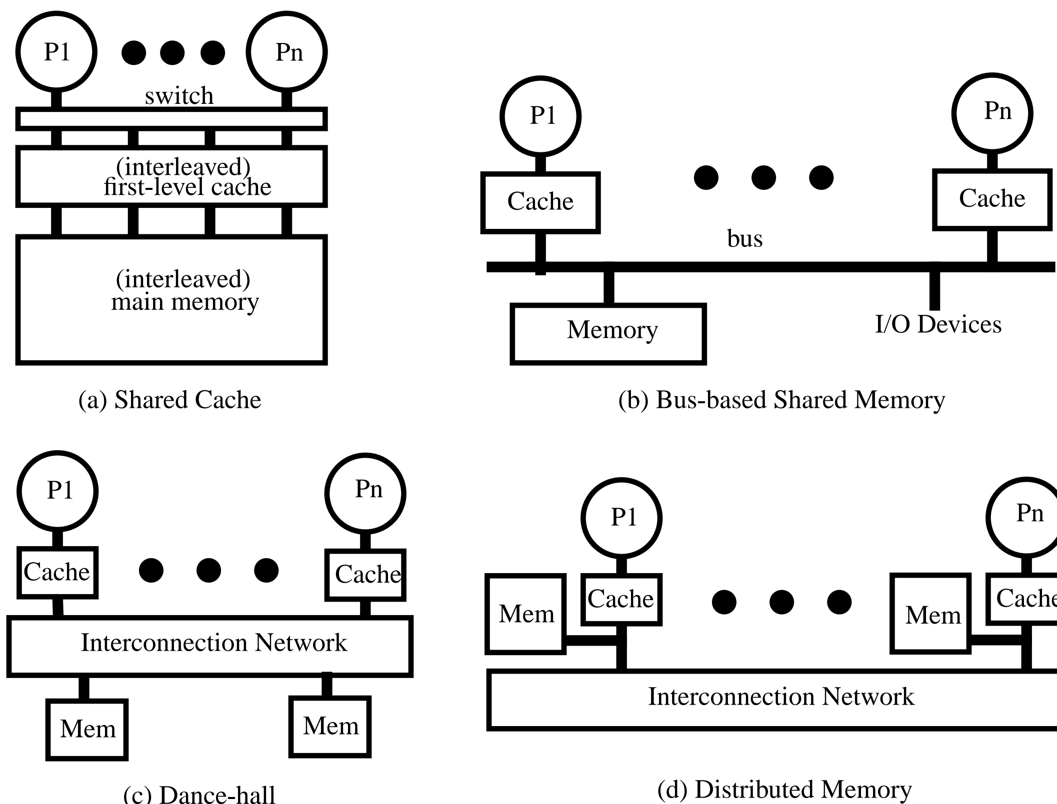


Figure 5-2 Common memory hierarchies found in multiprocessors.

is on the critical path that determines the latency of cache access and because the shared cache must deliver tremendous bandwidth to the processors pounding on it.

In the second category, the "bus-based shared memory" approach (Figure 5-2b), the interconnect is a shared bus located between the processor's private caches and the shared main-memory subsystem. This approach has been widely used for small- to medium-scale multiprocessors consisting of up to twenty or thirty processors. It is the dominant form of parallel machine sold today, and essentially all modern microprocessors have invested considerable design effort to be able to support "cache-coherent" shared memory configurations. For example, the Intel Pentium Pro processor can attach to a coherent shared bus without any glue logic, and low-cost bus-based machines that use these processors have greatly increased the popularity of this approach. The scaling limit for these machines comes primarily due to bandwidth limitations of the shared bus and memory system.

The last two categories are intended to be scalable to many processing nodes. The third category, the "dance-hall" approach, also places the interconnect between the caches and main memory, but the interconnect is now a scalable point-to-point network and memory is divided into many logical modules that connect to logically different points in the interconnect. This approach is symmetric, all of main memory is uniformly far away from all processors, but its limitation is that all of memory is indeed *far* away from all processors: Especially in large systems, several

"hops" or switches in the interconnect must be traversed to reach any memory module. The fourth category, the "distributed memory" approach, is not symmetric: A scalable interconnect is located between processing nodes but each node has its local portion of the global main memory to which it has faster access (Figure 5-2c). By exploiting locality in the distribution of data, the hope is that most accesses will be satisfied in the local memory and will not have to traverse the network. This design is therefore most attractive for scalable multiprocessors, and several chapters are devoted to the topic later in the book. Of course, it is also possible to combine multiple approaches into a single machine design—for example a distributed memory machine whose individual nodes are bus-based SMPs, or sharing a cache other than at the first level.

In all cases, caches play an essential role in reducing the average memory access time as seen by the processor and in reducing the bandwidth requirement each processor places on the shared interconnect and memory system. The bandwidth requirement is reduced because the data accesses issued by a processor that are satisfied in the cache do not have to appear on the bus; otherwise, every access from every processor would appear on the bus which would quickly become saturated. In all but the shared cache approach, each processor has at least one level of its cache hierarchy that is private and this raises a critical challenge, namely that of *cache coherence*. The problem arises when a memory block is present in the caches of one or more processors, and another processor modifies that memory block. Unless special action is taken, the former processors will continue to access the old, stale copy of the block that is in their caches.

Currently, most small-scale multiprocessors use a shared bus interconnect with per-processor caches and a centralized main memory, while scalable systems use physically distributed main memory. Dancehall and shared cache approaches are employed in relatively specific settings, that change as technology evolves. This chapter focuses on the logical design of multiprocessors that exploit the fundamental properties of a bus to solve the cache coherence problem. The next chapter expands on the hardware design issues associated with realizing these cache coherence techniques. The basic design of scalable distributed-memory multiprocessors will be addressed in Chapter 7, and issues specific to scalable cache coherence in Chapter 8.

In Section 5.2 describes the cache coherence problem for shared-memory architectures in detail. Coherence is a key hardware design concept and is a necessary part of our intuitive notion the memory abstraction. However, parallel software often makes stronger assumptions about how memory behaves. Section 5.3 extends the discussion of ordering begun in Chapter 1 and introduces the concept of memory consistency which defines the semantics of shared address space. This issue has become increasingly important in computer architecture and compiler design; a large fraction of the reference manuals for most recent instruction set architectures is devoted to the memory model. Section 5.4 presents what are called "snooping" or "snoopy" protocols for bus-based machines and shows how they satisfy the conditions for coherence as well as for a useful consistency model. The basic design space of snooping protocols is laid out in Section 5.5 and the operation of commonly used protocols is described at the state transition level. The techniques used for quantitative evaluation several design tradeoffs at this level are illustrated using aspects of the methodology for workload driven evaluation of Chapter 4.

The latter portions of the chapter examine the implications these cache coherent shared memory architectures have for software that runs on them. Section 5.6 examines how the low-level synchronization operations make use of the available hardware primitives on cache coherent multiprocessors, and how the algorithms can be tailored to use the machine efficiently. Section 5.7 discusses the implications for parallel programming more generally, putting together our knowledge of parallel programs from Chapters 2 and 3 and of bus-based cache-coherent architecture

from this chapter. In particular, it discusses how temporal and spatial data locality may be exploited to reduce cache misses and traffic on the shared bus.

5.2 Cache Coherence

Think for a moment about your intuitive model of what a memory should do. It should provide a set of locations holding values, and when a location is read it should return the latest value written to that location. This is the fundamental property of the memory abstraction that we rely on in sequential programs when we use memory to communicate a value from a point in a program where it is computed to other points where it is used. We rely on the same property of a memory system when using a shared address space to communicate data between threads or processes running on one processor. A read returns the latest value written to the location, regardless of which process wrote it. Caching does not interfere with the use of multiple processes on one processor, because they all see the memory through the same cache hierarchy. We would also like to rely on the same property when the two processes run on different processors that share a memory. That is, we would like the results of a program that uses multiple processes to be no different when the processes run on different physical processors than when they run (interleaved or multiprogrammed) on the same physical processor. However, when two processes see the shared memory through different caches, there is a danger that one may see the new value in its cache while the other still sees the old value.

5.2.1 The Cache Coherence Problem

The cache coherence problem in multiprocessors is both pervasive and performance critical. The problem is illustrated by the following example.

Example 5-1

Figure 5-3 shows three processors with caches connected via a bus to shared main memory. A sequence of accesses to location `u` is made by the processors. First, processor P1 reads `u` from main memory, bringing a copy into its cache. Then processor P3 reads `u` from main memory, bringing a copy into its cache. Then processor P3 writes location `u` changing its value from 5 to 7. With a write-through cache, this will cause the main memory location to be updated; however, when processor P1 reads location `u` again (action 4), it will unfortunately read the stale value 5 from its own cache instead of the correct value 7 from main memory. What happens if the caches are writeback instead of write through?

Answer

The situation is even worse with writeback caches. P3's write would merely set the dirty (or modified) bit associated with the cache block holding location `u` and would not update main memory right away. Only when this cache block is subsequently replaced from P3's cache would its contents be written back to main memory. Not only will P1 read the stale value, but when processor P2 reads location `u` (action 5) it will miss in its cache and read the stale value of 5 from main memory, instead of 7. Finally, if multiple processors write distinct values to location `u` in their write-back caches, the final value that will reach main memory

will be determined by the order in which the cache blocks containing u are replaced, and will have nothing to do with the order in which the writes to u occur.

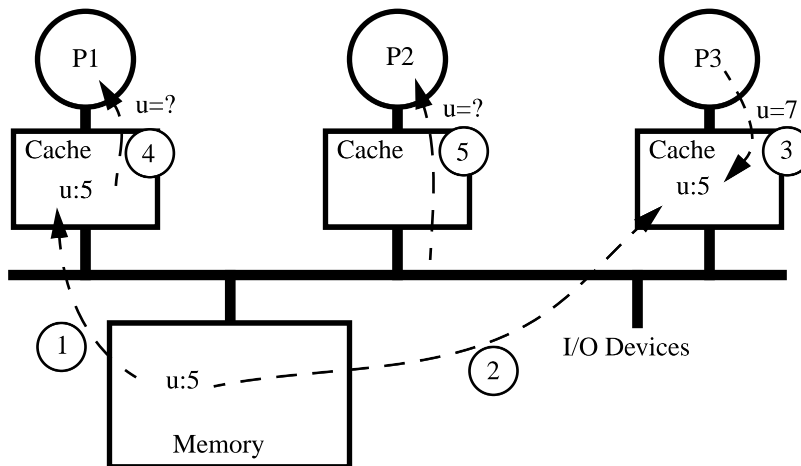


Figure 5-3 Example cache coherence problem

The figure shows three processors with caches connected by a bus to main memory. "u" is a location in memory whose contents are being read and written by the processors. The sequence in which reads and writes are done is indicated by the number listed inside the circles placed next to the arc. It is easy to see that unless special action is taken when P3 updates the value of u to 7, P1 will subsequently continue to read the stale value out of its cache, and P2 will also read a stale value out of main memory.

Cache coherence problems arise even in uniprocessors when I/O operations occur. Most I/O transfers are performed by DMA devices that move data between memory and the peripheral component. A DMA device sitting on the main memory bus may read a stale value for a location in main memory, because the latest value for that location is in the processor's write-back cache. Similarly, when the DMA device writes to a location in main memory, unless special action is taken, the processor may continue to see the old value if that location was previously present in its cache. Since I/O operations are much less frequent than memory operations, several simple solutions have been adopted in uniprocessors. For example, segments of memory space used for I/O may be marked as uncacheable, or the processor may always use uncached loads-stores for locations used to communicate with I/O devices. For I/O devices that transfer large blocks of data, such as disks, operating system support is often enlisted to ensure coherence. In many systems the page of memory from/to which the data is to be transferred is flushed by the operating system from the processor's cache before the I/O is allowed to proceed. In still other systems, all I/O traffic is made to flow through the processor cache hierarchy, thus maintaining coherence. This, of course, pollutes the cache hierarchy with data that may not be of immediate interest to the processor. Today, essentially all microprocessors provide support for cache coherence; in addition to making the chip "multiprocessor ready", this also solves the I/O coherence problem.

Clearly, the behavior described in Example 5-1 violates our intuitive notion of what a memory should do. Rreading and writing of shared variables is expected to be a frequent event in multiprocessors; it is the way that multiple processes belonging to a parallel application communicate with each other, so we do not want to disallow caching of shared data or invoke the operating system on all shared references. Rather, cache coherence needs to be addressed as a basic hardware design issue; for example, stale cached copies of a shared location must be eliminated when the location is modified, by either invalidating them or updating them with the new value. The oper-

ating system itself benefits greatly from transparent, hardware-supported coherent sharing of its data structures, and is in fact one of the most widely used parallel applications on these machines.

Before we explore techniques to provide coherence, it is useful to define the coherence property more precisely. Our intuitive notion, "each read should return the last value written to that location," is problematic for parallel architecture because "last" may not be well defined. Two different processors might write to the same location at the same instant, or one processor may write so soon after another that due to speed of light and other factors, there is not time to propagate the earlier update to the later writer. Even in the sequential case, "last" is not a chronological notion, but latest *in program order*. For now, we can think of program order in a single process as the order in which memory operations are presented to the processor by the compiler. The subtleties of program order will be elaborated on later, in Section 5.3. The challenge in the parallel case is that program order is defined for the operations in each individual process, and we need to make sense of the collection of program orders.

Let us first review the definitions of some terms in the context of uniprocessor memory systems, so we can extend the definitions for multiprocessors. By *memory operation*, we mean a single read (load), write (store), or read-modify-write access to a memory location. Instructions that perform multiple reads and writes, such as appear in many complex instruction sets, can be viewed as broken down into multiple memory operations, and the order in which these memory operations are executed is specified by the instruction. These memory operations within an instruction are assumed to execute *atomically* with respect to each other in the specified order, i.e. all aspects of one appear to execute before any aspect of the next. A memory operation *issues* when it leaves the processor's internal environment and is presented to the memory system, which includes the caches, write-buffers, bus, and memory modules. A very important point is that the processor only observes the state of the memory system by issuing memory operations; thus, our notion of what it means for a memory operation to be *performed* is that it appears to have taken place from the perspective of the processor. A write operation is said to *perform with respect to the processor* when a subsequent read by the processor returns the value produced by either that write or a later write. A read operation is said to perform with respect to the processor when subsequent writes issued by that processor cannot affect the value returned by the read. Notice that in neither case do we specify that the physical location in the memory chip has actually been accessed. Also, 'subsequent' is well defined in the sequential case, since reads and writes are ordered by the program order.

The same definitions for operations performing with respect to a processor apply in the parallel case; we can simply replace "the processor" by "a processor" in the definitions. The challenge for ordering, and for the intuitive notions of 'subsequent' and 'last', now is that we do not have one program order; rather, we have separate program orders for every process and these program orders interact when accessing the memory system. One way to sharpen our intuitive notion of a coherent memory system is to picture what would happen if there were no caches. Every write and every read to a memory location would access the physical location at main memory, and would be performed with respect to all processors at this point, so the memory would impose a serial order on all the read and write operations to the location. Moreover, the reads and writes to the location from any individual processor should be in program order within this overall serial order. We have no reason to believe that the memory system should interleave independent accesses from different processors in a particular way, so any interleaving that preserves the individual program orders is reasonable. We do assume some basic fairness; eventually the operations from each processor should be performed. Thus, our intuitive notion of "last" can be viewed as most recent in some hypothetical serial order that maintains the properties discussed above.

Since this serial order must be consistent, it is important that all processors see the writes to a location in the same order (if they bother to look, i.e. to read the location).

Of course, the total order need not actually be constructed at any given point in the machine while executing the program. Particularly in a system with caches, we do not want main memory to see all the memory operations, and we want to avoid serialization whenever possible. We just need to make sure that the program behaves as if some serial order was enforced.

More formally, we say that a multiprocessor memory system is *coherent* if the results of any execution of a program are such that, for each location, it is possible to construct a hypothetical serial order of all operations to the location (i.e., put all reads/writes issued by all processors into a total order) which is consistent with the results of the execution and in which:

1. operations issued by any particular processor occur in the above sequence in the order in which they were issued to the memory system by that processor, and
2. the value returned by each read operation is the value written by the last write to that location in the above sequence.

Implicit in the definition of coherence is the property that all writes to a location (from the same or different processes) are seen in the same order by all processes. This property is called *write serialization*. It means that if read operations by processor P1 to a location see the value produced by write *w1* (from P2, say) before the value produced by write *w2* (from P3, say), then reads by another processor P4 (or P2 or P3) should also not be able to see *w2* before *w1*. There is no need for an analogous concept of read serialization, since the effects of reads are not visible to any processor but the one issuing the read.

The results of a program can be viewed as the values returned by the read operations in it, perhaps augmented with an implicit set of reads to all locations at the end of the program. From the results, we cannot determine the order in which operations were actually executed by the machine, but only orders in which they appear to execute. In fact, it is not even important in what order things actually happen in the machine or when which bits change, since this is not detectable; all that matters is the order in which things appear to happen, as detectable from the results of an execution. This concept will become even more important when we discuss memory consistency models. Finally, the one additional definition that we will need in the multiprocessor case is that of an operation completing: A read or write operation is said to *complete* when it has performed with respect to all processors.

5.2.2 Cache Coherence Through Bus Snooping

A simple and elegant solution to the cache coherence problem arises from the very nature of a bus. The bus is a single set of wires connecting several devices, each of which can observe every bus transaction, for example every read or write on the shared bus. When a processor issues a request to its cache, the controller examines the state of the cache and takes suitable action, which may include generating bus transactions to access memory. Coherence is maintained by having each cache controller 'snoop' on the bus and monitor the transactions, as illustrated in Figure 5-4 [Goo83]. The snooping cache controller also takes action if a bus transaction is relevant to it, i.e. involves a memory block of which it has a copy in its cache. In fact, since the allocation and replacement of data in caches are managed at the granularity of a cache block (usually several words long) and cache misses fetch a block of data, most often coherence is maintained at

the granularity of a cache block as well. That is, either an entire cache block is in valid state or none of it is. Thus, a cache block is the granularity of allocation in the cache, of data transfer between caches and of coherence.

The key properties of a bus that support coherence are the following: All transactions that appear on the bus are visible to all cache controllers, and they are visible to the controllers in the same order (the order in which they appear on the bus). A coherence protocol must simply guarantee that all the necessary transactions in fact appear on the bus, in response to memory operations, and that the controllers take the appropriate actions when they see a relevant transaction.

The simplest illustration of maintaining coherence is a system that has single-level write-through caches. It is basically the approach followed by the first commercial bus-based SMPs in the mid 80's. In this case, every write operation causes a write transaction to appear on the bus, so every cache observes every write. If a snooping cache has a copy of the block, it either invalidates or updates its copy. Protocols that invalidate other cached copies on a write are called *invalidation-based protocols*, while those that update other cached copies are called *update-based protocols*. In either case, the next time the processor with the invalidated or updated copy accesses the block, it will see the most recent value, either through a miss or because the updated value is in its cache. The memory always has valid data, so the cache need not take any action when it observes a read on the bus.

Example 5-2 Consider the scenario that was shown in Figure 5-3. Assuming write through caches, show how the bus may be used to provide coherence using an invalidation-based protocol.

Answer When processor P3 writes 7 to location `u`, P3's cache controller generates a bus transaction to update memory. Observing this bus transaction as relevant, P1's cache controller invalidates its own copy of block containing `u`. The main memory controller will update the value it has stored for location `u` to 7. Subsequent reads to `u` from processors P1 and P2 (actions 4 and 5) will both miss in their private caches and get the correct value of 7 from the main memory.

In general, a snooping-based cache coherence scheme ensures that:

- all "necessary" transactions appear on the bus, and
- each cache monitors the bus for relevant transactions and takes suitable actions.

The check to determine if a bus transaction is relevant to a cache is essentially the same tag match that is performed for a request from the processor. The suitable action may involve invalidating or updating the contents or state of that memory block, and/or supplying the latest value for that memory block from the cache to the bus.

A *snoopy cache coherence protocol* ties together two basic facets of computer architecture: bus transactions and the state transition diagram associated with a cache block. Recall, a bus transaction consists of three phases: arbitration, command/address and data. In the arbitration phase, devices that desire to perform (or master) a transaction assert their bus request and the bus arbiter selects one of these and responds by asserting its grant signal. Upon grant, the device places the command, e.g. read or write, and the associated address on the bus command and address lines.

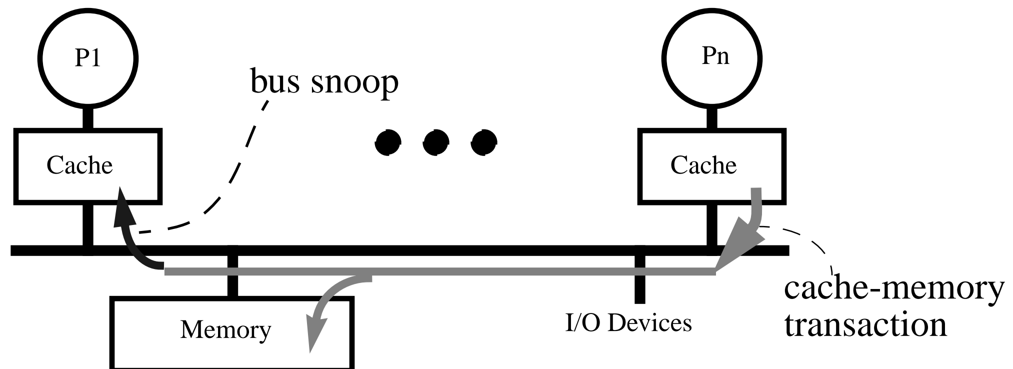


Figure 5-4 Snoopy cache-coherent multiprocessor

Multiple processors with private caches are placed on a shared bus. Each processor's cache controller continuously "snoops" on the bus watching for relevant transactions, and updates its state suitably to keep its local cache coherent.

All devices observe the address and one of them recognizes that it is responsible for the particular address. For a read transaction, the address phase is followed by data transfer. Write transactions vary from bus to bus in whether the data is transferred during or after the address phase. For most busses, the slave device can assert a wait signal to hold off the data transfer until it is ready. This wait signal is different from the other bus signals, because it is a wired-OR across all the processors, i.e., it is a logical 1 if any device asserts it. The master does not need to know which slave device is participating in the transfer, only that there is one and it is not yet ready.

The second basic notion is that each block in a uniprocessor cache has a state associated with it, along with the tag and data, indicating the disposition of the block, e.g., invalid, valid, dirty. The cache policy is defined by the *cache block state transition diagram*, which is a finite state machine. Transitions for a block occur upon access to an address that maps to the block. For a write-through, write-no-allocate cache [HeP90] only two states are required: valid and invalid. Initially all the blocks are invalid. (Logically, all memory blocks that are not resident in the cache can be viewed as being in either a special "not present" state, or in "invalid" state.) When a processor read operation misses, a bus transaction is generated to load the block from memory and the block is marked valid. Writes generate a bus transaction to update memory and the cache block if it is present in valid state. Writes never change the state of the block. (If a block is replaced, it may be marked invalid until the memory provides the new block, whereupon it becomes valid.) A write-back cache requires an additional state, indicating a "dirty" or modified block.

In a multiprocessor system, a block has a state in each cache, and these cache states change according to the state transition diagram. Thus, we can think of a block's "cache state" as being a vector of p states instead of a single state, where p is the number of caches. The cache state is manipulated by a set of p distributed finite state machines, implemented by the cache controllers. The state machine or state transition diagram that governs the state changes is the same for all blocks and all caches, but the current states of a block in different caches is different. If a block is not present in a cache, we can assume it to be in a special "not present" state or even in invalid state.

In a snoopy cache coherence scheme, each cache controller receives two sets of inputs: the processor issues memory requests, and the bus snooper informs about transactions from other caches. In response to either, the controller updates the state of the appropriate block in the cache according to the current state and the state transition diagram. It responds to the processor with requested data, potentially generating new bus transactions to obtain the data. It responds to bus transactions generated by others by updating its state, and sometimes intervenes in completing the transaction. Thus, a snoopy protocol is a distributed algorithm represented by a collection of such cooperating finite state machines. It is specified by the following components:

1. the set of states associated with memory blocks in the local caches;
2. the state transition diagram, which takes as inputs the current state and the processor request or observed bus transaction, and produces as output the next state for the cache block; and
3. the actual actions associated with each state transition, which are determined in part by the set of feasible actions defined by the bus, cache, and processor design.

The different state machines for a block do not operate independently, but are coordinated by bus transactions.

A simple invalidation-based protocol for a coherent write-through no-allocate cache is described by the state diagram in Figure 5-5. As in the uniprocessor case, each cache block has only two states: invalid (I) and valid (V) (the "not present" state is assumed to be the same as invalid). The transitions are marked with the input that causes the transition and the output that is generated with the transition. For example, when a processor read misses in the cache a BusRd transaction is generated, and upon completion of this transaction the block transitions up to the "valid" state. Whenever the processor issues a write to a location, a bus transaction is generated that updates that location in main memory with no change of state. The key enhancement to the uniprocessor state diagram is that when the bus snooper sees a write transaction for a memory block that is cached locally, it sets the cache state for that block to invalid thereby effectively discarding its copy. Figure 5-5 shows this bus-induced transition with a dashed arc. By extension, if any one cache generates a write for a block that is cached by any of the others, all of the others will invalidate their copies. The collection of caches thus implements a single writer, multiple reader discipline.

To see that this simple write-through invalidation protocol provides coherence, we need to show that a total order on the memory operations for a location can be constructed that satisfies the two conditions. Assume for the present discussion that memory operations are *atomic*, in that only one transaction is in progress on the bus at a time and a processor waits until its transaction is finished before issuing another memory operation. That is, once a request is placed on the bus, all phases of the transaction including the data response complete before any other request from any processor is allowed access to the bus. With single-level caches, it is natural to assume that invalidations are applied to the caches, and the write completes, during the bus transaction itself. (These assumptions will be relaxed when we look at protocol implementations in more detail and as we study high performance designs with greater concurrency.) We may assume that the memory handles writes and reads in the order they are presented to it by the bus. In the write-through protocol, all writes go on the bus and only one bus transaction is in progress at a time, so all writes to a location are serialized (consistently) by the order in which they appear on the shared bus, called the *bus order*. Since each snooping cache controller performs the invalidation during the bus transaction, invalidations are performed by all cache controllers in bus order.

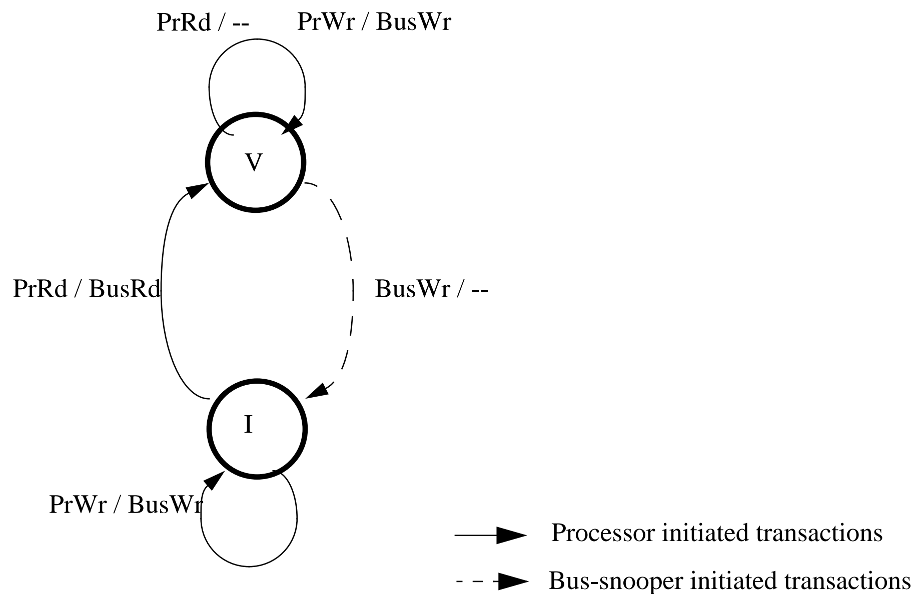


Figure 5-5 Snoopy coherence for a multiprocessor with write-through no-write-allocate caches.

There are two states, valid (V) and invalid (I) with intuitive semantics. The notation A/B (e.g. PrRd/BusRd) means if you observe A then generate transaction B. From the processor side, the requests can be read (PrRd) or write (PrWr). From the bus side, the cache controller may observe/generate transactions bus read (BusRd) or bus write (BusWr).

Processors "see" writes through read operations. However, reads to a location are not completely serialized, since read hits may be performed independently and concurrently in their caches without generating bus transactions. To see how reads may be inserted in the serial order of writes, and guarantee that all processors see writes in the same order (write serialization), consider the following scenario. A read that goes on the bus (a read miss) is serialized by the bus along with the writes; it will therefore obtain the value written by the most recent write to the location in bus order. The only memory operations that do not go on the bus are read hits. In this case, the value read was placed in the cache by either the most recent write to that location by the same processor, or by its most recent read miss (in program order). Since both these sources of the value appear on the bus, read hits also see the values produced in the consistent bus order.

More generally, we can easily construct a hypothetical serial order by observing the following partial order imposed by the protocol:

A memory operation M_2 is subsequent to a memory operation M_1 if the operations are issued by the same processor and M_2 follows M_1 in program order.

A read operation is subsequent to a write operation W if the read generates a bus transaction that follows that for W.

A write operation is subsequent to a read or write operation M if M generates a bus transaction and the transaction for the write follows that for M.

A write operation is subsequent to a read operation if the read does not generate a bus transaction (is a hit) and is not already separated from the write by another bus transaction.

The "subsequent" ordering relationship is transitive. An illustration of this partial order is depicted in Figure 5-6, where the bus transactions associated with writes segment the individual program orders. The partial order does not constrain the ordering of read bus transactions from different processors that occur between two write transactions, though the bus will likely establish a particular order. In fact, any interleaving of read operations in the segment between two writes is a valid serial order, as long as it obeys program order.

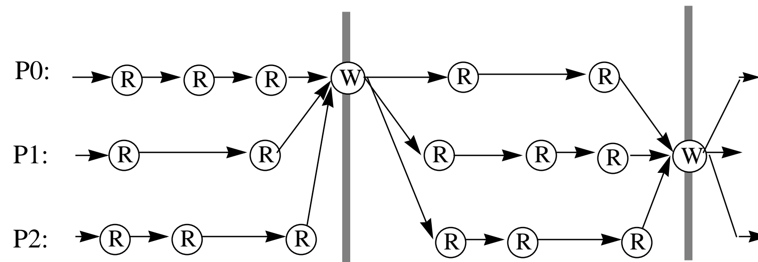


Figure 5-6 Partial order of memory operations for an execution with the write-through invalidation protocol

Bus transactions define a global sequence of events, between which individual processors read locations in program order. The execution is consistent with any total order obtained by interleaving the processor orders within each segment.

Of course, the problem with this simple write-through approach is that every store instruction goes to memory, which is why most modern microprocessors use write-back caches (at least at the level closest to the bus). This problem is exacerbated in the multiprocessor setting, since every store from every processor consumes precious bandwidth on the shared bus, resulting in poor scalability of such multiprocessors as illustrated by the example below.

Example 5-3 Consider a superscalar RISC processor issuing two instructions per cycle running at 200MHz. Suppose the average CPI (clocks per instruction) for this processor is 1, 15% of all instructions are stores, and each store writes 8 bytes of data. How many processors will a GB/s bus be able to support without becoming saturated?

Answer A single processor will generate 30 million stores per second (0.15 stores per instruction * 1 instruction per cycle * 1,000,000/200 cycles per second), so the total write-through bandwidth is 240Mbytes of data per second per processor (ignoring address and other information, and ignoring read misses). A GB/s bus will therefore support only about four processors.

For most applications, a write-back cache would absorb the vast majority of the writes. However, if writes do not go to memory they do not generate bus transactions, and it is no longer clear how the other caches will observe these modifications and maintain cache coherence. A somewhat more subtle concern is that when writes are allowed to occur into different caches concurrently there is no obvious order to the sequence of writes. We will need somewhat more sophisticated cache coherence protocols to make the "critical" events visible to the other caches and ensure write serialization.

Before we examine protocols for write-back caches, let us step back to the more general ordering issue alluded to in the introduction to this chapter and examine the semantics of a shared address space as determined by the memory consistency model.

5.3 Memory Consistency

Coherence is essential if information is to be transferred between processors by one writing a location that the other reads. Eventually, the value written will become visible to the reader, indeed all readers. However, it says nothing about when the write will become visible. Often, in writing a parallel program we want to ensure that a read returns the value of a particular write, that is we want to establish an order between a write and a read. Typically, we use some form of event synchronization to convey this dependence and we use more than one location.

