Chapter 4 Multiprocessor Memory Systems

Memory systems are a prominent component of multiprocessors; to a large extent they define the multiprocessor system architecture. For example, a major division in multiprocessor architectures is between shared memory and distributed memory. Because of high demands of multiprocessors for both capacity and bandwidth, main memories are larger and more complex than in uniprocessor systems. Furthermore, bandwidth and capacity demands are constantly increasing. Not only have main memory systems become complex, but multiprocessor systems often have elaborate, multi-level cache hierarchies. In this chapter we will describe the memory hierarchy from the L1 caches to main memory. We will defer the I/O system, disks and networks, until we discuss multiprocessor systems in later chapters. We will focus on shared memory implementations, primarily. Distributed memory organizations employ a subset of the shared memory techniques.

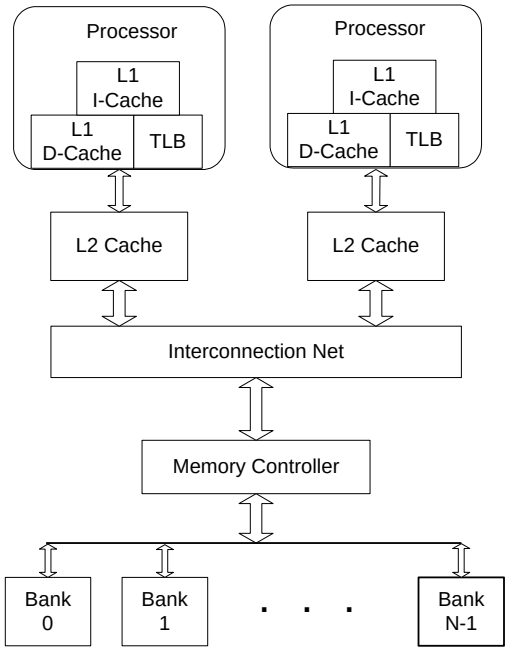


Figure 1. The major components of the memory hierarchy – extending from the L1 caches to main memory.

Figure 1 illustrates the major components of the memory system we will be discussing in this chapter. There is a cache hierarchy, consisting first of level 1 (L1) caches that are embedded in the processor core. In most processors, there is both an L1 instruction cache (I-Cache) and an L1 data cache (D-cache). There are also one or more translation lookaside buffers (TLBs) which translate virtual addresses to real addresses. The next level of the memory hierarchy is the L2 cache. These have become very commonplace in both client (desktop and laptop) and server systems. The L2 cache may be

shared among multiple processors, or each processor may have its own L2 cache, as shown in the figure. After the L2 cache, there may be an L3 cache; an L3 cache is more commonly found in server systems than in client systems. The lowest level of the cache hierarchy, in this case the L2 cache, is attached through an interconnection network, which may be a bus, or a more complex structure containing links and switches, to a memory controller. The memory controller drives the main memory which is composed of multiple, interleaved banks. The memory controller buffers memory requests, loads and stores, and presents them to the memory banks.

In this chapter, we will first proceed down the memory hierarchy, beginning with a description of cache memories and then main memory technology and organizations. This is followed by a detailed discussion of memory coherence implementations. Because the cache memories are such a central part of memory coherence, this is often referred to as *cache coherence*. Next is a discussion of memory ordering implementations, including both sequential consistency and some relaxed consistency models. Then, because of the considerable attention it is receiving for future multiprocessor systems, the implementation of transactional memory is discussed; in effect, this could be considered a relaxation of sequential consistency. Finally, there is a discussion of virtual memory, with emphasis on maintaining TLB coherence – a counterpart of cache coherence, and another aspect of a coherent memory implementation.

4.1 Cache Memories

Cache memories are an essential element of virtually every computer system built today. As such, they are usually covered in introductory texts on computer architecture, so we only survey them here. Later, in Section 4.3, we will discuss cache coherence in some detail. It is the coherence aspect of cache memories that is specific to multiprocessor systems.

Cache memories take advantage of *locality*. Temporal locality is present when a data item is accessed, then the same data item accessed again in the near future. Spatial locality is present when a data item is accessed, and a data item at a nearby memory address is accessed in the near future. Here "data" is used in a generic sense, it also includes instructions when they are held in a cache.

As was observed in Chapter 1, another basic aspect of cache design is that smaller memories are faster than large ones, so, coupling this fact with locality leads to caches which are small memories that hold data likely to be used in the near future. In effect caches implement a form of speculation; based on history, a prediction is made that the same, or nearby, data will be accessed again in the near future.

4.1.1 Fully Associative Caches

When memory data is to be placed in a cache memory, main memory is divided into equal sized *blocks*, or *lines*. To simplify addressing, memory lines typically are a power-of-two number of bytes in size. A memory line may consist of 32-256 bytes, for example. Cache memory is divided into *frames* that are of the same size as the memory lines. Then, the memory lines are moved in and out of the cache frames, depending on memory accessing patterns and implementation features of the cache memory, such as the line replacement algorithm. Hence, at any given time, a cache frame may be empty (is invalid) or it may hold a single memory line. During the time a memory line is residing in a cache frame, it is also referred to as a *cache line*.

Because the number of frames in a cache is much smaller than the number of memory lines, a cache holds a proper subset of the memory lines (in general, recently accessed lines). In a fully associative cache (Figure 2), there is a tag associated with each frame. This tag is the address of the memory line that is currently in the frame. To access a fully associative cache, a load or store instruction generates a memory address, and the tag bits of the memory address are compared with all the tags associated with

all the cache frames. If there is a match (called a *hit*), then the memory line is present in the corresponding cache frame and can be accessed immediately from the cache. If there is no tag match (a *miss*), then the requested data is not in the cache, and the addressed line must be accessed from memory (or the next lower cache if there is a multi-level cache hierarchy.

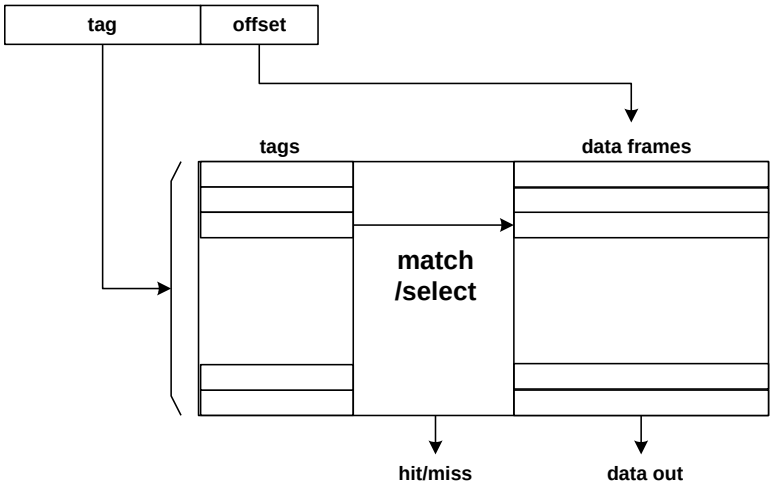


Figure 2. Fully Associative Cache Memory.

When the new line is brought from memory, it must be placed into one of the cache frames. Unless some frame is empty (invalid), this means that the line currently in one of the frames must be replaced. Consequently, there is a replacement algorithm which decides which line should be evicted to make room for the new line. Least Recently Used, FIFO, and Random are replacement algorithms that are commonly used. The replacement algorithm implemented in a particular design involves the tradeoff between hardware complexity and performance.

4.1.2 Direct Mapped Caches

In a fully associative cache, a given memory line can be in any of the cache frames. This means there is a large amount of tag comparison logic, because the tag of the memory address must be compared with the tags associated with every frame in the cache. To reduce the tag comparison logic substantially, a *direct mapped cache* can be used. In a direct mapped cache, a given memory line can only be placed in one particular frame (Figure 3). A memory address is divided into three fields, with the index field indicating which frame is the one that potentially holds the addressed line. Then, the tag field of a memory address is compared with only the tag for this one cache frame.

A disadvantage of a direct mapped cache, as compared with a fully associative cache, is that there may be more cache misses because there is much less flexibility regarding where lines may be placed. This can result in *localized thrashing*, where a number of memory lines compete for the same cache frame, even though other cache frames are not being actively used. On the other hand, the implementation of a direct mapped cache is cheaper (only one tag comparator is required), and it may save power and have a shorter access latency. Furthermore, there is no need for replacement algorithm hardware. A given line can only go in one place – there is no replacement decision to be made.

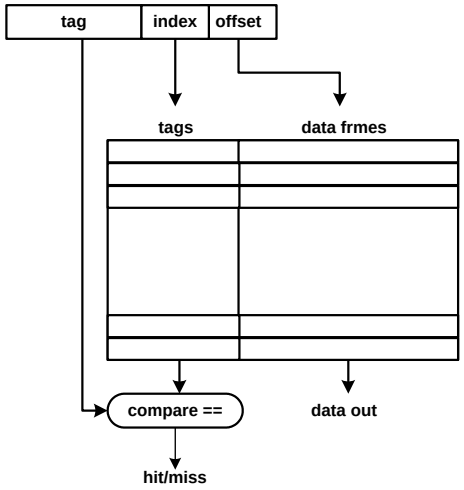


Figure 3. Direct Mapped Cache Memory.

4.1.3 Set Associative Caches

A set associative cache is a compromise between the fully associative and direct mapped cache. In a set associative cache, the frames are partitioned into a number of sets (usually a power of two). A set can hold two or more frames (if it holds a single frame, then the cache is direct-mapped). The index field of the address identifies the set containing the frames where the addressed memory line can potentially reside. Then, the tag of the address is compared against the tags for all the frames in the set to determine if there is a cache hit. Typically, set associative caches are 2-way associative to 8-way associative; the *way* is the number of frames in the set. The way does not have to be a power of two, and sometimes isn't. For example, one might design a 3-way or 5-way set associative cache – the determining factor is usually the total cache size.

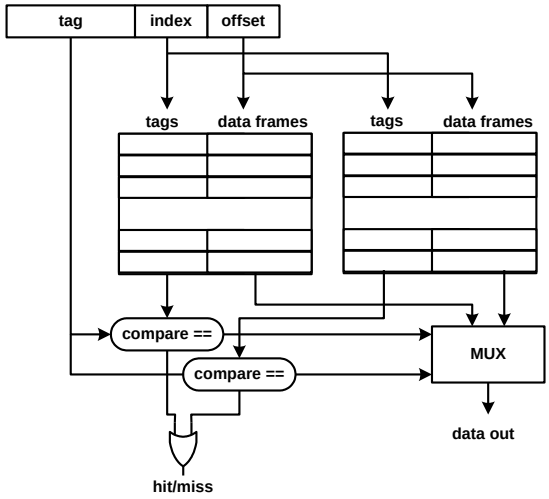


Figure 4. Set Associative Cache Memory.

