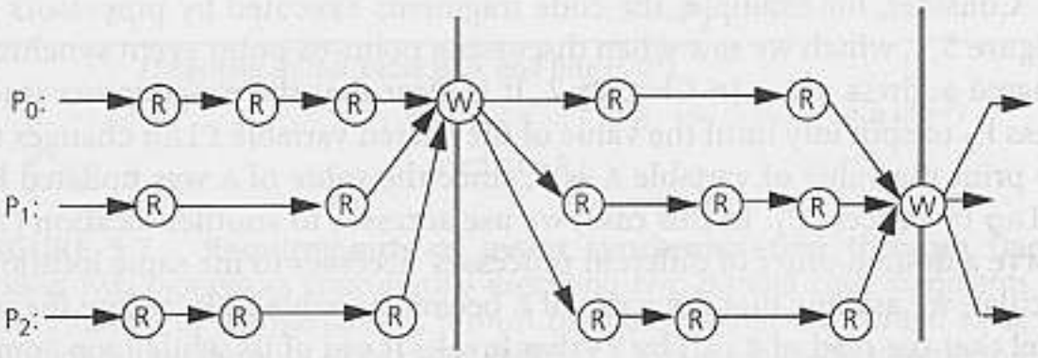

FIGURE 5.6 Partial order of memory operations for an execution with the write-through invalidation protocol. Write bus transactions define a global sequence of events between which individual processors read locations in program order. The execution is consistent with any total order obtained by interleaving the processor orders within each segment.

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 240 MB of data per second per processor. Even ignoring address and other information and ignoring read misses, a 1-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 ensure write propagation. Also, when writes to different caches are allowed to occur concurrently, no obvious ordering mechanism exists to sequence the writes. We will need somewhat more sophisticated cache coherence protocols to make the "critical" events visible to the other caches and to ensure write serialization.

The space of protocols for write-back caches is quite large. Before we examine it, 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.

5.2 MEMORY CONSISTENCY

Coherence, on which we have focused so far, is essential if information is to be transferred between processors by one writing to a location that the other reads. Eventually, the value written will become visible to the reader—indeed to all readers. However, coherence 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 memory location.

Consider, for example, the code fragments executed by processors P_1 and P_2 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 P_2 to spin idly until the value of the shared variable `flag` changes to 1 and then to print the value of variable `A` as 1, since the value of `A` was updated before that of `flag` by process P_1. In this case, we use accesses to another location (`flag`) to preserve a desired order of different processes' accesses to the same location (`A`). In particular, we assume that the write of `A` becomes visible to P_2 before the write to `flag` and that the read of `flag` by P_2 that breaks it out of its while loop completes before its read of `A` (a print operation is essentially a read). These program orders within P_1 and P_2's accesses to different locations are not implied by coherence, which, for example, only requires that the new value for `A` eventually become visible to process P_2, not necessarily before the new value of `flag` is observed.

The programmer might try to avoid this issue by using a barrier or other explicit event synchronization, 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. Even this approach has two potential problems, however. First, we are adding assumptions to the meaning of the barrier: not only do processes wait at the barrier until all of them have arrived, they also wait until all writes issued prior to the barrier have become visible to the 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 in the compiled code, not a special barrier operation. Coherence does not say anything at all about the order among these accesses.

Clearly, we expect more from a memory system than to "return the last value written" for each location. To establish order among accesses to the same 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 the same process. Coherence says nothing about the order in which writes to different locations become visible. Similarly, it says nothing about the order in which the reads issued to different locations by P_2 are performed with respect to P_1. Thus, coherence does not in itself prevent an answer of 0 from being printed by either example, which is certainly not what the programmer had in mind.