Consider, for example, the code fragments executed by processors P1 and P2 in Figure 5-7, which we saw when discussing point-to-point event synchronization in a shared address space in Chapter 2. It is clear that the programmer intends for process P2 to spin idly until the value of the shared variable `flag` changes to 1, and to then print the value of variable `A` as 1, since the value of `A` was updated before that of `flag` by process P1. In this case, we use accesses to another location (`flag`) to preserve order among different processes' accesses to the same location (`A`). In particular, we assume that the write of `A` becomes visible to P2 before the write to `flag`, and that the read of flag by P2 that breaks it out of its while loop completes before its read of `A`. These program orders within P1 and P2's accesses are not implied by coherence, which, for example, only requires that the new value for `A` eventually become visible to processor P2, not necessarily before the new value of `flag` is observed.

```
P1                       P2
/* Assume initial value of A and flag is 0 */
A = 1;              while (flag == 0); /* spin idly */
flag = 1;           print A;
```

Figure 5-7 Requirements of event synchronization through flags

The figure shows two processors concurrently executing two distinct code fragments. For programmer intuition to be maintained, it must be the case that the printed value of A is 1. The intuition is that because of program order, if flag equals 1 is visible to processor P2, then it must also be the case that A equals 1 is visible to P2.

The programmer might try to avoid this issue by using a barrier, as shown in Figure 5-8. We expect the value of `A` to be printed as 1, since `A` was set to 1 before the barrier (note that a print statement is essentially a read). There are two potential problems even with this approach. First, we are adding assumptions to the meaning of the barrier: Not only do processes wait at the barrier till all have arrived, but until all writes issued prior to the barrier have become visible to other processors. Second, a barrier is often built using reads and writes to ordinary shared variables (e.g. `b1` in the figure) rather than with specialized hardware support. In this case, as far as the machine is concerned it sees only accesses to different shared variables; coherence does not say anything at all about orders among these accesses.

```
        P1                            P2
        /* Assume initial value of A is 0 */
        A = 1;              ...
- — - BARRIER(b1)- — — — — — — -BARRIER(b1)  - — -
                              print A;
```

Figure 5-8 Maintaining orders among accesses to a location using explicit synchronization through barriers.

Clearly, we expect more from a memory system than "return the last value written" for each location. To establish order among accesses to the location (say `A`) by different processes, we sometimes expect a memory system to respect the order of reads and writes to *different* locations (`A` and `flag` or `A` and `b1`) issued by a given process. Coherence says nothing about the order in which the writes issued by P1 become visible to P2, since these operations are to different locations. Similarly, it says nothing about the order in which the reads issued to different locations by P2 are performed relative to P1. Thus, coherence does not in itself prevent an answer of 0 being printed by either example, which is certainly not what the programmer had in mind.

In other situations, the intention of the programmer may not be so clear. Consider the example in Figure 5-9. The accesses made by process P1 are ordinary writes, and `A` and `B` are not used as

```
P1                  P2
/* Assume initial values of A and B are 0 */
(1a) A = 1;         (2a) print B;
(1b) B = 2;         (2b) print A;
```

Figure 5-9 Orders among accesses without synchronization.

synchronization variables. We may intuitively expect that if the value printed for `B` is 2 then the value printed for `A` is 1 as well. However, the two print statements read different locations before printing them, so coherence says nothing about how the writes by P1 become visible. (This example is a simplification of Dekker's algorithm to determine which of two processes arrives at a critical point first, and hence ensure mutual exclusion. It relies entirely on writes to distinct locations becoming visible to other processes in the order issued.) Clearly we need something more than coherence to give a shared address space a clear semantics, i.e., an ordering model that programmers can use to reason about the possible results and hence correctness of their programs.

A *memory consistency model* for a shared address space specifies constraints on the order in which memory operations must appear to be performed (i.e. to become visible to the processors) with respect to one another. This includes operations to the same locations or to different locations, and by the same process or different processes, so memory consistency subsumes coherence.

5.3.1 Sequential Consistency

In discussing the fundamental design issues for a communication architecture in Chapter 1 (Section 1.4), we described informally a desirable ordering model for a shared address space: the reasoning one goes through to ensure a multithreaded program works under any possible interleaving on a uniprocessor should hold when some of the threads run in parallel on different processors. The ordering of data accesses within a process was therefore the program order, and that across processes was some interleaving of the program orders. That is, the multiprocessor case should not be able to cause values to become visible in the shared address space that no interleaving of accesses from different processes can generate. This intuitive model was formalized by Lamport as *sequential consistency* (SC), which is defined as follows [Lam79].[1]

> *Sequential Consistency* A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor occur in this sequence in the order specified by its program.

Figure 5-10 depicts the abstraction of memory provided to programmers by a sequentially consistent system [AdG96]. Multiple processes *appear* to share a *single* logical memory, even though in the real machine main memory may be distributed across multiple processors, each with their private caches and write buffers. Every processor appears to issue and complete memory operations one at a time and atomically in program order—that is, a memory operation does not appear to be issued until the previous one has completed—and the common memory appears to service these requests one at a time in an interleaved manner according to an arbitrary (but hopefully fair) schedule. Memory operations appear *atomic* in this interleaved order; that is, it

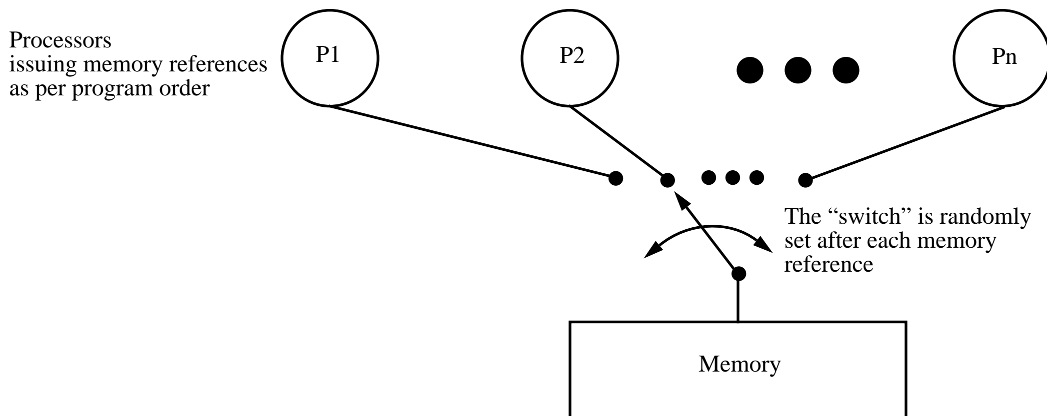


Figure 5-10 Programmer's abstraction of the memory subsystem under the sequential consistency model.

The model completely hides the underlying concurrency in the memory-system hardware, for example, the possible existence of distributed main memory, the presence of caches and write buffers, from the programmer.

1. Two closely related concepts in the software context are serializability[Papa79] for conccurent updates to a database and linearizability[HeWi87] for concurrent objects.

should appear globally (to all processors) as if one operation in the consistent interleaved order executes and completes before the next one begins.

As with coherence, it is not important in what order memory operations actually issue or even complete. What matters is that they appear to complete in an order that does not violate sequential consistency. In the example in Figure 5-9, under SC the result (0,2) for (A,B) would not be allowed under sequential consistency—preserving our intuition—since it would then appear that the writes of A and B by process P1 executed out of program order. However, the memory operations may actually execute and complete in the order 1b, 1a, 2b, 2a. It does not matter that they actually complete out of program order, since the results of the execution (1,2) is the same as if the operations were executed and completed in program order. On the other hand, the actual execution order 1b, 2a, 2b, 1a would not be sequentially consistent, since it would produce the result (0,2) which is not allowed under SC. Other examples illustrating the intuitiveness of sequential consistency can be found in Exercise 5.4. Note that sequential consistency does not obviate the need for synchronization. SC allows operations from different processes to be interleaved arbitrarily and at the granularity of individual instructions. Synchronization is needed if we want to preserve atomicity (mutual exclusion) across multiple memory operations from a process, or if we want to enforce certain orders in the interleaving across processes.

The term "program order" also bears some elaboration. Intuitively, program order for a process is simply the order in which statements appear in the source program; more specifically, the order in which memory operations appear in the assembly code listing that results from a straightforward translation of source statements one by one to assembly language instructions. This is not necessarily the order in which an optimizing compiler presents memory operations to the hardware, since the compiler may reorder memory operations (within certain constraints such as dependences to the same location). The programmer has in mind the order of statements in the program, but the processor sees only the order of the machine instructions. In fact, there is a "program order" at each of the interfaces in the communication architecture—particularly the programming model interface seen by the programmer and the hardware-software interface—and ordering models may be defined at each. Since the programmer reasons with the source program, it makes sense to use this to define program order when discussing memory consistency models; that is, we will be concerned with the consistency model presented by the system to the programmer.

Implementing SC requires that the system (software and hardware) preserve the intuitive constraints defined above. There are really two constraints. The first is the *program order* requirement discussed above, which means that it must appear as if the memory operations of a process become visible—to itself and others—in program order. The second requirement, needed to guarantee that the total order or interleaving is consistent for all processes, is that the operations appear atomic; that is, it appear that one is completed with respect to all processes before the next one in the total order is issued. The tricky part of this second requirement is making writes appear atomic, especially in a system with multiple copies of a block that need to be informed on a write. *Write atomicity*, included in the definition of SC above, implies that the position in the total order at which a write appears to perform should be the same with respect to all processors. It ensures that nothing a processor does *after* it has seen the new value produced by a write becomes visible to other processes before they too have seen the new value for that write. In effect, while coherence (write serialization) says that writes to the same location should appear to all processors to have occurred in the same order, sequential consistency says that all writes (to any location) should appear to all processors to have occurred in the same order. The following example shows why write atomicity is important.

Example 5-4 Consider the three processes in Figure 5-11. Show how not preserving write atomicity violates sequential consistency.

Answer Since P2 waits until A becomes 1 and then sets B to 1, and since P3 waits until B becomes 1 and only then reads value of A, from transitivity we would infer that P3 should find the value of A to be 1. If P2 is allowed to go on past the read of A and write B before it is guaranteed that P3 has seen the new value of A, then P3 may read the new value of B but the old value of A from its cache, violating our sequentially consistent intuition.

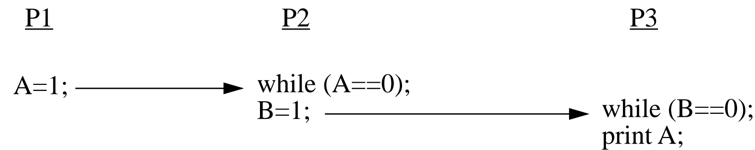


Figure 5-11 Example illustrating the importance of write atomicity for sequential consistency.

Each process's program order imposes a partial order on the set of all operations; that is, it imposes an ordering on the subset of the operations that are issued by that process. An interleaving (in the above sense) of the operations from different processes defines a total order on the set of all operations. Since the exact interleaving is not defined by SC, interleaving the partial (program) orders for different processes may yield a large number of possible total orders. The following definitions apply:

Sequentially Consistent Execution An execution of a program is said to be sequentially consistent if the results it produces are the same as those produced by any one of these possible total orders (interleavings as defined earlier). That is, there should exist a total order or interleaving of program orders from processes that yields the same result as that actual execution.

Sequentially Consistent System A system is sequentially consistent if any possible execution on that system corresponds to (produces the same results as) some possible total order as defined above.

Of course, an implicit assumption throughout is that a read returns the last value that was written to that same location (by any process) in the interleaved total order.

5.3.2 Sufficient Conditions for Preserving Sequential Consistency

It is possible to define a set of sufficient conditions that the system should obey that will guarantee sequential consistency in a multiprocessor—whether bus-based or distributed, cache-coherent or not. The following set, adapted from their original form [DSB86,ScD87], are commonly used because they are relatively simple without being overly restrictive:

1. Every process issues memory requests in the order specified by the program.
2. After a write operation is issued, the issuing process waits for the write to complete before issuing its next operation.

3. After a read operation is issued, the issuing process waits for the read to complete, *and* for the write whose value is being returned by the read to complete, before issuing its next operation. That is, if the write whose value is being returned has performed with respect to this processor (as it must have if its value is being returned) then the processor should wait until the write has performed with respect to all processors.

The third condition is what ensures write atomicity, and it is quite demanding. It is not a simple local constraint, because the load must wait until the logically preceding store has become globally visible. Note that these are sufficient, rather than necessary conditions. Sequential consistency can be preserved with less serialization in many situations. This chapter uses these conditions, but exploits other ordering constraints in the bus-based design to establish completion early.

With program order defined in terms of the source program, it is clear that for these conditions to be met the compiler should not change the order of memory operations that it presents to the processor. Unfortunately, many of the optimizations that are commonly employed in both compilers and processors violate the above sufficient conditions. For example, compilers routinely reorder accesses to different locations within a process, so a processor may in fact issue accesses out of the program order seen by the programmer. Explicitly parallel programs use uniprocessor compilers, which are concerned only about preserving dependences to the same location. Advanced compiler optimizations designed to improve performance—such as common sub-expression elimination, constant propagation, and loop transformations such as loop splitting, loop reversal, and blocking [Wol96]—can change the order in which different locations are accessed or even eliminate memory references.[1] In practice, to constrain compiler optimizations multithreaded and parallel programs annotate variables or memory references that are used to preserve orders. A particularly stringent example is the use of the `volatile` qualifier in a variable declaration, which prevents the variable from being register allocated or any memory operation on the variable from being reordered with respect to operations before or after it.

Example 5-5 How would reordering the memory operations in Figure 5-7 affect semantics in a sequential program (only one of the processes running), in a parallel program running on a multiprocessor, and in a threaded program in which the two processes are interleaved on the same processor. How would you solve the problem?

Answer The compiler may reorder the writes to `A` and `flag` with no impact on a sequential program. However, this can violate our intuition for both parallel programs and concurrent uniprocessor programs. In the latter case, a context switch can happen between the two reordered writes, so the process switched in may see the update to `flag` without seeing the update to `A`. Similar violations of intuition occur if the compiler reorders the reads of flag and A. For many compilers, we can avoid these reorderings by declaring the variable `flag` to be of type `volatile integer`

1. Note that register allocation, performed by modern compilers to eliminate memory operations, can be dangerous too. In fact, it can affect coherence itself, not just memory consistency. For the flag synchronization example in Figure 5-7, if the compiler were to register allocate the flag variable for process P2, the process could end up spinning forever: The cache-coherence hardware updates or invalidates only the memory and the caches, not the registers of the machine, so the write propagation property of coherence is violated.

instead of just `integer`. Other solutions are also possible, and will be discussed in Chapter 9.

Even if the compiler preserves program order, modern processors use sophisticated mechanisms like write buffers, interleaved memory, pipelining and out-of-order execution techniques [HeP90]. These allow memory operations from a process to issue, execute and/or complete out of program order. These architectural and compiler optimizations work for sequential programs because there the appearance of program order requires that dependences be preserved only among accesses to the same memory location, as shown in Figure 5-12.

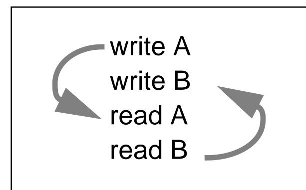


Figure 5-12 Preserving orders in a sequential program running on a uniprocessor.

Only the orders corresponding to the two dependence arcs must be preserved. The first two operations can be reordered without a problem, as can the last two or the middle two.

Preserving sequential consistency in multiprocessors is quite a strong requirement; it limits compiler reordering and out of order processing techniques. The problem is that the out of order processing of operations to shared variables by a process can be detected by other processes. Several weaker consistency models have been proposed, and we will examine these in the context of large scale shared address space machines in Chapter 6. For the purposes of this chapter, we will assume the compiler does not perform optimizations that violate the sufficient conditions for sequential consistency, so the program order that the processor sees is the same as that seen by the programmer. On the hardware side, to satisfy the sufficient conditions we need mechanisms for a processor to detect completion of its writes so it may proceed past them—completion of reads is easy, it is when the data returns to the processor—and mechanisms to preserve write atomicity. For all the protocols and systems considered in this chapter, we will see how they satisfy coherence (including write serialization), how they can satisfy sequential consistency (in particular, how write completion is detected and write atomicity is guaranteed), and what shortcuts can be taken while still satisfying the sufficient conditions.

The serialization imposed by transactions appearing on the shared bus is very useful in ordering memory operations. The reader should verify that the 2-state write-through invalidate protocol discussed above actually provides sequential consistency, not just coherence, quite easily. The key observation is that writes and read misses to all locations, not just to individual locations, are serialized in bus order. When a read obtains the value of a write, the write is guaranteed to have completed, since it caused a previous bus transaction. When a write is performed, all previous writes have completed.

5.4 Design Space for Snooping Protocols

The beauty of snooping-based cache coherence is that the entire machinery for solving a difficult problem boils down to a small amount of extra interpretation on events that naturally occur in the system. The processor is completely unchanged. There are no explicit coherence operations inserted in the program. By extending the requirements on the cache controller and exploiting the properties of the bus, the loads and stores that are inherent to the program are used implicitly to keep the caches coherent and the serialization of the bus maintains consistency. Each cache controller observes and interprets the memory transactions of others to maintain its internal state. Our initial design point with write-through caches is not very efficient, but we are now ready to study the design space for snooping protocols that make efficient use of the limited bandwidth of the shared bus. All of these use write-back caches, allowing several processors to write to different blocks in their local caches concurrently without any bus transactions. Thus, extra care is required to ensure that enough information is transmitted over the bus to maintain coherence. We will also see how the protocols provide sufficient ordering constraints to maintain write serialization and a sequentially consistent memory model.

Recall that with a write-back cache on a uniprocessor, a processor write miss causes the cache to read the entire block from memory, update a word, and retain the block as *modified* (or *dirty*) so it may be written back on replacement. In a multiprocessor, this modified state is also used by the protocols to indicate exclusive ownership of the block by a cache. In general, a cache is said to be the *owner* of a block if it must supply the data upon a request for that block [SwS86]. A cache is said to have an *exclusive* copy of a block if it is the only cache with a valid copy of the block (main memory may or may not have a valid copy). Exclusivity implies that the cache may modify the block without notifying anyone else. If a cache does not have exclusivity, then it cannot write a new value into the block before first putting a transaction on the bus to communicate with others. The data may be in the writer's cache in a valid state, but since a transaction must be generated this is called a write miss just like a write to a block that is not present or invalid in the cache. If a cache has the block in modified state, then clearly it is both the owner and has exclusivity. (The need to distinguish ownership from exclusivity will become clear soon.)

On a write miss, then, a special form of transaction called a *read-exclusive* is used to tell other caches about the impending write and to acquire a copy of the block with exclusive ownership. This places the block in the cache in modified state, where it may now be written. Multiple processors cannot write the same block concurrently, which would lead to inconsistent values: The read-exclusive bus transactions generated by their writes will be serialized by the bus, so only one of them can have exclusive ownership of the block at a time. The cache coherence actions are driven by these two types of transactions: read and read-exclusive. Eventually, when a modified block is replaced from the cache, the data is written back to memory, but this event is not caused by a memory operation to that block and is almost incidental to the protocol. A block that is not in modified state need not be written back upon replacement and can simply be dropped, since memory has the latest copy. Many protocols have been devised for write-back caches, and we will examine the basic alternatives.

We also consider update-based protocols. Recall that in update-based protocols, when a shared location is written to by a processor its value is updated in the caches of all other processors holding that memory block[1]. Thus, when these processors subsequently access that block, they can do so from their caches with low latency. The caches of all other processors are updated with a single bus transaction, thus conserving bandwidth when there are multiple sharers. In contrast, with

invalidation-based protocols, on a write operation the cache state of that memory block in all other processors' caches is set to invalid, so those processors will have to obtain the block through a miss and a bus transaction on their next read. However, subsequent writes to that block by the same processor do not create any further traffic on the bus, until the block is read by another processor. This is attractive when a single processor performs multiple writes to the same memory block before other processors read the contents of that memory block. The detailed tradeoffs are quite complex and they depend on the workload offered to the machine; they will be illustrated quantitatively in Section 5.5. In general, invalidation-based strategies have been found to be more robust and are therefore provided as the default protocol by most vendors. Some vendors provide an update protocol as an option to be used selectively for blocks corresponding to specific data structures or pages.

The choices made for the protocol, update versus invalidate, and caching strategies directly affect the choice of states, the state transition diagram, and the associated actions. There is substantial flexibility available to the computer architect in the design task at this level. Instead of listing all possible choices, let us consider three common coherence protocols that will illustrate the design options.

5.4.1 A 3-state (MSI) Write-back Invalidation Protocol

The first protocol we consider is a basic invalidation-based protocol for write-back caches. It is very similar to the protocol that was used in the Silicon Graphics 4D series multiprocessor machines [BJS88]. The protocol uses the three states required for any write-back cache in order to distinguish valid blocks that are unmodified (clean) from those that are modified (dirty). Specifically, the states are *invalid* (I), *shared* (S), and *modified* (M). Invalid has the obvious meaning. Shared means the block is present in unmodified state in this cache, main memory is up-to-date, and zero or more other caches may also have an up-to-date (shared) copy. Modified, also called *dirty*, was discussed earlier; it means that only this processor has a valid copy of the block in its cache, the copy in main memory is stale, and no other cache may have a valid copy of the block (in either shared or modified state). Before a shared or invalid block can be written and placed in the modified state, all the other potential copies must be invalidated via a read-exclusive bus transaction. This transaction serves to order the write as well as cause the invalidations and hence ensure that the write becomes visible to others.

The processor issues two types of requests: reads (PrRd) and writes (PrWr). The read or write could be to a memory block that exists in the cache or to one that does not. In this latter case, the block currently in the cache will have to be replaced by the newly requested block, and if the existing block is in modified state its contents will have to be written back to main memory.

We assume that the bus allows the following transactions:

Bus Read (BusRd): The cache controller puts the address on the bus and asks for a copy that it does not intend to modify. The memory system (possibly another cache) supplies the data.

1. This is a write-broadcast scenario. Read-broadcast designs have also been investigated, in which the cache containing the modified copy flushes it to the bus on a read, at which point all other copies are updated too.

This transaction is generated by a PrRd that misses in the cache, and the processor expects a data response as a result.

Bus Read-Exclusive (BusRdX): The cache controller puts the address on the bus and asks for an exclusive copy that it intends to modify. The memory system (possibly another cache) supplies the data. All other caches need to be invalidated. This transaction is generated by a PrWr to a block that is either not in the cache or is in the cache but not in modified state. Once the cache obtains the exclusive copy, the write can be performed in the cache. The processor may require an acknowledgment as a result of this transaction.

Writeback (BusWB): The cache controller puts the address and the contents for the memory block on the bus. The main memory is updated with the latest contents. This transaction is generated by the cache controller on a writeback; the processor does not know about it, and does not expect a response.

The Bus Read-Exclusive (sometimes called *read-to-own*) is a new transaction that would not exist except for cache coherence. The other new concept needed to support write-back protocols is that in addition to changing the state of cached blocks, the cache controller can intervene in the bus transaction and "flush" the contents of the referenced block onto the bus, rather than allow the memory to supply the data. Of course, the cache controller can also initiate new bus transactions as listed above, supply data for writebacks, or pick up data supplied by the memory system.

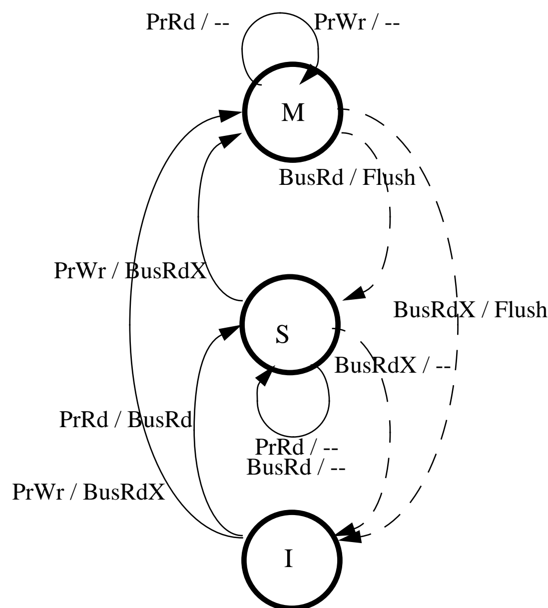


Figure 5-13 Basic 3-state invalidation protocol.

I, S, and M stand for Invalid, Shared, and Modified states respectively. The notation A / B means that if we observe from processor-side or bus-side event A, then in addition to state change, generate bus transaction or action B. "--" means null action. Transitions due to observed bus transactions are shown in dashed arcs, while those due to local processor actions are shown in bold arcs. If multiple A / B pairs are associated with an arc, it simply means that multiple inputs can cause the same state transition. For completeness, we need to specify actions from each state corresponding to each observable event. If such transitions are not shown, it means that they are uninteresting and no action needs to be taken. Replacements and the writebacks they may cause are not shown in the diagram for simplicity.

State Transitions

The state transition diagram that governs a block in each cache in this snoopy protocol is as shown in Figure 5-13. The states are organized so that the closer the state is to the top the more tightly the block is bound to that processor. A processor read to a block that is ‘invalid’ (including not present) causes a BusRd transaction to service the miss. The newly loaded block is promoted from invalid to the ‘shared’ state, whether or not any other cache holds a copy. Any other caches with the block in the ‘shared’ state observe the BusRd, but take no special action, allowing the memory to respond with the data. However, if a cache has the block in the ‘modified’ state (there can only be one) and it observes a BusRd transaction on the bus, then it must get involved in the transaction since the memory copy is stale. This cache flushes the data onto the bus, in lieu of memory, and demotes its copy of the block to the shared state. The memory and the requesting cache both pick up the block. This can be accomplished either by a direct cache-to-cache transfer across the bus during this BusRd transaction, or by signalling an error on the BusRd transaction and generating a write transaction to update memory. In the latter case, the original cache will eventually retry its request and obtain the block from memory. (It is also possible to have the flushed data be picked up only by the requesting cache but not by memory, leaving memory still out-of-date, but this requires more states [SwS86]).

Writing into an invalid block is a write miss, which is serviced by first loading the entire block and then modifying the desired bytes within it. The write miss generates a read exclusive bus transaction, which causes all other cached copies of the block to be invalidated, thereby granting the requesting cache exclusive ownership of the block. The block is raised to the ‘modified’ state and is then written. If another cache later requests exclusive access, then in response to its BusRdX transaction this block will be demoted to the invalid state after flushing the exclusive copy to the bus.

The most interesting transition occurs when writing into a shared block. As discussed earlier this is treated essentially like a write miss, using a read-exclusive bus transaction to acquire exclusive ownership, and we will refer to it as a miss throughout the book. The data that comes back in the read-exclusive can be ignored in this case, unlike when writing to an invalid or not present block, since it is already in the cache. In fact, a common optimization to reduce data traffic in bus protocols is to introduce a new transaction, called a *bus upgrade* or BusUpgr, for this situation. A BusUpgr which obtains exclusive ownership just like a BusRdX, by causing other copies to be invalidated, but it does not return the data for the block to the requestor. Regardless of whether a BusUpgr or a BusRdX is used (let us continue to assume BusRdX), the block transitions to modified state. Additional writes to the block while it is in the modified state generate no additional bus transactions.

A replacement of a block from a cache logically demotes a block to invalid (not present) by removing it from the cache. A replacement therefore causes the state machines for two blocks to change states in that cache: the one being replaced from its current state to invalid, and the one being brought in from invalid to its new state. The latter state change cannot take place before the former, which requires some care in implementation. If the block being replaced was in modified state, the replacement transition from M to I generates a write-back transaction. No special action is taken by the other caches on this transaction. If the block being replaced was in shared or invalid state, then no action is taken on the bus. Replacements are not shown in the state diagram for simplicity.

Note that to specify the protocol completely, for each state we must have outgoing arcs with labels corresponding to all observable events (the inputs from the processor and bus sides), and also actions corresponding to them. Of course, the actions can be null sometimes, and in that case we may either explicitly specify null actions (see states S and M in the figure), or we may simply omit those arcs from the diagram (see state I). Also, since we treat the not-present state as invalid, when a new block is brought into the cache on a miss the state transitions are performed as if the previous state of the block was invalid.

Example 5-6 Using the MSI protocol, show the state transitions and bus transactions for the scenario depicted in Figure 5-3.

Answer The results are shown in Figure 5-14.

Proc. Action	State in P1	State in P2	State in P3	Bus Action	Data Supplied By
1. P1 reads u	S	--	--	BusRd	Memory
2. P3 reads u	S	--	S	BusRd	Memory
3. P3 writes u	I	--	M	BusRdX	Memory
4. P1 reads u	S	--	S	BusRd	P3's cache
5. P2 reads u	S	S	S	BusRd	Memory

Figure 5-14 The 3-state invalidation protocol in action for processor transactions shown in Figure 5-3.

The figure shows the state of the relevant memory block at the end of each transaction, the bus transaction generated, if any, and the entity supplying the data.

With write-back protocols, a block can be written many times before the memory is actually updated. A read may obtain data not from memory but rather from a writer's cache, and in fact it may be this read rather than a replacement that causes memory to be updated. Also, write hits do not appear on the bus, so the concept of "performing a write" is a little different. In fact, it is often simpler to talk, equivalently, about the write being "made visible." A write to a shared or invalid block is made visible by the bus read-exclusive transaction it triggers. The writer will "observe" the data in its cache; other processors will experience a cache miss before observing the write. Write hits to a modified block are visible to other processors, but are observed by them only after a miss through a bus transaction. Formally, in the MSI protocol, the write to a non-modified block is performed or made visible when the BusRdX transaction occurs, and to a modified block when the block is updated in the writer's cache.

Satisfying Coherence

Since both reads and writes to local caches can take place concurrently with the write-back protocol, it is not obvious that it satisfies the conditions for coherence, much less sequential consistency. Let's examine coherence first. The read-exclusive transaction ensures that the writing

cache has the only valid copy when the block is actually written in the cache, just like a write in the write-through protocol. It is followed immediately by the corresponding write being performed in the cache, before any other bus transactions are handled by that cache controller.The only difference from a write-through protocol, with regard to ordering operations to a location, is that not all writes generate bus transactions. However, the key here is that between two transactions for that block that do appear on the bus, only one processor can perform such write hits; this is the processor, say P, that performed the most recent read-exclusive bus transaction *w* for the block. In the serialization, this write hit sequence therefore appears in program order between *w* and the next bus transaction for that block. Reads by processor P will clearly see them in this order with respect to other writes. For a read by another processor, there is at least one bus transaction that separates its completion from the completion of these write hits. That bus transaction ensures that this read also sees the writes in the consistent serial order. Thus, reads by all processors see all writes in the same order.

Satisfying Sequential Consistency

To see how sequential consistency is satisfied, let us first appeal to the definition itself and see how a consistent global interleaving of all memory operations may be constructed. The serialization for the bus, in fact, defines a total order on bus transactions for all blocks, not just those to a single block. All cache controllers observe read and read-exclusive transactions in the same order, and perform invalidations in this order. Between consecutive bus transactions, each processor performs a sequence of memory operations (read and writes) in program order. Thus, any execution of a program defines a natural partial order:

> A memory operation M_j is subsequent to operation M_i if (i) the operations are issued by the same processor and M_j follows M_i in program order, or (ii) M_j generates a bus transaction that follows the memory operation for M_i.

This partial order looks graphically like that of Figure 5-6, except the local sequence within a segment has reads and writes and both read-exclusive and read bus transactions play important roles in establishing the orders. Between bus transactions, any interleaving of the sequences from different processors leads to a consistent total order. In a segment between bus transactions, a processor can observe writes by other processors, ordered by previous bus transactions that it generated, as well as its own writes ordered by program order.

We can also see how SC is satisfied in terms of the sufficient conditions, which we will return to when we look further at implementing protocols. Write completion is detected when the read-exclusive bus transaction occurs on the bus and the write is performed in the cache. The third condition, which provides write atomicity, is met because a read either (i) causes a bus transaction that follows that of the write whose value is being returned, in which case the write must have completed globally before the read, or (ii) follows in program order such a read by the same processor, or (iii) follows in program order on the same processor that performed the write, in which case the processor has already waited for the read-exclusive transaction and the write to complete globally. Thus, all the sufficient conditions are easily guaranteed.

Lower-Level Design Choices

To illustrate some of the implicit design choices that have been made in the protocol, let us examine more closely the transition from the M state when a BusRd for that block is observed. In Figure 5-13 we transition to state S and the contents of the memory block are placed on the bus.

While it is imperative that the contents are placed on the bus, it could instead have transitioned to state I. The choice of going to S versus I reflects the designer's assertion that it is more likely that the original processor will continue reading that block than it is that the new processor will soon write to that memory block. Intuitively, this assertion holds for mostly-read data, which is common in many programs. However, a common case where it does not hold is for a flag or buffer that is used to transfer information back and forth between two processes: one processor writes it, the other reads it and modifies it, then the first reads it and modifies it, and so on. The problem with betting on read sharing in this case is that each write is preceded by an invalidate, thereby increasing the latency of the ping-pong operation. Indeed the coherence protocol used in the early Synapse multiprocessor made the alternate choice of directly going from M to I state on a BusRd. Some machines (Sequent Symmetry (model B) and the MIT Alewife) attempt to adapt the protocol when the access pattern indicates such migratory data [CoF93,DDS94]. The implementation of the protocol can also affect the performance of the memory system, as we will see later in the chapter.