4.1.4 Write Policies

Thus far, we have primarily been discussing caches in terms of performing load instructions (reading data from the cache). Of course, an instruction can also store to a cache line. When the address of a

store instruction hits in the cache, there are two possible actions that may be taken. The first is that the data in the cache line is updated (written to), and the next level in the memory hierarchy (for example main memory) is updated as well. This is a *write-through* policy; the Cache Read&M "passes through" to memory (or the next level of the memory hierarchy).

Alternatively, when a store instruction hits in the cache, only the cache line is written, with the next level of the memory hierarchy remaining unchanged. This is a *write-back* policy. With a write back policy, the copy of the data in the next level of the memory hierarchy becomes stale, and the copy of the line in the cache is referred to as being *dirty*. When a dirty line is later evicted from the cache, then any updated parts of the line (or, more commonly, the entire line) are written back to memory (or the next level of them memory hierarchy).

If a store instruction misses in the cache, then, again, there are two choices. One is simply to update the memory line in memory, and not allocate a frame in the cache for the line being written. The other policy is to allocate a cache frame (by evicting the line currently in the frame) and bring the line in from the next level of the memory hierarchy. Then the cache line is written into the allocated frame. These two policies are referred to as *write allocate* and *no allocate*.

The choice of write-back versus write-through and write-allocate versus no-allocate are independent choices, strictly speaking, and all four combinations of policies are possible. However, in practice, a write-through cache typically uses no-allocate policy and a write-back cache typically uses a write allocate policy.

An advantage of write-back caches is that in most cases less memory bandwidth is required; repeated writes to the same line do not require writes to memory. Furthermore, the combination of write-back and write-allocate policies means that the unit of data transferred between memory and the cache is always a complete line; individual bytes and words are seldom (if ever) transferred. Hence the data path from memory may be simplified somewhat.

4.1.5 Buffering

There are a number of buffers that are used in combination with a cache memory. One set of buffers includes the load/store buffers discussed in Chapter 3. Another buffer is the *write-back buffer*. A combination of a write-back and a cache line allocate (for either a read or write) may require two transfers to/from memory (if the line being evicted is dirty). If a load is being performed, then getting the new line into the cache is on the critical performance path. Hence, the sequence of events is as shown in Figure 5. First, the new line is accessed, and in parallel the dirty line to be evicted is read from the cache and placed into the write-back buffer. This makes room for the new line. After the new line has been transferred, then the dirty line can be written to memory from the write-back buffer.

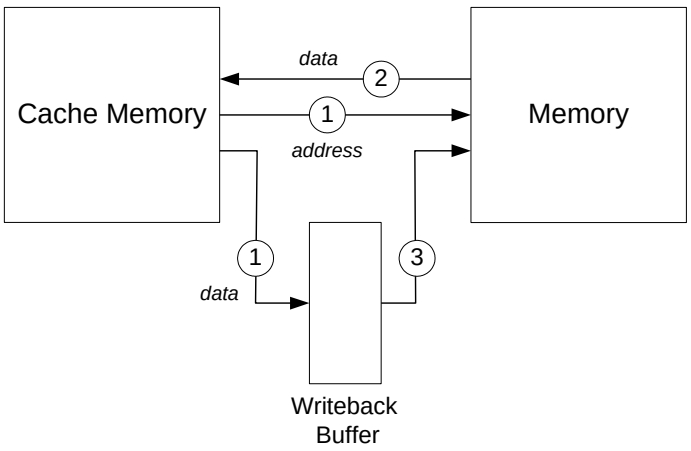


Figure 5. Operation of writeback buffer. First, the line is accessed from memory while the line is evicted from cache; second, the line from memory is written into the cache; third, the evicted line is written back to memory.

4.2 Main Memory Organization

In a modern ISA, memory addresses used by load and store instructions are *logical* or *virtual* addresses, which are translated into *real* addresses by address translation hardware. Our emphasis in this section will be on real memory systems because main memory hardware, as well as most, if not all of the cache hierarchy, is accessed via real addresses. Issues related to address translation are covered in Section 4.7.

The basic components of a main memory system are illustrated in Figure 6. The figure shows a single memory controller, but a multiprocessor system could have multiple memory controllers. A memory controller manages a set of interleaved memory banks. Memory interleaving is a technique fundamental to high performance main memory systems. Multiple banks with address interleaving provide a mechanism for achieving high memory bandwidth from DRAM (Dynamic Random Access Memory) chips which individually have relatively low bandwidth. If addresses are distributed among the memory banks, then the banks can be accessed in parallel with their access latencies being overlapped (see Figure 7).

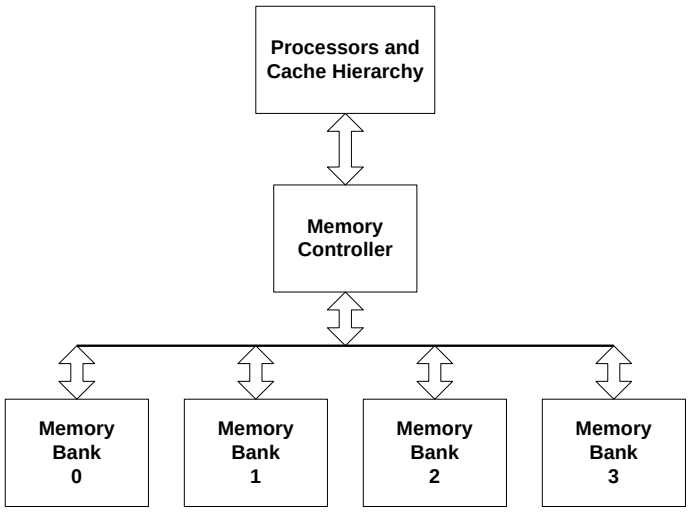


Figure 6. The basic components of memory systems are memory controllers and interleaved memory banks.

Because the distribution of memory addresses is a statistical property of a program, memory banks are usually interleaved on low order address bits so that the probability of a bank conflict (addressing a bank that is already in use) is reduced. For example, in Figure 7, there are four memory banks (0 through 3) and the low order bits of the addresses in the example address stream are 0, 3, 2, 1. Because these four addresses are to different memory banks, the memory accesses can be almost completely overlapped.

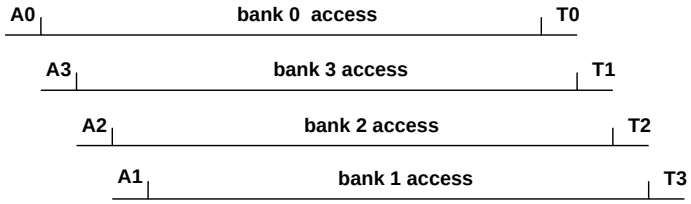


Figure 7. Example of an address stream accessing an interleaved memory.

In this section, we first describe DRAM technology, and then turn to main memory organization, including interleaved (multi-bank) memories and memory controllers. Most of the concepts in this section are similar for uniprocessors and multiprocessors. The underlying DRAM chips are the same, for example; however, memory system organizations are often of a larger scale in multiprocessors because multiprocessor systems tend to have both larger memories and greater bandwidth demands.

4.2.1 DRAM Technology

DRAMs have been used for main memories for many years. They exploit different technology tradeoffs than microprocessors; i.e., they are designed for density rather than speed. Consequently, DRAMs have not kept up with processors with respect to access speed, thereby increasing the importance of cache hierarchies and multi-bank main memories. Over the years there have been a series of innovations in DRAMs in order to keep the performance gap between processors and memories at least manageable. We will first cover the operation of basic Asynchronous DRAMs and then describe important enhancements that have led to today's Double Data Rate Synchronous DRAMs (DDR SDRAMS).

ASYNCHRONOUS DRAMS

Traditional asynchronous DRAMs are illustrated in Figure 8. They have a time-multiplexed address bus that alternately provides row and column addresses, a bi-directional data bus, and control signals. These buses and signals are illustrated in Figure 8, although not all the control signals are not shown. The example DRAM in Figure 8 contains a total of 256 Mbits in what is called a 16M x 16 configuration. The data bus width is 16 bits and the total size is 256M; so, logically speaking it appears to contain 16M locations, each 16 bits wide. Physically, however, the storage array in the example contains 16K rows and 16K columns. A DRAM access first reads an entire row of 16K bits, and then selects the 16 output data bits beginning at an addressed column within the row.

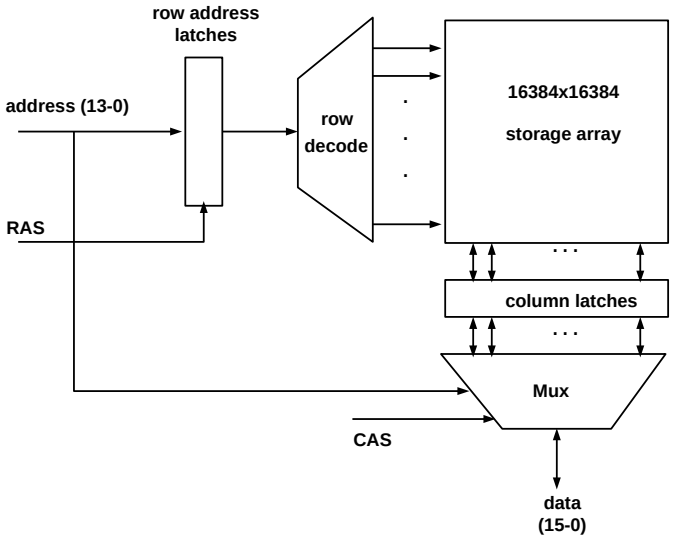


Figure 8. Diagram of an asyncronous DRAM chip.

The signal sequence for reading an asynchronous DRAM is shown in Figure 9. First, the row address is placed on the address lines and the *row address select* or *row access strobe* (RAS) control signal is activated. This causes the row to be accessed from the data array. Then, the column address is placed on the address lines and the *column address select* (CAS) signal is activated; this causes the addressed bits to be placed on the output data pins. The column address bits select the bits to be placed on the data lines. The elapsed time to input the address and read out the data is the *access time*.

The row access of a DRAM is destructive, that is, the process of reading the data causes it to be destroyed. Hence, after a read operation, the row data must be written back into the memory array. Also, the row access lines are pre-charged in order to make the next access faster. This means that after an access is performed there is a delay before another access can be made. The total time between two consecutive accesses is the *cycle time*. The cycle time, roughly speaking, is about twice the access time.