In other situations, the programmer's intention may not be so clear. Consider the example in Figure 5.9. The accesses made by process P_1 are ordinary writes, and `A` and `B` are not used as flags or synchronization variables. Should we intuitively expect that if the value printed for `B` is 2, then the value printed for `A` is 1? Whatever the answer, the two print statements read different locations and coherence says nothing about the order in which the writes by P_1 become visible to P_2. This example is in fact a fragment from Dekker's algorithm (Tanenbaum and Woodhull 1997) to determine which of two processes arrives first at a critical point as a step in ensuring mutual exclusion. The algorithm relies on writes to distinct locations by a process becoming visible to other processes in the order in which they appear in the

P_1	P_2
*/*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 =1` is visible to process P_2, then it must also be the case that `A = 1` is visible to P_2.

P_1	P_2
*/*Assume initial value of A is 0*/*	
`A = 1;`	`. . .`
`- - - BARRIER(b1)- - - - - -`	`BARRIER(b1)- - - - - - - -`
	`print A;`

FIGURE 5.8 Maintaining order among accesses to a location using explicit synchronization through barriers. As in Figure 5.7, the programmer expects the value printed for `A` to be 1 since passing the barrier should imply that the write of `A` by P_1 has already completed and is therefore visible to P_2.

P_1	P_2
*/*Assume initial values of A and B are 0*/*	
`(1a) A = 1;`	`(2a) print B;`
`(1b) B = 2;`	`(2b) print A;`

FIGURE 5.9 Order among accesses without synchronization. Here it is less clear what a programmer should expect since neither a flag nor any other explicit event synchronization is used.

program. Clearly, we need something more than coherence to give a shared address space a clear semantics, that is, an ordering model that programmers can use to reason about the possible results and hence the 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 in this sense memory consistency subsumes coherence.

5.2.1 Sequential Consistency

In the discussion in Chapter 1 of fundamental design issues for a communication architecture, Section 1.4 described informally a desirable ordering model for a shared address space: the reasoning that allows a multithreaded program to work 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 to processes in the shared address space in a manner that no sequential interleaving of accesses from different processes can generate. This intuitive model was formalized by Lamport as *sequential consistency* (SC), which is defined as follows (Lamport 1979):[1]

> 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 (Adve and Gharachorloo 1996). It is similar to the machine model we used to introduce coherence, though now it applies to multiple memory locations. 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 own private caches and buffers. Every process 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 from that process has completed. In addition, 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 should appear globally (to all processes) 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 for sequential consistency is that they appear to complete in a manner that satisfies the constraints just described. In the example in Figure 5.9, under SC the result (0,2) for (A,B) would not be allowed—preserving our intuition—since it would then appear that the writes of A and B by process P_1 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) are 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

1. Two closely related concepts in software systems are serializability (Papadimitriou 1979) for concurrent updates to a database and linearizability (Herlihy and Wing 1987) for concurrent objects.