5.4.2 A 4-state (MESI) Write-Back Invalidation Protocol

A serious concern with our MSI protocol arises if we consider a sequential application running on a multiprocessor; such multiprogrammed use in fact constitutes the most common workload on small-scale multiprocessors. When it reads in and modifies a data item, in the MSI protocol two bus transactions are generated even though there are never any sharers. The first is a BusRd that gets the memory block in S state, and the second is a BusRdX (or BusUpgr) that converts the block from S to M state. By adding a state that indicates that the block is the only (exclusive) copy but is not modified and loading the block in this state, we can save the latter transaction since the state indicates that no other processor is caching the block. This new state, called exclusive-clean or exclusive-unowned or sometime simply exclusive, indicates an intermediate level of binding between shared and modified. It is exclusive, so unlike the shared state it can perform a write and move to modified state without further bus transactions; but it does imply ownership, so unlike in the modified state the cache need not reply upon observing a request for the block (memory has a valid copy). Variants of this MESI protocol are used in many modern microprocessors, including the Intel Pentium, PowerPC 601, and the MIPS R4400 used in the Silicon Graphics Challenge multiprocessors. It was first published by researchers at the University of Illinois at Urbana-Champaign [PaP84] and is often referred to as the Illinois protocol [ArB86].

The MESI protocol consists of four states: *modified*(M) or dirty, *exclusive-clean* (E), *shared* (S), and *invalid* (I). I and M have the same semantics as before. E, the exclusive-clean or just exclusive state, means that only one cache (this cache) has a copy of the block, and it has not been modified (i.e., the main memory is up-to-date). S means that potentially two or more processors have this block in their cache in an unmodified state.

State Transitions

When the block is first read by a processor, if a valid copy exists in another cache then it enters the processor's cache in the S state as usual. However, if no other cache has a copy at the time (for example, in a sequential application), it enters the cache in the E state. When that block is written by the same processor, it can directly transition from E to M state without generating another bus transaction, since no other cache has a copy. If another cache had obtained a copy in the meantime, the state of the block would have been downgraded from E to S by the snoopy protocol.

This protocol places a new requirement on the physical interconnect of the bus. There must be an additional signal, the shared signal (S), available to the controllers in order to determine on a BusRd if any other cache currently holds the data. During the address phase of the bus transaction, all caches determine if they contain the requested block and, if so, assert the shared signal. This is a wired-OR line, so the controller making the request can observe whether there are any other processors caching the referenced memory block and thereby decide whether to load a requested block in the E state or the S state.

Figure 5-15 shows the state transition diagram for the MESI protocol, still assuming that the BusUpgr transaction is not used. The notation BusRd(S) means that when the bus read transac-

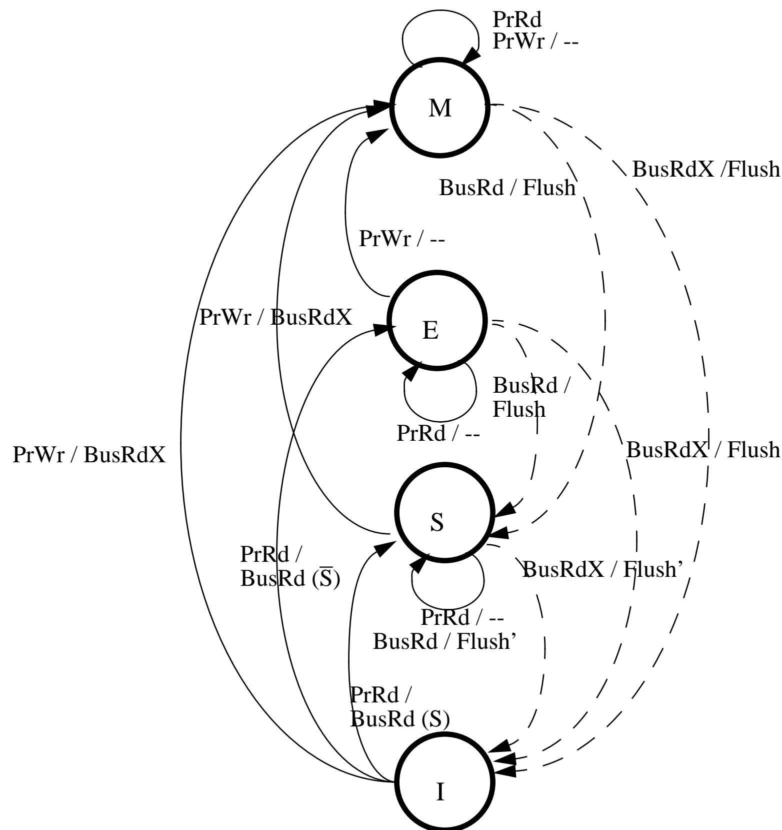


Figure 5-15 State transition diagram for the Illinois MESI protocol.

M, E, S, I stand for the Modified (dirty), Exclusive, Shared and Invalid states respectively. The notation is the same as that in Figure 5-13. The E state helps reduce bus traffic for sequential programs where data are not shared. Whenever feasible, the protocol also makes caches, rather than main memory, supply data for BusRd and BusRdX transactions. Since multiple processors may have a copy of the memory block in their cache, we need to select only one to supply the data on the bus. Flush’ is true only for that processor; the remaining processors take null action.

tion occurred the signal “S” was asserted, and BusRd($\overline{S}$) means “S” was unasserted. A plain BusRd means that we don’t care about the value of S for that transition. A write to a block in any state will elevate the block to the M state, but if it was in the E state no bus transaction is required. Observing a BusRd will demote a block from E to S, since now another cached copy exists. As usual, observing a BusRd will demote a block from M to S state and cause the block to be flushed on to the bus; here too, the block may be picked up only by the requesting cache and not by main memory, but this will require additional states beyond MESI [SwS86]. Notice that it

is possible for a block to be in the S state even if no other copies exist, since copies may be replaced (S -> I) without notifying other caches. The arguments for coherence and sequential consistency being satisfied here are the same as in the MSI protocol.

Lower-Level Design Choices

An interesting question for the protocol is who should supply the block for a BusRd transaction when both the memory and another cache have a copy of it. In the original Illinois version of the MESI protocol the cache, rather than main memory, supplied the data, a technique called *cache-to-cache sharing*. The argument for this approach was that caches being constructed out of SRAM, rather than DRAM, could supply the data more quickly. However, this advantage is not quite present in many modern bus-based machines, in which intervening in another processor's cache to obtain data is often more expensive than obtaining the data from main memory. Cache-to-cache sharing also adds complexity to a bus-based protocol. Main memory must wait until it is sure that no cache will supply the data before driving the bus, and if the data resides in multiple caches then there needs to be a selection algorithm to determine which one will provide the data. On the other hand, this technique is useful for multiprocessors with physically distributed memory, as we will see in Chapter 6, because the latency to obtain the data from a nearby cache may be much smaller than that for a far away memory unit. This effect can be especially important for machines constructed as a network of SMP nodes, because caches within the SMP node may supply the data. The Stanford DASH multiprocessor [LLJ+92] used such cache-to-cache transfers for this reason.

5.4.3 A 4-state (Dragon) Write-back Update Protocol

We now look at a basic update-based protocol for writeback caches, an enhanced version of which is used in the SUN SparcServer multiprocessors [Cat94]. This protocol was first proposed by researchers at Xerox PARC for their Dragon multiprocessor system [McC84,TLS88].

The Dragon protocol consists of four states: *exclusive-clean* (E), *shared-clean* (SC), *shared-modified* (SM), and *modified*(M). Exclusive-clean (or exclusive) again means that only one cache (this cache) has a copy of the block, and it has not been modified (i.e., the main memory is up-to-date). The motivation for adding the E state in Dragon is the same as that for the MESI protocol. SC means that potentially two or more processors (including this cache) have this block in their cache, and main memory may or may not be up-to-date. SM means that potentially two or more processor's have this block in their cache, main memory is not up-to-date, and it is this processor's responsibility to update the main memory at the time this block is replaced from the cache. A block may be in SM state in only one cache at a time. However, it is quite possible that one cache has the block in SM state while others have it in SC state. Or it may be that no cache has it in SM state but some have it in SC state. This is why when a cache has the block in SC state memory may or may not be up to date; it depends on whether some cache has it in SM state. M means, as before, that the block is modified (dirty) in this cache alone, main memory is stale, and it is this processor's responsibility to update main memory on replacement. Note that there is no explicit invalid (I) state as in the previous protocols. This is because Dragon is an update-based protocol; the protocol always keeps the blocks in the cache up-to-date, so it is always okay to use the data present in the cache. However, if a block is not present in a cache at all, it can be imagined to be there in a special invalid or not-present state.[1]

The processor requests, bus transactions, and actions for the Dragon protocol are similar to the Illinois MESI protocol. The processor is still assumed to issue only read (PrRd) and write (PrWr) requests. However, given that we do not have an invalid state in the protocol, to specify actions when a new memory block is first requested by the processor, we add two more request types: processor read-miss (PrRdMiss) and write-miss (PrWrMiss). As for bus transactions, we have bus read (BusRd), bus update (BusUpd), and bus writeback (BusWB). The BusRd and BusWB transactions have the usual semantics as defined for the earlier protocols. BusUpd is a new transaction that takes the specific word written by the processor and broadcasts it on the bus so that all other processors' caches can update themselves. By only broadcasting the contents of the specific word modified rather than the whole cache block, it is hoped that the bus bandwidth is more efficiently utilized. (See Exercise for reasons why this may not always be the case.) As in the MESI protocol, to support the E state there is a shared signal (S) available to the cache controller. This signal is asserted if there are any processors, other than the requestor, currently caching the referenced memory block. Finally, as for actions, the only new capability needed is for the cache controller to update a locally cached memory block (labeled Update) with the contents that are being broadcast on the bus by a relevant BusUpd transaction.

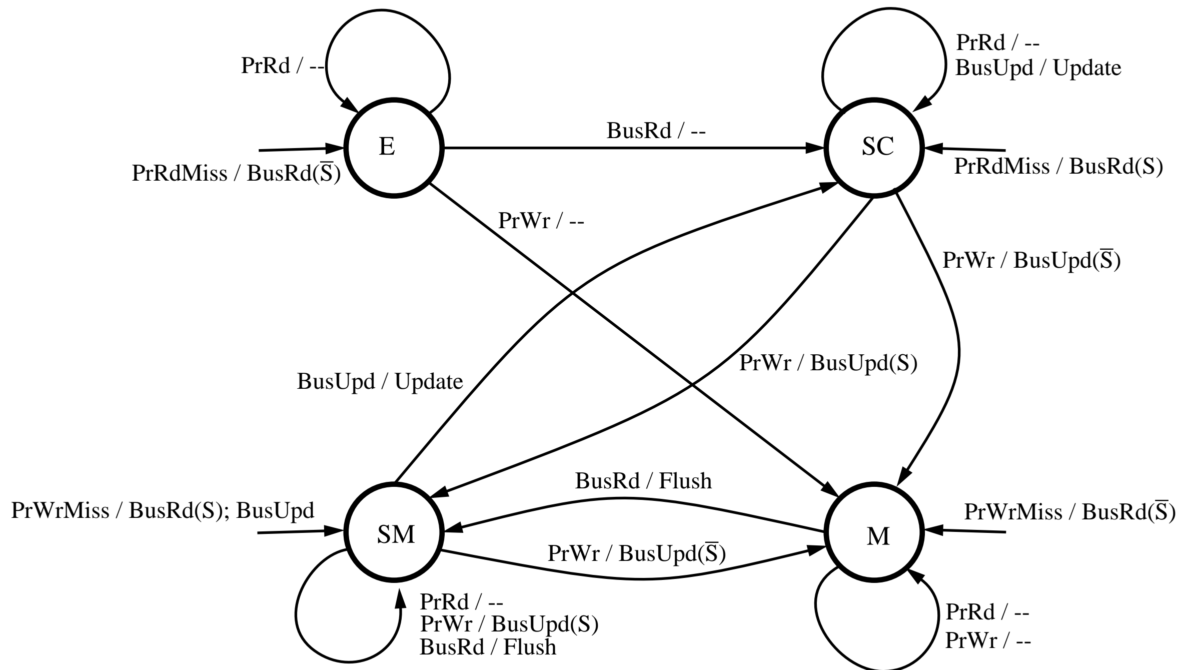


Figure 5-16 State-transition diagram for the Dragon update protocol.

The four states are valid-exclusive (E), shared-clean (SC), shared-modified (SM), and modified (M). There is no invalid (I) state because the update protocol always keeps blocks in the cache up-to-date.

1. Logically there is another state as well, but it is rather crude. A mode bit is provided with each block to force a miss when the block is accessed. Initialization software reads data into every line in the cache with the miss mode turned on, to ensure that the processor will miss the first time it references a block that maps to that line.

State Transitions

Figure 5-16 shows the state transition diagram for the Dragon update protocol. To take a processor-centric view, one can also explain the diagram in terms of actions taken when a processor incurs a read-miss, a write-hit, or a write-miss (no action is ever taken on a read-hit).

Read Miss: A BusRd transaction is generated. Depending on the status of the shared-signal (S), the block is loaded in the E or SC state in the local cache. More specifically, if the block is in M or SM states in one of the other caches, that cache asserts the shared signal and supplies the latest data for that block on the bus, and the block is loaded in local cache in SC state (it is also updated in main memory; if we did not want to do this, we would need more states). If the other cache had it in state M, it changes its state to SM. If no other cache has a copy, then the shared line remains unasserted, the data are supplied by the main memory and the block is loaded in local cache in E state.

Write: If the block is in the SM or M states in the local cache, then no action needs to be taken. If the block is in the E state in the local cache, then it internally changes to M state and again no further action is needed. If the block is in SC state, however, a BusUpd transaction is generated. If any other caches have a copy of the data, they update their cached copies, and change their state to SC. The local cache also updates its copy of the block and changes its state to SM. If no other cache has a copy of the data, the shared-signal remains unasserted, the local copy is updated and the state is changed to M. Finally, if on a write the block is not present in the cache, the write is treated simply as a read-miss transaction followed by a write transaction. Thus first a BusRd is generated, and then if the block was also found in other caches (i.e., the block is loaded locally in the SC state), a BusUpd is generated.

Replacement: On a replacement (arcs not shown in the figure), the block is written back to memory using a bus transaction only if it is in the M or SM state. If it is in the SC state, then either some other cache has it in SM state, or noone does in which case it is already valid in main memory.

Example 5-7 Using the Dragon update protocol, show the state transitions and bus transactions for the scenario depicted in Figure 5-3.

Answer The results are shown in Figure 5-17. We can see that while for processor actions 3 and 4 only one word is transferred on the bus in the update protocol, the whole memory block is transferred twice in the invalidation-based protocol. Of course, it is easy to construct scenarios where the invalidation protocol does much better than the update protocol, and we will discuss the detailed tradeoffs later in Section 5.5.

Lower-Level Design Choices

Again, many implicit design choices have been made in this protocol. For example, it is feasible to eliminate the shared-modified state. In fact, the update-protocol used in the DEC Firefly multiprocessor did exactly that. The reasoning was that every time the BusUpd transaction occurs, the main memory can also update its contents along with the other caches holding that block, and therefore, a shared-modified state is not needed. The Dragon protocol was instead based on the assumption that since caches use SRAM and main memory uses DRAM, it is much quicker to update the caches than main memory, and therefore it is inappropriate to wait for main memory

Proc. Action	State in P1	State in P2	State in P3	Bus Action	Data Supplied By
1. P1 reads u	E	--	--	BusRd	Memory
2. P3 reads u	SC	--	SC	BusRd	Memory
3. P3 writes u	SC	--	SM	BusUpd	P3
4. P1 reads u	SC	--	SM	null	--
5. P2 reads u	SC	SC	SM	BusRd	P3

Figure 5-17 The Dragon update protocol in action for processor transactions shown in Figure 5-3.

The figure shows the state of the relevant memory block at the end of each transaction, the bus transaction generated, if any, and the entity supplying the data.

to be updated on all BusUpd transactions. Another subtle choice, for example, relates to the action taken on cache replacements. When a shared-clean block is replaced, should other caches be informed of that replacement via a bus-transaction, so that if there is only one remaining cache with a copy of the memory block then it can change its state to valid-exclusive or modified? The advantage of doing this would be that the bus transaction upon the replacement may not be in the critical path of a memory operation, while the later bus transaction it saves might be.

Since all writes appear on the bus in an update protocol, write serialization, write completion detection, and write atomicity are all quite straightforward with a simple atomic bus, a lot like they were in the write-through case. However, with both invalidation and update based protocols, there are many subtle implementation issues and race conditions that we must address, even with an atomic bus and a single-level cache. We will discuss these in Chapter 6, as well as more realistic scenarios with pipelined buses, multi-level cache hierarchies, and hardware techniques that can reorder the completion of memory operations like write buffers. Nonetheless, we can quantitatively examine protocol tradeoffs at the state diagram level that we have been considering so far.

5.5 Assessing Protocol Design Tradeoffs

Like any other complex system, the design of a multiprocessor requires many decisions to be made. Even when a processor has been picked, the designer must decide on the maximum number of processors to be supported by the system, various parameters of the cache hierarchy (e.g., number of levels in the hierarchy, and for each level the cache size, associativity, block size, and whether the cache is write-through or writeback), the design of the bus (e.g., width of the data and address buses, the bus protocol), the design of the memory system (e.g., interleaved memory banks or not, width of memory banks, size of internal buffers), and the design of the I/O subsystem. Many of the issues are similar to those in uniprocessors [Smi82], but accentuated. For example, a write-through cache standing before the bus may be a poor choice for multiprocessors because the bus bandwidth is shared by many processors, and memory may need to be interleaved more because it services cache misses from multiple processors. Greater cache associativity may also be useful in reducing conflict misses that generate bus traffic.

A crucial new design issue for a multiprocessor is the cache coherence protocol. This includes protocol class (invalidation or update), protocol states and actions, and lower-level implementation tradeoffs that we will examine later. Protocol decisions interact with all the other design issues. On one hand, the protocol influences the extent to which the latency and bandwidth characteristics of system components are stressed. On the other, the performance characteristics as well as organization of the memory and communication architecture influence the choice of protocols. As discussed in Chapter 4, these design decisions need to be evaluated relative to the behavior of real programs. Such evaluation was very common in the late 1980s, albeit using an immature set of parallel programs as workloads [ArB86,AgG88,EgK88,EgK89a,EgK89b].

Making design decisions in real systems is part art and part science. The art is the past experience, intuition, and aesthetics of the designers, and the science is workload-driven evaluation. The goals are usually to meet a cost-performance target and to have a balanced system so that no individual resource is a performance bottleneck yet each resource has only minimal excess capacity. This section illustrates some key protocol tradeoffs by putting the workload driven evaluation methodology from Chapter 4 to action.

The basic strategy is as follows. The workload is executed on a simulator of a multiprocessor architecture, which has two coupled parts: a reference generator and a memory system simulator. The reference generator simulates the application program's processes and issues memory references to the simulator. The simulator simulates the operation of the caches, bus and state machines, and returns timing information to the reference generator, which uses this information to determine from which process to issue the next reference. The reference generator interleaves references from different processes, preserving timing relationships from the simulator as well as the synchronization relationships in the parallel program. By observing the state transitions in the simulator, we can determine the frequency of various events such as cache misses and bus transactions. We can then evaluate the effect of protocol choices in terms of other design parameters such as latency and bandwidth requirements.

Choosing parameters according to the methodology of Chapter 4, this section first establishes the basic state transition characteristics generated by the set of applications for the 4-state, Illinois MESI protocol. It then illustrates how to use these frequency measurements to obtain a preliminary quantitative analysis of design tradeoffs raised by the example protocols above, such as the use of the fourth, exclusive state in the MESI protocol and the use of BusUpgr rather then BusRdX transactions for the S->M transition. It also illustrates more traditional design issues, such as how the cache block size—the granularity of both coherence and communication—impacts the latency and bandwidth needs of the applications. To understand this effect, we classify cache misses into categories such as cold, capacity, and sharing misses, examine the effect of block size on each, and explain the results in light of application characteristics. Finally, this understanding of the applications is used to illustrate the tradeoffs between invalidation-based and update-based protocols, again in light of latency and bandwidth implications.

One important note is that this analysis is based on the frequency of various important events, not the absolute times taken or therefore the performance. This approach is common in studies of cache architecture, because the results transcend particular system implementations and technology assumptions. However, it should be viewed as only a preliminary analysis, since many detailed factors that might affect the performance tradeoffs in real systems are abstracted away. For example, measuring state transitions provides a means of calculating miss rates and bus traffic, but realistic values for latency, overhead, and occupancy are needed to translate the rates into the actual bandwidth requirements imposed on the system. And the bandwidth requirements

themselves do not translate into performance directly, but only indirectly by increasing the cost of misses due to contention. To obtain an estimate of bandwidth requirements, we may artificially assume that every reference takes a fixed number of cycles to complete. The difficulty is incorporating contention, because it depends on the timing parameters used and on the burstiness of the traffic which is not captured by the frequency measurements.

The simulations used in this section do not model contention, and in fact they do not even pretend to assume a realistic cost model. All memory operations are assumed to complete in the same amount of time (here a single cycle) regardless of whether they hit or miss in the cache. There are Three main reasons for this. First, the focus is on understanding inherent protocol behavior and tradeoffs in terms of event frequencies, not so much on performance. Second, since we are experimenting with different cache block sizes and organizations, we would like the interleaving of references from application processes on the simulator to be the same regardless of these choices; i.e. all protocols and block sizes should see the same trace of references. With execution-driven simulation, this is only possible if we make the cost of every memory operation the same in the simulations (e.g., one cycle), whether it hits or misses. Otherwise, if a reference misses with a small cache block but hits with a larger one (for example) then it will be delayed by different amounts in the interleaving in the two cases. It would therefore be difficult to determine which effects are inherently due to the protocol and which are due to the particular parameter values chosen. The third reason is that realistic simulations that model contention would take a lot more time. The disadvantage of using this simple model is that the timing model may affect some of the frequencies we observe; however, this effect is small for the applications we study.

5.5.1 Workloads

The illustrative workloads for coherent shared address space architectures include six parallel applications and computational kernels and one multiprogrammed workload. The parallel programs run in batch mode with exclusive access to the machine and do not include operating system activity, at least in the simulations, while the multiprogrammed workload includes operating system activity. The number of applications we use is relatively small, but they are primarily for illustration and we try to choose programs that represent important classes of computation and have widely varying characteristics.

The parallel applications used here are taken from the SPLASH2 application suite (see Appendix A) and were described in previous chapters. Three of them (Ocean, Barnes-Hut, and Raytrace) were used as case studies for developing parallel programs in Chapters 2 and 3. The frequencies of basic operations for the applications is given in Table 4-1. We now study them in more detail to assess design tradeoffs in cache coherency protocols.

Bandwidth requirement under the MESI protocol

Driving the address traces for the workloads depicted in Table 4-1 through a cache simulator modeling the Illinois MESI protocol generates the state-transition frequencies in Table 5-1. The data are presented as number of state-transitions of a particular type per 100 references issued by the processors. Note that in the tables, a new state NP (not-present) is introduced. This addition helps to clarify transitions where on a cache miss, one block is replaced (creating a transition from one of I, E, S, or M to NP) and a new block is brought in (creating a transition from NP to one of I, E, S, or M). For example, we can distinguish between writebacks done due to replacements (M -> NP transitions) and writebacks done because some other processor wrote to a modi-

fied block that was cached locally (M -> I transitions). Note that the sum of state transitions can be greater than 100, even though we are presenting averages per 100 references, because some references cause multiple state transitions. For example, a write-miss can cause two-transitions in the local processor's cache (e.g., S -> NP for the old block, and NP -> M) for the incoming block, plus transitions in other processor's cache invalidating their copies (I/E/S/M -> I).[1] This low-level state-transition frequency data is an extremely useful way to answer many different kinds of "what-if" questions. As a first example, we can determine the bandwidth requirement these applications would place on the memory system.

Table 5-1 State transitions per 1000 data memory references issued by the applications. The data assumes 16 processors, 1 Mbyte 4-way set-associative caches, 64-byte cache blocks, and the Illinois MESI coherence protocol.

Appln.	From/To	NP	I	E	S	M
Barnes-Hut	NP	0	0	0.0011	0.0362	0.0035
	I	0.0201	0	0.0001	0.1856	0.0010
	E	0.0000	0.0000	0.0153	0.0002	0.0010
	S	0.0029	0.2130	0	97.1712	0.1253
	M	0.0013	0.0010	0	0.1277	902.782
LU	NP	0	0	0.0000	0.6593	0.0011
	I	0.0000	0	0	0.0002	0.0003
	E	0.0000	0	0.4454	0.0004	0.2164
	S	0.0339	0.0001	0	302.702	0.0000
	M	0.0001	0.0007	0	0.2164	697.129
Ocean	NP	0	0	1.2484	0.9565	1.6787
	I	0.6362	0	0	1.8676	0.0015
	E	0.2040	0	14.0040	0.0240	0.9955
	S	0.4175	2.4994	0	134.716	2.2392
	M	2.6259	0.0015	0	2.2996	843.565
Radiosity	NP	0	0	0.0068	0.2581	0.0354
	I	0.0262	0	0	0.5766	0.0324
	E	0	0.0003	0.0241	0.0001	0.0060
	S	0.0092	0.7264	0	162.569	0.2768
	M	0.0219	0.0305	0	0.3125	839.507
	NP	0	0	0.0030	1.3153	5.4051

1. For the Multiprog workload, to speed up the simulations a 32Kbyte instruction cache is used as a filter, before passing the instruction references to the 1 Mbyte unified instruction and data cache. The state transition frequencies for the instruction references are computed based only on those references that missed in the L1 instruction cache. This filtering does not affect any of the bus-traffic data that we will generate using these numbers. Also, for Multiprog we present data separately for kernel instructions, kernel data references, user instructions, and user data references. A given reference may produce transitions of multiple types. For example, if a kernel instruction miss causes a modified user-data-block to be written back, then we will have one transition for kernel instructions from NP -> E/S and another transition for user data reference category from M -> NP.

Table 5-1 State transitions per 1000 data memory references issued by the applications. The data assumes 16 processors, 1 Mbyte 4-way set-associative caches, 64-byte cache blocks, and the Illinois MESI coherence protocol.

Appln.	From/To	NP	I	E	S	M
Radix	I	0.0485	0	0	0.4119	1.7050
	E	0.0006	0.0008	0.0284	0.0001	0
	S	0.0109	0.4156	0	84.6671	0.3051
	M	0.0173	4.2886	0	1.4982	906.945
Raytrace	NP	0	0	1.3358	0.15486	0.0026
	I	0.0242	0	0.0000	0.3403	0.0000
	E	0.8663	0	29.0187	0.3639	0.0175
	S	1.1181	0.3740	0	310.949	0.2898
	M	0.0559	0.0001	0	0.2970	661.011
Multiprog User Data References	NP	0	0	0.1675	0.5253	0.1843
	I	0.2619	0	0.0007	0.0072	0.0013
	E	0.0729	0.0008	11.6629	0.0221	0.0680
	S	0.3062	0.2787	0	214.6523	0.2570
	M	0.2134	0.1196	0	0.3732	772.7819
Multiprog User Instruction References	NP	0	0	3.2709	15.7722	0
	I	0	0	0	0	0
	E	1.3029	0	46.7898	1.8961	0
	S	16.9032	0	0	981.2618	0
	M	0	0	0	0	0
Multiprog Kernel Data References	NP	0	0	1.0241	1.7209	4.0793
	I	1.3950	0	0.0079	1.1495	0.1153
	E	0.5511	0.0063	55.7680	0.0999	0.3352
	S	1.2740	2.0514	0	393.5066	1.7800
	M	3.1827	0.3551	0	2.0732	542.4318
Multiprog Kernel Instruction References	NP	0	0	2.1799	26.5124	0
	I	0	0	0	0	0
	E	0.8829	0	5.2156	1.2223	0
	S	24.6963	0	0	1075.2158	0
	M	0	0	0	0	0

Example 5-8 Suppose that the integer-intensive applications run at 200 MIPS per processor, and the floating-point intensive applications at 200 MFLOPS per processor. Assuming that cache block transfers move 64 bytes on the data lines and that each bus

transaction involves six bytes of command and address on the address lines, what is the traffic generated per processor?

Answer The first step is to calculate the amount of traffic per instruction. We determine what bus-action needs to be taken for each of the possible state-transitions and how much traffic is associated with each transaction. For example, a M --> NP transition indicates that due to a miss a cache block needs to be written back. Similarly, an S -> M transition indicates that an upgrade request must be issued on the bus. The bus actions corresponding to all possible transitions are shown in Table 5-2. All transactions generate six bytes of address traffic and 64 bytes of address traffic, except BusUpgr, which only generates address traffic.

Table 5-2 Bus actions corresponding to state transitions in Illinois MESI protocol.

From/To	NP	I	E	S	M
NP	--	--	BusRd	BusRd	BusRdX
I	--	--	BusRd	BusRd	BusRdX
E	--	--	--	--	--
S	--	--	not poss.	--	BusUpgr
M	BusWB	BusWB	not poss.	BusWB	--

At this point in the analysis the cache parameters and cache protocol are pinned down (they are implicit in the state-transition data since they were provided to the simulator in order to generate the data.) and the bus design parameters are pinned down as well. We can now compute the traffic generated by the processors. Using Table 5-2 we can convert the transitions per 1000 memory references in Table 5-1 to bus transactions per 1000 memory references, and convert this to address and data traffic by multiplying by the traffic per transaction. Using the frequency of memory accesses in Table 4-1 we convert this to traffic per instruction, or per MFLOPS. Finally, multiplying by the assuming processing rate, we get the address and data bandwidth requirement for each application. The result of this calculation is shown by left-most bar in Figure 5-18.

The calculation in the example above gives the average bandwidth requirement under the assumption that the bus bandwidth is enough to allow the processors to execute at full rate. In practice. This calculation provides a useful basis for sizing the system. For example, on a machine such as the SGI Challenge with 1.2 GB/s of data bandwidth, the bus provides sufficient bandwidth to support 16 processors on the applications other than Radix. A typical rule of thumb is to leave 50% “head room” to allow for burstiness of data transfers. If the Ocean and Multiprog workloads were excluded, the bus could support up to 32 processors. If the bandwidth is not sufficient to support the application, the application will slow down due to memory waits. Thus, we would expect the speedup curve for Radix to flatten out quite quickly as the number of processors grows. In general a multiprocessor is used for a variety of workloads, many with low per-processor bandwidth requirements, so the designer will choose to support configurations of a size that would overcommit the bus on the most demanding applications.

5.5.2 Impact of Protocol Optimizations

Given this base design point, we can evaluate protocol tradeoffs under common machine parameter assumptions.

Example 5-9

We have described two invalidation protocols in this chapter -- the basic 3-state invalidation protocol and the Illinois MESI protocol. The key difference between them is the existence of the valid-exclusive state in the latter. How much bandwidth savings does the E state buy us?

Answer

The main advantage of the E state is that when going from E->M, no traffic need be generated. In this case, in the 3-state protocol we will have to generate a BusUpgr transaction to acquire exclusive ownership for that memory block. To compute bandwidth savings, all we have to do is put a BusUpgr for the E->M transition in Table 5-2, and then recompute the traffic as before. The middle bar in Figure 5-18 shows the resulting bandwidth requirements.

The example above illustrates how anecdotal rationale for a more complex design may not stand up to quantitative analysis. We see that, somewhat contrary to expectations, the E state offers negligible savings. This is true even for the Multiprog workload, which consists primarily of sequential jobs. The primary reason for this negligible gain is that the fraction of E->M transitions is quite small. In addition, the BusUpgr transaction that would have been needed for the S->M transition takes only 6 bytes of address traffic and no data traffic.

Example 5-10

Recall that for the 3-state protocol, for a write that finds the memory block in shared state in the cache we issue a BusUpgr request on the bus, rather than a BusRdX. This saves bandwidth, as no data need be transferred for a BusUpgr, but it does complicate the implementation, since local copy of data can be invalidated in the cache before the upgrade request comes back. The question is how much bandwidth are we saving for taking on this extra complexity.

Answer

To compute the bandwidth for the less complex implementation, all we have to do is put in BusRdX in the E->M and S->M transitions in Table 5-2 and then recompute the bandwidth numbers. The results for all applications are shown in the right-most bar in Figure 5-18. While for most applications the difference in bandwidth is small, Ocean and Multiprog kernel-data references show that it can be as large as 10-20% on demanding applications.