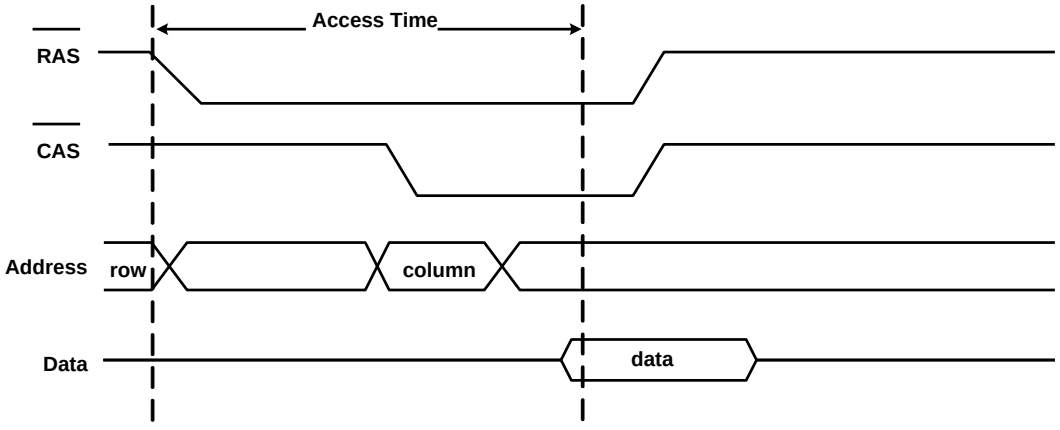


Figure 9. Sequence for reading data from an asynchronous DRAM.

The example RAM chip shown in Figure 8 has a 16 bit wide data path, although DRAM chips can be of other widths (a power of two). To form a memory with datapath wider than that of a single DRAM,

a number of DRAM chips are placed in parallel. For example, to build a 64-bit wide memory, four of the x16 chips are used. Combining chips in this way leads to today's DIMM (Dual Inline Memory Module) memory packaging. A DIMM contains a number of DRAM chips that collectively provide a memory of a given width, e.g., 64-bits.

At their core, today's DRAMs still look like the basic DRAMs just described, however they have additional, surrounding circuitry that provides a synchronous interface capable of transmitting data on both clock edges, and they contain multiple, internal memory banks. These features are discussed in the next subsection.

MODERN DOUBLE DATA RATE SYNCHRONOUS DRAMS (DDR SDRAMS)

Modern DRAMs (Figure 10) contain a number of innovations to increase memory bandwidth and reduce access delay. One bandwidth improvement follows from the observation that the on-chip bandwidth is very high; in the asynchronous DRAM of Figure 8, 16K bits are read from the memory array in parallel. However, this is reduced to only 16 bits at the chip's pins. Hence, one way of improving bandwidth is to access the row once and then read from multiple columns within the row. First, the contents of a row are latched, then a sequence of columns can be read quickly, thereby amortizing the cost of the relatively slow, but wide, memory array access.

A second performance enhancer is the division of the data array into multiple banks, each with its own latched row buffer. By using multiple memory arrays, the individual arrays are smaller than would be the case with one large array, so wire delays (both row and column) are reduced, thereby reducing the array access time. Furthermore, multiple, interleaved banks have their usual advantage: they increase bandwidth by allowing overlapped data accesses (Figure 7). Finally, once a row has been read, it can be maintained in the row buffer indefinitely, and, if the next access should be to the same row, then the access time is reduced. In essence, the row buffers can act as a cache memory on the DRAM chip.

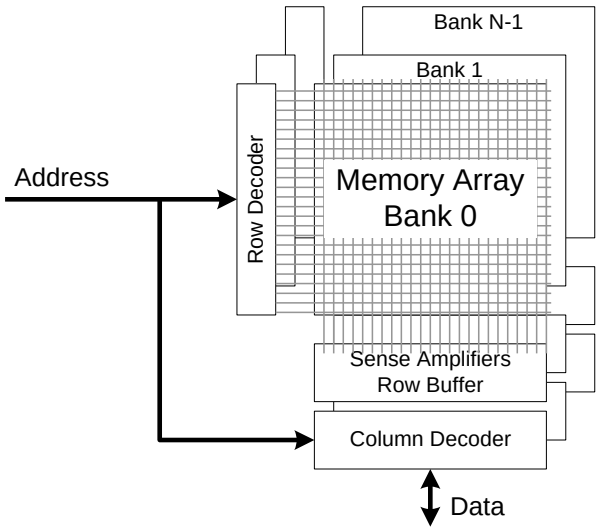


Figure 10. The organization of a DDR SDRAM.

A third performance enhancer, to expedite the transmission of data off the memory chip, is a synchronous chip interface (to avoid the round-trip delays of asynchronous signaling). Off-chip bandwidth is increased even more by transmitting data on both clock edges, that is, at what is called *double data rate*.

Figure 11 illustrates a DDR SDRAM read access. To simplify the description of DDR SDRAM operation, abstraction is used, so that combinations of control form a single *command*. A command is directed at a memory bank. An *activate* command reads an addressed row from the bank's memory array and places it into the row buffer, thereby *opening* the row. Once a row is open, any number of *read* and *write* commands can be issued to transfer data into and out of the row. Figure 11 shows a read command to an open row. After the read command is shown, there is a fixed, on-chip CAS delay before data is ready; in Figure 11 this is two cycles. Then, data is transferred on both clock edges. A *precharge* command closes a row by writing it back to the memory array and precharging the bank for the next row activation.

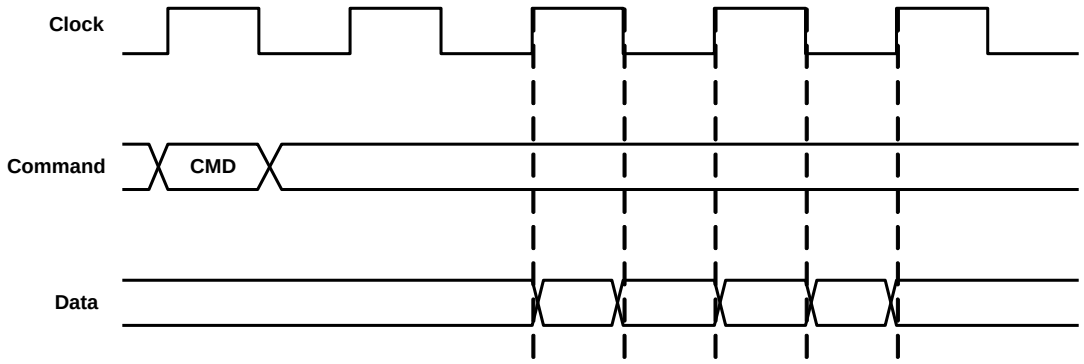


Figure 11. Timing for a read access from a DDR SDRAM; address lines are not shown. CMD is a collection of control signals.

A state diagram for control of a SDRAM bank is shown in Figure 12. A bank can be in the idle state where the bank is pre-charged and ready for a new row access. A row access is invoked by a row activation command. After a number of cycles (depending on the particular SDRAM), the row is open or active. Then, a column access command will read data from the row. This can be done repeatedly. A bank precharge command will return the bank to the idle state.

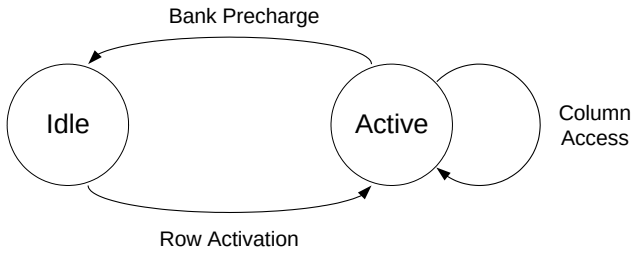


Figure 12. State diagram for controlling a memory bank in a SDRAM.

4.2.2 Memory Systems

As mentioned above, a number of DRAM chips are combined in a DIMM to form a wider memory module that can provide 32, 64, or 128 bits per access. For increasing total memory capacity, multiple DIMMs can be combined; when this is done, each DIMM forms a *rank* of memory. A memory system with three ranks (DIMMs) is shown in Figure 13. If each of the DRAMs is 16 bits wide, then the four chips in the DIMM provide a 64-bit data path. And, if each of the DRAM chips has four banks, then

they form a four-way interleaved memory system. In this system, the highest order address bits are decoded in the memory controller to select the rank (DIMM). The other bits address bits are shared by all the DRAM chips.

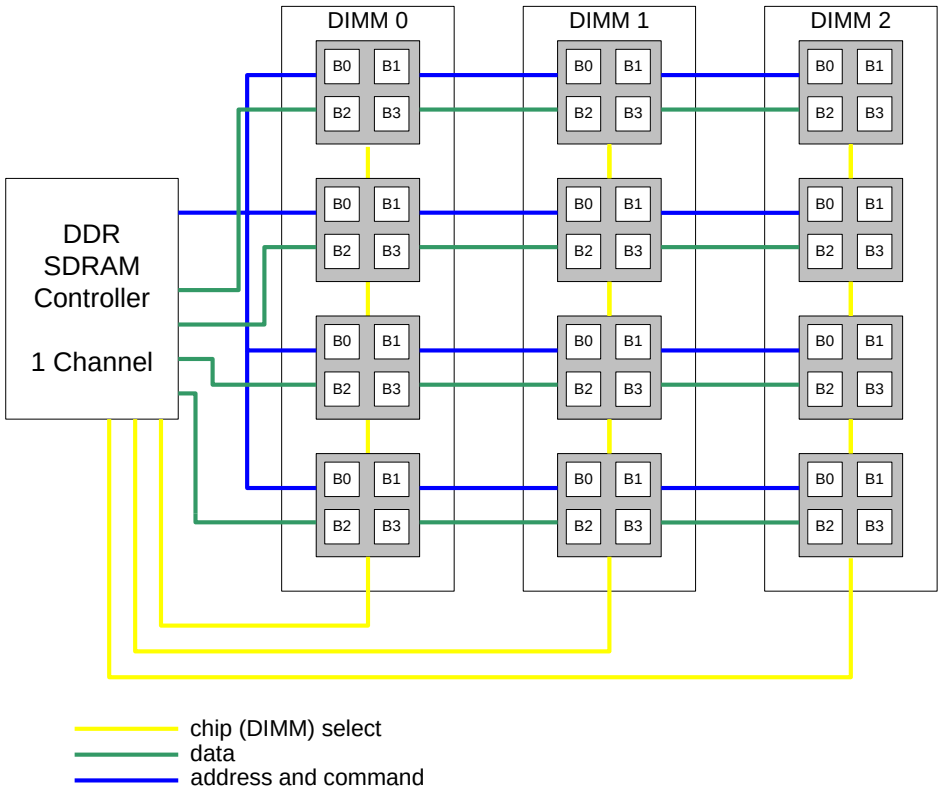


Figure 13. A memory system consisting of three ranks (DIMMs). The memory contains four banks per DIMM.