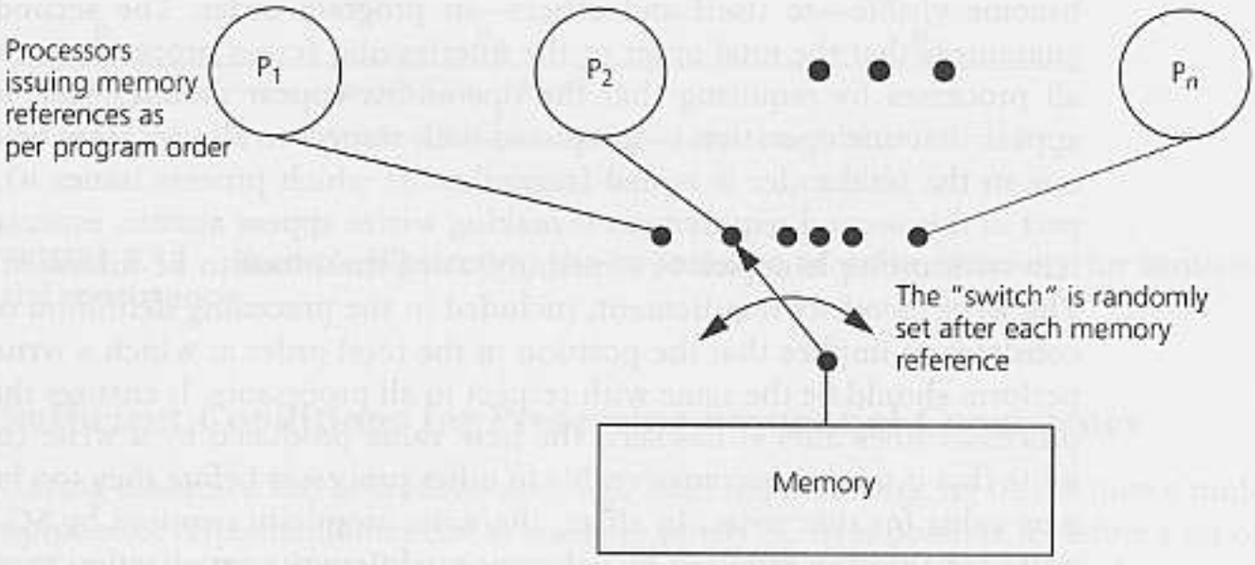

FIGURE 5.10 Programmer's abstraction of the memory subsystem under the sequential consistency model. The model completely hides the underlying concurrency in the memory system hardware (e.g., the possible existence of distributed main memory, the presence of caches and write buffers) from the programmer.

in Exercise 5.6. Note that SC does not obviate the need for synchronization. The reason is that SC allows operations from different processes to be interleaved arbitrarily and does so 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 constraints on 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 according to the source code that the process executes; more specifically, it is the order in which memory operations occur in the assembly code that results from a straightforward translation of source statements one by one to machine 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 preserving dependences to the same location). The programmer has in mind the order of statements in the source 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 parallel computer 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 language and the underlying system to the programmer.

Implementing SC requires that the system (software and hardware) preserve the intuitive constraints defined previously. There are really two constraints. The first is the program order requirement: memory operations of a process must appear to

become visible—to itself and others—in program order. The second constraint guarantees that the total order or the interleaving across processes is consistent for all processes by requiring that the operations appear atomic. That is, it should appear that one operation is completed with respect to all processes before the next one in the total order is issued (regardless of which process issues it). The tricky part of this second requirement is making writes appear atomic, especially in a system with multiple copies of a memory word that need to be informed on a write. The *write atomicity* requirement, included in the preceding definition of sequential consistency, 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 (e.g., another write that it issues) becomes visible to other processes before they too have seen the new value for that write. In effect, the write atomicity required by SC extends the write serialization required by coherence: while write serialization says that writes to the same location should appear to all processors to have occurred in the same order, write atomicity says that all writes (to any location) should appear to all processors to have occurred in the same order. Example 5.4 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 P_2 waits until A becomes 1 and then sets B to 1, and since P_3 waits until B becomes 1 and only then reads the value of A, from transitivity we would infer that P_3 should find the value of A to be 1. If P_2 is allowed to go on past the read of A and write B before it is guaranteed that P_3 has seen the new value of A, then P_3 may read the new value of B but read the old value of A (e.g., from its cache), violating our sequentially consistent intuition. ■

More formally, 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 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 therefore 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 the possible total orders (interleavings) as defined earlier. That is, a total order or interleaving of program orders from processes should exist that yields the same result as the actual execution.
- *Sequentially consistent system.* A system is sequentially consistent if any possible execution on that system is sequentially consistent.

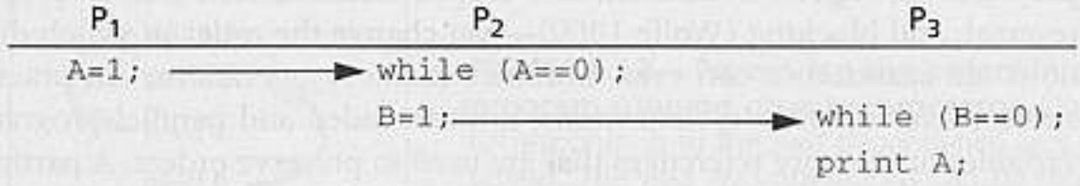

FIGURE 5.11 Example illustrating the importance of write atomicity for sequential consistency