The performance impact of these differences in bandwidth requirement depend on how the bus transactions are actually implemented. However, this high level analysis indicates to the designer where more detailed evaluation is required.

Finally, as we had discussed in Chapter 3, given the input data-set sizes we are using for the above applications, it is important that we run the Ocean, Raytrace, Radix applications for smaller 64 Kbyte cache sizes. The raw state-transition data for this case are presented in Table 5-3 below, and the per-processor bandwidth requirements are shown in Figure 5-19. As we can see,

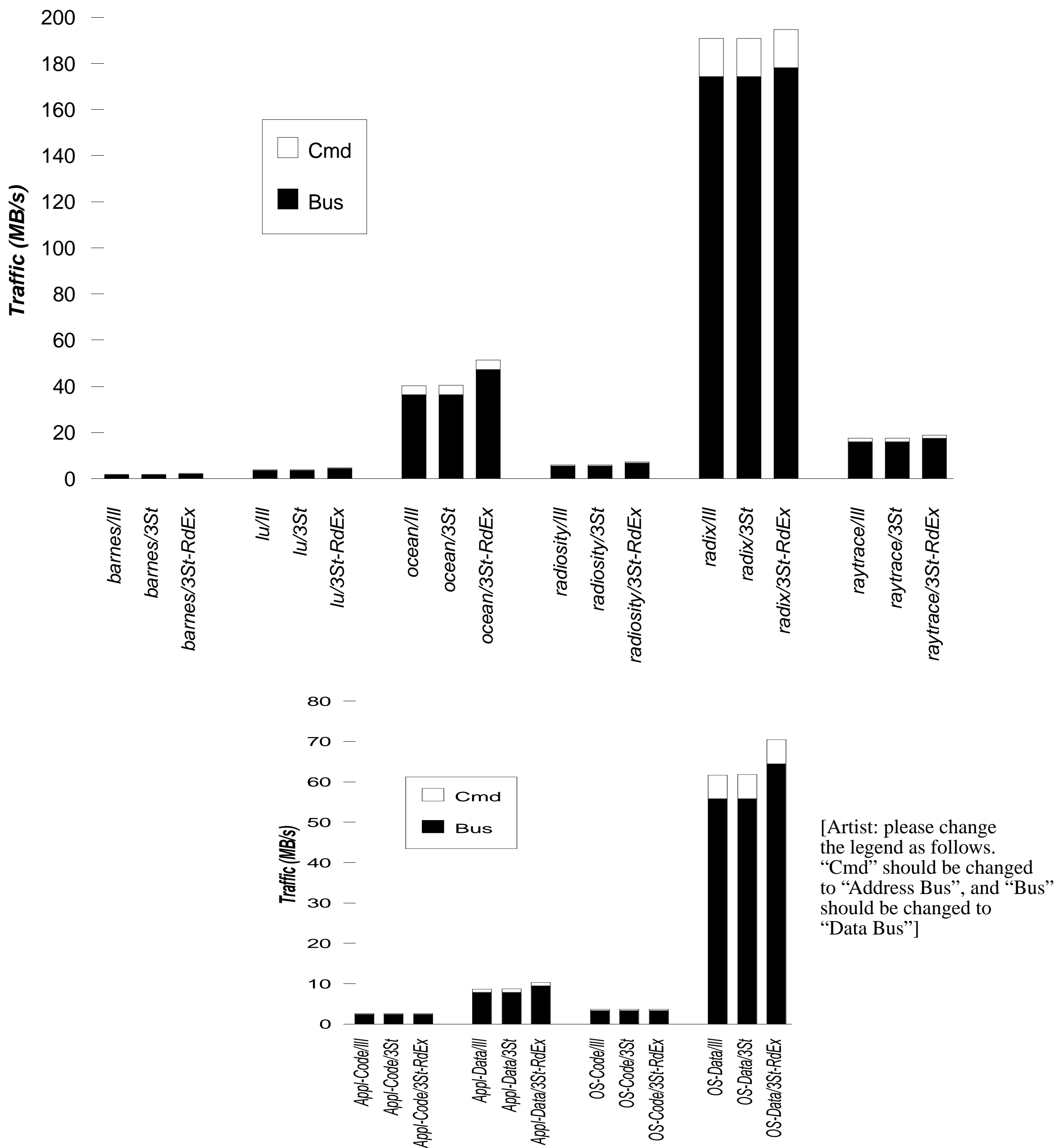


Figure 5-18 Per processor bandwidth requirements for the various applications assuming 200MIPS/MFLOPS processors.

The top bar-chart shows data for SPLASH-2 applications and the bottom chart data for the Multiprog workload. The traffic is split into data traffic and address+command bus traffic. The leftmost bar shows traffic for the Illinois MESI protocol, the middle bar for the case where we use the basic 3-state invalidation protocol without the E state as described in Section 5.4.1, and the right most bar shows the traffic for the 3-state protocol when we use BusRdX instead of BusUpgr for S -> M transitions.

Table 5-3 State transitions per 1000 memory references issued by the applications. The data assumes 16 processors, 64 Kbyte 4-way set-associative caches, 64-byte cache blocks, and the Illinois MESI coherence protocol.

Appln.	From/To	NP	I	E	S	M
Ocean	NP	0	0	26.2491	2.6030	15.1459
	I	1.3305	0	0	0.3012	0.0008
	E	21.1804	0.2976	452.580	0.4489	4.3216
	S	2.4632	1.3333	0	113.257	1.1112
	M	19.0240	0.0015	0	1.5543	387.780
Radix	NP	0	0	3.5130	0.9580	11.3543
	I	1.6323	0	0.0001	0.0584	0.5556
	E	3.0299	0.0005	52.4198	0.0041	0.0481
	S	1.4251	0.1797	0	56.5313	0.1812
	M	8.5830	2.1011	0	0.7695	875.227
Raytrace	NP	0	0	7.2642	3.9742	0.1305
	I	0.0526	0	0.0003	0.2799	0.0000
	E	6.4119	0	131.944	0.7973	0.0496
	S	4.6768	0.3329	0	205.994	0.2835
	M	0.1812	0.0001	0	0.2837	660.753

not having one of the critical working sets fit in the processor cache can dramatically increase the bandwidth required. A 1.2 Gbyte/sec bus can now barely support 4 processors for Ocean and Radix, and 16 processors for Raytrace.

5.5.3 Tradeoffs in Cache Block Size

The cache organization is a critical performance factor of all modern computers, but it is especially so in multiprocessors. In the uniprocessor context, cache misses are typically categorized into the "three-Cs": compulsory, capacity, and conflict misses [HiS89,PH90]. Many studies have examined how cache size, associativity, and block size affect each category of miss. *Compulsory misses*, or *cold misses*, occur on the first reference to a memory block by a processor. *Capacity misses* occur when all the blocks that are referenced by a processor during the execution of a program do not fit in the cache (even with full associativity), so some blocks are replaced and later accessed again. *Conflict* or *collision misses* occur in caches with less than full associativity, when the collection of blocks referenced by a program that map to a single cache set do not fit in the set. They are misses that would not have occurred in a fully associative cache.

Capacity misses are reduced by enlarging the cache. Conflict misses are reduced by increasing the associativity or increasing the number of blocks (increasing cache size or reducing block size). Cold misses can only be reduced by increasing the block size, so that a single cold miss will bring in more data that may be accessed as well. What makes cache design challenging is that these factors trade-off against one another. For example, increasing the block size for a fixed cache capacity will reduce the number of blocks, so the reduced cold misses may come at the cost of increased conflict misses. In addition, variations in cache organization can affect the miss penalty or the hit time, and therefore cycle time.

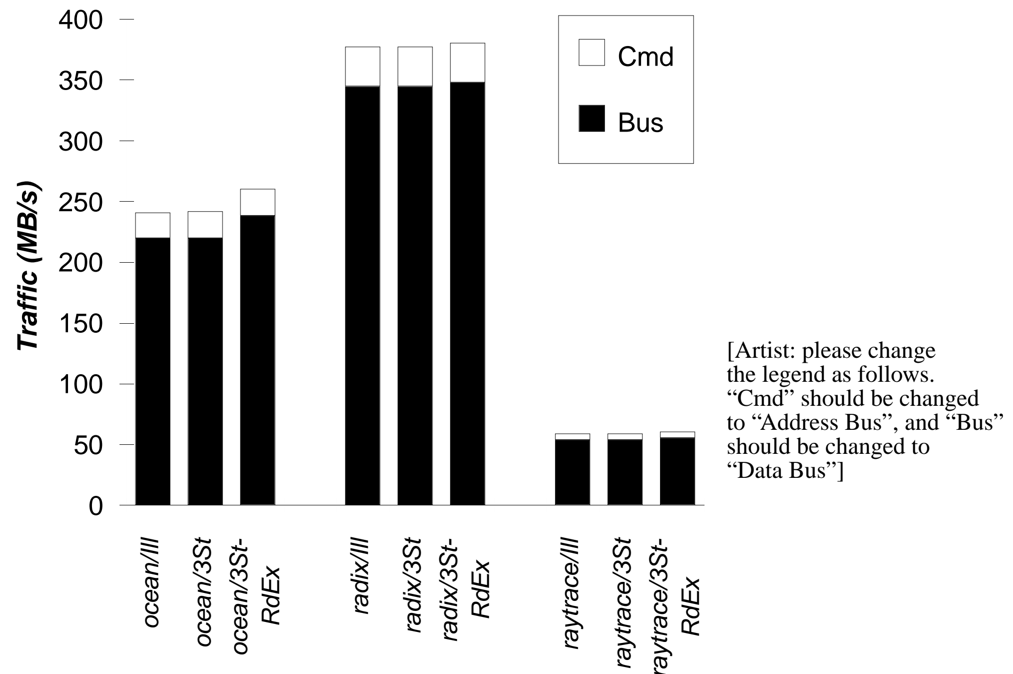


Figure 5-19 Per processor bandwidth requirements for the various applications assuming 200MIPS/MFLOPS processors and 64 Kbyte caches.

The traffic is split into data traffic and address+cmd bus traffic. The leftmost bar shows traffic for the Illinois MESI protocol, the middle bar for the case where we use the basic 3-state invalidation protocol without the E state as described in Section 5.4.1, and the right most bar shows the traffic for the 3-state protocol when we use BusRdX instead of BusUpgr for S -> M transitions.

Cache-coherent multiprocessors introduce a fourth category of misses: *coherence misses*. These occur when blocks of data are shared among multiple caches, and are of two types: true sharing and false sharing. True sharing occurs when a data word produced by one processor is used by another. False sharing occurs when independent data words for different processors happen to be placed in the same block. The cache block size is the granularity (or unit) of fetching data from the main memory, and it is typically also used as the granularity of coherence. That is, on a write by a processor, the whole cache block is invalidated in other processors' caches. A *true sharing miss* occurs when one processor writes some words in a cache block, invalidating that block in another processor's cache, and then the second processor reads one of the modified words. It is called a "true" sharing miss because the miss truly communicates newly defined data values that are used by the second processor; such misses are "essential" to the correctness of the program in an invalidation-based coherence protocol, regardless of interactions with the machine organization. On the other hand, when one processor writes some words in a cache block and then another processor reads (or writes) different words in the same cache block, the invalidation of the block and subsequent cache miss occurs as well, even though no useful values are being communicated between the processors. These misses are thus called *false-sharing misses* [DSR+93]. As cache block size is increased, the probability of distinct variables being accessed by different processors but residing on the same cache block increases. Technology pushes in the direction of large cache block sizes (e.g., DRAM organization and access modes, and the need to obtain high-band-

width data transfers by amortizing overhead), so it is important to understand the potential impact of false sharing misses and how they may be avoided.

True-sharing misses are inherent to a given parallel decomposition and assignment; like cold misses, the only way to decrease them is by increasing the block size and increasing spatial locality of communicated data. False sharing misses, on the other hand, are an example of the artifactual communication discussed in Chapter 3, since they are caused by interactions with the architecture. In contrast to true sharing and cold misses, false sharing misses can be decreased by reducing the cache block size, as well as by a host of other optimizations in software (orchestration) and hardware. Thus, there is a fundamental tension in determining the best cache block size, which can only be resolved by evaluating the options against real programs.

A Classification of Cache Misses

The flowchart in Figure 5-20 gives a detailed algorithm for classification of misses.[1] Understanding the details is not very important for now—it is enough for the rest of the chapter to understand the definitions above—but it adds insight and is a useful exercise. In the algorithm, we define the *lifetime* of a block as the time interval during which the block remains valid in the cache, that is, from the occurrence of the miss until its invalidation, replacement, or until the end of simulation. Observe that we cannot classify a cache miss when it occurs, but only when the fetched memory block is replaced or invalidated in the cache, because we need to know if during the block's lifetime the processor used any words that were written since the last true-sharing or essential miss. Let us consider the simple cases first. Cases 1 and 2 are straightforward cold misses occurring on previously unwritten blocks. Cases 7 and 8 reflect false and true sharing on a block that was previously invalidated in the cache, but not discarded. The type of sharing is determined by whether the word(s) modified since the invalidation are actually used. Case 11 is a straightforward capacity (or conflict) miss, since the block was previously replaced from the cache and the words in the block have not been accessed since last modified. All of the other cases refer to misses that occur due to a combination of factors. For example, cases 4 and 5 are cold misses because this processor has never accessed the block before, however, some other processor has written the block, so there is also sharing (false or true, depending on which words are accessed). Similarly, we can have sharing (false or true) on blocks that were previously discarded due to capacity. Solving only one of the problems may not necessarily eliminate such misses. Thus, for example, if a miss occurs due to both false-sharing and capacity problems, then eliminating the false-sharing problem by reducing block size will likely not eliminate that miss. On the other hand, sharing misses are in some sense more fundamental than capacity misses, since they will remain there even if the size of cache is increased to infinity.

Example 5-11 To illustrate the definitions that we have just given, Table 5-4 below shows how to classify references issued by three processors P1, P2, and P3. For this example, we

1. In this classification we do not distinguish conflict from capacity misses, since they both are a result of the available resources becoming full and the difference between them (set versus entire cache) does not shed additional light on multiprocessor issues.

assume that each processor's cache consists of a single 4-word cache block. We also assume that the caches are all initially empty.

Table 5-4 Classifying misses in an example reference stream from 3 processors. If multiple references are listed in the same row, we assume that P1 issues before P2 and P2 issues before P3. The notation ld/st wi refers to load/store of word i. The notation Pi.j points to the memory reference issued by processor i at row j.

Seq.	P1	P2	P3	Miss Classification
1	ld w0		ld w2	P1 and P3 miss; but we will classify later on replace/ inval
2			st w2	P1.1: pure-cold miss; P3.2: upgrade
3		ld w1		P2 misses, but we will classify later on replace/inval
4		ld w2	ld w7	P2 hits; P3 misses; P3.1: cold miss
5	ld w5			P1 misses
6		ld w6		P2 misses; P2.3: cold-true-sharing miss (w2 accessed)
7		st w6		P1.5: cold miss; p2.7: upgrade; P3.4: pure-cold miss
8	ld w5			P1 misses
9	ld w6		ld w2	P1 hits; P3 misses
10	ld w2	ld w1		P1, P2 miss; P1.8: pure-true-share miss; P2.6: cold miss
11	st w5			P1 misses; P1.10: pure-true-sharing miss
12			st w2	P2.10: capacity miss; P3.11: upgrade
13			ld w7	P3 misses; P3.9: capacity miss
14			ld w2	P3 misses; P3.13: inval-cap-false-sharing miss
15	ld w0			P1 misses; P1.11: capacity miss

Impact of Block Size on Miss Rate

Applying the classification algorithm of Figure 5-20 to simulated runs of a workload we can determine how frequently the various kinds of misses occur in programs and how the frequencies change with variations in cache organization, such as block size. Figure 5-21 shows the decomposition of the misses for the example applications running on 16 processors, 1 Mbyte 4-way set associative caches, as the cache block size is varied from 8 bytes to 256 bytes. The bars show the four basic types of misses, cold misses (cases 1 and 2), capacity misses (case 11), true-sharing misses, (cases 4, 6, 8, 10, 12), and false-sharing misses (cases 3, 5, 7, and 9). In addition, the figure shows the frequency of *upgrades,* which are writes that find the block in the cache but in shared state. They are different than the other types of misses in that the cache already has the valid data so all that needs to be acquired is exclusive ownership, and they are are not included in the classification scheme of Figure 5-20. However, they are still usually considered to be misses since they generate traffic on the interconnect and can stall the processor.

While the table only shows data for the default data-sets, in practice it is very important to examine the results as input data-set size and number of processors are scaled, before drawing conclusions about the false-sharing or spatial locality of an application. The impact of such scaling is elaborated briefly in the discussion below.

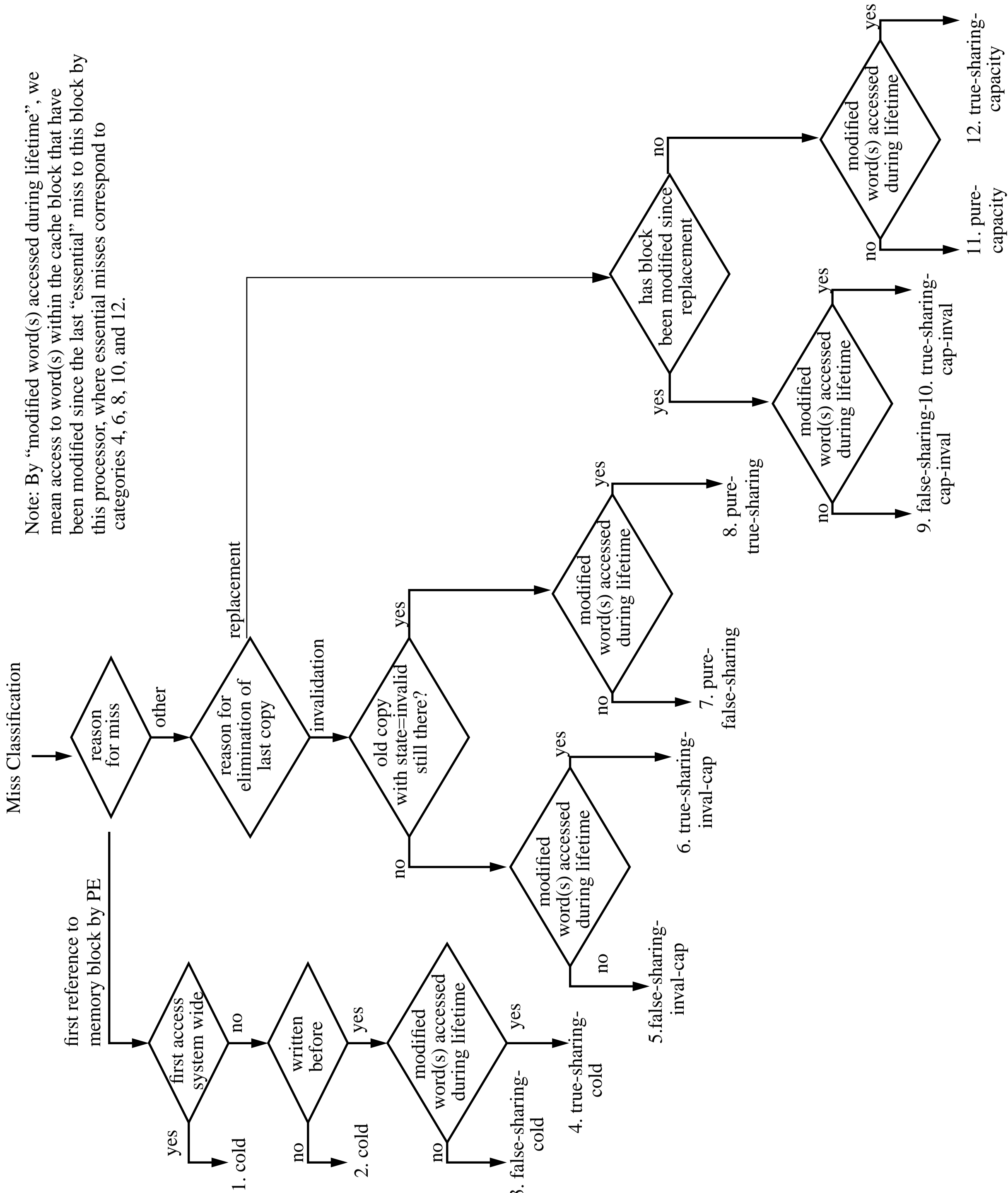


Figure 5-20 A classification of cache misses for shared-memory multiprocessors.

The four basic categories of cache misses are cold, capacity, true-sharing, and false-sharing misses. Many mixed categories arise because there may be multiple causes for the miss. For example, a block may be first replaced from processor A’s cache, then be written to by processor B, and then be read back by processor A, making it a capacity-cum-invalidation false/true sharing miss.

[Artist: please change
the legend text as follows:
UPGMR --> Upgrade
FSMR--> False Sharing
TSMR --> True Sharing
CAPMR --> Capacity
COLDMR --> Cold]

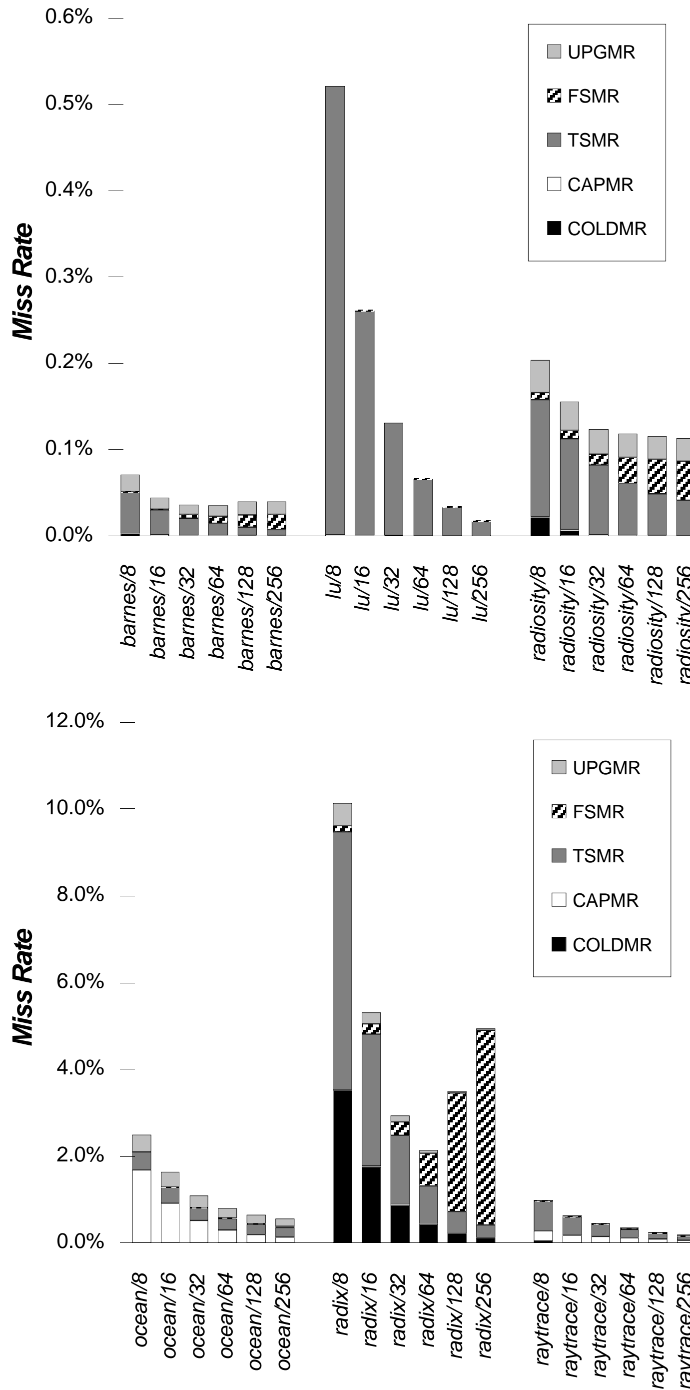


Figure 5-21 Breakdown of application miss rates as a function of cache block size for 1MB per-processor caches.

Conflict misses are included in capacity misses. The breakdown and behavior of misses varies greatly across applications, but there are some sommon trends. Cold misses and capacity misses tend to decrease quite quickly with block size due to spatial locality. True sharing misses. True sharing misses also tend to decrease, while false sharing misses increase. While the false sharing component is usually small for small block sizes, it sometimes remains small and sometimes increases very quickly.

For each individual application the miss characteristics change with block size much as we would expect, however, the overall characteristics differ widely across the sample programs. Cold, capacity and true sharing misses decrease with block size because the additional data brought in with each miss is accessed before the block is replaced, due to spatial locality. However, false sharing misses tend to increase with block size. In all cases, true sharing is a significant fraction of the misses, so even with ideal, infinite caches the miss rate and bus bandwidth will not go to zero. However, the size of the true sharing component varies significantly. Furthermore, some applications show substantial increase in false sharing with block size, while others show almost none. Below we investigate the properties of the applications that give rise to these differences observed at the machine level.

Relation to Application Structure

Multi-word cache blocks exploit spatial locality by prefetching data surrounding the accessed address. Of course, beyond a point larger cache blocks can hurt performance by: (i) prefetching unneeded data (ii) causing increased conflict misses, as the number of distinct blocks that can be stored in a finite-cache decreases with increasing block size; and (iii) causing increased false-sharing misses. Spatial locality in parallel programs tends to be lower than in sequential programs because when a memory block is brought into the cache some of the data therein will belong to another processor and will not be used by the processor performing the miss. As an extreme example, some parallel decompositions of scientific programs assign adjacent elements of an array to be handled by different processors to ensure good load balance, and in the process substantially decrease the spatial locality of the program.

The data in Figure 5-21 show that LU and Ocean have both good spatial locality and little false sharing. There is good spatial locality because the miss-rates drop proportionately to increases in cache block size and there are essentially no false sharing misses. This is in large part because these matrix codes use architecturally-aware data-structures. For example, a grid in Ocean is not represented as a single 2-D array (which can introduce substantial false-sharing), but as a 2-D array of blocks each of which is itself a 2-D array. Such structuring, by programmers or compilers, ensures that most accesses are unit-stride and over small blocks of data, and thus the nice behavior. As for scaling, the spatial locality for these applications is expected to remain good and false-sharing non-existent both as the problem size is increased and as the number of processors is increased. This should be true even for cache blocks larger than 256 bytes.

The graphics application Raytrace also shows negligible false sharing and somewhat poor spatial locality. The false sharing is small because the main data structure (collection of polygons constituting the scene) is read-only. The only read-write sharing happens on the image-plane data structure, but that is well controlled and thus only a small factor. This true-sharing miss rate reduces well with increasing cache block size. The reason for the poor spatial locality of capacity misses (although the overall magnitude is small) is that the access pattern to the collection of polygons is quite arbitrary, since the set of objects that a ray will bounce off is unpredictable. As for scaling, as problem size is increased (most likely in the form of more polygons) the primary effect is likely to be larger capacity miss rates; the spatial locality and false-sharing should not change. A larger number of processors is in most ways similar to having a smaller problem size, except that we may see slightly more false sharing in the image-plane data structure.

The Barnes-Hut and Radiosity applications show moderate spatial locality and false sharing. These applications employ complex data structures, including trees encoding spatial information

and arrays where records assigned to each processor are not contiguous in memory. For example, Barnes-Hut operates on particle records stored in an array. As the application proceeds and particles move in physical space, particle records get reassigned to different processors, with the result that after some time adjacent particles most likely belong to different processors. True sharing misses then seldom see good spatial locality because adjacent records within a cache block are often not touched. For the same reason, false sharing becomes a problem at large block sizes, as different processors write to records that are adjacent within a cache block. Another reason for the false-sharing misses is that the particle data structure (record) brings together (i) fields that are being modified by the owner of that particle (e.g., the current force on this particle), and (ii) fields that are read-only by other processors and that are not being modified in this phase (e.g., the current position of the particle). Since these two fields may fall in the same cache block for large block sizes, false-sharing results. It is possible to eliminate such false-sharing by splitting the particle data structure but that is not currently done, as the absolute magnitude of the miss-rate is small. As problem size is scaled and number of processors is scaled, the behavior of Barnes-Hut is not expected to change much. This is because the working set size changes very slowly (as log of number of particles), spatial locality is determined by the record size of one particle and thus remains the same, and finally because the sources of false sharing are not sensitive to number of processors. Radiosity is, unfortunately, a much more complex application whose behavior is difficult to reason about with larger data sets or more processors; the only option is to gather empirical data showing the growth trends.

The poorest sharing behavior is exhibited by Radix, which not only has a very high miss rate (due to cold and true sharing misses), but which gets significantly worse due to false sharing misses for block sizes of 128 bytes or more. The false-sharing in Radix arises as follows. Consider sorting 256K keys, using a radix of 1024, and 16 processors. On average, this results in 16 keys per radix per processor (64 bytes of data), which are then written to a contiguous portion of a global array at a random (for our purposes) starting point. Since 64 bytes is smaller than the 128 byte cache blocks, the high potential for false-sharing is clear. As the problem size is increased (e.g., to 4 million keys above), it is clear that we will see much less false sharing. The effect of increasing number of processors is exactly the opposite. Radix illustrates clearly that it is not sufficient to look at a given problem size, a given number of processors, and based on that draw conclusions of whether false-sharing or spatial locality are or are not a problem. It is very important to understand how the results are dependent on the particular parameters chosen in the experiment and how these parameters may vary in reality.

Data for the Multiprog workload for 1 Mbyte caches are shown in Figure 5-22. The data are shown separately for user-code, user-data, kernel-code, and kernel data. As one would expect, for code, there are only cold and capacity misses. Furthermore, we see that the spatial locality is quite low. Similarly, we see that the spatial locality in data references is quite low. This is true for the application data misses, because `gcc` (the main application causing misses in Multiprog) uses a large number of linked lists, which obviously do not offer good spatial locality. It is also somewhat interesting that we have an observable fraction of true-sharing misses, although we are running only sequential applications. They arise due to process migration; when a sequential process migrates from one processor to another and then references memory blocks that it had written while it was executing on the other processor, they appear as true sharing misses. We see that the kernel data misses, far outnumber the user data misses and have just as poor spatial locality. While the spatial locality in cold and capacity misses is quite reasonable, the true-sharing misses do not decrease at all for kernel data.

[Artist: same change to legend as in previous figure.]

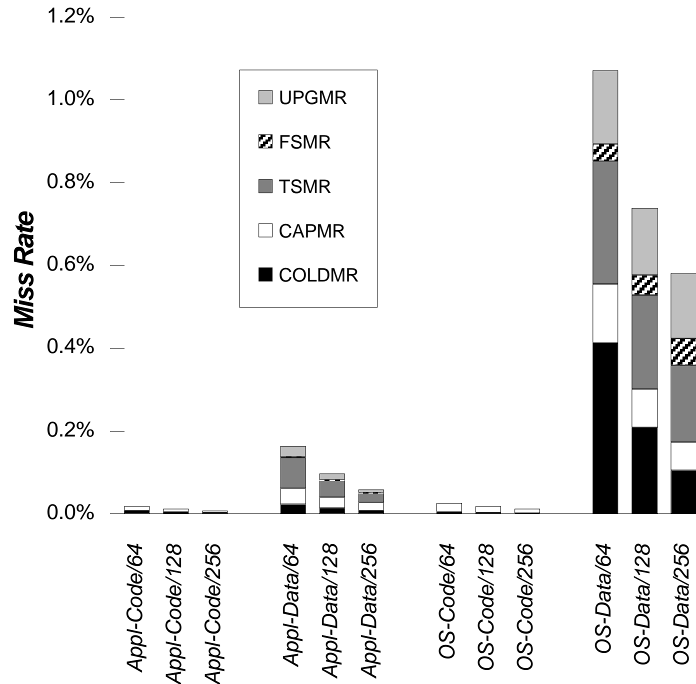


Figure 5-22 Breakdown of miss rates for Multiprog as a function of cache block size.

The results are for 1MB caches. Spatial locality for true sharing misses is much better for the applications than for the OS.

It is also interesting to look at the behavior of Ocean, Radix, and Raytrace for smaller 64 Kbyte caches. The miss-rate results are shown in Figure 5-23. As expected, the overall miss rates are higher and capacity misses have increased substantially. However, the spatial locality and false-sharing trends are not changed significantly as compared to the results for 1 Mbyte caches, mostly because these properties are fairly fundamental to data structure and algorithms used by a program and are not too sensitive to cache size. The key new interaction is for the Ocean application, where the capacity misses indicate somewhat poor spatial locality. The reason is that because of the small cache size data blocks are being thrown out of the cache due to interference before the processor has had a chance to reference all of the words in a cache block. Results for false sharing and spatial locality for other applications can be found in the literature [TLH94,JeE91].