For purposes of addressing the memory system, one way of partitioning a real memory address is as follows: rank, row, bank, column [byte]. Beginning with low order address bits, the byte bits (three of them in a 64-bit wide memory) ordinarily don't go out to the memory banks. Main memory systems usually operate at a coarser granularity than individual bytes (although one could design a memory system where individual bytes can be accessed). The next address bits are the column bits. As addresses are incremented to sequence through consecutive addresses, the column bits change first, sweeping through a given row. The next higher order address bits identify different banks. This enables all the rows holding a larger block of contiguous data to be open simultaneously. This is also in keeping with the viewpoint that that the row buffers serve as a direct-mapped cache of sorts. Then, the rows are addressed by higher order bits, and the highest bits identify the rank.

Another approach is to interleave on the rank bits before the row bits. This will further increase the size of the contiguous block of data that can be held in row buffers. In the presence of spatial locality, this will likely increase the row hit rate. However, this would mean that the number of ranks should be a power of two (or there will be unpopulated "holes" scattered throughout real memory). This power-of-two rank feature would slightly reduce the flexibility with which a memory system can be expanded.

Overall, one such set of DIMMs that is served by a memory controller and channel is an interleaved memory with as many banks as supported by the SDRAM chips. In a full system, there can be multi-

ple memory channels, potentially supporting a large number of interleaved banks. One could design a multi-channel memory system in what is called a *dance-hall* organization (Figure 14). However, this makes the memory controller a centralized resource with relatively high bandwidth requirements. A more common organization today is to support one or two channels with a memory controller on each processor (CMP) chip. This way, memory control is de-centralized and the memory controllers can be integrated onto processor chips. Then, as larger systems are constructed, the number of controllers (and memory bandwidth) expands linearly with the number of processors. Figure 15 shows such a multi-chip system with four CMP chips that supports four memory channels. This system architecture leads to a degree of non-uniformity of memory access, but the non-uniformity is acceptably small.

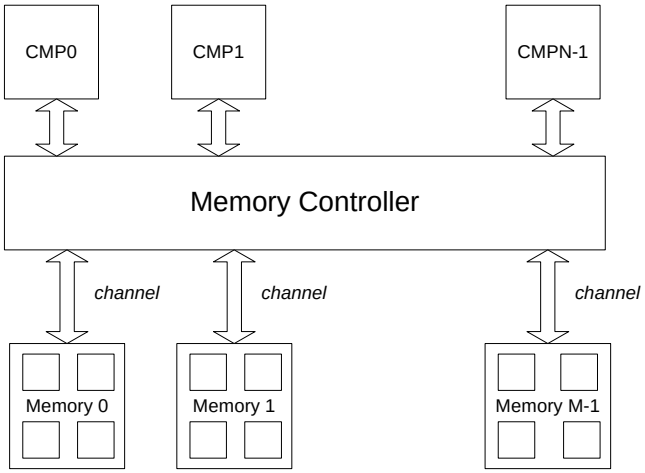


Figure 14. Multiprocessor system with CMP chips and memory modules arranged in a dance-hall configuration.

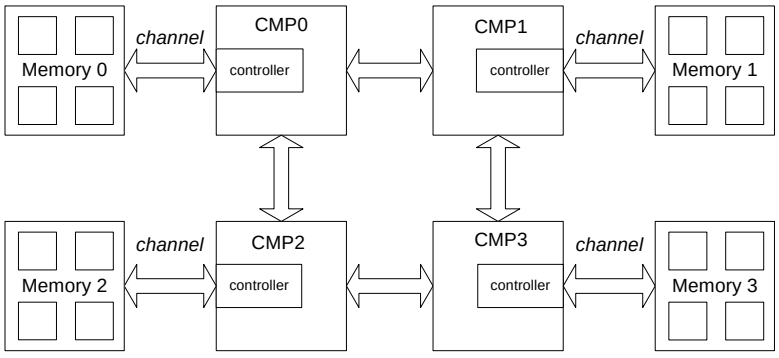


Figure 15. Multiprocessor system with integrated memory controllers and memory sharing via interconnected CMPs.

4.2.3 Memory Controllers

A *memory controller* provides the interface between the cache hierarchy and main memory. More specifically, a memory controller translates memory read and write requests into sequences of SDRAM commands. The read and write requests are usually in response to cache memory activity (misses and Cache Write-backs) or prefetch requests. Figure 16 illustrates the basic structure of a high-performance

memory controller. It consists of a memory scheduler, transaction buffer, a write buffer, and a read buffer. The transaction buffer holds the state of each memory request, e.g., the request type and request identifier. The write buffer temporarily holds cache lines being written to memory, and the read buffer temporarily holds cache lines read from memory while they are in transit to the requesting cache memory.

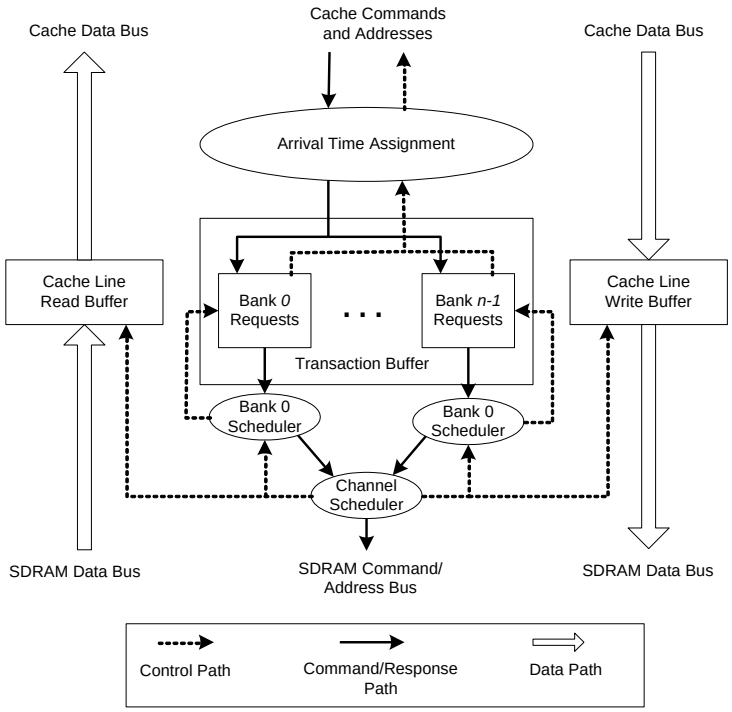


Figure 16. A memory controller.

The *memory scheduler* (shown in the center of Figure 16) is the core of the memory controller. It keeps track of the state of the SDRAM memory banks (Figure 12), and it reorders and interleaves memory requests to optimize memory latency and memory bandwidth utilization. Optimal scheduling is fairly complex because of timing differences in accessing data from a memory bank row that is already open versus data from a bank with a different open row. In the former case, it is only necessary to access addressed column(s) from the open row. In the latter case, the row must first be closed (a bank precharge in Figure 12), then the accessed row must be activated, and only then can the data access be performed. A third, intermediate case occurs if the bank is precharged, but there is no open row. These cases are sometimes referred to as a *row hit* (the addressed row is already in a row buffer), a *row miss* (a different row is in the row buffer), and an *empty row* (there is no row in the row buffer and the memory is pre-charged.) In order of increasing access latency, a row hit is fastest, an empty row is second, and a row miss is slowest.

Figure 17 illustrates the type of optimization that a memory scheduler might perform; this example is adapted from Rixner et al. [22]. Memory accesses are represented as a triple (bank, row, column). In this example, it is assumed that the bank precharge and row activation operations take three memory cycles, and the column access takes one memory cycle. Down the left side is a sequence of memory addresses as they are received by the memory controller. The first access (0,0,0) is to bank 0, row, 0, column 0; this access starts at time 0; it is assumed that bank 0 has a row miss. The sequence of

precharge, activate, and column access takes a total of seven cycles. Because the next three accesses are to bank 0, they must all wait for the first access to complete before they can start. Consequently, the fifth access, to bank 1 can begin its access at cycle 1. Now, as noted above, the second access is also to bank 0, but it is to row 1. Meanwhile, the third access, also to bank 0, is to row 0; consequently, the memory controller lets the third access pass the second, so it can take advantage of the already-open row 0. The third access then only needs to perform a column access.

At cycle 8, the memory controller has the choice of letting either the second or fifth access use the memory channel's data bus (the data bus can only be used by one access at a time). Here, the controller opts for letting the second access have the bus. This will then let bank 0 cycle (precharge and activate) so that the second access can be serviced. The remaining cycles are similarly serviced out-of-order in such a way that the overall time is 20 cycles total. In contrast, if these accesses are serviced in a strictly FIFO fashion, then 56 cycles are required.

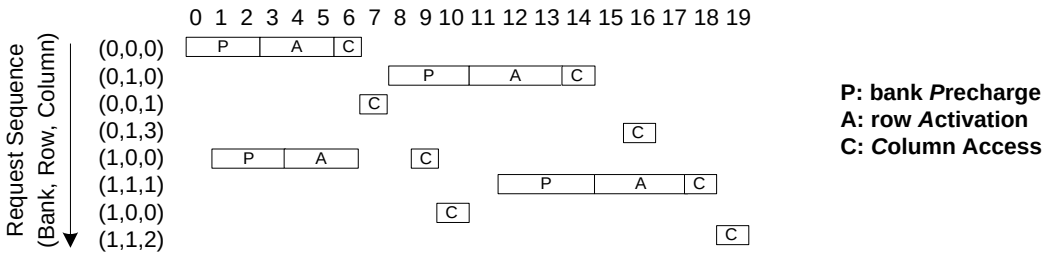


Figure 17. Example of optimal memory access scheduling.

A commonly used bank scheduling algorithm is First-Ready First-Come-First-Served (FR-FCFS) . With FR-FCFS, the first access that is ready is given priority; if there are multiple ready accesses then the first to arrive is given priority. The example in Figure 17 uses FR-FCFS. Actually, there is a family of FR-FCFS policies [22]. These policies can also prioritize according to types of commands. For example, Rixner et al. [22] show that a good FR-FCFS policy has three priority levels: 1) prioritize ready commands over commands that are not ready, 2) prioritize ready CAS commands over ready RAS commands, and 3) prioritize the command with the earliest arrival time (i.e. the time memory request arrived at the memory controller). This priority would let the fifth access (1,0,0) go ahead of the second (0,1,0) in the above example; if this is done, the total time is reduced by one cycle. Prioritizing first-ready CAS commands exploits already open rows and is essential for utilizing the data bus bandwidth efficiently. In addition to these priorities, a controller may also prioritize reads over writes.

To implement a scheduling policy, the memory scheduler has a logical priority queue and a bank scheduler for each SDRAM bank in the memory system. These queues can be implemented as a single hardware structure, although Figure 16 shows them as logically separate structures. The bank scheduler selects the pending request with the highest priority and generates a sequence of SDRAM commands to read (write) the request's data from (to) memory. The bank scheduler also tracks the bank's timing constraints to ensure that the sequence of SDRAM commands conforms to the SDRAM's specification. When an SDRAM command is ready (with respect to the bank's timing constraints), the bank scheduler sends the command to the channel scheduler.

