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

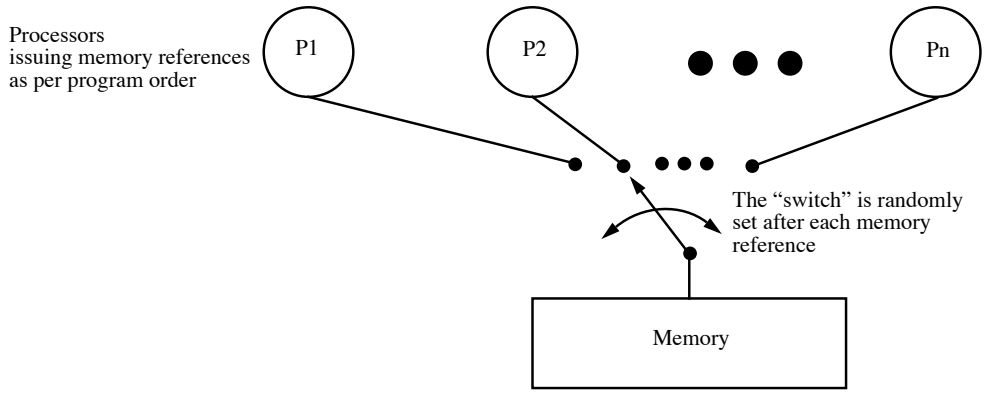

Figure 5-10 Programmer’s abstraction of the memory subsystem under the sequential consistency model.

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.

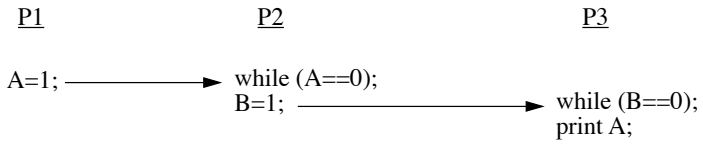

Figure 5-11 Example illustrating the importance of write atomicity for sequential consistency.

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.

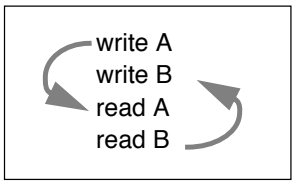

Figure 5-12 Preserving orders in a sequential program running on a uniprocessor.

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