Impact of Block Size on Bus Traffic

Let us briefly examine impact of cache-block size on bus traffic rather than miss rate. Note that while the number of misses and total traffic generated are clearly related, their impact on observed performance can be quite different. Misses have a cost that may contribute directly to performance, even though modern microprocessors try hard to hide the latency of misses by overlapping it with other activities. Traffic, on the other hand, affects performance only indirectly by causing contention and hence increasing the cost of other misses. For example, if an application program's misses are halved by increasing the cache block size but the bus traffic is doubled, this might be a reasonable trade-off if the application was originally using only 10% of the available bus and memory bandwidth. Increasing the bus/memory utilization to 20% is unlikely to

[Artist: same change to legend as in previous figure.]

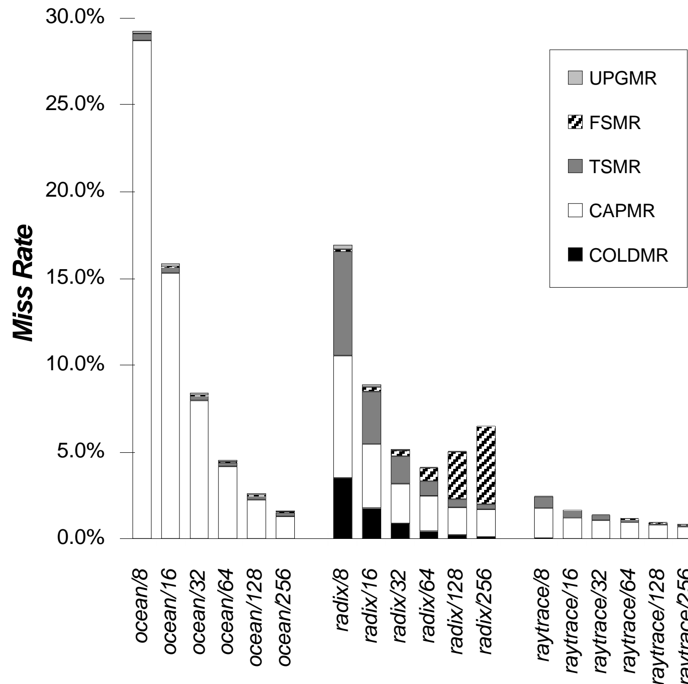


Figure 5-23 Breakdown of application miss-rates as a function of cache block size for 64 Kbyte caches.

Capacity misses are now a much larger fraction of the overall miss rate. Capacity miss rate decreases differently with block size for different applications.

increase the miss latencies significantly. However, if the application was originally using 75% of the bus/memory bandwidth, then increasing the block size is probably a bad idea under these circumstances

Figure 5-24 shows the total bus traffic in bytes/instruction or bytes/FLOP as block size is varied. There are three points to observe from this graph. Firstly, although overall miss rate decreases for most of the applications for the block sizes presented, traffic behaves quite differently. Only LU shows monotonically decreasing traffic for these block sizes. Most other applications see a doubling or tripling of traffic as block size becomes large. Secondly, given the above, the overall traffic requirements for the applications are still small for very large block sizes, with the exception of Radix. Radix's large bandwidth requirements (~650 Mbytes/sec per-processor for 128-byte cache blocks, assuming 200 MIPS processor) reflect its false sharing problems at large block sizes. Finally, the constant overhead for each bus transaction comprises a significant fraction of total traffic for small block sizes. Hence, although actual data traffic increases as we increase the block size, due to poor spatial locality, the total traffic is often minimized at 16-32 bytes. Figure 5-25 shows the data for Multiprog. While the bandwidth increase from 64 byte cache blocks to 128 byte blocks is small, the bandwidth requirements jump substantially at 256 byte cache blocks (primarily due to kernel data references). Finally, in Figure 5-26 we show traffic results for 64 Kbyte caches. Notice that for Ocean, even 64 and 128 byte cache blocks do not look so bad.

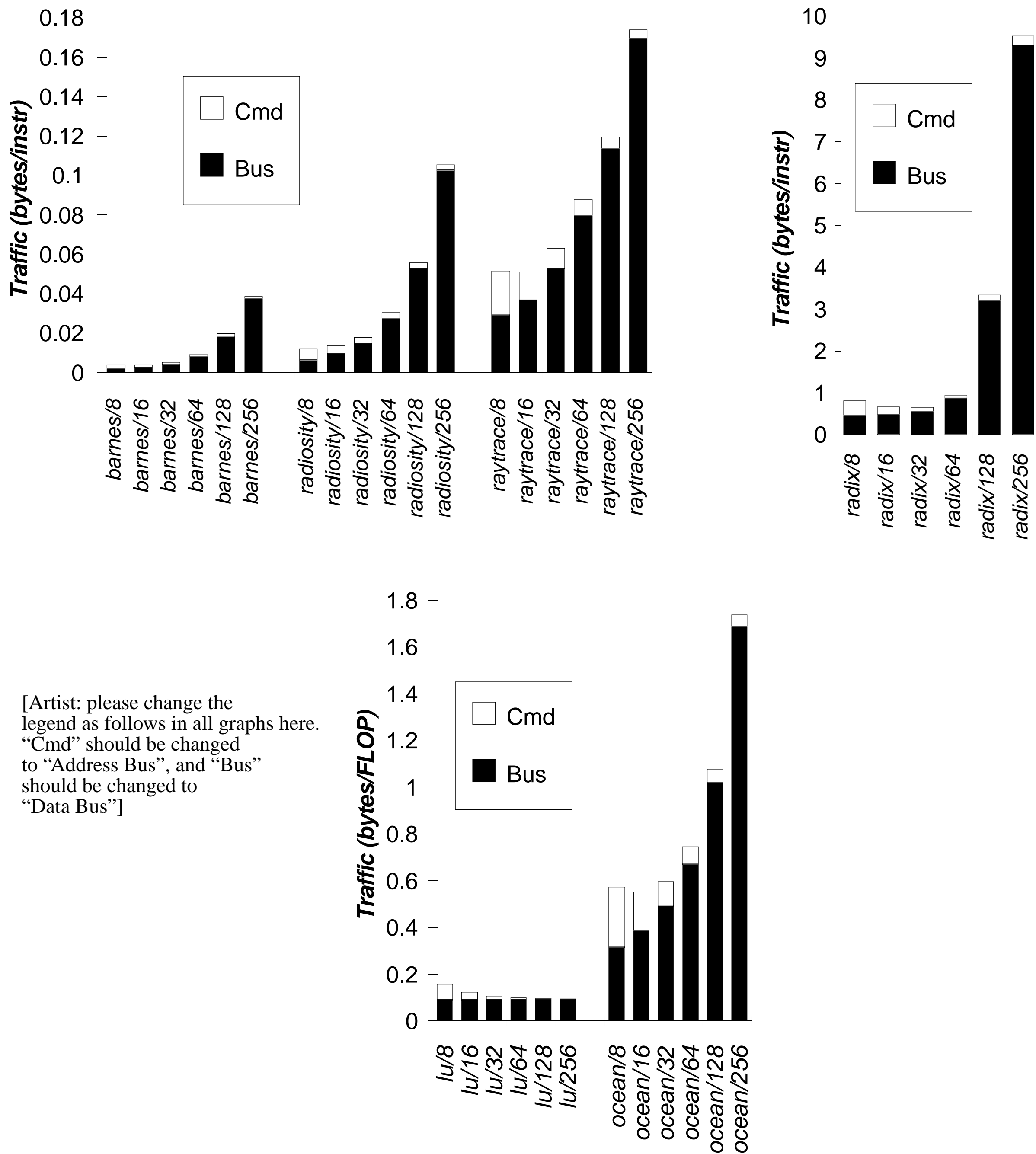


Figure 5-24 Traffic (in bytes/instr or bytes/flop) as a function of cache block size with 1 Mbyte per-processor caches.

Data traffic increases quite quickly with block size when communication misses dominate, except for applications like LU that have excellent spatial locality on all types of misses. Address and command bus traffic tends to decrease with block size since the miss rate and hence number of blocks transferred decreases.

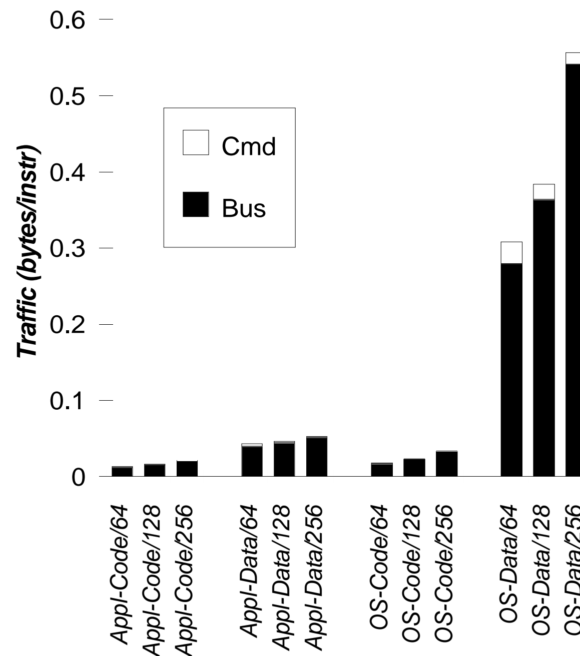


Figure 5-25 Traffic in bytes/instruction as a function of cache block size for Multiprog with 1Mbyte caches.

Traffic increasese quickly with block size for data references from the OS kernel.

Alleviating the Drawbacks of Large Cache Blocks

The trend towards larger cache block sizes is driven by the increasing gap between processor performance and memory access time. The larger block size amortizes the cost of the bus transaction and memory access time across a greater amount of data. The increasing density of processor and memory chips makes it possible to employ large first level and second level caches, so the prefetching obtained through a larger block size dominates the small increase in conflict misses. However, this trend potentially bodes poorly for multiprocessor designs, because false sharing becomes a larger problem. Fortunately there are hardware and software mechanisms that can be employed to counter the effects of large block size.

The software techniques to reduce false sharing are discussed later in the chapter. These essentially involve organizing data structures or work assignment so that data accessed by different processes is not at nearby addresses. One prime example is the use of higher dimensional arrays so blocks are contiguous. Some compiler techniques have also been developed to automate some of these techniques of laying out data to reduce false sharing [JeE91].

A natural hardware mechanism is the use of sub-blocks. Each cache block has a single address tag but distinct state bits for each of several sub-blocks. One subblock may be valid while others are invalid (or dirty). This technique is used in many machines to reduce the memory access time on a read miss, by resuming the processor when the accessed sub-block is present, or to reduce the amount of data that is copied back to memory on a replacement. Under false sharing, a write by one processor may invalidate the sub-block in another processors cache while leaving the other sub-blocks valid. Alternatively, small cache blocks can be used, but on a miss prefetch

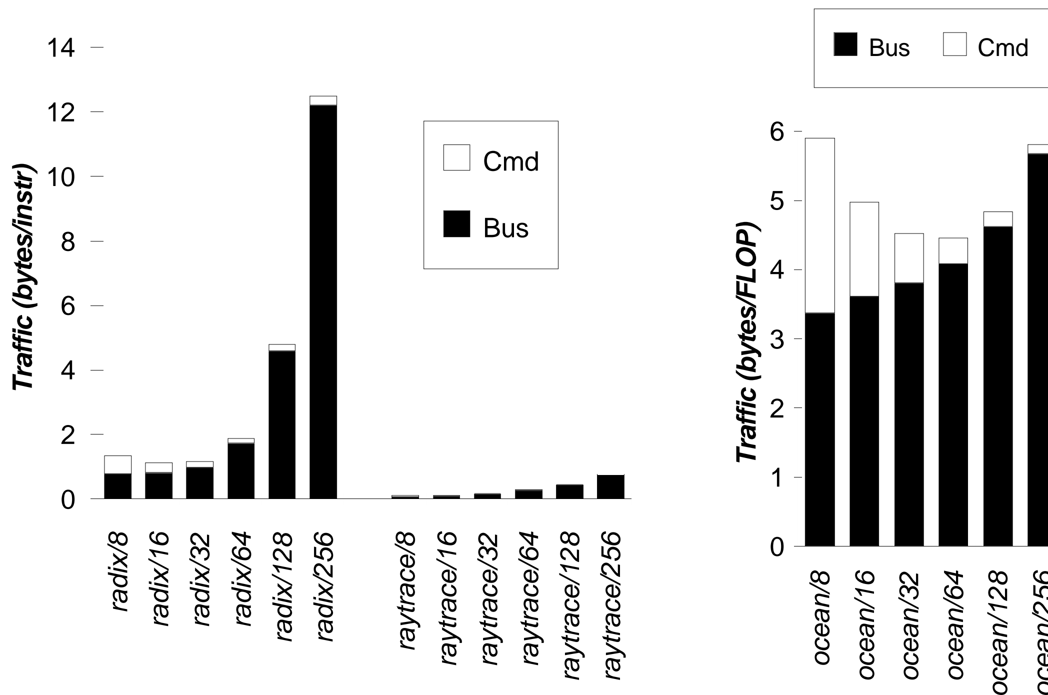


Figure 5-26 Traffic (bytes/instr or bytes/flop) as a function of cache block size with 64 Kbyte per-processor caches.

Traffic increases more slowly now for Ocean than with 1MB caches, since the capacity misses that now dominate exhibit excellent spatial locality (traversal of a process's assigned subgrid). However, traffic in Radix increases quickly once the threshold block size that causes false sharing is exceeded.

blocks beyond the accessed block. Proposals have also been made for caches with adjustable block size [DuL92].

A more subtle hardware technique is to delay generating invalidations until multiple writes have been performed. Delaying invalidations and performing them all at once reduces the occurrence of intervening read misses to false shared data. However, this sort of technique can change the memory consistency model in subtle ways, so further discussion is deferred until Chapter 6 where we consider weaker consistency models in the context of large scale machines. The other hardware technique that is important to consider is the use of update protocols, which push updates into other cached blocks, rather than invalidating those blocks. This option has been the subject of considerable study.

5.5.4 Update-based vs. Invalidation-based Protocols

A larger design issue is the question of whether writes should cause cached copies to be updated or invalidated. This has been the subject of considerable debate and various companies have taken different stands and, in fact, changed their position from one design to the next. The controversy on this topic arises because the relative performance of machines using an update-based versus an invalidation-based protocol depends strongly on the sharing patterns exhibited by the workload that is run and on the assumptions on cost of various underlying operations. Intuitively,

if the processors that were using the data before it was updated are likely to continue to use it in the future, updates should perform better than invalidates. However, if the processor holding the old data is never going to use it again, the updates are useless and just consume interconnect and controller resources. Invalidates would clean out the old copies and eliminate the apparent sharing. This latter 'pack rat' phenomenon is especially irritating under multiprogrammed use of an SMP, when sequential processes migrate from processor to processor so the apparent sharing is simply a shadow of the old process footprint. It is easy to construct cases where either scheme does substantially better than the other, as illustrated by the following example.

Example 5-12 Consider the following two program reference patterns:

Pattern-1: Repeat k times, processor-1 writes new value into variable V and processors 2 through N read the value of V. This represents a one-producer many-consumer scenario, that may arise for example when processors are accessing a highly-contended spin-lock.

Pattern-2: Repeat k times, processor-1 writes M-times to variable V, and then processor-2 reads the value of V. This represents a sharing pattern that may occur between pairs of processors, where the first accumulates a value into a variable, and then when the accumulation is complete it shares the value with the neighboring processor.

What is the relative cost of the two protocols in terms of the number of cache-misses and the bus traffic that will be generated by each scheme? To make the calculations concrete, assume that an invalidation/upgrade transaction takes 6 bytes (5 bytes for address, plus 1 byte command), an update takes 14 bytes (6 bytes for address and command, and 8 bytes of data), and that a regular cache-miss takes 70 bytes (6 bytes for address and command, plus 64 bytes of data corresponding to cache block size). Also assume that N=16, M=10, k=10, and that initially all caches are empty.

Answer With an update scheme, on pattern 1 the first iteration on all N processors will incur a regular cache miss. In subsequent k-1 iterations, no more misses will occur and only one update per iteration will be generated. Thus, overall, we will see: misses = N = 16; traffic = N x RdMiss + (k-1). Update = 16 x 70 + 10 x 14 = 1260 bytes.

With an invalidate scheme, in the first iteration all N processors will incur a regular cache miss. In subsequent k-1 iterations, processor-1 will generate an upgrade, but all others will experience a read miss. Thus, overall, we will see: misses = N + (k-1) x (N-1) = 16 + 9 x 15 = 151; traffic = misses x RdMiss + (k-1). Upgrade = 151 x 70 + 9 x 6 = 10,624 bytes.

With an update scheme on pattern 2, in the first iteration there will be two regular cache misses, one for processor-1 and the other for processor-2. In subsequent k-1 iterations, no more misses will be generated, but M updates will be generated each iteration. Thus, overall, we will see: misses = 2; traffic = 2. RdMiss + M x (k-1). Update = 2 x 70 + 10 x 9 x 14 = 1400 bytes.

With an invalidate scheme, in the first iteration there will be two regular cache miss. In subsequent k-1 iterations, there will be one upgrade plus one regular miss each

iteration. Thus, overall, we will see: misses = 2 + (k-1) = 2 + 9 = 11; traffic = misses x RdMiss + (k-1) x Upgrade = 11 x 70 + 9 x 6 = 824 bytes.

The example above shows that for pattern-1 the update scheme is substantially superior, while for pattern-2 the invalidate scheme is substantially superior. This anecdotal data suggests that it might be possible design schemes that capture the advantages of both update and invalidate protocols. The success of such schemes will depend on the sharing patterns for real parallel programs and workloads. Let us briefly explore the design options and then employ the workload driven evaluation of Chapter 4.

Combining Update and Invalidation-based Protocols

One way to take advantage of both update and invalidate protocols is to support both in hardware and then to decide dynamically at a page granularity, whether coherence for any given page is to be maintained using an update or an invalidate protocol. The decision about choice of protocol to use can be indicated by making a system call. The main advantage of such schemes is that they are relatively easy to support; they utilize the TLB to indicate to the rest of the coherence subsystem which of the two protocols to use. The main disadvantage of such schemes is the substantial burden they put on the programmer. The decision task is also made difficult because of the coarse-granularity at which control is made available.

An alternative is to make the decision about choice of protocol at a cache-block granularity, by observing the sharing behavior at run time. Ideally, for each write, one would like to be able to peer into the future references that will be made to that location by all processors and then decide (in a manner similar to what we used above to compute misses and traffic) whether to invalidate other copies or to do an update. Since this information is obviously not available, and since there are substantial perturbations to an ideal model due to cache replacements and multi-word cache blocks, a more practical scheme is needed. So called “competitive” schemes change the protocol for a block between invalidate and update based on observed patterns at runtime. The key attribute of any such scheme is that if a wrong decision is made once for a cache block, the losses due to that wrong decision should be kept bounded and small [KMR+86]. For example, if a block is currently using update mode, it should not remain in that mode if one processor is continuously writing to it but none of the other processors are reading values from that block.

As an example, one class of schemes that has been proposed to bound the losses of update protocols works as follows [GSD95]. Starting with the base Dragon update protocol as described in Section 5.4.3, associate a count-down counter with each block. Whenever a given cache block is accessed by the local processor, the counter value for that block is reset to a threshold value, k. Every time an update is received for a block, the counter is decremented. If the counter goes to zero, the block is locally invalidated. The consequence of the local invalidation is that the next time an update is generated on the bus, it may find that no other cache has a valid (shared) copy, and in that case (as per the Dragon protocol) that block will switch to the modified (exclusive) state and will stop generating updates. If some other processor accesses that block, the block will again switch to shared state and the protocol will again start generating updates. A related approach implemented in the Sun SPARCcenter 2000 is to selectively invalidate with some probability, which is a parameter set when configuring the machine [Cat94]. Other mixed approaches may also be used. For example, one system uses an invalidation-based protocol for first-level caches and by default an update-based protocol for the second-level caches. However, if the L2 cache receives a second update for the block while the block in the L1 cache is still invalid, then

the block is invalidated in the L2 cache as well. When the block is thus invalidated in all other L2 caches, writes to the block are no longer placed on the bus.

Workload-driven Evaluation

To assess the tradeoffs among invalidate, update, and the mixed protocols just described, Figure 5-27 shows the miss rates, by category for four applications using 1 MBytes, 4-way set associative caches with a 64 byte block size. The mixed protocol used is the first one discussed in the previous paragraph. We see that for applications with significant capacity miss rates, this category increases with an update protocol. This make sense, because the protocol keeps data in processor caches that would have been removed by an invalidation protocol. For applications with significant true-sharing or false-sharing miss rates, these categories decrease with an invalidation protocol. After a write update, the other caches holding the blocks can access them without a miss. Overall, for these categories the update protocol appears to be advantageous and the mixed protocol falls in between. The category that is not shown in this figure is the upgrade and update operations for these protocols. This data is presented in Figure 5-28. Note that the scale of the graphs have changed because update operations are roughly four times more prevalent than misses. It is useful to separate these operations from other misses, because the way they are handled in the machine is likely to be different. Updates are a single word write, rather than a full cache block transfer. Because the data is being pushed from where it is being produced, it may arrive at the consumer before it is even needed, so the latency of these operations may be less critical than misses.

[ARTIST: same change to legend as before (fig 5-22, except please get rid of UPGMR altogether since it is not included in this figure.]

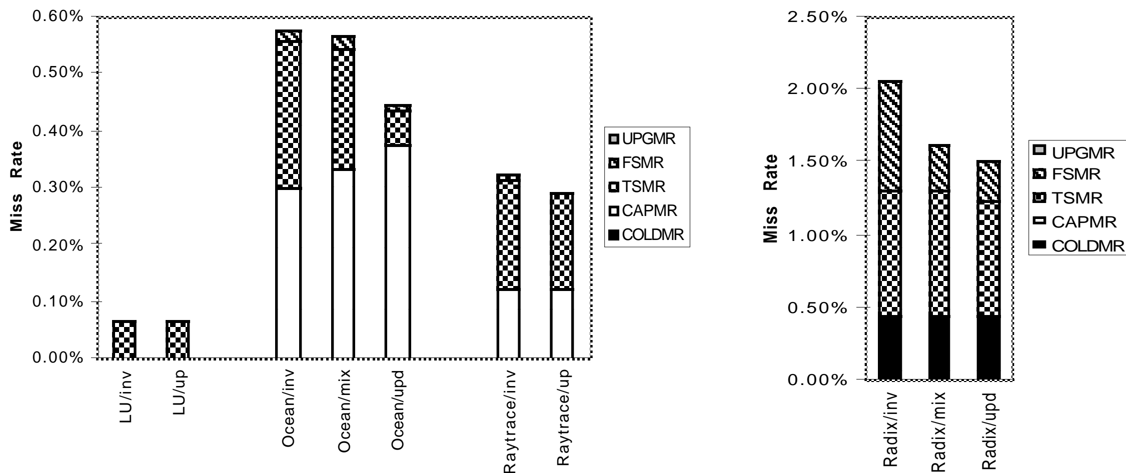


Figure 5-27 Miss rates and their decomposition for invalidate, update, and hybrid protocols.

1Mbyte caches, 64byte cache blocks, 4-way set-associativity, and threshold k=4 for hybrid protocol.

The traffic associated with updates is quite substantial. In large part this occurs where multiple writes are made to the same block. With the invalidate protocol, the first of these writes may

cause an invalidation, but the rest can simply accumulate in the block and be transferred in one bus transaction on a flush or a write-back. Sophisticated update schemes might attempt to delay the update to achieve a similar effect (by merging writes in the write buffer), or use other techniques to reduce traffic and improve performance [DaS95]. However, the increased bandwidth demand, the complexity of supporting updates, the trend toward larger cache blocks and the pack-rat phenomenon with sequential workloads underly the trend away from update-based protocols in the industry. We will see in Chapter 8 that update protocols also have some other prob-

[ARTIST: Please remove FSMR, TSMR, CAPMR and COLDMR from legend, and please change the only remaining one, (UPGMR) to Upgrade/Update].

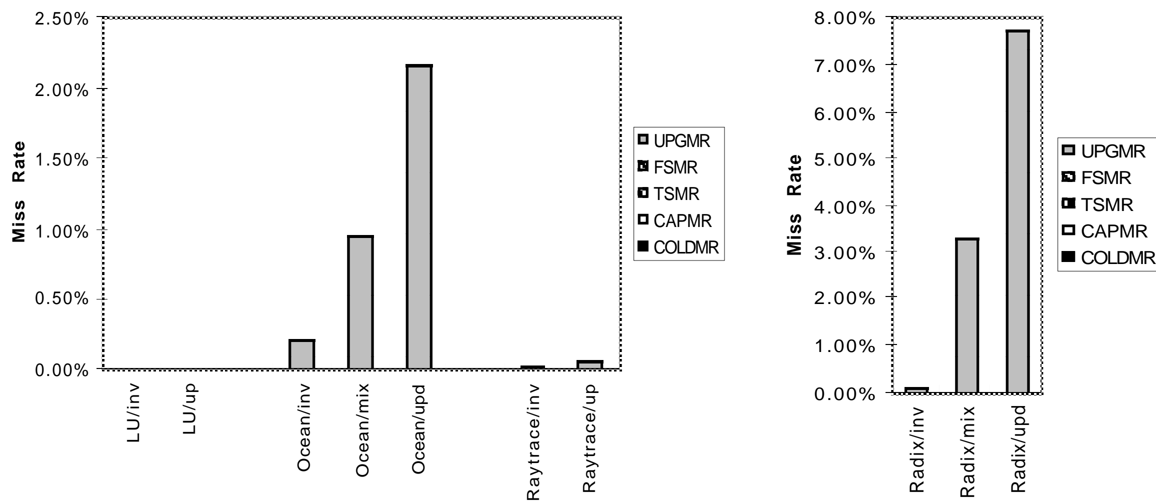


Figure 5-28 Upgrade and update rates for invalidate, update, and mixed protocols

1Mbyte, 64byte, 4-way, and k=4

lems for scalable cache-coherent architectures, making it less attractive for microprocessors to support them.

5.6 Synchronization

A critical interplay of hardware and software in multiprocessors arises in supporting synchronization operations, mutual exclusion, point-to-point events and global events, and there has been considerable debate over the years on what hardware primitives should be provided in multiprocessor machines to support these synchronization operations. The conclusions have changed from time to time, with changes in technology and machine design style. Hardware support has the advantage of speed, but moving functionality to software has the advantage of flexibility and adaptability to different situations. The classic work of Dijkstra [Dij65] and Knuth [Knu66] shows that it is possible to provide mutual exclusion with only atomic read and write operations (assuming a sequentially consistent memory). However, all practical synchronization operations rely on some *atomic read-modify-write* machine primitive, in which the value of a memory loca-

tion is read, modified and written back atomically without intervening accesses to the location by other processors. Simple or sophisticated synchronization algorithms can be built using these primitives.

The history of instruction set design offers a glimpse into the evolving hardware support for synchronization. One of the key instruction set enhancements in the IBM 370 was the inclusion of a sophisticated atomic instruction, the *compare-and-swap* instruction, to support synchronization in multiprogramming on uniprocessor or multiprocessor systems. The compare&swap compares the value in a memory location with the value in a specified register, and if they are equal swaps the value of the location with the value in a second register. The Intel x86 allows any instruction to be prefixed with a lock modifier to make it atomic, so with the source and destination being memory operands much of the instruction set can be used to implement various atomic operations involving even more than one memory location. Advocates of high level language architecture have proposed that the user level synchronization operations, such a locks and barriers, should be supported at the machine level, not just the atomic read-modify-write primitives; i.e. the synchronization "algorithm" itself should be implemented in hardware. The issue became very active with the reduced instruction set debates, since the operations that access memory were scaled back to simple loads and stores with only one memory operand. The SPARC approach was to provide atomic operations involving a register and a memory location, e.g., a simple swap (atomically swap the contents of the specified register and memory location) and a compare-and-swap, while MIPS left off atomic primitives in the early instruction sets, as did the IBM Power architecture used in the RS6000. The primitive that was eventually incorporated in MIPS was a novel combination of a special load and a conditional store, described below, which allow higher level synchronization operations to be constructed without placing the burden of full read-modify-write instructions on the machine implementor. In essence, the pair of instructions can be used to implement atomic exchange (or higher-level operations) instead of a single instruction. This approach was later incorporated in the PowerPC and DEC Alpha architectures, and is now quite popular. Synchronization brings to light an unusually rich family of tradeoffs across the layers of communication architecture.

The focus of this section is how synchronization operations can be implemented on a bus-based cache-coherent multiprocessor. In particular, it describes the implementation of mutual exclusion through lock-unlock pairs, point-to-point event synchronization through flags, and global event synchronization through barriers. Not only is there a spectrum of high level operations and of low level primitives that can be supported by hardware, but the synchronization requirements of applications vary substantially. Let us begin by considering the components of a synchronization event. This will make it clear why supporting the high level mutual exclusion and event operations directly in hardware is difficult and is likely to make the implementation too rigid. Given that the hardware supports only the basic atomic state transitions, we can examine role of the user software and system software in synchronization operations, and then examine the hardware and software design tradeoffs in greater detail.

5.6.1 Components of a Synchronization Event

There are three major components of a given type of synchronization event:

an *acquire method*: a method by which a process tries to acquire the right to the synchronization (to enter the critical section or proceed past the event synchronization)

a *waiting algorithm*: a method by which a process waits for a synchronization to become available when it is not. For example, if a process tries to acquire a lock but the lock is not free (or proceed past an event but the event has not yet occurred), it must somehow wait until the lock becomes free.

a *release method*: a method for a process to enable other processes to proceed past a synchronization event; for example, an implementation of the UNLOCK operation, a method for the last processes arriving at a barrier to release the waiting processes, or a method for notifying a process waiting at a point-to-point event that the event has occurred.

The choice of waiting algorithm is quite independent of the type of synchronization: What should a processor that has reached the acquire point do while it waits for the release to happen? There are two choices here: *busy-waiting* and *blocking*. Busy-waiting means that the process spins in a loop that repeatedly tests for a variable to change its value. A release of the synchronization event by another processor changes the value of the variable, allowing the process to proceed. Under blocking, the process does not spin but simply blocks (suspends) itself and releases the processor if it finds that it needs to wait. It will be awoken and made ready to run again when the release it was waiting for occurs. The tradeoffs between busy-waiting and blocking are clear. Blocking has higher overhead, since suspending and resuming a process involves the operating system, and suspending and resuming a thread involves the runtime system of a threads package, but it makes the processor available to other threads or processes with useful work to do. Busy-waiting avoids the cost of suspension, but consumes the processor and memory system bandwidth while waiting. Blocking is strictly more powerful than busy waiting, because if the process or thread that is being waited upon is not allowed to run, the busy-wait will never end.[1] Busy-waiting is likely to be better when the waiting period is short, whereas, blocking is likely to be a better choice if the waiting period is long and if there are other processes to run. Hybrid waiting methods can be used, in which the process busy-waits for a while in case the waiting period is short, and if the waiting period exceeds a certain threshold, blocks allowing other processes to run. The difficulty in implementing high level synchronization operations in hardware is not the acquire and release components, but the waiting algorithm. Thus, it makes sense to provide hardware support for the critical aspects of the acquire and release and allow the three components to be glued together in software. However, there remains a more subtle but very important hardware/software interaction in how the spinning operation in the busy-wait component is realized.

5.6.2 Role of User, System Software and Hardware

Who should be responsible for implementing the internals of high-level synchronization operations such as locks and barriers? Typically, a programmer wants to use locks, events, or even higher level operations and not have to worry about their internal implementation. The implementation is then left to the system, which must decide how much hardware support to provide and how much of the functionality to implement in software. Software synchronization algorithms using simple atomic exchange primitives have been developed which approach the speed of full hardware implementations, and the flexibility and hardware simplification they afford are

1. This problem of denying resources to the critical process or thread is one problem that is actually made simpler in with more processors. When the processes are timeshared on a single processor, strict busy-waiting without preemption is sure to be a problem. If each process or thread has its own processor, it is guaranteed not to be a problem. Realistic multiprogramming environments on a limited set of processors fall somewhere in between.

very attractive. As with other aspects of the system design, the utility of faster operations depends on the frequency of the use of those operations in the applications. So, once again, the best answer will be determined by a better understanding of application behavior.

Software implementations of synchronization constructs are usually included in system libraries. Many commercial systems thus provide subroutines or system calls that implement lock, unlock or barrier operations, and perhaps some types of other event synchronization. Good synchronization library design can be quite challenging. One potential complication is that the same type of synchronization (lock, barrier), and even the same synchronization variable, may be used at different times under very different runtime conditions. For example, a lock may be accessed with low-contention (a small number of processors, maybe only one, trying to acquire the lock at a time) or with high-contention (many processors trying to acquire the lock at the same time). The different scenarios impose different performance requirements. Under high-contention, most processes will spend time waiting and the key requirement of a lock algorithm is that it provide high lock-unlock bandwidth, whereas under low-contention the key goal is to provide low latency for lock acquisition. Since different algorithms may satisfy different requirements better, we must either find a good compromise algorithm or provide different algorithms for each type of synchronization among which a user can choose. If we are lucky, a flexible library can at runtime choose the best implementation for the situation at hand. Different synchronization algorithms may also rely on different basic hardware primitives, so some may be better suited to a particular machine than others. A second complication is that these multiprocessors are often used for multiprogrammed workloads where process scheduling and other resource interactions can change the synchronization behavior of the processes in a parallel program. A more sophisticated algorithm that addresses multiprogramming effects may provide better performance in practice than a simple algorithm that has lower latency and higher bandwidth in the dedicated case. All of these factors make synchronization a critical point of hardware/software interaction.