The channel scheduler scans the banks' ready commands and issues the command with the highest priority. When a command is issued, the channel scheduler acknowledges the appropriate bank scheduler, and the bank scheduler updates its bank state machine appropriately. The channel scheduler also

tracks the state of the address bus, data bus, and ranks to ensure there are no channel scheduling conflicts and that no rank timing constraints are violated.

Another policy consideration is what to do when a bank has finished all accesses and there are no additional accesses in the bank's queue. One option is to always leave the row in its open state. This is an *open row* policy. The other option is to close the row by pre-charging it so that it is ready for the next access; this is a *closed row* policy. If a bank is left in the open (active) state after all pending accesses have been completed, and the memory controller later accesses data in the same row, then only a column access is required. On the other hand, if a different row is accessed, then there must be a bank precharge command followed by a row activation command before the data can be accessed. In the example of Figure 17, we implicitly assumed an open row policy (all the banks needed to be precharged as part of the first access). With a closed row policy, the bank is closed by precharging it immediately after all pending accesses are complete, then, when the next access takes place, a precharge is not needed. Only a row activation will be required, leading to a shorter access latency if it is to a different row. On the other hand, if the access is to the same row, then an opportunity for a fast column access was lost.

Whether an open row or closed row policy is better clearly depends on the likelihood that the next access to a bank will be to the same row as the immediately preceding access. Generally speaking if only a single thread is accessing the memory (as would be the case in some desktop client computers), then there is good likelihood that this will be the case due to spatial locality. On the other hand, if the memory is servicing multiple, concurrent threads as would be the case in a server, then a closed row policy is likely to perform better. This issue, as well as a number of others, was studied by Natarajan et al. [18] for a server environment.

4.3 Memory (Cache) Coherence

Memory coherence was discussed in Chapter 2 as an architecture feature. In this section, we discuss the implementation of memory coherence. Caches are such a large part of the memory coherence problem, that "cache coherence" is often used synonymously "memory coherence". Because a memory line potentially may be present in a number of caches at the same time, and updates (writes) may be performed on the cached copies, memory incoherence is an obvious hazard that must be avoided. In this section we will discuss the cache coherence problem and a number of solutions.

Let's begin with a simple example that illustrates the nature of the cache coherence problem. Consider the sequence illustrated in Figure 18. Initially, the variable A in memory has a value of zero. In Figure 18a both thread 0 (on processor 0) and thread 1 read the variable A so that copies of the line containing A are copied into their caches. Then, after some intermediate computation, process 0 writes variable A with a 1 (Figure 18b). Assuming caches implement a write-through policy, then the value of 1 is also propagated to the copy of A in main memory. Finally, in Figure 18c, thread 1 reads A; it gets a hit in its cache and reads a 0. At this point, 0 is a "stale" value, and thread 1 does not read the most-up-to-date copy. Memory coherence has been violated.

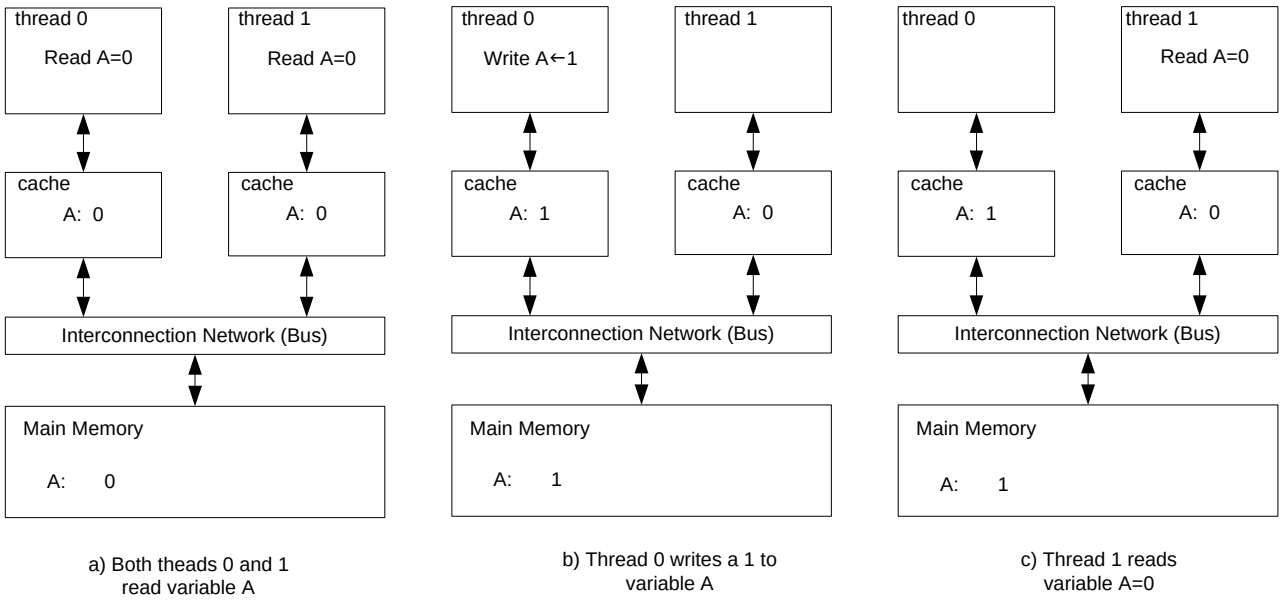


Figure 18. Example where memory coherence is violated.

There are a number of ways that cache incoherence can occur in a computer system. One way is thread communication through shared memory as in the example just given. Thread migration is another; this occurs when the OS schedules a thread on a different processor than it previously ran. If some of its data was in the previous processor's cache, then incoherence can occur. In this case, writeback caches may make incoherence more likely. Finally, I/O can cause incoherence when the I/O system write to memory, making cached copies of the same memory locations stale. In some systems, the I/O coherence problem is solved by an OS-invoked cache flush preceding and/or following an I/O operation. However, with very large caches, a full cache flush can lead to significant performance loss in the presence of an I/O intensive program. Hence, a solution is to design coherence hardware that includes the I/O path along with the processors in the system.

There are two basic policies for maintaining cache coherence, one involves avoidance of stale copies by invalidating them before they can become stale, and the other involves updating copies so that they do not become stale. These two policies are illustrated in Figure 19. These examples use the same sequence of reading and writing variable A as in Figure 18. In an invalidate protocol (Figure 19a), the copy in thread 1's cache is invalidated at the time thread 0 writes a 1 to A. The copy in main memory may either be updated (if the caches are write-through) or it will become stale (if the caches are write-back). Then, when thread 1 attempts to read A, it misses in its cache, and, depending on the protocol, it will either read the updated copy from memory or it will acquire the up-to-date copy from thread 0's cache. These alternatives will be discussed later. In an update protocol (Figure 19b), thread 0's write of a 1 to variable A causes the copy in process 1's cache to be updated immediately. Consequently, when thread 1 reads A, it hits in its cache and gets the updated version.

Both update and invalidate protocols have been used in practice, although invalidate protocols tend to be more common. Which is used depends on anticipated sharing patterns and other implementation-specific design considerations. These will be discussed further as the various protocols are described in more detail.

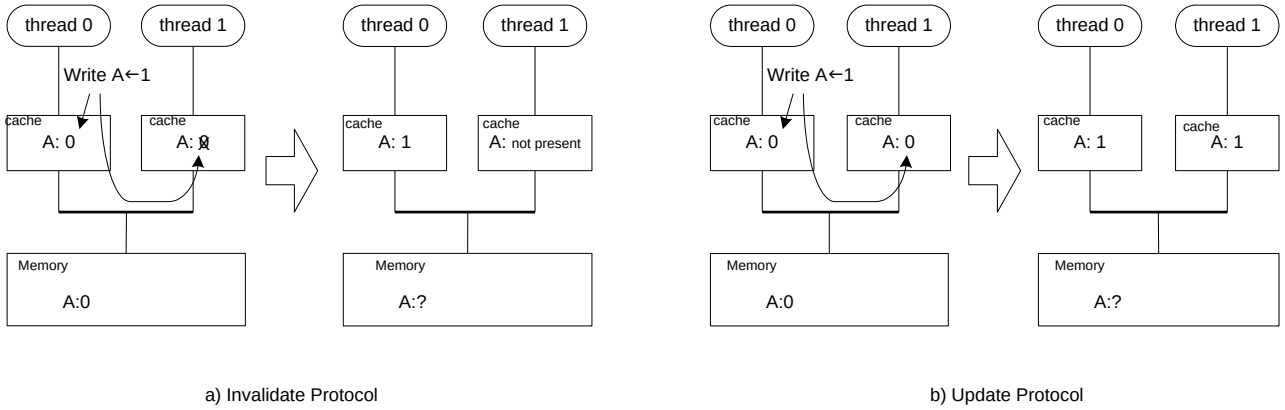


Figure 19. Examples of a) Invalidate coherence protocols and b) Update coherence protocols.

In a system with a main memory and caches, there can be many copies of a given memory line. We refer to a memory line as being *valid* if it is known to be an up-to-date copy; otherwise the copy is invalid. In a system with *n* caches, we can represent the overall state of a given memory line with an *n+1* element vector (numbered 0 to *n*). Element $0 \le i \le n-1$ is the state (valid or invalid) of the memory line in cache *i*. Element *n* (the last element of the *n+1* element vector) is the state of the copy in main memory. For example, if caches 0 and 3 have a valid copy, and memory has a valid copy, then the state vector is: <1,0,0,1,0,… , 1>.

In the coherence protocols we consider, the local caches and/or main memory contain logic that collectively tracks this state in order to make sure that valid copies are always used. Individual caches and memory may only have an incomplete summary of the global state, but collectively, the summaries contain enough state information to make valid coherence decisions. Furthermore, all of the summaries contain a consistent view of the global state.

We will refer to the state summary available to a given cache or memory as the *local state* (or simply "state" when it is unambiguous). The local state indicates the validity of that cache's local copy of the line as well as some additional information regarding the state of other copies. Definitions and examples of local state that are used in a number of coherence protocols follow. Without loss of generality, we assume that cache 0 is the *local* cache, i.e., the one upon which we are focusing. An X in one of the state vectors indicates that the value may be either a 0 or a 1.

Invalid (I): <0,X,X,X....X> -- the local cache does not have a valid copy; if the line is accessed, the local cache will signal a miss.

Shared (S): <1,X,X,X,…,1> -- the local cache has a valid copy, main memory has a valid copy, and other caches may or may not have valid copies.

Modified(M): <1,0,0,..0,…0> -- the local cache has the only valid copy.

Exclusive(E): <1,0,0,..0,…1> -- the local cache has a valid copy of the line, no other caches do, and main memory has a valid copy.