5.2.2 Sufficient Conditions for Preserving Sequential Consistency

Having discussed the definitions and high-level requirements, let us see how a multiprocessor implementation can be made to satisfy SC. It is possible to define a set of sufficient conditions that will guarantee sequential consistency in a multiprocessor—whether bus-based or distributed, cache-coherent or not. The following set, adapted from its original form (Dubois, Scheurich, and Briggs 1986; Scheurich and Dubois 1987), is relatively simple:

1. Every process issues memory operations in program order.
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 is quite demanding. It is not a simple local constraint because the read must wait until the logically preceding write has become globally visible. Note that these are sufficient, rather than necessary, conditions. Sequential consistency can be preserved with less serialization in many situations, as we shall see.

With program order defined in terms of the source program, it is important that the compiler should not change the order of memory operations that it presents to the hardware (processor). Otherwise, sequential consistency from the programmer's perspective may be compromised even before the hardware gets involved. Unfortunately, many of the optimizations that are commonly employed in both compilers and processors violate these 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 that greatly improve performance—such as common subexpression elimination, constant

propagation, register allocation, and loop transformations like loop splitting, loop reversal, and blocking (Wolfe 1989)—can change the order in which different locations are accessed or can even eliminate memory operations.[2] In practice, to constrain these 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 in program order. Example 5.5 illustrates these issues.

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 (or multithreaded) 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` instead of just `integer`. Other solutions are also possible and are 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 (Hennessy and Patterson 1996). These allow memory operations from a process to issue, execute, and/or complete out of program order. Like compiler optimizations, these architectural optimizations work for sequential programs because the appearance of program order in these programs requires that dependences be preserved only among accesses to the same memory location, as shown in Figure 5.12. The problem in parallel programs is that the out-of-order processing of operations to different shared variables by a process can be detected by other processes.

Preserving the sufficient conditions for SC in multiprocessors is quite a strong requirement since it limits compiler reordering and out-of-order processing techniques. Several weaker consistency models have been proposed and techniques have been developed to satisfy SC while relaxing the sufficient conditions. We will examine these approaches in the context of scalable shared address space machines in Chapter 9. For the purposes of this chapter, we assume the compiler does not reorder memory operations, so the program order that the processor sees is the same as

2. Note that register allocation, performed by modern compilers to eliminate memory operations, 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 P_2, 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.

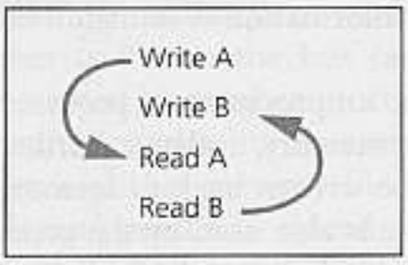

FIGURE 5.12 Preserving the orders in a sequential program running on a uniprocessor. Only the orders corresponding to the two dependence arcs must be preserved. The first two operations can be reordered without a problem, as can the last two or the middle two.

that seen by the programmer. On the hardware side, we assume that the sufficient conditions must be satisfied. To do this, we need mechanisms for a processor to detect completion of its writes so it may proceed past them (completion of reads is easy; a read completes when the data returns to the processor) and mechanisms to satisfy the condition that preserves write atomicity. For all the protocols and systems considered in this chapter, we 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.

For bus-based machines, the serialization imposed by transactions appearing on the shared bus is very useful in ordering memory operations. It is easy to verify that the two-state write-through invalidation protocol discussed previously actually provides sequential consistency—not just coherence—quite easily. The key observation to extend the arguments made for coherence in that system 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, thus ensuring write atomicity. When a write is performed with respect to any processor, all previous writes in bus order have completed.

5.3 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 applying a small amount of extra interpretation to events that naturally occur in the system. The processor is completely unchanged. No explicit coherence operations must be inserted in the program. By extending the requirements on the cache controller and exploiting the properties of the bus, the reads and writes that are inherent to the program are used implicitly to keep the caches coherent, and the serialization provided by the bus maintains consistency. Each cache controller observes and interprets the bus transactions generated by 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 processors to write to different blocks in their local caches concurrently without any bus transactions. Thus,