5.6.3 Mutual Exclusion

Mutual exclusion (lock/unlock) operations are implemented using a wide range of algorithms. The simple algorithms tend to be fast when there is little contention for the lock, but inefficient under high contention, whereas sophisticated algorithms that deal well with contention have a higher cost in the low contention case. After a brief discussion of hardware locks, the simplest algorithms for memory-based locks using atomic exchange instructions are described. Then, we discuss how the simplest algorithms can be implemented by using the special load locked and conditional store instruction pairs to synthesize atomic exchange, in place of atomic exchange instructions themselves, and what the performance tradeoffs are. Next, we discuss more sophisticated algorithms that can be built using either method of implementing atomic exchange.

Hardware Locks

Lock operations can be supported entirely in hardware, although this is not popular on modern bus-based machines. One option that was used one some older machines was to have a set of lock lines on the bus, each used for one lock at a time. The processor holding the lock asserts the line, and processors waiting for the lock wait for it to be released. A priority circuit determines whichgets the lock next when there are multiple requestors. However, this approach is quite inflexible since only a limited number of locks can be in use at a time and waiting algorithm is fixed (typically busy-wait with abort after timeout). Usually, these hardware locks were used only by the operating system for specific purposes, one of which was to implement a larger set of software

locks in memory, as discussed below. The Cray xMP provided an interesting variant of this approach. A set of registers were shared among the processors, including a fixed collection of lock registers[**XMP**]. Although the architecture made it possible to assign lock registers to user processes, with only a small set of such registers it was awkward to do so in a general purpose setting and, in practice, the lock registers were used primarily to implement higher level locks in memory.

Simple Lock Algorithms on Memory Locations

Consider a lock operation used to provide atomicity for a critical section of code. For the acquire method, a process trying to obtain a lock must check that the lock is free and if it is then claim ownership of the lock. The state of the lock can be stored in a binary variable, with 0 representing free and 1 representing busy. A simple way of thinking about the lock operation is that a process trying to obtain the lock should check if the variable is 0 and if so set it to 1 thus marking the lock busy; if the variable is 1 (lock is busy) then it should wait for the variable to turn to 0 using the waiting algorithm. An unlock operation should simply set the variable to 0 (the release method). Assembly-level instructions for this attempt at a lock and unlock are shown below (in our pseudo-assembly notation, the first operand always specifies the destination if there is one).

```
lock:    ld   register, location    /* copy location to register */
         cmp  location, #0          /* compare with 0 */
         bnz  lock                  /* if not 0, try again */
         st   location, #1          /* store 1 into location to mark it locked */
         ret                        /* return control to caller of lock */
```

and

```
unlock:  st location, #0            /* write 0 to location */
         ret                        /* return control to caller */
```

The problem with this lock, which is supposed to provide atomicity, is that it needs atomicity in its own implementation. To illustrate this, suppose that the lock variable was initially set to 0, and two processes P0 and P1 execute the above assembly code implementations of the lock operation. Process P0 reads the value of the lock variable as 0 and thinks it is free, so it enters the critical section. Its next step is to set the variable to 1 marking the lock as busy, but before it can do this process P1 reads the variable as 0, thinks the lock is free and enters the critical section too. We now have two processes simultaneously in the same critical section, which is exactly what the locks were meant to avoid. Putting the store of 1 into the location just after the load of the location would not help. The two-instruction sequence—reading (testing) the lock variable to check its state, and writing (setting) it to busy if it is free—is not atomic, and there is nothing to prevent these operations from different processes from being interleaved in time. What we need is a way to atomically test the value of a variable *and* set it to another value if the test succeeded (i.e. to atomically read and then conditionally modify a memory location), and to return whether the atomic sequence was executed successfully or not. One way to provide this atomicity for user processes is to place the lock routine in the operating system and access it through a system call, but this is expensive and leaves the question of how the locks are supported for the system itself. Another option is to utilize a hardware lock around the instruction sequence for the lock routine, but this also tends to be very slow compared to modern processors.

Hardware Atomic Exchange Primitives

An efficient, general purpose solution to the lock problem is to have an *atomic read-modify-write* instruction in the processor's instruction set. A typical approach is to have and atomic exchange instruction, in which a value at a location specified by the instruction is read into a register, and another value—that is either a function of the value read or not—is stored into the location, all in an atomic operation. There are many variants of this operation with varying degrees of flexibility in the nature of the value that can be stored. A simple example that works for mutual exclusion is an atomic *test&set* instruction. In this case, the value in the memory location is read into a specified register, and the constant 1 is stored into the location atomically if the value read is 0 (1 and 0 are typically used, though any other constants might be used in their place). Given such an instruction, with the mnemonic t&s, we can write a lock and unlock in pseudo-assembly language as follows:

```
lock:    t&s  register, location    /* copy location to reg, and if 0 set location to 1 */
         bnz  register, lock        /* compare old value returned with 0 */
                                    /* if not 0, i.e. lock already busy, try again */
         ret                        /* return control to caller of lock */
```

and

```
unlock:  st location, #0            /* write 0 to location */
         ret                        /* return control to caller */
```

The lock implementation keeps trying to acquire the lock using test&set instructions, until the test&set returns zero indicating that the lock was free when tested (in which case the test&set has set the lock variable to 1, thus acquiring it). The unlock construct simply sets the location associated with the lock to 0, indicating that the lock is now free and enabling a subsequent lock operation by any process to succeed. A simple mutual exclusion construct has been implemented in software, relying on the fact that the architecture supports an atomic test&set instruction.

More sophisticated variants of such atomic instructions exist, and as we will see are used by different software synchronization algorithms. One example is a *swap* instruction. Like a test&set, this reads the value from the specified memory location into the specified register, but instead of writing a fixed constant into the memory location it writes whatever value was in the register to begin with. That is, it atomically exchanges or swaps the values in the memory location and the register. Clearly, we can implement a lock as before by replacing the test&set with a swap instruction as long as we ensure that the value in the register is 1 before the swap instruction is executed.

Another example is the family of so-called *fetch&op* instructions. A fetch&op instruction also specifies a location and a register. It atomically reads the value of the location into the register, and writes into the location the value obtained by applying to the current value of the location the operation specified by the fetch-and-op instruction. The simplest forms of fetch&op to implement are the fetch&increment and fetch&decrement instructions, which atomically read the current value of the location into the register and increment (or decrement) the value in the location by one. A fetch&add would take another operand which is a register or value to add into the previous value of the location. More complex primitive operations are possible. For example, the *compare&swap* operation takes two register operands plus a memory location; it compares the

value in the location with the contents of the first register operand, and if the two are equal it swaps the contents of the memory location with the contents of the second register.

Performance Issues

Figure 5-29 shows the performance of a simple test&set lock on the SGI Challenge.[1] Performance is measured for the following pseudocode executed repeatedly in a loop:

```
lock(L); critical-section(c); unlock(L);
```

where c is a delay parameter that determines the size of the critical section (which is only a delay, with no real work done). The benchmark is configured so that the same total number of locks are executed as the number of processors increases, reflecting a situation where there is a fixed number of tasks, independent of the number of processors. Performance is measured as the time per lock transfer, i.e., the cumulative time taken by all processes executing the benchmark divided by the number of times the lock is obtained. The uniprocessor time spent in the critical section itself (i.e. c times the number of successful locks executed) is subtracted from the total execution time, so that only the time for the lock transfers themselves (or any contention caused by the lock operations) is obtained. All measurements are in microseconds.

The upper curve in the figure shows the time per lock transfer with increasing number of processors when using the test&set lock with a very small critical section (ignore the curves with "backoff" in their labels for now). Ideally, we would like the time per lock acquisition to be independent of the number of processors competing for the lock, with only one uncontended bus transaction per lock transfer, as shown in the curve labelled ideal. However, the figure shows that performance clearly degrades with increasing number of processors. The problem with the test&set lock is that every attempt to check whether the lock is free to be acquired, whether successful or not, generates a write operation to the cache block that holds the lock variable (writing the value to 1); since this block is currently in the cache of some other processor (which wrote it last when doing its test&set), a bus transaction is generated by each write to invalidate the previous owner of the block. Thus, all processors put transactions on the bus repeatedly. The resulting contention slows down the lock considerably as the number of processors, and hence the frequency of test&sets and bus transactions, increases. The high degree of contention on the bus and the resulting timing dependence of obtaining locks causes the benchmark timing to vary sharply across numbers of processors used and even across executions. The results shown are for a particular, representative set of executions with different numbers of processors.

The major reason for the high traffic of the simple test&set lock above is the waiting method. A processor waits by repeatedly issuing test&set operations, and every one of these test&set operations includes a write in addition to a read. Thus, processors are consuming precious bus bandwidth even while waiting, and this bus contention even impedes the progress of the one process that is holding the lock (as it performs the work in its critical section and attempts to release the

1. In fact, the processor on the SGI Challenge, which is the machine for which synchronization performance is presented in this chapter, does not provide a test&set instruction. Rather, it uses alternative primitives that will be described later in this section. For these experiments, a mechanism whose behavior closely resembles that of test&set is synthesized from the available primitives. Results for real test&set based locks on older machines like the Sequent Symmetry can be found in the literature [GrT90, MCS87].

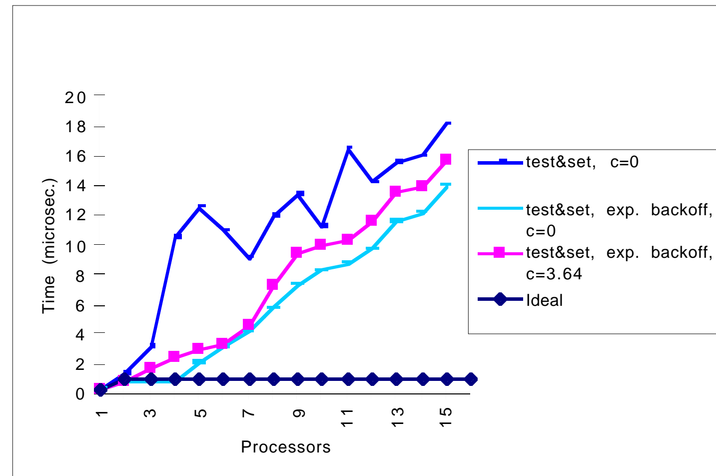


Figure 5-29 Performance of test&set locks with increasing number of competing processors on the SGI Challenge.

The Y axis is the time per lock-unlock pair, excluding the critical section of size c microseconds. The "exp. backoff" refers to exponential backoff, which will be discussed shortly. The irregular nature of the top curve is due to the timing-dependence of the contention effects caused. Note that since the processor on the SGI Challenge does not provide anatomic read-modify-write primitive but rather more sophisticated primitives discussed later in this section, the behavior of a test&set is simulated using those primitives for this experiment. Performance of locks that use test&set and test&set with backoff on older systems can be found in the literature [GrT90, MCS91].

lock).There are two simple things we can do to alleviate this traffic. First, we can reduce the frequency with which processes issue test&set instructions while waiting; second, we can have processes busy-wait only with read operations so they do not generate invalidations and misses until the lock is actually released. Let us examine these two possibilities, called the test&set lock with backoff and the test-and-test&set lock.

Test&set Lock with Backoff. The basic idea with backoff is to insert a delay after an unsuccessful attempt to acquire the lock. The delay between test&set attempts should not be too long, otherwise processors might remain idle even when the lock becomes free. But it should be long enough that traffic is substantially reduced. A natural question is whether the delay amount should be fixed or should vary. Experimental results have shown that good performance is obtained by having the delay vary "exponentially"; i.e. the delay after the first attempt is a small constant k, and then increases geometrically so that after the ith iteration it is $k*c^i$ where c is another constant. Such a lock is called a test&set lock with exponential backoff. Figure 5-29 also shows the performance for the test&set lock with backoff for two different sizes of the critical section and the starting value for backoff that appears to perform best. Performance improves, but still does not scale very well. Performance results using a real test&set instruction on older machines can be found in the literature [GrT90, MCS91]. See also Exercise 5.6, which discusses why the performance with a null critical section is worse than that with a non-zero critical section when backoff is used.

Test-and-test&set Lock. A more subtle change to the algorithm is have it use instructions that do not generate as much bus traffic while busy-waiting. Processes busy-wait by repeatedly *reading* with a standard load, not a test&set, the value of the lock variable until it turns from 1 (locked) to 0 (unlocked). On a cache-coherent machine, the reads can be performed in-cache by all processors, since each obtains a cached copy of the lock variable the first time it reads it. When the lock is released, the cached copies of all waiting processes are invalidated, and the next

read of the variable by each process will generate a read miss. The waiting processes will then find that the lock has been made available, and will only then generate a test&set instruction to actually try to acquire the lock.

Before examining other lock algorithms and primitives, it is useful to articulate some performance goals for locks and to place the above locks along them. The goals include:

Low latency If a lock is free and no other processors are trying to acquire it at the same time, a processor should be able to acquire it with low latency.

Low traffic Suppose many or all processors try to acquire a lock at the same time. They should be able to acquire the lock one after the other with as little generation of traffic or bus transactions as possible. High traffic can slow down lock acquisitions due to contention, and can also slow down unrelated transactions that compete for the bus.

Scalability Related to the previous point, neither latency nor traffic should scale quickly with the number of processors used. Keep in mind that since the number of processors in a bus-based SMP is not likely to be large, it is not asymptotic scalability that is important.

Low storage cost The information needed for a lock should be small and should not scale quickly with the number of processors.

Fairness Ideally, processors should acquire a lock in the same order as their requests are issued. At least, starvation or substantial unfairness should be avoided.

Consider the simple atomic exchange or test&set lock. It is very low-latency if the same processor acquires the lock repeatedly without any competition, since the number of instructions executed is very small and the lock variable will stay in that processor's cache. However, as discussed earlier it can generate a lot of bus traffic and contention if many processors compete for the lock. The scalability of the lock is poor with the number of competing processors. The storage cost is low (a single variable suffices) and does not scale with the number of processors. The lock makes no attempt to be fair, and an unlucky processor can be starved out. The test&set lock with backoff has the same uncontended latency as the simple test&set lock, generates less traffic and is more scalable, takes no more storage, and is no more fair. The test-and-test&set lock has slightly higher uncontended overhead than the simple test&set lock (it does a read in addition to a test&set even when there is no competition), but generates much less bus traffic and is more scalable. It too requires negligible storage and is not fair. Exercise 5.6 asks you to count the number of bus transactions and the time required for each type of lock.

Since a test&set operation and hence a bus transaction is only issued when a processor is notified that the lock is ready, and thereafter if it fails it spins on a cache block, there is no need for backoff in the test-and-test&set lock. However, the lock does have the problem that all processes rush out and perform both their read misses and their test&set instructions at about the same time when the lock is released. Each of these test&set instructions generates invalidations and subsequent misses, resulting in $O(p^2)$ bus traffic for p processors to acquire the lock once each. A random delay before issuing the test&set could help to stagger at least the test&set instructions, but it would increase the latency to acquire the lock in the uncontended case.

Improved Hardware Primitives: Load-locked, Store-conditional

Several microprocessors provide a pair of instructions called load locked and store conditional to implement atomic operations, instead of a atomic read-modify-write instructions like test&set. Let us see how these primitives can be used to implement simple lock algorithms and improve

performance over a test-and-test&set lock. Then, we will examine sophisticated lock algorithms that tend to use more sophisticated atomic exchange operations, which can be implemented either as atomic instructions or with load locked and store conditional.

In addition to spinning with reads rather than read-modify-writes, which test-and-test&set accomplishes, it would be nice to implement read-modify-write operations in such a way that failed attempts to complete the read-modify-write do not generate invalidations. It would also be nice to have a single primitive that allows us to implement a range of atomic read-modify-write operations—such as test&set, fetch&op, compare&swap—rather than implement each with a separate instruction. Using a pair of special instructions rather than a single instruction to implement atomic access to a variable, let's call it a synchronization variable, is one way to achieve both goals. The first instruction is commonly called *load-locked* or *load-linked* (LL). It loads the synchronization variable into a register. It may be followed by arbitrary instructions that manipulate the value in the register; i.e. the modify part of a read-modify-write. The last instruction of the sequence is the second special instruction, called a *store-conditional* (SC). It writes the register back to the memory location (the synchronization variable) *if and only if* no other processor has written *to that location* since this processor completed its LL. Thus, if the SC succeeds, it means that the LL-SC pair has read, perhaps modified in between, and written back the variable atomically. If the SC detects that an intervening write has occurred to the variable, it fails and does not write the value back (or generate any invalidations). This means that the atomic operation on the variable has failed and must be retried starting from the LL. Success or failure of the SC is indicated by the condition codes or a return value. How the LL and SC are actually implemented will be discussed later; for now we are concerned with their semantics and performance.

Using LL-SC to implement atomic operations, lock and unlock subroutines can be written as follows, where reg1 is the register into which the current value of the memory location is loaded, and reg2 holds the value to be stored in the memory location by this atomic exchange (reg2 could simply be 1 for a lock attempt, as in a test&set). Many processors may perform the LL at the same time, but only the first one that manages to put its store conditional on the bus will succeed in its SC. This processor will have succeeded in acquiring the lock, while the others will have failed and will have to retry the LL-SC.

```
lock:    ll   reg1, location       /* load-linked the location to reg1 */
         bnz  reg1, lock           /* if location was locked (nonzero), try again*/
         sc   location, reg2       /* store reg2 conditionally into location*/
         beqz  lock                /* if SC failed, start again*/
         ret                       /* return control to caller of lock */
```

and

```
unlock:  st location, #0           /* write 0 to location */
         ret                       /* return control to caller */
```

If the location is 1 (nonzero) when a process does its load linked, it will load 1 into reg1 and will retry the lock starting from the LL without even attempt the store conditional.

It is worth noting that the LL itself is not a lock, and the SC itself is not an unlock. For one thing, the completion of the LL itself does not guarantee obtaining exclusive access; in fact LL and SC are used together to implement a lock operation as shown above. For another, even a successful LL-SC pair does not guarantee that the instructions between them (if any) are executed atomi-

cally with respect to those instructions on other processors, so in fact those instructions do not constitute a critical section. All that a successful LL-SC guarantees is that no conflicting writes *to the synchronization variable* accessed by the LL and SC themselves intervened between the LL and SC. In fact, since the instructions between the LL and SC are executed but should not be visible if the SC fails, it is important that they do not modify any important state. Typically, they only manipulate the register holding the synchronization variable—for example to perform the op part of a fetch&op—and do not modify any other program variables (modification of the register is okay since the register will be re-loaded anyway by the LL in the next attempt). Microprocessor vendors that support LL-SC explicitly encourage software writers to follow this guideline, and in fact often provide guidelines on what instructions are possible to insert with guarantee of correctness given their implementations of LL-SC. The number of instructions between the LL and SC should also be kept small to reduce the probability of the SC failing. On the other hand, the LL and SC can be used directly to implement certain useful operations on shared data structures. For example, if the desired function is a shared counter, it makes much more sense to implement it as the natural sequence (LL, register op, SC, test) than to build a lock and unlock around the counter update.

Unlike the simple test&set, the spin-lock built with LL-SC does not generate invalidations if either the load-linked indicates that the lock is currently held or if the SC fails. However, when the lock is released, the processors spinning in a tight loop of load-locked operations will miss on the location and rush out to the bus with read transactions. After this, only a single invalidation will be generated for a given lock acquisition, by the processor whose SC succeeds, but this will again invalidate all caches. Traffic is reduced greatly from even the test-and-test&set case, down from $O(p^2)$ to $O(p)$ per lock acquisition, but still scales quickly with the number of processors. Since spinning on a locked location is done through reads (load-locked operations) there is no analog of a test-and-test&set to further improve its performance. However, backoff can be used between the LL and SC to further reduce bursty traffic.

The simple LL-SC lock is also low in latency and storage, but it is not a fair lock and it does not reduce traffic to a minimum. More advanced lock algorithms can be used that both provide fairness and reduce traffic. They can be built using both atomic read-modify-write instructions or LL-SC, though of course the traffic advantages are different in the two cases. Let us consider two of these algorithms.

Advanced Lock Algorithms

Especially when using a test&set to implement locks, it is desirable to have only one process attempt to obtain the lock when it is released (rather than have them all rush out to do a test&set and issue invalidations as in all the above cases). It is even more desirable to have only one process even incur a read miss when a lock is released. The ticket lock accomplishes the first purpose, while the array-based lock accomplishes both goals but at a little cost in space. Both locks are fair, and grant the lock to processors in FIFO order.

Ticket Lock. The ticket lock operates just like the ticket system in the sandwich line at a grocery store, or in the teller line at a bank. Every process wanting to acquire the lock takes a number, and busy-waits on a global *now-serving* number—like the number on the LED display that we watch intently in the sandwich line—until this *now-serving* number equals the number it obtained. To release the lock, a process simply increments the *now-serving* number. The atomic primitive needed is a fetch&increment, which a process uses to obtain its ticket number from a shared counter. It may be implemented as an atomic instruction or using LL-SC. No test&set is needed

to actually obtain the lock upon a release, since only the unique process that has its ticket number equal *now-serving* attempts to enter the critical section when it sees the release. Thus, the atomic primitive is used when a process first reaches the lock operation, not in response to a release. The acquire method is the fetch&increment, the waiting algorithm is busy-waiting for *now-serving* to equal the ticket number, and the release method is to increment *now-serving*. This lock has uncontended overhead about equal to the test-and-test&set lock, but generates much less traffic. Although every process does a fetch and increment when it first arrives at the lock (presumably not at the same time), the simultaneous test&set attempts upon a release of the lock are eliminated, which tend to be a lot more heavily contended. The ticket lock also requires constant and small storage, and is fair since processes obtain the lock in the order of their fetch&increment operations. However, like the simple LL-SC lock it still has a traffic problem. The reason is that all processes spin on the same variable (now-serving). When that variable is written at a release, all processors' cached copies are invalidated and they all incur a read miss. (The simple LL-SC lock was somewhat worse in this respect, since in that case another invalidation and set of read misses occurred when a processor succeeded in its SC.) One way to reduce this bursty traffic is to introduce a form of backoff. We do not want to use exponential backoff because we do not want all processors to be backing off when the lock is released so none tries to acquire it for a while. A promising technique is to have each processor backoff from trying to read the now-serving counter by an amount proportional to when it expects its turn to actually come; i.e. an amount proportional to the difference in its ticket number and the *now-serving* counter it last read. Alternatively, the array-based lock eliminates this extra read traffic upon a release completely, by having every process spin on a distinct location.

Array-based Lock. The idea here is to use a fetch&increment to obtain not a value but a unique location to busy-wait on. If there are p processes that might possibly compete for a lock, then the lock contains an array of p locations that processes can spin on, ideally each on a separate memory block to avoid false-sharing. The acquire method then uses a fetch&increment operation to obtain the next available location in this array (with wraparound) to spin on, the waiting method spins on this location, and the release method writes a value denoting "unlocked" to the next location in the array after the one that the releasing processor was itself spinning on. Only the processor that was spinning on that location has its cache block invalidated, and its consequent read miss tells it that it has obtained the lock. As in the ticket lock, no test&set is needed after the miss since only one process is notified when the lock is released. This lock is clearly also FIFO and hence fair. It's uncontended latency is likely to be similar to that of the test-and-test&set lock (a fetch&increment followed by a read of the assigned array location), and it is more scalable than the ticket lock since only one process incurs the read miss. It's only drawback for a bus-based machine is that it uses $O(p)$ space rather than $O(1)$, but with both p and the proportionality constant being small this is usually not a very significant drawback. It has a potential drawback for distributed memory machines, but we shall discuss this and lock algorithms that overcome this drawback in Chapter 7.

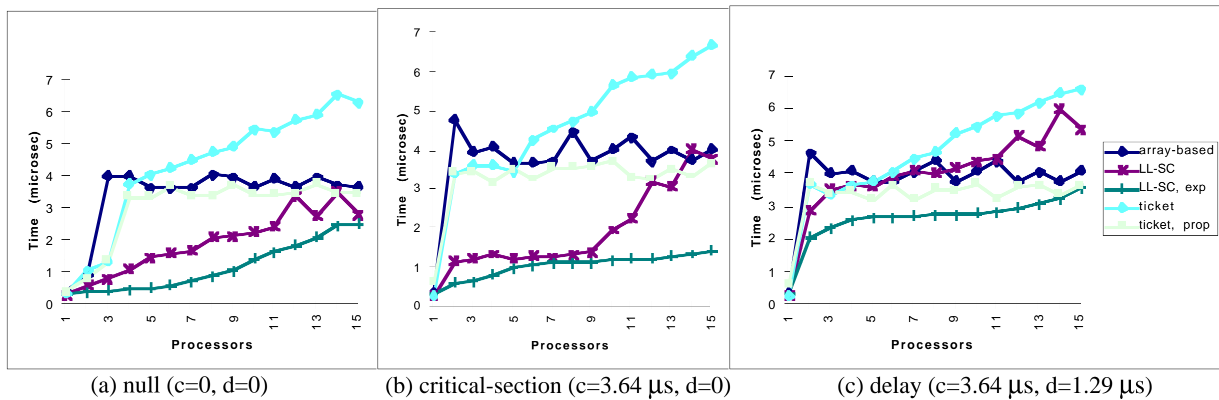
Performance

Let us briefly examine the performance of the different locks on the SGI Challenge, as shown in Figure 5-30. All locks are implemented using LL-SC, since the Challenge provides only these and not atomic instructions. The test&set locks are implemented by simulating a test&set using LL-SC, just as they were in Figure 5-29, and are shown as leaping off the graph for reference[1]. In particular, every time an SC fails a write is performed to another variable on the same cache block, causing invalidations as a test&set would. Results are shown for a somewhat more param-

eterized version of the earlier code for test&set locks, in which a process is allowed to insert a delay between its release of the lock and its next attempt to acquire it. That is, the code is a loop over the following body:

```
lock(L); critical_section(c); unlock(L); delay(d);
```

Let us consider three cases—(i) c=0, d=0, (ii) c=3.64 µs, d=0, and (iii) c=3.64 µs, d=1.29 µs—called *null*, *critical-section*, and *delay*, respectively. The delays c and d are inserted in the code as round numbers of processor cycles, which translates to these microsecond numbers. Recall that in all cases c and d (multiplied by the number of lock acquisitions by each processor) are subtracted out of the total time, which is supposed to measure the total time taken for a certain number of lock acquisitions and releases only (see also Exercise 5.6).



(a) null (c=0, d=0) (b) critical-section (c=3.64 µs, d=0) (c) delay (c=3.64 µs, d=1.29 µs)

Figure 5-30 Performance of locks on the SGI Challenge, for three different scenarios.

Consider the null critical section case. The first observation, comparing with Figure 5-29, is that all the subsequent locks we have discussed are indeed better than the test&set locks as expected. The second observation is that the simple LL-SC locks actually perform better than the more sophisticated ticket lock and array-based lock. For these locks, that don't encounter so much contention as the test&set lock, with reasonably high-bandwidth busses the performance of a lock is largely determined by the number of bus transactions between a release and a successful acquire. The reason that the LL-SC locks perform so well, particularly at lower processor counts, is that they are not fair, and the unfairness is exploited by architectural interactions! In particular, when a processor that does a write to release a lock follows it immediately with the read (LL) for its next acquire, it's read and SC are likely to succeed in its cache before another processor can read the block across the bus. (The bias on the Challenge is actually more severe, since the releasing processor can satisfy its next read from its write buffer even before the corresponding read-exclusive gets out on the bus.) Lock transfer is very quick, and performance is good. As the number of processors increases, the likelihood of self-transfers decreases and bus traffic due to invalidations and read misses increases, so the time per lock transfer increases. Exponential backoff helps reduce the burstiness of traffic and hence slows the rate of scaling.

1. This method of simulating test&set with LL-SC may lead to somewhat worse performance than a true test&set primitive, but it conveys the trend.

At the other extreme (c=3.64, d=1.29), we see the LL-SC lock doing not quite so well, even at low processor counts. This is because a processor waits after its release before trying to acquire the lock again, making it much more likely that some other waiting processor will acquire the lock before it. Self-transfers are unlikely, so lock transfers are slower even at two processors. It is interesting that performance is particularly worse for the backoff case at small processor counts when the delay d between unlock and lock is non-zero. This is because with only a few processors, it is quite likely that while a processor that just released the lock is waiting for d to expire before doing its next acquire, the other processors are in a backoff period and not even trying to acquire the lock. Backoff must be used carefully for it to be successful.

Consider the other locks. These are fair, so every lock transfer is to a different processor and involves bus transactions in the critical path of the transfer. Hence they all start off with a jump to about 3 bus transactions in the critical path per lock transfer even when two processors are used. Actual differences in time are due to what exactly the bus transactions are and how much of their latency can be hidden from the processor. The ticket lock without backoff scales relatively poorly: With all processors trying to read the now-serving counter, the expected number of bus transactions between the release and the read by the correct processor is $p/2$, leading to the observed linear degradation in lock transfer critical path. With successful proportional backoff, it is likely that the correct processor will be the one to issue the read first after a release, so the time per transfer does not scale with p. The array-based lock also has a similar property, since only the correct processor issues a read, so its performance also does not degrade with more processors.

The results illustrate the importance of architectural interactions in determining the performance of locks, and that simple LL-SC locks perform quite well on busses that have high enough bandwidth (and realistic numbers of processors for busses). Performance for the unfair LL-SC lock scales to become as bad as or a little worse than for the more sophisticated locks beyond 16 processors, due to the higher traffic, but not by much because bus bandwidth is quite high. When exponential backoff is used to reduce traffic, the simple LL-SC lock delivers the best average lock transfer time in all cases. The results also illustrate the difficulty and the importance of sound experimental methodology in evaluating synchronization algorithms. Null critical sections display some interesting effects, but meaningful comparison depend on what the synchronization patterns look like in practice in real applications. An experiment to use LL-SC but guarantee round-robin acquisition among processors (fairness) by using an additional variable showed performance very similar to that of the ticket lock, confirming that unfairness and self-transfers are indeed the reason for the better performance at low processor counts.

Lock-free, Non-blocking, and Wait-free Synchronization

An additional set of performance concerns involving synchronization arise when we consider that the machine running our parallel program is used in a multiprogramming environment. Other processes run for periods of time or, even if we have the machine to ourselves, background daemons run periodically, processes take page faults, I/O interrupts occur, and the process scheduler makes scheduling decisions with limited information on the application requirements. These events can cause the rate at which processes make progress to vary considerably. One important question is how the program as a whole slows down when one process is slowed. With traditional locks the problem can be serious because if a process holding a lock stops or slows while in its critical section, all other processes may have to wait. This problem has received a good deal of attention in work on operating system schedulers and in some cases attempts are made to avoid preempting a process that is holding a lock. There is another line of research that takes the view

that lock-based operations are not very robust and should be avoided. If a process dies while holding a lock, other processes hang. It can be observed that many of the lock/unlock operations are used to support operations on a data structure or object that is shared by several processes, for example to update a shared counter or manipulate a shared queue. These higher level operation can be implemented directly using atomic primitives without actually using locks.

A shared data structure is *lock-free* if its operations do not require mutual exclusion over multiple instructions. If the operations on the data structure guarantee that some process will complete its operation in finite amount of time, even if other processes halt, the data structure is *non-blocking*. If the data structure operations can guantee that every (non-faulty) process will complete its operation in a finite amount of time, then the data structure is *wait-free*.[Her93]. There is a body of literature that investigates the theory and practice of such data structures, including requirements on the basic atomic primitives[Her88], general purpose techniques for translating sequential operations to non-blocking concurrent operations[Her93], specific useful lock-free data structures [Val95,MiSc96], operating system implementations [MaPu91,GrCh96] and proposals for architectural support[HeMo93]. The basic approach is to implement updates to a shared object by reading a portion of the object to make a copy, updating the copy, and then performing an operation to commit the change only if no conflicting updates have been made. As a simple example, consider a shared counter. The counter is read into a register, a value is added to the register copy and the result put in a second register, then a compare-and-swap is performed to update the shared counter only if its value is the same as the copy. For more sophisticated data structures a linked structure is used and typically the new element is linked into the shared list if the insert is still valid. These techniques serve to limit the window in which the shared data structure is in an inconsistent state, so they improve the robustness, although it can be difficult to make them efficient.