Owned(O): <1,X,X,X,….X> -- the local cache has a valid copy, all the other caches and memory may or may not have a valid copy. When in the O state, all the other caches must be either in the I state or the S state. Note that <1,X,1,X,… 0> is a state that is included in O, but is not included in any of the others.

The local cache maintains state information (a few bits) for each line frame. For all of the states except I, the state bits hold the state for the line currently in the frame. For the I state, however, there is no frame holding the line; in fact, it is the absence of the cache line from any frame that indicates that the line's local state is I.

What a coherence protocol does is this: when a memory load or store instruction is issued to the cache, the local cache controller determines the local state of the line. Then, for certain local states, the controller may have to force a global state change and get a valid copy of the line if it does not already have one. This implies an interconnection network connecting the caches and memory over which state change requests, responses, and line copies can be transmitted.

Specific cache coherence algorithms are very closely associated with the memory interconnection structure. The reason is that memory coherence is equivalent to sequential consistency for a single memory variable as was pointed out in Chapter 2. So, conceptually, most memory coherence implementations are based on an ordering point model as shown for SC in Figure 19 of Chapter 2. For a given memory location, there can be a number of potential ordering points, or even virtual ordering points. The ordering point places the state changes (and related cache accesses) into a well-ordered time sequence. The interconnection structure and the ordering point influence the coherence algorithm significantly. One straightforward possibility is to use a bus to order state changes. Another possibility is to use an ordering point at the memory controller, and doing so enables a wide variety of interconnection structures besides a bus. In following subsections, we will discuss both classes of coherence algorithms, beginning with bus-based coherence algorithms.

4.3.1 Bus-Based Coherence

In bus-based coherence, a shared bus is used as the ordering point for implementing cache coherence. Because all accesses must arbitrate for the bus, and can only use it one at a time, it is a natural ordering point. However, for implementing cache coherence a bus is more than just an ordering point. In particular, a shared bus lets the caches see what all the other caches are doing on the bus. That is, when one cache communicates with memory, the other caches can "snoop" on the bus transaction that is taking place. Hence, bus based coherence protocols are often referred to as *snooping protocols*.

In a snooping coherence protocol, each cache keeps track of the local state for each of the memory lines: this is a summary of the global cache line state as described above. The cache frames each contain a small field for keeping track of the local state. When a memory load or store instruction is issued to the cache, the cache controller (Figure 20) uses the frame's state fields to determine the local state. Then, based on its local state, the cache controller will either 1) allow the requested operation to be performed immediately, or 2) it will first have to invoke a global state change and get a valid copy of the line if it does not already have one.

In bus-based coherence, the cache controller does this by gaining access to the bus and sending a request that informs the other cache controllers and memory controller what it would like to do. Then, they may do one or more of the following 1) respond with information regarding their state 2) modify their state, 3) supply a valid copy of the line. Following the response, the local cache controller performs the requested memory operation and may change its local state. In a bus-based protocol, main memory doesn't hold any explicit state information (the summaries in the cache controllers are sufficient), but, depending on how the cache controllers respond, the memory controller may supply a valid copy of the requested line. In summary, as illustrated in Figure 20, a cache controller responds to both processor initiated actions and actions initiated by other, remote cache controllers via the bus (bus actions).

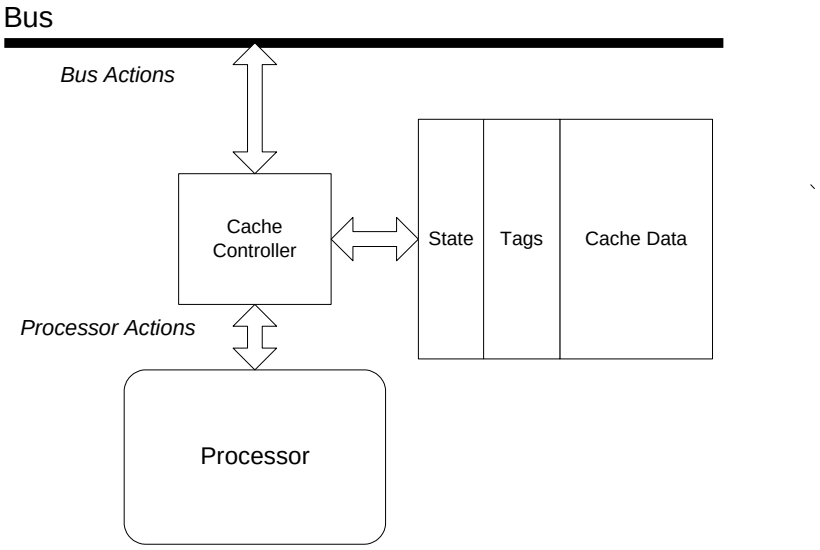


Figure 20. A cache controller monitors actions on the bus and actions made by its local processor. Based on what it observes, the cache controller may initiate bus actions of its own and/or modify the state information associated with the cache frames.

THE MSI COHERENCE PROTOCOL

There are a number of coherence protocols of varying complexity and performance optimality. These protocols have different numbers of states; we will initially study a simple protocol based on three states (M, S, and I) as defined above. Then, in subsequent subsections, we will extend this protocol to more advanced, optimized protocols containing additional states.

The local state information maintained in a cache frame field indicates whether its line is in either the M or S state. Any memory line not present in a given cache is Invalid with respect to that cache. In essence, it is the absence of either an M or S state that indicates the I state. Thus, every line always has some state with respect to all of the caches at all times. If the cache frame holds no valid line, then the state bits associated with the cache frame mark the *frame* as invalid (lower case *i*– this is a different type of invalid than the I state). The invalidity of a frame is typically encoded with the same set of bits as a line's M or S state information. Although it is a subtle point, when a cache frame's bits are marked invalid, this is really saying that the frame is *empty*; it is not making an explicit statement about the state of any of the memory lines. However, on occasion, the state of a line contained in a cache frame is changed to I by evicting the line by setting the frame's state to *i*. The absence of a memory line from a cache is what indicates that the memory line is Invalid with respect to that cache. The importance of making this distinction will be more evident when we discuss the implementation of the coherence protocols.

Figure 21 is an example, where the states of three memory lines, A, B, and C are given. Line A is present in main memory, cache 0, and cache 1. Therefore it is Valid in memory, and in the S state in caches 0 and 1. Because it is not present in cache 2, the memory line's state with respect to cache 2 is Invalid. An up-to-date copy of line B is present only in cache 0; therefore, its state in cache 0 is M, and its state everywhere else is Invalid. Finally, line C is in none of the caches, so its state with respect to all of the caches is I. The copy in memory is Valid.

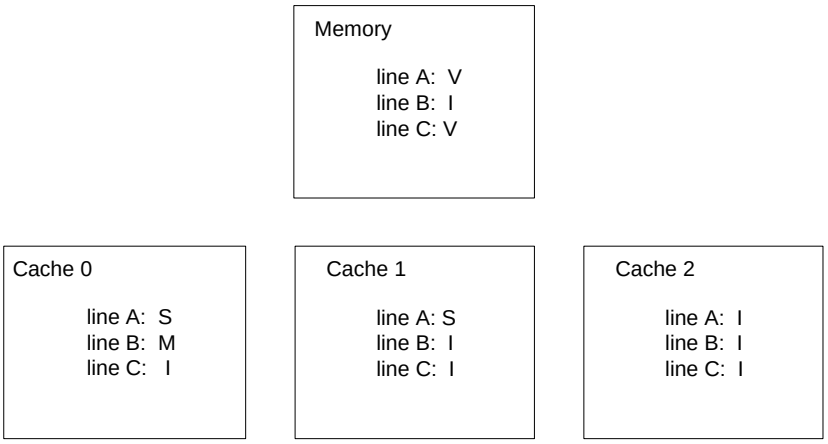


Figure 21. Example showing states of memory lines in caches and in main memory.

A coherence action may be initiated by a local processor when it initiates one of the following events.

Processor Read – the processor wants to perform a read for a given memory line.

Processor Write – the processor wants to perform a write to a given memory line.

Eviction – the processor wants to remove the memory line from the cache frame where it is being held; or, more often, the cache controller wants to remove a line to make room for a new line being brought into the cache.

For the MSI protocol described here, the caches are managed as write-allocate, write-back caches. This is fairly common in bus-based protocols because a write-back cache implies less bus traffic. As with most buses, the shared bus contains address, data, and control signals. In response to one of the processor actions listed above, a cache controller may invoke a corresponding bus action. The bus control signals indicate the type of bus action. Brief descriptions of the bus actions follow.

Cache Read – a copy of a memory line is requested by a processor's cache controller; the address is placed on the bus by the requesting cache controller, and, in response, either main memory or another processor's cache controller will provide a copy of the line. If a cache controller provides the copy of the line, it first signals its intention by activating a *memory inhibit* signal, which prevents the memory controller from responding.

Cache Read&M (Read and Modify) – a copy of a memory line is requested by a processor's cache controller; the address is placed on the bus by the requesting controller, and, in response, either the memory controller or another processor's cache controller will provide a copy of the line; at the time the request is made, an additional bus control signal indicates that line will be written (Modified) by the requesting processor after it is received. The memory inhibit signal is used in the same way as for the Cache Read.

Cache Upgrade – the address of a memory line is placed on the bus; a control signal indicates that that the local processor will write to the corresponding line. No data is transferred on the bus for this bus action.

Cache Write-back – the address of a memory line and the line's data are placed on the bus. The data is written to main memory.

The actions to be performed by the cache controller can be specified by a state transition diagram or a state table. We will use the table format. The inputs to the coherence algorithm signal either actions that are initiated by the local processor or bus actions initiated by a remote processor's cache controller. The bus actions monitored by the cache controller are three of the four given above: Cache Read, Cache Read&M and Cache Upgrade.

For the three-state (MSI) protocol, the state transition table is shown in Table 1. The left-hand column marks the current state for a given memory line, and across the top of the table in italics are the possible events that may take place. The entries in the table contain the actions (responses) taken by the cache controller for a given input event and a given state. These actions are the outputs of the state table. The table entry also shows the next state for the memory line. Some combinations of states and actions can never occur. For example, a line in the invalid state cannot be evicted from a cache; the state transition table is blank for impossible current state/action combinations.

Table 1. State transition table for the MSI coherence protocol.