Having discussed the options for mutual exclusion on bus-based machines, let us move on to point-to-point and then barrier event synchronization.

5.6.4 Point-to-point Event Synchronization

Point-to-point synchronization within a parallel program is often implemented using busy-waiting on ordinary variables as flags. If we want to use blocking instead of busy-waiting, we can use semaphores just as they are used in concurrent programming and operating systems [TaW97].

Software Algorithms

Flags are control variables, typically used to communicate the occurrence of a synchronization event, rather than to transfer values. If two processes have a producer-consumer relationship on the shared variable *a*, then a flag can be used to manage the synchronization as shown below:

P1		P2	
a = f(x);	/* set a */	while (flag is 0) do nothing;	
flag = 1;		b = g(a);	/* use a */

If we know that the variable *a* is initialized to a certain value, say 0, which will be changed to a new value we are interested in by this production event, then we can use *a* itself as the synchronization flag, as follows:

```
            P1                                          P2
a = f(x);            /* set a */          while (a is 0) do nothing;
                                          b = g(a);          /* use a */
```

This eliminates the need for a separate flag variable, and saves the write to and read of that variable.

Hardware Support: Full-empty Bits

The last idea above has been extended in some research machines—although mostly machines with physically distributed memory—to provide hardware support for fine-grained producer-consumer synchronization. A bit, called a full-empty bit, is associated with every word in memory. This bit is set when the word is "full" with newly produced data (i.e. on a write), and unset when the word is "emptied" by a processor consuming those data (i.e. on a read). Word-level producer-consumer synchronization is then accomplished as follows. When the producer process wants to write the location it does so only if the full-empty bit is set to empty, and then leaves the bit set to full. The consumer reads the location only if the bit is full, and then sets it to empty. Hardware preserves the atomicity of the read or write with the manipulation of the full-empty bit. Given full-empty bits, our example above can be written without the spin loop as:

```
            P1                                          P2
a = f(x);            /* set a */             b = g(a);          /* use a */
```

Full-empty bits raise concerns about flexibility. For example, they do not lend themselves easily to single-producer multiple-consumer synchronization, or to the case where a producer updates a value multiple times before a consumer consumes it. Also, should all reads and writes use full-empty bits or only those that are compiled down to special instructions? The latter requires support in the language and compiler, but the former is too restrictive in imposing synchronization on all accesses to a location (for example, it does not allow asynchronous relaxation in iterative equation solvers, see Chapter 2). For these reasons and the hardware cost, full-empty bits have not found favor in most commercial machines.

Interrupts

Another important kind of event is the interrupt conveyed from an I/O device needing attention to a processor. In a uniprocessor machine there is no question where the interrupt should go, but in an SMP any processor can potentially take the interrupt. In addition, there are times when one processor may need to issue an interrupt to another. In early SMP designs special hardware was provided to monitor the priority of the process on each processor and deliver the I/O interrupt to the processor running at lowest priority. Such measures proved to be of small value and most modern machines use simple arbitration strategies. In addition, there is usually a memory mapped interrupt control region, so at kernel level any processor can interrupt any other by writing the interrupt information at the associated address.

5.6.5 Global (Barrier) Event Synchronization

Software Algorithms

Software algorithms for barriers are typically implemented using locks, shared counters and flags. Let us begin with a simple barrier among *p* processes, which is called a centralized barrier since it uses only a single lock, a single counter and a single flag.

Centralized Barrier

A shared counter maintains the number of processes that have arrived at the barrier, and is therefore incremented by every arriving process. These increments must be mutually exclusive. After incrementing the counter, the process checks to see if the counter equals *p*, i.e. if it is the last process to have arrived. If not, it busy waits on the flag associated with the barrier; if so, it writes the flag to release the waiting processes. A simple barrier algorithm may therefore look like:

```
struct bar_type {
     int counter;
     struct lock_type lock;
     int flag = 0;
} bar_name;

BARRIER (bar_name, p)
{
     LOCK(bar_name.lock);
     if (bar_name.counter == 0)
          bar_name.flag = 0;            /* reset flag if first to reach*/
     mycount = bar_name.counter++;  /* mycount is a private variable*/
     UNLOCK(bar_name.lock);
     if (mycount == p) {                /* last to arrive */
          bar_name.counter = 0;         /* reset counter for next barrier */
          bar_name.flag = 1;            /* release waiting processes */
     }
     else
          while (bar_name.flag == 0) {};/* busy wait for release */
}
```

Centralized Barrier with Sense Reversal

Can you see a problem with the above barrier? There is one. It occurs when the same barrier (barrier bar_name above) is used consecutively. That is, each processor executes the following code:

```
some computation...
BARRIER(bar1, p);
some more computation...
BARRIER(bar1, p);
```

The first process to enter the barrier the second time re-initializes the barrier counter, so that is not a problem. The problem is the flag. To exit the first barrier, processes spin on the flag until it is set to 1. Processes that see flag change to one will exit the barrier, perform the subsequent computation, and enter the barrier again. Suppose one processor P_x gets stuck while in the spin loop; for example, it gets swapped out by the operating system because it has been spinning too long. When it is swapped back in, it will continue to wait for the flag to change to 1. However, in the meantime other processes may have entered the second instance of the barrier, and the first of these will have reset the flag to 0. Now the flag will only get set to 1 again when all p processes have registered at the new instance of the barrier, which will never happen since P_x will never leave the spin loop and get to this barrier instance.

How can we solve this problem? What we need to do is prevent a process from entering a new instance of a barrier before all processes have exited the previous instance of the same barrier. One way is to use another counter to count the processes that leave the barrier, and not let a process enter a new barrier instance until this counter has turned to p for the previous instance. However, manipulating this counter incurs further latency and contention. A better solution is the following. The main reason for the problem in the previous case is that the flag is reset before all processes reach the next instance of the barrier. However, with the current setup we clearly cannot wait for all processes to reach the barrier before resetting the flag, since that is when we actually set the flag for the release. The solution is to have processes wait for the flag to obtain a different value in consecutive instances of the barrier, so for example processes may wait for the flag to turn to 1 in one instance and to turn to 0 in the next instance. A private variable is used per process to keep track of which value to wait for in the current barrier instance. Since by the semantics of a barrier a process cannot get more than one barrier ahead of another, we only need two values (0 and 1) that we toggle between each time, so we call this method sense-reversal. Now the flag need not be reset when the first process reaches the barrier; rather, the process stuck in the old barrier instance still waits for the flag to reach the old release value (sense), while processes that enter the new instance wait for the other (toggled) release value. The value of the flag is only changed when all processes have reached the barrier instance, so it will not change before processes stuck in the old instance see it. Here is the code for a simple barrier with sense-reversal.

```
BARRIER (bar_name, p)
{
     local_sense = !(local_sense);        /* toggle private sense variable */
     LOCK(bar_name.lock);
     mycount = bar_name.counter++;        /* mycount is a private variable*/
     if (bar_name.counter == p) {         /* last to arrive */
          UNLOCK(bar_name.lock);
          bar_name.counter = 0;           /* reset counter for next barrier */
          bar_name.flag = local_sense;    /* release waiting processes */
     }
     else {
          UNLOCK(bar_name.lock);
          while (bar_name.flag != local_sense) {};/* busy wait for release */
     }
}
```

The lock is not released immediately after the increment of the counter, but only after the condition is evaluated; the reason for this is left as an exercise (see Exercise 5.7) We now have a correct barrier that can be reused any number of times consecutively. The remaining issue is performance, which we examine next. (Note that the LOCK/UNLOCK protecting the increment of the counter can be replaced more efficiently by a simple LL-SC or atomic increment operation.

Performance Issues

The major performance goals for a barrier are similar to those for locks:

Low latency (small critical path length) Ignoring contention, we would like the number of operations in the chain of dependent operations needed for *p* processors to pass the barrier to be small.

Low traffic Since barriers are global operations, it is quite likely that many processors will try to execute a barrier at the same time. We would like the barrier algorithm to reduce the number of bus transactions and hence the possible contention.

Scalability Related to traffic, we would like to have a barrier that scales well to the number of processors we may want to support.

Low storage cost We would of course like to keep the storage cost low.

Fairness This is not much of an issue since all processors get released at about the same time, but we would like to ensure that the same processor does not always become the last one to exit the barrier, or to preserve FIFO ordering.

In a centralized barrier, each processor accesses the lock once, so the critical path length is at least *p*. Consider the bus traffic. To complete its operation, a centralized barrier involving *p* processors performs *2p* bus transactions to get the lock and increment the counter, two bus transactions for the last processor to reset the counter and write the release flag, and another *p-1* bus transactions to read the flag after it has been invalidated. Note that this is better than the traffic for

even a test-and-test&set lock to be acquired by p processes, because in that case each of the p releases causes an invalidation which results in $O(p)$ processes trying to perform the test&set again, thus resulting in $O(p^2)$ bus transactions. However, the contention resulting from these competing bus transactions can be substantial if many processors arrive at the barrier simultaneously, and barriers can be expensive.

Improving Barrier Algorithms for a Bus

Let us see if we can devise a better barrier for a bus. One part of the problem in the centralized barrier is that all processors contend for the same lock and flag variables. To address this, we can construct barriers that cause less processors to contend for the same variable. For example, processors can signal their arrival at the barrier through a software combining tree (see Chapter 3, Section 3.4.2). In a binary combining tree, only two processors notify each other of their arrival at each node of the tree, and only one of the two moves up to participate at the next higher level of the tree. Thus, only two processors access a given variable. In a network with multiple parallel paths, such as those found in larger-scale machines, a combining tree can perform much better than a centralized barrier since different pairs of processors can communicate with each other in different parts of the network in parallel. However, with a centralized interconnect like a bus, even though pairs of processors communicate through different variables they all generate transactions and hence contention on the same bus. Since a binary tree with p leaves has approximately $2p$ nodes, a combining tree requires a similar number of bus transactions to the centralized barrier. It also has higher latency since it requires $\log p$ steps to get from the leaves to the root of the tree, and in fact on the order of p serialized bus transactions. The advantage of a combining tree for a bus is that it does not use locks but rather simple read and write operations, which may compensate for its larger uncontended latency if the number of processors on the bus is large. However, the simple centralized barrier performs quite well on a bus, as shown in Figure 5-31. Some of the other, more scalable barriers shown in the figure for illustration will be discussed, along with tree barriers, in the context of scalable machines in Chapter 7.

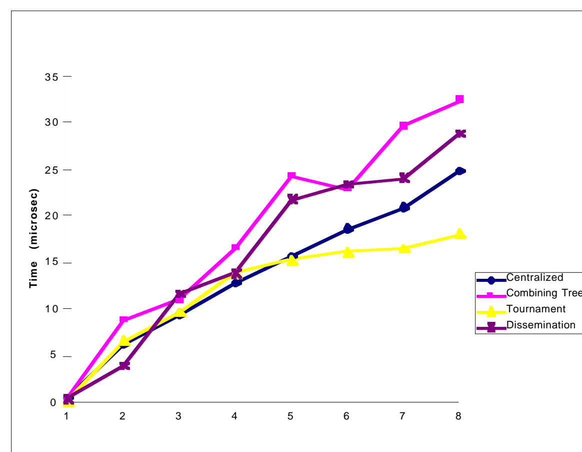


Figure 5-31 Performance of some barriers on the SGI Challenge.

Performance is measured as average time per barrier over a loop of many consecutive barriers. The higher critical path latency of the combining barrier hurts it on a bus, where it has no traffic and contention advantages.

Hardware Primitives

Since the centralized barrier uses locks and ordinary reads/writes, the hardware primitives needed depend on what lock algorithms are used. If a machine does not support atomic primitives well, combining tree barriers can be useful for bus-based machines as well.

A special bus primitive can be used to reduce the number of bus transactions in the centralized barrier. This optimization takes advantage of the fact that all processors issue a read miss for the same value of the flag when they are invalidated at the release. Instead of all processors issuing a separate read miss bus transaction, a processor can monitor the bus and abort its read miss before putting it on the bus if it sees the response to a read miss to the same location (issued by another processor that happened to get the bus first), and simply take the return value from the bus. In the best case, this “piggybacking” can reduce the number of read miss bus transactions from p to one.

Hardware Barriers

If a separate synchronization bus is provided, it can be used to support barriers in hardware too. This takes the traffic and contention off the main system bus, and can lead to higher-performance barriers. Conceptually, a wired-AND is enough. A processor sets its input to the gate high when it reaches the barrier, and waits till the output goes high before it can proceed. (In practice, reusing barriers requires that more than a single wire be used.) Such a separate hardware mechanism for barriers is particularly useful if the frequency of barriers is very high, as it may be in programs that are automatically parallelized by compilers at the inner loop level and need global synchronization after every innermost loop. However, it can be difficult to manage when not all processors on the machine participate in the barrier. For example, it is difficult to dynamically change the number of processors participating in the barrier, or to adapt the configuration when processes are migrated among processors by the operating system. Having multiple participating processes per processor also causes complications. Current bus-based multiprocessors therefore do not tend to provide special hardware support, but build barriers in software out of locks and shared variables.

5.6.6 Synchronization Summary

While some bus-based machines have provided full hardware support for synchronization operations such as locks and barriers, issues related to flexibility and doubts about the extent of the need for them has led to a movement toward providing simple atomic operations in hardware and synthesizing higher level synchronization operations from them in software libraries. The programmer can thus be unaware of these low-level atomic operations. The atomic operations can be implemented either as single instructions, or through speculative read-write instruction pairs like load-locked and store-conditional. The greater flexibility of the latter is making them increasingly popular. We have already seen some of the interplay between synchronization primitives, algorithms, and architectural details. This interplay will be much more pronounced when we discuss synchronization for scalable shared address space machines in the coming chapters. Theoretical research has identified the properties of different atomic exchange operations in terms of the time complexity of using them to implement synchronized access to variables. In particular, it is found that simple operations like test&set and fetch&op are not powerful enough to guarantee that the time taken by a processor to access a synchronized variable is independent of the number

of processors, while more sophisticated atomic operations like compare&swap and an memory-to-memory swap are [Her91].

5.7 Implications for Software

So far, we have looked at architectural issues and how architectural and protocol tradeoffs are affected by workload characteristics. Let us now come full circle and examine how the architectural characteristics of these small-scale machines influence parallel software. That is, instead of keeping the workload fixed and improving the machine or its protocols, we keep the machine fixed and examine how to improve parallel programs. Improving synchronization algorithms to reduce traffic and latency was an example of this, but let us look at the parallel programming process more generally, particularly the data locality and artifactual communication aspects of the orchestration step.

Of the techniques discussed in Chapter 3, those for load balance and inherent communication are the same here as in that general discussion. In addition, one general principle that is applicable across a wide range of computations is to try to assign computation such that as far as possible only one processor writes a given set of data, at least during a single computational phase. In many computations, processors read one large shared data structure and write another. For example, in Raytrace processors read a scene and write an image. There is a choice of whether to partition the computation so the processors write disjoint pieces of the destination structure and read-share the source structure, or so they read disjoint pieces of the source structure and write-share the destination. All other considerations being equal (such as load balance and programming complexity), it is usually advisable to avoid write-sharing in these situations. Write-sharing not only causes invalidations and hence cache misses and traffic, but if different processes write the same words it is very likely that the writes must be protected by synchronization such as locks, which are even more expensive.

The structure of communication is not much of a variable: With a single centralized memory, there is little incentive to use explicit large data transfers, so all communication is implicit through loads and stores which lead to the transfer of cache blocks. DMA can be used to copy large chunks of data from one area of memory to another faster than loads and stores, but this must be traded off against the overhead of invoking DMA and the possibility of using other latency hiding techniques instead. And with a zero-dimensional network topology (a bus) mapping is not an issue, other than to try to ensure that processes migrate from one processor to another as little as possible, and is invariably left to the operating system. The most interesting issues are related to temporal and spatial locality: to reduce the number of cache misses, and hence reduce both latency as well as traffic and contention on the shared bus.

With main memory being centralized, temporal locality is exploited in the processor caches. The specialization of the working set curve introduced in Chapter 3 is shown in Figure 5-32. All capacity-related traffic goes to the same local bus and centralized memory. The other three kinds of misses will occur and generate bus traffic even with an infinite cache. The major goal is to have working sets fit in the cache hierarchy, and the techniques are the same as those discussed in Chapter 3.

For spatial locality, a centralized memory makes data distribution and the granularity of allocation in main memory irrelevant (only interleaving data among memory banks to reduce conten-

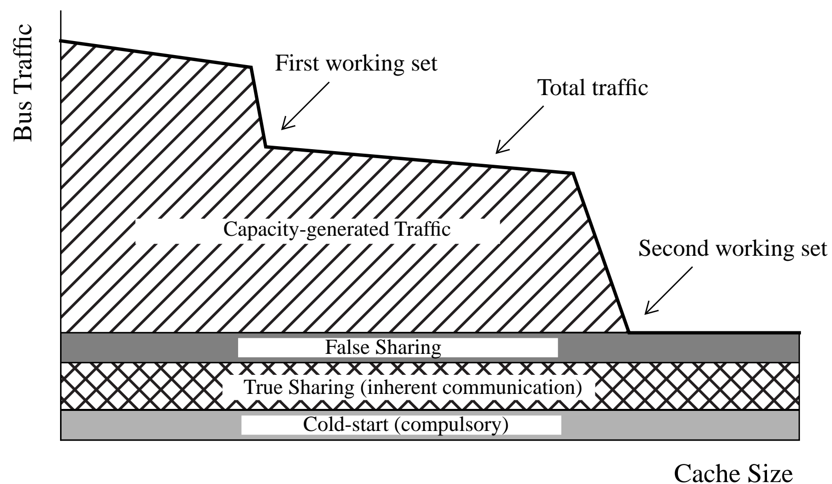


Figure 5-32 Data traffic on the shared bus, and its components as a function of cache size.

The points of inflection indicate the working sets of the program.

tion may be an issue just as in uniprocessors). The ill effects of poor spatial locality are *fragmentation*, i.e. fetching unnecessary data on a cache block, and *false sharing*. The reasons are that the *granularity of communication* and the *granularity of coherence* are both cache blocks, which are larger than a word. The former causes fragmentation in communication or data transfer, and the latter causes false sharing (we assume here that techniques like subblock dirty bits are not used, since they are not found in actual machines). Let us examine some techniques to alleviate these problems and effectively exploit the prefetching effects of long cache blocks, as well as techniques to alleviate cache conflicts by better spatial organization of data. There are many such techniques in a programmer's "bag of tricks", and we only provide a sampling of the most general ones.

Assign tasks to reduce spatial interleaving of access patterns. It is desirable to assign tasks such that each processor tends to access large contiguous chunks of data. For example, if an array computation with *n* elements is to be divided among *p* processors, it is better to divide it so that each processor accesses *n/p* contiguous elements rather than use a finely interleaved assignment of elements. This increases spatial locality and reduces false sharing of cache blocks. Of course, load balancing or other constraints may force us to do otherwise.

Structure data to reduce spatial interleaving of access patterns. We saw an example of this in the equation solver kernel in Chapter 3, when we used higher-dimensional arrays to keep a processor's partition of an array contiguous in the address space. There, the motivation was to allow proper distribution of data among physical memories, which is not an issue here. However, the same technique also helps reduce false sharing, fragmentation of data transfer, and conflict misses as shown in Figures 5-33 and 5-34, all of which cause misses and traffic on the bus. Consider false sharing and fragmentation for the equation solver kernel and the Ocean application. A cache block larger than a single grid element may straddle a column-oriented partition boundary as shown in Figure 5-33(a). If the block is larger than two grid elements, as is likely, it can cause communication due to false sharing. It is easiest to see if we assume for a moment that there is no inherent communication in the algorithm; for example, suppose in each sweep a process simply

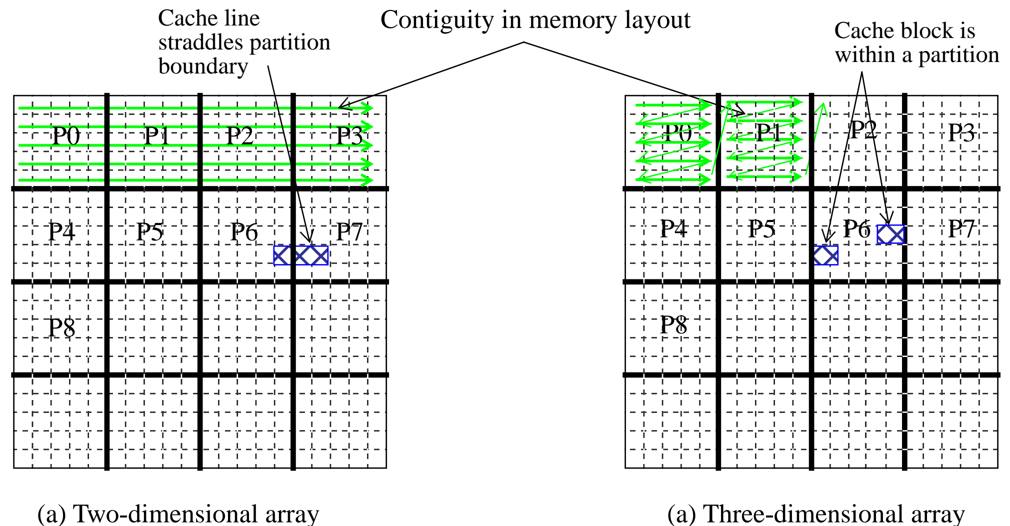


Figure 5-33 Reducing false sharing and fragmentation by using higher-dimensional arrays to keep partitions contiguous in the address space.

In the two-dimensional array case, cache blocks straddling partition boundaries cause both fragmentation (a miss brings in data from the other processor's partition) as well as false sharing.

added a constant value to each of its assigned grid elements, instead of performing a nearest-neighbor computation. A cache block straddling a partition boundary would be false-shared as different processors wrote different words on it. This would also cause fragmentation in communication, since a process reading its own boundary element and missing on it would also fetch other elements in the other processor's partition that are on the same cache block, and that it does not need. The conflict misses problem is explained in Figure 5-34. The issue in all these cases is also non-contiguity of partitions, only at a finer granularity (cache line) than for allocation (page). Thus, a single data structure transformation helps us solve all our spatial locality related problems in the equation solver kernel. Figure 5-35 illustrates the impact of using higher-dimensional arrays to represent grids or blocked matrices in the Ocean and LU applications on the SGI Challenge, where data distribution in main memory is not a issue. The impact of conflicts and false sharing on uniprocessor and multiprocessor performance is clear.

Beware of conflict misses. Figure 5-33 illustrates how allocating power of two sized arrays can cause cache conflict problems—which can become quite pathological in some cases—since the cache size is also a power of two. Even if the logical size of the array that the application needs is a power of two, it is often useful to allocate a larger array that is not a power of two and then access only the amount needed. However, this can interfere with allocating data at page granularity (also a power of two) in machines with physically distributed memory, so we may have to be careful. The cache mapping conflicts in these array or grid examples are within a data structure accessed in a predictable manner, and can thus be alleviated in a structured way. Mapping conflicts are more difficult to avoid when they happen *across* different major data structures (e.g. across different grids used by the Ocean application), where they may have to be alleviated by ad hoc padding and alignment. However, they are particularly insidious in a shared address space when they occur on seemingly harmless shared variables or data structures that a programmer is not inclined to think about, but that can generate artifactual communication. For example, a fre-

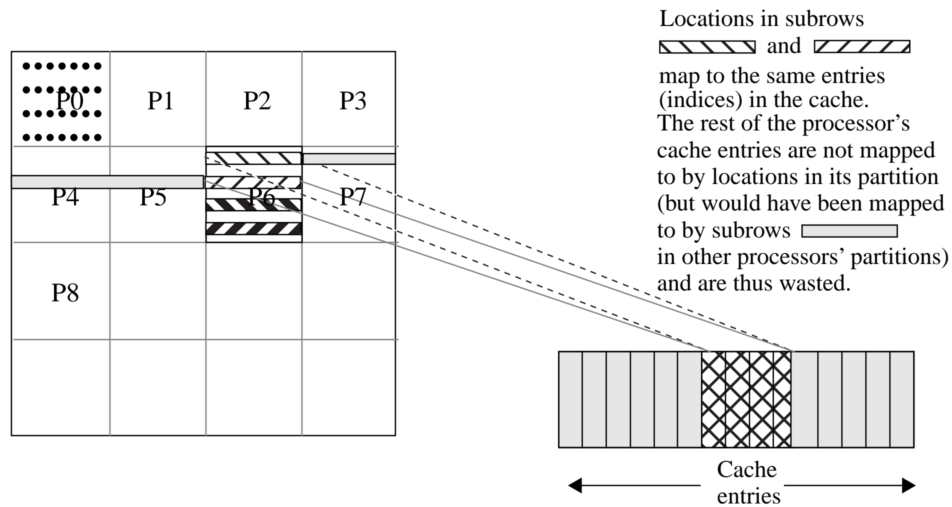


Figure 5-34 Cache mapping conflicts caused by a two-dimensional array representation in a direct-mapped cache.

The figure shows the worst case, in which the separation between successive subrows in a process's partition (i.e. the size of a full row of the 2-d array) is exactly equal to the size of the cache, so consecutive subrows map directly on top of one another in the cache. Every subrow accessed knocks the previous subrow out of the cache. In the next sweep over its partition, the processor will miss on every cache block it references, even if the cache as a whole is large enough to fit a whole partition. Many intermediately poor cases may be encountered depending on grid size, number of processors and cache size. *Since the cache size in bytes is a power of two, sizing the dimensions of allocated arrays to be powers of two is discouraged.*

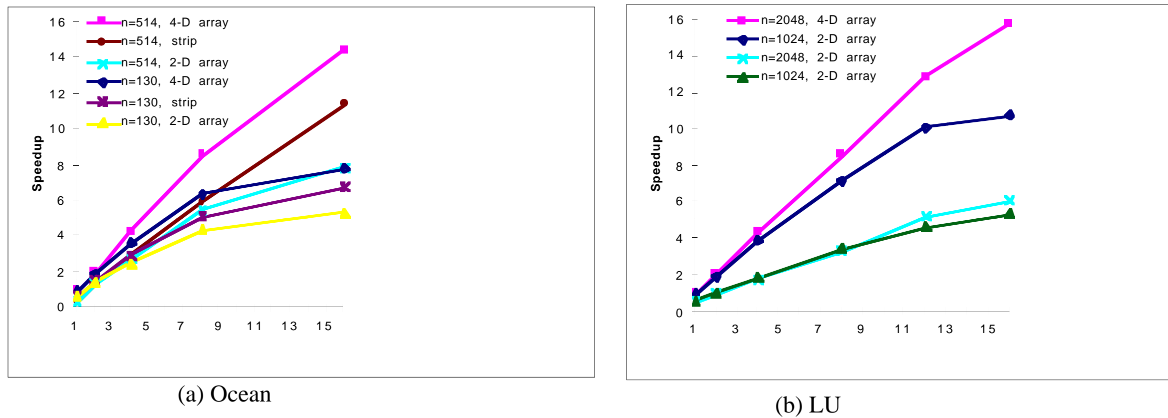


Figure 5-35 Effects of using 4-D versus 2-D arrays to represent two-dimensional grid or matrix data structures on the SGI Challenge.

Results are shown for different problem sizes for the Ocean and LU applications.

quently accessed pointer to an important data structure may conflict in the cache with a scalar variable that is also frequently accessed during the same computation, causing a lot of traffic. Fortunately, such problems tend to be infrequent in modern caches. In general, efforts to exploit locality can be wasted if attention is not paid to reducing conflict misses.

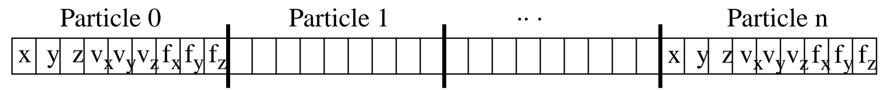
Use per-processor heaps: It is desirable to have separate heap regions for each processor (or process) from which it allocates data dynamically. Otherwise, if a program performs a lot of very small memory allocations, data used by different processors may fall on the same cache block.

Copy data to increase locality: If a processor is going to reuse a set of data that are otherwise allocated non-contiguously in the address space, it is often desirable to make a contiguous copy of the data for that period to improve spatial locality and reduce cache conflicts. Copying requires memory accesses and has a cost, and is not useful if the data are likely to reside in the cache anyway. But otherwise the cost can be overcome by the advantages. For example, in blocked matrix factorization or multiplication, with a 2-D array representation of the matrix a block is not contiguous in the address space (just like a partition in the equation solver kernel); it is therefore common to copy blocks used from another processor's assigned set to a contiguous temporary data structure to reduce conflict misses. This incurs a cost in copying, which must be traded off against the benefit of reducing conflicts. In particle-based applications, when a particle moves from one processor's partition to another, spatial locality can be improved by moving the attributes of that particle so that the memory for all particles being handled by a processor remains contiguous and dense.

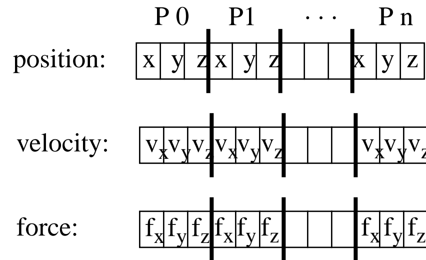
Pad arrays: Beginning parallel programmers often build arrays that are indexed using the process identifier. For example, to keep track of load balance, one may have an array each entry of which is an integer that records the number of tasks completed by the corresponding processor. Since many elements of such an array fall into a single cache block, and since these elements will be updated quite often, false sharing becomes a severe problem. One solution is to pad each entry with dummy words to make its size as large as the cache block size (or, to make the code more robust, as big as the largest cache block size on anticipated machines), and then align the array to a cache block. However, padding many large arrays can result in significant waste of memory, and can cause fragmentation in data transfer. A better strategy is to combine all such variables for a process into a record, pad the entire record to a cache block boundary, and create an array of records indexed by process identifier.

Determine how to organize arrays of records: Suppose we have a number of logical records to represent, such as the particles in the Barnes-Hut gravitational simulation. Should we represent them as a single array of n particles, each entry being a record with fields like position, velocity, force, mass, etc. as in Figure 5-36(a)? Or should we represent it as separate arrays of size n, one per field, as in Figure 5-36(b)? Programs written for vector machines, such as traditional Cray computers, tend to use a separate array for each property of an object, in fact even one per field per physical dimension x, y or z. When data are accessed by field, for example the x coordinate of all particles, this increases the performance of vector operations by making accesses to memory unit stride and hence reducing memory bank conflicts. In cache-coherent multiprocessors, however, there are new tradeoffs, and the best way to organize data depends on the access patterns. Consider the update phase of the Barnes-Hut application. A processor reads and writes only the position and velocity fields of all its assigned particles, but its assigned particles are not contiguous in the shared particle array. Suppose there is one array of size n (particles) per field or property. A double-precision three-dimensional position (or velocity) is 24 bytes of data, so several of these may fit on a cache block. Since adjacent particles in the array may be read and written by other processors, poor spatial locality and false sharing can result. In this case it is better to have a single array of particle records, where each record holds all information about that particle.

Now consider the force-calculation phase of the same Barnes-Hut application. Suppose we use an organization by particle rather than by field as above. To compute the forces on a particle, a



(a) Organization by particle



(a) Organization by property or field

Figure 5-36 Alternative data structure organizations for record-based data.

processor reads the position values of many other particles and cells, and it then updates the force components of its own particle. In updating force components, it invalidates the position values of this particle from the caches of other processors that are using and reusing them, due to false sharing, even though the position values themselves are not being modified in this phase of computation. In this case, it would probably be better if we were to split the single array of particle records into two arrays of size *n* each, one for positions (and perhaps other fields) and one for forces. The force array itself could be padded to reduce false-sharing in it. In general, it is often beneficial to split arrays to separate fields that are used in a read-only manner in a phase from those fields whose values are updated in the same phase. Different situations dictate different organizations for a data structure, and the ultimate decision depends on which pattern or phase dominates performance.

Align arrays: In conjunction with the techniques discussed above, it is often necessary to align arrays to cache block boundaries to achieve the full benefits. For example, given a cache block size of 64 bytes and 8 byte fields, we may have decided to build a single array of particle records with x, y, z, fx, fy, and fx fields, each 48-byte record padded with two dummy fields to fill a cache block. However, this wouldn't help if the array started at an offset of 32 bytes from a page in the virtual address space, as this would mean that the data for each particle will now span two cache blocks. Alignment is easy to achieve by simply allocating a little extra memory and suitably adjusting the starting address of the array.

As seen above, the organization, alignment and padding of data structures are all important to exploiting spatial locality and reducing false sharing and conflict misses. Experienced programmers and even some compilers use these techniques. As we discussed in Chapter 3, these locality and artifactual communication issues can be significant enough to overcome differences in inherent communication, and can cause us to revisit our algorithmic partitioning decisions for an application (recall strip versus subblock decomposition for the simple equation solver as discussed in Chapter 3, Section 3.2.2).