	Action and Next State					
Current State	***Processor Read***	***Processor Write***	***Eviction***	***Cache Read***	***Cache Read&M***	***Cache Upgrade***
I	*Cache Read* Acquire Copy → S	*Cache Read&M* Acquire Copy → M		No Action → I	No Action → I	No Action → I
S	No Action → S	*Cache Upgrade* → M	No Action → I	No Action → S	*Invalidate Frame* → I	*Invalidate Frame* → I
M	No Action → M	No Action → M	*Cache Write-back* → I	Memory inhibit; Supply data; → S	*Invalidate Frame;* Memory inhibit; Supply data; → I	

Sequences of actions always start at the local processor when it attempts to read, write, or evict a line in its cache. Depending on the line's state in the processor's cache, the cache controller may (or may not) output an action (request) onto the bus. This request, carried over the bus, will usually act as an input event for the other (remote) cache controllers and/or main memory. Depending their cache line states, the remote cache controllers may (or may not) perform an action involving a state change or placing a copy of the line onto the bus. After bus cycle is complete, the local cache controller will then allow the processor to complete its read or write. Because the sequences of actions flow from a local processor through the local controller over the bus to the remote controller and back, we will describe the protocol with a similar flow. The columns on the left half of the table correspond to events initiated by the local processor's cache controller; on the right are events that come in over the bus.

If the processor attempts to read a certain memory line and the copy of the line its local cache is either S or M, then there will be a cache hit. The data is read from the cache and no further action needs to be taken. If the accessed line is not in the S or M states, then there is a cache miss (the line is in the I state). The controller initiates a Cache Read to effect a state change and acquire a copy of the accessed memory line. This Cache Read action will be propagated on the bus, and the other cache controllers will see it as an input event (refer now to the Cache Read column of the table). If a remote cache has

the line in the I or S state, no further action is required by the remote cache controller. The memory controller will also see the Cache Read and will provide a copy of the requested line. On the other hand, if a remote cache has the line in the M state, then it has the only valid copy of the data, and its cache controller must provide a copy of the data to the other cache (and main memory should not provide a copy). The remote cache controller accomplishes this by activating the memory inhibit bus control signal. This signal will prevent the memory controller from responding. Then, the remote cache controller will place its copy of the line on the bus; both the memory controller and the local cache will take the data off the bus. Because the remote cache no longer has the only valid copy of the line, it changes its local state to S. Finally, back at the local cache controller, when the Cache Read is completed, the line in the local cache will be placed in the S state (and the frame holding the line is marked accordingly).

If the processor attempts to write a certain memory line and the line is in the M state with respect to the processor's cache, then there will be a cache hit and the local processor can write to the cache; no further action is required. If the line is in the I state, then there is a cache miss and the controller initiates a Cache Read&M to acquire a copy of the accessed memory line. The Cache Read&M action will be propagated on the bus, and the other cache controllers will see it as an input event (refer now to the Cache Read&M column of the table). If a remote cache does not contain the line (it is in the I state), no action is required by that remote cache controller. If the remote cache has the line in the S state, then the frame holding the line is invalidated, causing the line's state to change to I, but nothing more needs to be done by the remote controller. If the remote cache has the line in the M state, then it has the only valid copy of the data, and it must provide a copy of the data to the requesting cache. The remote cache controller accomplishes this by activating the memory inhibit signal and then placing a copy of the line on the bus. If, on the other hand, none of the cache controllers signal "memory inhibit" then main memory will supply a copy of the data. Note that for a Cache Read&M, main memory does not need to get a copy of the line if one of the remote cache controllers signals memory inhibit. The reason is that the requesting cache will immediately modify the line after it is received, thereby making any memory copy Invalid. After the remote cache controller responds with a copy of the line, the line's frame is marked invalid (empty), causing the line to transition to the I state in the remote cache. Finally, when the Cache Read&M is completed, then the line in the local cache will be in the M state, and the local write can proceed.

If the local processor attempts a write and the line is in the S state with respect to its cache, then it could issue a Cache Read&M, just as if the line were in the I state. This would trigger a sequence of actions that would yield the desired result. However, this is inefficient because some other cache controller or the memory controller would be forced to provide a copy of the data on the bus. This is unnecessary because the local cache already has a copy of the line; the only problem is that its copy is in the S state, so there may be other copies. Consequently, the Cache Upgrade action is provided to get rid of (invalidate) any other copies, while avoiding the redundant data copy operation over the bus. Referring to the Cache Upgrade column, any remote cache with the line in the S state removes the line from its cache (thereby transitioning its state to I). The case where a remote cache has the line in the M state is impossible, because the M state implies there is only one copy, and the local cache has a copy in the S state.

The last processor action occurs when the processor evicts (removes) a line from its cache to make room for some other line. If the line is in the S state, then the local cache controller can simply remove the line from its frame by overwriting it with a new line (and state); nothing more is required. If the line is in the M state, then the local controller must perform a Write-back bus operation to copy the line back into main memory (at that point, memory will have the only valid copy of the line). Finally,

the table entry for the Eviction event when the line is in the I state is a condition that can never happen. If the line is not in the cache in the first place (as indicated by its I state), then it cannot be evicted.

The operation of the MSI protocol is illustrated with the example in Figure 22. In the figure, three threads, running on three processors, access the same cache line. Initially, the line is only in main memory, so the line is Invalid in all three caches. The first thread, T0, reads the line. This causes a cache miss and the cache controller issues a Cache Read (CR) to get a copy of the line. Memory will provide a copy, and the copy in the thread's cache, C0, becomes shared (S). Next, T0 performs a write to the line. This is a cache hit, but the local state is S, indicating there could potentially be other copies (although there aren't, C0's cache controller has no way of knowing this). Consequently, the cache controller issues a Cache Upgrade (CU) bus action. When the bus cycle is over, the line transitions to the M state in cache C0 as it is guaranteed to be I elsewhere. Next, T2 performs a read from the line. This will cause a cache miss in C2. C2's controller will issue a Cache Read action. This causes C0 to provide a copy of the line and to "downgrade" the line's state in its cache to S. Finally, T1 performs a write to the line. This causes a Cache Read&M (CRM) cycle which will acquire a copy of the line from memory and will cause the other cache controllers to remove copies of the line from their caches.

Thread Event	Bus Action	Data From	global state	local states:		
				C0	C1	C2
0. Initially:			<0,0,0,1>	I	I	I
1. T0 read→	CR	Memory	<1,0,0,1>	S	I	I
2. T0 write→	CU		<1,0,0,0>	M	I	I
3. T2 read→	CR	C0	<1,0,1,1>	S	I	S
4. T1 write→	CRM	Memory	<0,1,0,0>	I	M	I

Figure 22. Example of MSI coherence protocol.

THE MESI COHERENCE PROTOCOL

The MSI protocol, as we have observed, is a fairly basic protocol, even though we did add the Cache Upgrade action as an optimization. Some potential inefficiency remains, however. We observe that there will be cases where a local cache has a copy of a line in the S state, and no other cache has a copy of the line. Whenever this is the case, the MSI protocol will initiate a Cache Upgrade action if the local processor attempts to write to the line. Because no other cache has a copy of the line, however, this bus cycle isn't really needed. In practice, this particular situation happens fairly often, for example when a thread first loads a variable, modifies it, and then stores it.

To avoid the Cache Upgrade cycle, an additional state, the Exclusive (E) state is added to the protocol, leading to the MESI coherence protocol described in Table 2. The E state gives the local cache controller a more precise summary of the global state. In the E state the line is the same as the copy in memory, but it is the only cached copy. To support the MESI protocol, there is also an additional bus control signal, *sharer*, which a remote cache controller activates in response to a Cache Read when it has a copy of the line in the S state. When there is a read to a line that misses in the cache, and none of the remote caches activates the sharer signal, then the local cache knows there are no other cached copies, and the line is known to be in the E state.

Then, subsequently, if the line is written to, and is in the E state, a Cache Upgrade is not needed because there are no other shared copies to invalidate. The local cache can simply write to the line, and change its state to M. If, on the other hand, the line is in the E state and some remote cache controller issues a Cache Read, then the line's state is changed to S because it is no longer an exclusive copy.

Table 2. State transition table for the MESI coherence protocol.

	Action and Next State					
Current State	***Processor Read***	***Processor Write***	***Eviction***	***Cache Read***	***Cache Read&M***	***Cache Upgrade***
I	*Cache Read* If no sharers: → E If sharers: → S	*Cache Read&M* → M		No Action → I	No Action → I	No Action → I
S	No Action → S	*Cache Upgrade* → M	No Action → I	Respond Shared: → S	No Action → I	No Action → I
E	No Action → E	No Action → M	No Action → I	Respond Shared; → S	No Action → I	
M	No Action → M	No Action → M	*Cache Write-back* data → I	Respond Shared; Write back data; → S	Respond Shared; Write back data; → I	

Figure 23, is the first three lines of Figure 22 given earlier. It demonstrates the key difference between the MSI and MESI protocols. When thread T0 reads the line and then writes it, only one bus cycle is needed -- to bring in the initial copy of the line. The write can be done immediately and does not require any bus activity; however, there is a local state change from E to M.

Thread Event	Bus Action	Data From	global state	local states: C0	C1	C2
0. Initially:			<0,0,0,1>	I	I	I
1. T0 read→	CR	Memory	<1,0,0,1>	E	I	I
2. T0 write→	none		<1,0,0,0>	M	I	I

Figure 23. Example of MESI coherence protocol.

THE MOESI COHERENCE PROTOCOL

Acquiring a copy of a line from memory may be much slower than getting it from one of the other caches. Hence, when there is a copy of a line in a remote cache it is better for the local cache to take it from the remote cache than from main memory. In the MSI and MESI protocols, this is often not done. For example, when a line is in the S or M state in a remote cache, the memory controller first

performs a memory access and then supplies a copy of the line. When there are multiple cached copies (in the S state), only one cache should attempt to respond with a copy of the line, however. This leads to the addition of the O (Owner) state to the MESI protocol, yielding the MOESI protocol. There is also a change to the S state:

Shared (S): <1,X,X,X,…,X> -- the local cache has a valid copy, main memory may or may not ahve a valid copy, and other caches may or may not have valid copies. If main memory does not have a valid copy then there must be some cache in the O state.

The owner is the one holder of a cached copy designated to supply it to other caches. The state transition table for the MOESI protocol is in Table 3.

Table 3. State transition table for the MOESI coherence protocol.

	Action and Next State					
Current State	***Processor Read***	***Processor Write***	***Eviction***	***Cache Read***	***Cache Read&M***	***Cache Upgrade***
I	*Cache Read* If no sharers: → E If sharers: → S	*Cache Read&M* → M		No Action → I	No Action → I	No Action → I
S	No Action → S	*Cache Upgrade* → M	No Action → I	Respond shared; → S	No Action → I	No Action → I
E	No Action → E	No Action → M	No Action → I	Respond shared; Supply data; → S	Respond shared; Supply data; → I	
O	No Action → O	*Cache Upgrade* → M	*Cache Write-back* → I	Respond shared; Supply data; → O	Respond shared; Supply data; → I	
M	No Action → M	No Action → M	*Cache Write-back* → I	Respond shared; Supply data; → O	Respond shared; Supply data; → I	

Figure 24 is an example of the MOESI protocol in action. It illustrates coherence activity for the same memory access sequence as in Figure 22. The O state come into play when T2 performs a read of the line and C0 has the line in the M state. In the MESI protocol (Table 2), C0's cache controller would supply the line, write a copy of the line back to memory, and enter the S state. With the MOESI protocol, there is no need to write the data back to memory. Instead C0's controller supplies the data to C2 and enters the O state; it is now the line's owner. Later, when T1 writes to the line (and misses in its cache), C0's cache controller will again supply the line (no need for a memory access). C0 will invalidate the line, and C1's copy of the line is in the M state. This makes C1 the *de facto* owner, although because it is in the M state, it must have the only copy, so the O state is not used.