5.8 Concluding Remarks

Symmetric shared-memory multiprocessors are a natural extension of uniprocessor workstations and personal-computers. Sequential applications can run totally unchanged, and yet benefit in performance by getting a larger fraction of a processor's time and from the large amount of shared main-memory and I/O capacity typically available on such machines. Parallel applications are also relative easy to bring up, as all local and shared data are directly accessible from all processors using ordinary loads and stores. Computationally intensive portions of a sequential application can be selectively parallelized. A key advantage for multiprogrammed workloads is the fine granularity at which resources can be shared among application processes. This is true both temporally, in that processors and/or main-memory pages can be frequently reallocated among different application processes, and also physically, in that main-memory may be split among applications at the granularity of an individual page. Because of these appealing features, all major vendors of computer systems, from workstation suppliers like SUN, Silicon Graphics, Hewlett Packard, Digital, and IBM to personal computer suppliers like Intel and Compaq are producing and selling such machines. In fact, for some of the large workstation vendors, these multiprocessors constitute a substantial fraction of their revenue stream, and a still larger fraction of their net profits due to the higher margins [Hep90] on these higher-end machines.

The key technical challenge in the design of such machines is the organization and implementation of the shared memory subsystem, which is used for communication between processors in addition to handling all regular memory accesses. The majority of small-scale machines found today use the system bus as the interconnect for communication, and the challenge then becomes how to keep shared data in the private caches of the processors coherent. There are a large variety of options available to the system architect, as we have discussed, including the set of states associated with cache blocks, choice of cache block size, and whether updates or invalidates are used. The key task of the system architect is to make choices that will perform well based on predominant sharing patterns expected in workloads, and those that will make the task of implementation easier.

As processor, memory-system, integrated-circuit, and packaging technology continue to make rapid progress, there are questions raised about the future of small-scale multiprocessors and importance of various design issues. We can expect small-scale multiprocessors to continue to be important for at least three reasons. The first is that they offer an attractive cost-performance point. Individuals or small groups of people can easily afford them for use as a shared resource or a compute- or file-server. Second, microprocessors today are designed to be multiprocessor-ready, and designers are aware of future microprocessor trends when they begin to design the next generation multiprocessor, so there is no longer a significant time lag between the latest microprocessor and its incorporation in a multiprocessor. As we saw in Chapter 1, the Intel Pentium processor line plugs in "gluelessly" into a shared bus. The third reason is that the essential software technology for parallel machines (compilers, operating systems, programming languages) is maturing rapidly for small-scale shared-memory machines, although it still has a substantial way to go for large-scale machines. Most computer systems vendors, for example, have parallel versions of their operating systems ready for their bus-based multiprocessors. The design issues that we have explored in this chapter are fundamental, and will remain important with progress in technology. This is not to say that the optimal design choices will not change.

We have explored many of the key design aspects of bus-based multiprocessors at the "logical level" involving cache block state transitions and complete bus transactions. At this level the

approach appears to be a rather simple extension of traditional cache controllers. However, much of the difficulty in such designs, and many of the opportunities for optimization and innovation occur at the more detailed "physical level." The next chapter goes down a level deeper into the hardware design and organization of bus-based cache-coherent multiprocessors and some of the natural generalizations.

5.9 References

[AdG96] Sarita V. Adve and Kourosh Gharachorloo. Shared Memory Consistency Models: A Tutorial. IEEE Computer, vol. 29, no. 12, December 1996, pp. 66-76.

[AgG88] Anant Agarwal and Anoop Gupta. Memory-reference Characteristics of Multiprocessor Applications Under MACH. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 215-225, May 1988.

[ArB86] James Archibald and Jean-Loup Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems*, 4(4):273-298, November 1986.

[BaW88] Jean-Loup Baer and Wen-Hann Wang. On the Inclusion Properties for Multi-level Cache Hierarchies. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pp. 73--80, May 1988.

[BJS88] F. Baskett, T. Jermoluk, and D. Solomon, The 4D-MP graphics superworkstation: Computing + graphics = 40 MIPS + 40 MFLOPS and 100,000 lighted polygons per second. *Proc of the 33rd IEEE Computer Society Int. Conf. - COMPCOM 88*, pp. 468-71, Feb. 1988.

[BRG+89] David Black, Richard Rashid, David Golub, Charles Hill, Robert Baron. Translation Lookaside Buffer Consistency: A Software Approach. In *Proceedings of the3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, April 1989.

[CAH+93] Ken Chan, et al. Multiprocessor Features of the HP Corporate Business Servers. In *Proceedings of COMPCON*, pp. 330-337, Spring 1993.

[CoF93] Alan Cox and Robert Fowler. Adaptive Cache Coherency for Detecting Migratory Shared Data. In *Proceedings of the 20th International Symposium on Computer Architecture*, pp. 98-108, May 1993.

[Dah95] Fredrik Dahlgren. Boosting the Performance of Hybrid Snooping Cache Protocols. In *Proceedings of the 22nd International Symposium on Computer Architecture*, June 1995, pp. 60-69.

[DDS94] Fredrik Dahlgren and Michel Dubois and Per Stenstrom. Combined Performance Gains of Simple Cache Protocol Extensions. In *Proceedings of the 21st International Symposium on Computer Architecture*, April 1994, pp. 187-197.

[DSB86] Michel Dubois, Christoph Scheurich and Faye A. Briggs. Memory Access Buffering in Multiprocessors. In *Proceedings of the 13th International Symposium on Computer Architecture*, June 1986, pp. 434-442.

[DSR+93] Michel Dubois, Jonas Skeppstedt, Livio Ricciulli, Krishnan Ramamurthy, and Per Stenstrom. The Detection and Elimination of Useless Misses in Multiprocessors. In *Proceedings of the 20th International Symposium on Computer Architecture*, pp. 88-97, May 1993.

[DuL92] C. Dubnicki and T. LeBlanc. Adjustable Block Size Coherent Caches. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp. 170-180, May 1992.

[EgK88] Susan Eggers and Randy Katz. A Characterization of Sharing in Parallel Programs and its Application to Coherency Protocol Evaluation. In *Proceedings of the 15th Annual International Sym-*

posium on Computer Architecture, pp. 373-382, May 1988.

[EgK89a] Susan Eggers and Randy Katz. The Effect of Sharing on the Cache and Bus Performance of Parallel Programs. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 257-270, May 1989.

[EgK89b] Susan Eggers and Randy Katz. Evaluating the Performance of Four Snooping Cache Coherency Protocols. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pp. 2-15, May 1989.

[FCS+93] Jean-Marc Frailong, et al. The Next Generation SPARC Multiprocessing System Architecture. In *Proceedings of COMPCON*, pp. 475-480, Spring 1993.

[GaW93] Mike Galles and Eric Williams. Performance Optimizations, Implementation, and Verification of the SGI Challenge Multiprocessor. In Proceedings of 27th Annual Hawaii International Conference on Systems Sciences, January 1993.

[Goo83] James Goodman. Using Cache Memory to Reduce Processor-Memory Traffic. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pp. 124-131, June 1983.

[Goo87] James Goodman. Coherency for Multiprocessor Virtual Address Caches. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, October 1987.

[GrCh96] M. Greenwald and D.R. Cheriton. The Synergy Between Non-blocking Synchronization and Operating System Structure, Proceedings of the Second Symposium on Operating System Design and Implementation. USENIX, Seattle, October, 1996, pp 123-136.

[GSD95] H. Grahn, P. Stenstrom, and M. Dubois, Implementation and Evaluation of Update-based Protocols under Relaxed Memory Consistency Models. In Future Generation Computer Systems, 11(3): 247-271, June 1995.

[GrT90] Gary Granuke and Shreekant Thakkar. Synchronization Algorithms for Shared Memory Multiprocessors. IEEE Computer, vol 23, no. 6, pp. 60-69, June 1990.

[HeP90] John Hennessy and David Patterson. Computer Architecture: A Quantitative Approach. Morgan Kaufmann Publishers, 1990.

[Her88] Maurice Herlihy. Impossibility and Universality Results for Wait-Free Synchronization. Seventh ACM SIGACTS-SIGOPS Symposium on Principles of Distributed Computing, August 88.

[Her93] M. Herlihy. A Methodology for Implementing Highly Concurrent Data Objects ACM Transactions on Programming Languages and Systems, 15(5), 745-770, November, 1993

[HeWi87] Maurice Herlihy and Janet Wing. Axioms for Concurrent Objects, Proc. of the 14th ACM Symp. on Principles of Programming Languages, pp. 13-26, Jan. 1987.

[HiS87] Mark D. Hill and Alan Jay Smith. Evaluating Associativity in CPU Caches. IEEE Transactions on Computers, vol. C-38, no. 12, December 1989, pp. 1612-1630.

[HEL+86] Mark Hill et al. Design Decisions in SPUR. *IEEE Computer*, 19(10):8-22, November 1986.

[HeMo93] M.P. Herlihy and J.E.B. Moss. Transactional Memory: Architectural support for lock-free data structures. 1993 20th Annual Symposium on Computer Architecture San Diego, Calif. pp. 289-301. May 1993.

[JeE91] Tor E. Jeremiassen and Susan J. Eggers. Eliminating False Sharing. In *Proceedings of the 1991 International Conference on Parallel Processing*, pp. 377-381.

[KMR+86] A. R. Karlin, M. S. Manasse, L. Rudolph and D. D. Sleator. Competitive Snoopy Caching. In Proceedings of the 27th Annual IEEE Symposium on Foundations of Computer Science, 1986.

[Kro81] D. Kroft. Lockup-free Instruction Fetch/Prefetch Cache Organization. In *Proceedings of 8th International Symposium on Computer Architecture*, pp. 81-87, May 1981.

[Lam79] Leslie Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. IEEE Transactions on Computers, vol. C-28, no. 9, September 1979, pp. 690-691.

[Lau94] James Laudon. Architectural and Implementation Tradeoffs for Multiple-Context Processors. Ph.D. Thesis, Computer Systems Laboratory, Stanford University, 1994.

[Lei92] C. Leiserson, et al. The Network Architecture of the Connection Machine CM-5. Symposium of Parallel Algorithms and Architectures, 1992.

[LLJ+92] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. The DASH Prototype: Logic Overhead and Performance. IEEE Transactions on Parallel and Distributed Systems, 4(1):41-61, January 1993.

[MaPu91] H. Massalin and C. Pu. A lock-free multiprocessor OS kernel. Technical Report CUCS-005-01, Computer Science Department, Columbia University, October 1991.

[McC84] McCreight, E. The Dragon Computer System: An Early Overview. Technical Report, Xerox Corporation, September 1984.

[MCS91] John Mellor-Crummey and Michael Scott. Algorithms for Scalable Synchronization on Shared Memory Mutiprocessors. ACM Transactions on Computer Systems, vol. 9, no. 1, pp. 21-65, February 1991.

[MiSc96] M. Michael and M. Scott, Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms", Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing, Philadelphia, PA, pp 267-276, May 1996.

[Mip91] MIPS R4000 User's Manual. MIPS Computer Systems Inc. 1991.

[PaP84] Mark Papamarcos and Janak Patel. A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pp. 348-354, June 1984.

[Papa79] C. H. Papadimitriou. The Serializability of Conccurent Database Updates, Journal of the ACM, 26(4):631-653, Oct. 1979.

[Ros89] Bryan Rosenburg. Low-Synchronization Translation Lookaside Buffer Consistency in Large-Scale Shared-Memory Multiprocessors. In *Proceedings of the Symposium on Operating Systems Principles*, December 1989.

[ScD87] Christoph Scheurich and Michel Dubois. Correct Memory Operation of Cache-based Multiprocessors. In *Proceedings of the 14th International Symposium on Computer Architecture*, June 1987, pp. 234-243.

[SFG+93] Pradeep Sindhu, et al. XDBus: A High-Performance, Consistent, Packet Switched VLSI Bus. In *Proceedings of COMPCON*, pp. 338-344, Spring 1993.

[SHG93] Jaswinder Pal Singh, John L. Hennessy and Anoop Gupta. Scaling Parallel Programs for Multiprocessors: Methodology and Examples. *IEEE Computer*, vol. 26, no. 7, July 1993.

[Smi82] Alan Jay Smith. Cache Memories. *ACM Computing Surveys*, 14(3):473-530, September 1982.

[SwS86] Paul Sweazey and Alan Jay Smith. A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus. In *Proceedings of the 13th International Symposium on Computer Architecture*, pp. 414-423, May 1986.

[TaW97] Andrew S. Tanenbaum and Albert S. Woodhull, Operating System Design and Implementation (Second Edition), Prentice Hall, 1997.

[Tel90] Patricia Teller. Translation-Lookaside Buffer Consistency. *IEEE Computer*, 23(6):26-36, June 1990.

[TBJ+88] M. Thompson, J. Barton, T. Jermoluk, and J. Wagner. Translation Lookaside Buffer Synchronization in a Multiprocessor System. In *Proceedings of USENIX Technical Conference,* February

1988.

[TLS88] Charles Thacker, Lawrence Stewart, and Edwin Satterthwaite, Jr. Firefly: A Multiprocessor Workstation, IEEE Transactions on Computers, vol. 37, no. 8, Aug. 1988, pp. 909-20.

[TLH94] Josep Torrellas, Monica S. Lam, and John L. Hennessy. False Sharing and Spatial Locality in Multiprocessor Caches. *IEEE Transactions on Computers*, 43(6):651-663, June 1994.

[Val95] J. Valois, Lock-Free Linked Lists Using Compare-and-Swap, Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing, Ottawa, Ont. Canada, pp 214-222, August 20-23, 1995

[WBL89] Wen-Hann Wang, Jean-Loup Baer and Henry M. Levy. Organization and Performance of a Two-Level Virtual-Real Cache Hierarchy. In Proceedings of the 16th Annual International Symposium on Computer Architecture, pp. 140-148, June 1989.

[WeS94] Shlomo Weiss and James Smith. Power and PowerPC. Morgan Kaufmann Publishers Inc. 1994.

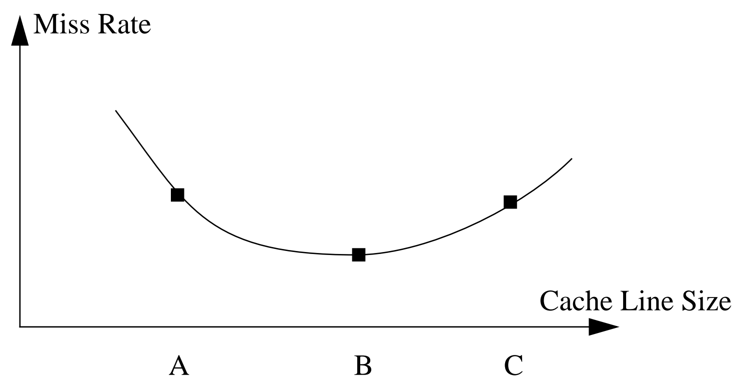
[WEG+86] David Wood, et al. An In-cache Address Translation Mechanism. In *Proceedings of the 13th International Symposium on Computer Architecture*, pp. 358-365, May 1986.

5.10 Exercises

5.1 Short Answer.

a. Is the cache-coherence problem an issue with processor-level registers as well? Given that registers are not kept consistent in hardware, how do current systems guarantee the desired semantics of a program under this hardware model?

b. Consider the following graph indicating the miss rate of an application as a function of cache block size. As might be expected, the curve has a U-shaped appearance. Consider the three points A, B, and C on the curve. Indicate under what circumstances, if any, each may be a sensible operating point for the machine (i.e. the machine might give better performance at that point rather than at the other two points).



5.2 Bus Traffic

Assume the following average data memory traffic for a bus-based shared memory multiprocessor:

private reads70%

private writes20%

shared reads 8%

shared writes2%

Also assume that 50% of the instructions (32-bits each) are either loads or stores. With a split instruction/data cache of 32Kbyte total size, we get hit rates of 97% for private data, 95% for shared data and 98.5% for instructions. The cache line size is 16 bytes.

We want to place as many processors as possible on a bus that has 64 data lines and 32 address lines.

The processor clock is twice as fast as that of the bus, and before considering memory penalties the processor CPI is 2.0. How many processors can the bus support without saturating if we use

(i) write-through caches with write-allocate strategy?

(ii) write-back caches?

Ignore cache consistency traffic and bus contention. The probability of having to replace a dirty block in the write-back caches is 0.3. For reads, memory responds with data 2 cycles after being presented the address. For writes, both address and data are presented to memory at the same time. Assume that the bus does not have split transactions, and that processor miss penalties are equal to the number of bus cycles required for each miss.

5.3 Update versus invalidate

a. For the memory reference streams given below, compare the cost of executing them on a bus-based machine that (i) supports the Illinois MESI protocol, and (ii) the Dragon protocol. Explain the observed performance differences in terms of the characteristics of the streams and the coherence protocols.

stream-1: r1 w1 r1 w1 r2 w2 r2 w2 r3 w3 r3 w3

stream-2: r1 r2 r3 w1 w2 w3 r1 r2 r3 w3 w1

stream-3: r1 r2 r3 r3 w1 w1 w1 w1 w2 w3

In the above streams, all of the references are to the same location: r/w indicates read or write, and the digit refers to the processor issuing the reference. Assume that all caches are initially empty, and use the following cost model: (i) read/write cache-hit: 1 cycle; (ii) misses requiring simple transaction on bus (BusUpgr, BusUpd): 3 cycles, and (iii) misses requiring whole cache block transfer = 6 cycles. Assume all caches are write allocated.

b. As miss latencies increase, does an update protocol become more or less preferable as compared to an invalidate protocol? Explain.

c. One of the key features of the update protocol is that instead of whole cache blocks, individual words are transferred over the bus. Given the design of the SGI Powerpath-2 bus in Section 6.6, estimate the peak available bandwidth if *all* bus transactions were updates. Assume that all updates contain 8 bytes of data, and that they also require a 5-cycle transaction. Then, given the miss-decomposition data for various applications in Figure 5-21, estimate the peak available bus bandwidth on the Powerpath-2 bus for these applications.

d. In a multi-level cache hierarchy, would you propagate updates all the way to the first-level cache or only to the second-level cache? Explain the tradeoffs.

e. Why is update-based coherence not a good idea for multiprogramming workloads typically found on multiprocessor compute servers today?

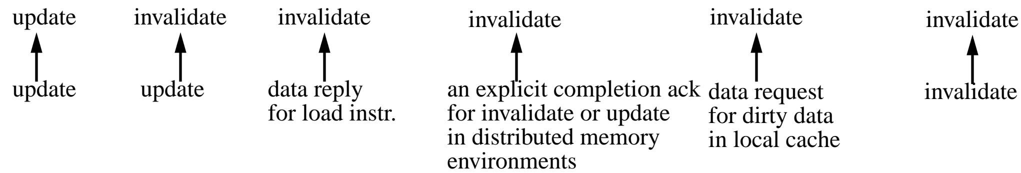
f. To provide an update protocol as an alternative, some machines have given control of the type of protocol to the software at the granularity of page, that is, a given page can be kept coherent either using an update scheme or an invalidate scheme. An alternative to page-based control is to provide special opcodes for writes that will cause updates rather than invalidates. Comment on the advantages and disadvantages.

5.4 Sequential Consistency.

a. Consider incoming transactions (requests and responses) from the bus down into the cache hierarchy, in a system with two-level caches, and outgoing transactions from the cache hierarchy to the bus. To ensure SC when invalidations are acknowledged early (as soon as they appear on the bus), what ordering constraints must be maintained in each direction among the ones described below, and which ones can be reordered? Now con-

sider the incoming transactions at the first level cache, and answer the question for these. Assume that the processor has only one outstanding request at a time.

Orderings in the upward direction (from bus towards the caches and the processor):



Orderings in the downward direction (from the processor and caches towards the bus):

- processor's single outstanding request can be reordered w.r.t. other replies being generated by the caches;
- replies to requests (where this caches had data in dirty state) can be reordered, since these distinct requests must have been from different processors (as each processor can have only one outstanding load/store request).
- a writeback can be reorderedw.r.t. above data replies for dirty cache blocks.

b. Given the following code segments, say what results are possible (or not possible) under SC. Assume that all variables are initialized to 0 before this code is reached. [

(i)

P1	P2	P3
A = 1	u = A	v = B
	B = 1	w = A

(ii)

P1	P2	P3	P4
A = 1	u = A	B = 1	w = B
	v = B		x = A

(iii) In the following sequence, first consider the operations within a dashed box to be part of the same instruction, say a fetch&increment. Then, suppose they are separate instructions. Answer the questions for both cases.

P1	P2
u = A	v = A
A = u+1	A = v+1

c. Is the reordering problem due to write buffers, discussed in Section 5.3.2, also a problem for concurrent programs on a uniprocessor? If so, how would you prevent it; if not, why not?

d. Can a read complete before a previous write issued by the same processor to the same location has completed (for example, if the write has been placed in the writer's write buffer but has not yet become visible to other processors), and still provide a coherent memory system? If so, what value should the read return? If not, why not? Can this be done and still guarantee sequential consistency?

e. If we cared only about coherence and not about sequential consistency, could we declare a write to complete as soon as the processor is able to proceed past it?

f. Are the sufficient conditions for SC necessary? Make them less constraining (i) as much as possible and (ii) in a reasonable intermediate way, and comment on the effects on implementation complexity.

5.5 Consider the following conditions proposed as sufficient conditions for SC:

- Every process issues memory requests in the order specified by the program.
- After a read or write operation is issued, the issuing process waits for the operation to complete before issuing its next operation.
- Before a processor P_j can return a value written by another processor P_i, all operations that were performed with respect to P_i before it issued the store must also be performed with respect to P_j.

Are these conditions indeed sufficient to guarantee SC executions? If not, construct a counter-example, and say why the conditions that were listed in the chapter are indeed sufficient in that case. [Hint: to construct the example, think about in what way these conditions are different from the ones in the chapter. The example is not obvious.]

5.6 Synchronization Performance

a. Consider a 4-processor bus-based multiprocessor using the Illinois MESI protocol. Each processor executes a test&set lock to gain access to a null critical section. Assume the test&set instruction always goes on the bus, and it takes the same time as a normal read transaction. Assume that the initial condition is such that processor-1 has the lock, and that processors 2, 3, and 4 are spinning on their caches waiting for the lock to be released. The final condition is that every processor gets the lock once and then exits the program. Considering only the bus transactions related to lock/unlock code:

(i) What is the least number of transactions executed to get from the initial to the final state?

(ii) What is the worst case number of transactions?

(iii) Repeat parts a and b assuming the Dragon protocol.

b. What are the main advantages and disadvantages of exponential backoff in locks?

c. Suppose all 16 processors in a bus-based machine try to acquire a test-and-test&set lock simultaneously. Assume all processors are spinning on the lock in their caches and are invalidated by a release at time 0. How many bus transactions will it take until all processors have acquired the lock if all the critical sections are empty (i.e. each processor simply does a LOCK and UNLOCK with nothing in between). Assuming that the bus is fair (services pending requests before new ones) and that every bus transaction takes 50 cycles, how long would it take before the *first* processor acquires and releases the lock. How long before the last processor to acquire the lock is able to acquire and release it? What is the best you could do with an unfair bus, letting whatever processor you like win an arbitration regardless of the order of the requests? Can you improve the performance by choosing a different (but fixed) bus arbitration scheme than a fair one?

d. If the varaibles used for implementing locks are not cached, will a test&test&set lock still generate less traffic than a test&set lock? Explain your answer.

e. For the same machine configuration in the previous exercise with a fair bus, answer the same questions about the number of bus transactions and the time needed for the first and last processors to acquire and release the lock when using a ticket lock. Then do the same for the array-based lock.

f. For the performance curves for the test&set lock with exponential backoff shown in Figure 5-29 on page 321, why do you think the curve for the non-zero critical section was a little worse than the curve for the null critical section?

g. Why do we use d to be smaller than c in our lock experiments. What problems might occur in measurement if we used d larger than c. (Hint: draw time-lines for two processor executions.) How would you expect the results to change if we used much larger values for c and d?

5.7 Synchronization Algorithms

a. Write pseudo code (high-level plus assembly) to implement the ticket lock and array-based lock using (i) fetch&increment, (ii) LL/SC.

b. Suppose you did not have a fetch&increment primitive but only a fetch&store (a simple atomic exchange). Could you implement the array-based lock with this primitive? Describe the resulting lock algorithm?

c. Implement a compare&swap operation using LL-SC.

d. Consider the barrier algorithm with sense reversal that was described in Section 5.6.5. Would there be a problem be if the UNLOCK statement were placed just after the increment of the counter, rather than after each branch of the if condition? What would it be?

e. Suppose that we have a machinethat supports full/empty bits on every word in hardware. This particular machine allows for the following C-code functions:

ST_Special(locn, val) writes val to data location locn and sets the full bit. If the full bit was already set, a trap is signalled.

int LD_Special(locn) waits until the data location's full bit is set, loads the data, clears the full bit, and returns the data as the result.

Write a C function swap(i,j) that uses the above primitives to atomically swap the contents of two locations A[i] and A[j]. You should allow high concurrency (if multiple PEs want to swap distinct paris of locations, they should be able to do so concurrently). You must avoid deadlock.

f. The fetch-and-add atomic operation can be used to implement barriers, semaphores and other synchronization mechanisms. The semantics of `fetch-and-add` is such that it returns the value before the addition. Use the fetch-and-add primitive to implement a barrier operation suitable for a shared memory multiprocessor. To use the barrier, a processor must execute: BARRIER(BAR, N), where `N` is the number of processes that need to arrive at the barrier before any of them can proceed. Assume that `N` has the same value in each use of barrier `BAR`. The barrier should be capable of supporting the following code:

```
    while (condition) {
        Compute stuff
        BARRIER(BAR, N);
}
```

1. A proposed solution for implementing the barrier is the following:

```
BARRIER(Var B: BarVariable, N: integer)
{
    if (fetch-and-add(B, 1) = N-1) then
        B := 0;
    else
        while (B <> 0) do {};
}
```

What is the problem with the above code? Write the code for `BARRIER` in away that avoids the problem.

g. Consider the following implementation of the BARRIER synchronization primitive, used repeatedly within an application, at the end of each phase of computation. Assume that `bar.releasing` and `bar.count` are initially zero and `bar.lock` is initially unlocked. The barrier is described here as a function instead of a macro, for simplicity. Note that this function takes no arguments:

```
struct bar_struct {
   LOCKDEC(lock);
   int count, releasing;
} bar;
...

BARRIER()
{
   LOCK(bar.lock);
   bar.count++;

   if (bar.count == numProcs) {
      bar.releasing = 1;
      bar.count--;
   } else {
      UNLOCK(bar.lock);
      while (! bar.releasing)
         ;
      LOCK(bar.lock);
      bar.count--;
      if (bar.count == 0) {
         bar.releasing = 0;
      }
   }
   UNLOCK(bar.lock);
}
```

(i) This code fails to provide a correct barrier. Describe the problem with this implementation.

(ii) Change the code *as little as possible* so it provides a correct barrier implementation. Either clearly indicate your changes on the code above, or thoroughly describe the changes below.

5.8 Protocol Enhancements. Consider migratory data, which are shared data objects that bounce around among processors, with each processor reading and then writing them. Under the standard MESI protocol, the read miss and the write both generate bus transactions.

a. Given the data in Table 5-1 estimate max bandwidth that can be saved when using upgrades (BusUpgr) instead of BusRdX.

b. It is possible to enhance the state stored with cache blocks and the state-transition diagram, so that such read shortly-followed-by write accesses can be recognized, so that migratory blocks can be directly brought in exclusive state into the cache on the first read miss. Suggest the extra states and the state-transition diagram extensions to achieve above. Using the data in Table 5-1, Table 5-2, and Table 5-3 compute the bandwidth sav-

ings that can be achieved. Are there any other benefits than bandwidth savings? Discuss program situations where the migratory protocol may hurt performance.

c. The Firefly update protocol eliminates the SM state present in the Dragon protocol by suitably updating main memory on updates. Can we further reduce the states in the Dragon and/or Firefly protocols by merging the E and M states. What are the tradeoffs?

d. Instead of using writeback caches in all cases, it has been observed that processors sometimes write only one word in a cache line. To optimize for this case, a protocol has been proposed with the following characteristics:

* On the initial write of a line, the processor writes through to the bus, and accepts the line into the Reserved state.

* On a write for a line in the Reserved state, the line transitions to the dirty state, which uses writeback instead of writethrough.

(i) Draw the state transitions for this protocol, using the INVALID, VALID, RESERVED and MODIFIED states. Be sure that you show an arc for each of BusRead, BusWrite, ProcRead, and ProcWrite for each state. Indicate the action that the processor takes after a slash (e.g. BusWrite/WriteLine). Since both word and line sized writes are used, indicate FlushWord or FlushLine.

(ii) How does this protocol differ from the 4-state Illinois protocol?

(iii) Describe concisely why you think this protocol is not used on a system like the SGI Challenge.

e. On a snoopy bus based cache coherent multiprocessor, consider the case when a processor writes a line shared by many processors (thus invalidating their caches). If the line is subsequently reread by the other processors, each will miss on the line. Researchers have proposed a read-broadcast scheme, in which if one processor reads the line, all other processors with invalid copies of the line read it into their second level cache as well. Do you think this is a good protocol extension? Give at least two reasons to support your choice and at least one that argues the opposite.

5.9 Miss classification Classify the misses in the following reference stream from three processors into the categories shown in Figure 5-20 (follow the format in Table 5-4). Assume that each processor's cache consists of a *single* 4-word cache block.

Seq. No.	P1	P2	P3
1	st w0		st w7
2	ld w6	ld w2	
3		ld w7	
4	ld w2	ld w0	
5		st w2	
6	ld w2		
7	st w2	ld w5	ld w5
8	st w5		
9		ld w3	ld w7
10		ld w6	ld w2
11		ld w2	st w7
12	ld w7		

Seq. No.	P1	P2	P3
13	ld w2		
14		ld w5	
15			ld w2

5.10 Software implications and programming.

a. You are given a bus-based shared-memory machine. Assume that the processors have a cache block size of 32 bytes. Now consider the following simple loop:

```
for i = 0 to 16
for j = 0 to 255 {
A[j] = do_something(A[j]);
}
```

(i) Under what conditions would it be better to use a dynamically-scheduled loop?

(ii) Under what conditions would it be better to use a statically-scheduled loop?

(iii) For a dynamically-scheduled inner loop, how many iterations should a PE pick each time if A is an array of 4-byte integers?

b. You are writing an image processing code, where the image is represented as a 2-d array of pixels. The basic iteration in this computation looks like:

```
for i = 1 to 1024
for j = 1 to 1024
 newA[i,j] = (A[i,j-1]+A[i-1,j]+A[i,j+1]+A[i+1,j])/4;
```

Assume `A` is a matrix of 4-byte single precision floats stored in row-major order (i.e `A[i,j]` and `A[i,j+1]` are at consecutive addresses in memory). `A` starts at memory location 0. You are writing this code for a 32 processor machine. Each PE has a 32 Kbyte direct-mapped cache and the cache blocks are 64 bytes.

(i) You first try assigning 32 rows of the matrix to each processor in an interleaved way. What is the actual ratio of computation to bus traffic that you expect? State any assumptions.

(ii) Next you assign 32 contiguous rows of the matrix to each processor. Answer the same questions as above.

(iii) Finally, you use a contiguous assignment of columns instead of rows. Answer the same questions now.

(iv) Suppose the matrix `A` started at memory location 32 rather than address 0. If we use the same decomposition as in (iii), do you expect this to change the actual ratio of computation to traffic generated in the machine? If yes, will it increase or decrease and why? If not, why not?

c. Consider the following simplified N-body code using an O(N^2) algorithm. Estimate the number of misses per time-step in the steady-state. Restructure the code using techniques suggested in the chapter to increase spatial locality and reduce false-sharing. Try and make your restructuring robust with respect to number of processors and cache block size.

As a baseline, assume 16 processors, 1 Mbyte direct-mapped caches with a 64-byte block size. Estimate the misses for the restructured code. State all assumptions that you make.

```
typedef struct moltype {
      double x_pos, y_pos, z_pos; /* position components */
      double x_vel, y_vel, z_vel; /* velocity components */
      double x_f, y_f, z_f; /* force components */
      } molecule;

#define numMols 4096
#define numPEs 16
molecule mol[numMols]

main()
{
... declarations ...
      for (time=0; time < endTime; time++)
            for (i=myPID; i < numMols; i+=numPEs)
            {
                  for (j=0; j < numMols; j++)
                  {
                        x_f[i] += x_fn(reads position of mols i & j);
                        y_f[i] += y_fn(reads position of mols i & j);
                        z_f[i] += z_fn(reads position of mols i & j);
                  }
                  barrier(numPEs);
                  for (i=myPID; i < numMols; i += numPEs)
                  {
                        writes velocity and position components
                        of mol[i] based on force on mol[i];
                  }
                  barrier(numPEs);
            }
}
```