Thread Event	Bus Action	Data From	global state	local states		
				C0	C1	C2
0. Initially:			<0,0,0,1>	I	I	I
1. T0 read→	CR	Memory	<1,0,0,1>	E	I	I
2. T0 write→	none		<1,0,0,0>	M	I	I
3. T2 read→	CR	C0	<1,0,1,0>	O	I	S
4. T1 write→	CRM	C0	<0,1,0,0>	I	M	I

Figure 24.Example of MOESI coherence protocol.

DRAGON: AN UPDATE PROTOCOL

We now consider cache update protocols that are implemented with bus snooping. In practice, update protocols are less commonly used than invalidate protocols. Which is better, invalidate or update, depends on memory access patterns, and the patterns that favor invalidate protocols typically out-weigh the ones that favor update protocols. Memory access patterns are discussed below in **Section X**. Some cache implementations implement both types of protocol, with a mode bit selecting which is used.

Probably the best-known update protocol was first used on the Xerox PARC Dragon multiprocessor system [3]. This system was a research prototype. Some of the Dragon developers left Xerox and developed a multiprocessor system at Digitial Equipment Corp. In that system, another uptdate protocoll, the Firefly protocol, was developed [26]. In this section we will describe both protocols.

For the Dragon protocol, the local cache states follow. The state information for a given memory line is on the left; this is followed by a vector showing the global information that the state summarizes along with a brief written description.

Invalid (I): <0,X,X,X....X> -- The local cache does not have a valid copy; accessing the line will cause a cache miss. This is similar to the I state in an invalidate protocol.

Shared Clean (Sc): <1,X,X,X,…,X> -- The local cache has a valid copy of the line; there is no information regarding other copies of the line. This state is the same as the O state for an invalidate protocol.

Shared Modified (Sm): <1,X,X,X…,0> -- The local cache has a valid copy of the line and main memory does not. When in this state, it is the local cache controller's responsibility to update memory if the line is evicted. Only one cache at a time can have a line in the Sm state. This is a new state, not present in an invalidate protocol.

Exclusive(E): <1,0,0,..0,…1> -- The only valid copies of the line are in the local cache and in memory. This state is the same as the E state for an invalidate protocol.

Modified(M): <1,0,0,..0,…0> -- The only valid copy of the line is in the local cache. This state is the same as the M state for an invalidate protocol.

In the Dragon protocol the cache controllers can perform a Cache Read, just as in the invalidate protocols. However, instead of a Cache Read&M action, the Dragon protocol uses a *Cache Update*. The Cache Update takes data the just written by a local processor and broadcasts it on the bus so that the other cache controllers can update their copy of the line. It is this feature that distinguishes an update protocol from an invalidate protocol.

Table 4 is the state diagram for the Dragon update protocol. Besides the update feature, which is evident in the state diagram, another feature of the protocol is that there may be multiple sharers of a given line, with no clean copy in memory. This means that one of the sharers must supply the line on a Cache Read command. This is done my marking only one of the sharers as Sm, while the others are in the Sc state. The most recent sharer to update a line has the line in the Sm state. The sharer with the line in the Sm state is then the one that supplies the line on a Cache Read. When all the shared copies of a line are clean (in the Sc state), then memory also has a clean copy, and it is the memory controller's responsibility to provide the line to a requestor.

Table 4. State transition table for the Dragon update coherence protocol.

	Action and Next State				
Current State	***Processor Read***	***Processor Write***	***Eviction***	***Cache Read***	***Cache Update***
I	*Cache Read* If no sharers: → E If sharers: → Sc	*Cache Read* If no sharers: → M If sharers: *Cache Update* → Sm		→ I	→ I
Sc	No Action → Sc	*Cache Update* If no sharers: →M If sharers: → Sm	No Action → I	Respond Shared; → Sc	Respond shared; Update copy; → Sc
E	No Action → E	No Action → M	No Action → I	Respond shared; Supply data → Sc	
Sm	No Action → Sm	*Cache Update* If no sharers: →M If sharers: → Sm	*Cache Write-back* → I	Respond shared; Supply data; → Sm	Respond shared; Update copy; → Sc
M	No Action → M	No Action → M	*Cache Write-back* → I	Respond shared; Supply data; → Sm	

As with the MESI and MOESI invalidate protocols, the Dragon protocol bus has a *shared* control signal. In response to a Cache Read or Cache Update, the shared signal is activated by cache controllers whose caches have a valid copy of the accessed line. This tells the requesting cache controller that it should place the line being accessed in one of the shared states. However, it appears that a shared bus signal is not adequate by itself. There must be some way to distinguish between the case where all the sharers are in the Sc state (so that the memory controller provides the line) and the case where there is

an Sm sharer (or a cache with the line in the M or E state) in which case a cache controller must provide the data (memory does not have an up-to-date copy). This set of alternative responses suggests that a single shared line is not enough. There has to be a shared signal and a memory inhibit signal. The shared signal would be used as in Table 4. The cases where a cache controller supplies the line would also have to inhibit memory controller so that it does not attempt to supply the line.

Also note that if there is a write that misses in the cache, the local processor must invoke two bus actions. The first is a Cache Read, which acquires a copy of the line. This is followed by a Cache Update when the local cache controller performs the write to the newly acquired copy. It appears that these must be atomic bus actions. Otherwise, if some other processor accesses the line in between the read and the update, the final state could be erroneous.

An example of the Dragon Protocol is in Figure 25. This is the same sequence of cache accesses as in previous examples, with an additional access by T0. The initial read and write by thread T0 are similar to the MOESI protocol. The update protocol distinguishes itself when T2 performs a read. Then this happens, cache controller C0 supplies the line. C0's controller marks the line as Sm, because it is the most recent writer to the now-shared line, and C2's controller marks it as Sc. Then, when T1 performs a write, C1's controller first performs a Cache Read. In response, C0 supplies a copy of the line. Then, controller C1 immediately performs a Cache Update. This causes the other sharers of the line to update their copies. When the update is finished, C1, the most recent writer, has the line in the Sm state and the other sharers have it in the Sc state. At some point, if the copy of the line that is in the Sm state is evicted from its cache, then the line is written back to memory. The other copies are in the Sc state, but at this point memory has a clean copy, so the line is in a consistent global state. Finally, when T0 performs a read, it will hit in the cache; this illustrates the advantage of an update policy. An invalidate policy would cause a cache miss.

Thread Event	Bus Action	Data From	global state	local states		
				C0	C1	C2
0. Initially:			<0,0,0,1>	I	I	I
1. T0 read→	CR	Memory	<1,0,0,1>	E	I	I
2. T0 write→	none		<1,0,0,0>	M	I	I
3. T2 read→	CR	C0	<1,0,1,0>	Sm	I	Sc
4. T1 write→	CR,CU	C0	<1,1,1,0>	Sc	Sm	Sc
5. T0 read→	none (hit)	C0	<1,1,1,0>	Sc	Sm	Sc

Figure 25. Example of Dragon update coherence protocol.

FIREFLY PROTOCOL

The Firefly protocol has similar states to Dragon (although the states are given different names). These names are given below. The Firefly designers observed that a line can be in a cache either shared or exclusive, and either clean or dirty, so they used two status bits to identify the four combinations. They assume that every cache frame always holds some valid line, so there is no way to mark an invalid (empty) frame. This led them to state that there is no I state in the protocol. However, this confuses

the state of a frame with the state of the line that is being held in a frame. In fact, any memory line that is not present in a cache is in the I state with respect to that cache.

The decision to assume that every frame always holds a valid line is a design decision that is in a sense orthogonal to the protocol. However, the lack of the ability to mark a frame as being empty (invalid) means that at boot time there is a sequence of cache reads to fill all the caches with lines so that the state information is consistent. This saves one bit of state information per frame; at the time the Firefly was implemented, this may have been considered significant. Today, however, one would probably add a state bit that directly indicates an invalid (empty) frame.

Invalid (I): <0,X,X,X....X>

Shared Clean (Sc): <1,X,X,X,…,X>

Shared Modified (Sm): <1,X,X,X…,0>

Exclusive Clean (Ec): <1,0,0,..0,…1> -- the same as the E state in the Dragon protocol.

Exclusive Modified(Em): <1,0,0,..0,…0> -- the same as the M state in the Dragon protocol.

In the Firefly Protocol, the two controller-invoked actions (besides eviction) are the Cache Read and Cache Read&M. Whenever a thread writes to a line that may be shared, then there is a Cache Read&M that also updates memory. Hence, the Firefly protocol performs Write-Through updates. This is the major difference with respect to the Dragon protocol and it generates more memory write traffic than the Dragon protocol. The State table for the Firefly protocol is in Table 5.

Table 5. State transition table for the Firefly update coherence protocol.

	Action and Next State				
Current State	***Processor Read***	***Processor Write***	***Eviction***	***Cache Read***	***Cache Read&M***
I	*Cache Read* If no sharers: → Ec If sharers: → Sc	*Cache Read* If no sharers: → Em If sharers: *Cache Update* → Sm		No Action → I	No Action → I
Sc	No Action → Sc	*Cache Read&M* If no sharers: →*Ec* If sharers: → Sc	No Action → I	Respond Shared; → Sc	Respond Shared → Sc
Ec	No Action → Ec	No Action → Em	No Action → I	Respond shared; → Sc	Respond Shared → Sc
Sm	No Action → Sm	*Cache Read&M* If no sharers: →Ec If sharers: → Sc	*Cache Write-back* → I	Respond shared; Supply data; → Sm	Respond shared; Supply data; → Sc
Em	No Action	No Action	*Cache Write-back*	Respond shared; Supply data;	Respond shared; Supply data;

	→ Em	→ Em	→ I	→ Sm	→ Sc

NASTY REALITIES

When a coherence protocol is implemented, the state changes in the table do not happen instantaneously and atomically. After the processor performs a read or write and the cache controller decides some coherence action is necessary, several cycles may pass before the shared bus is acquired by the controller. Because the bus is the ordering point for the protocol, the indicated state change can only occur after the bus has been acquired. But, between the time the controller decides to make a bus request and time it actually acquires the bus, some other cache controller could acquire the bus and perform some action that would cause a line to change state. For example, in the MESI protocol, a local cache may have a line in the S state when the thread running on the local processor performs a write. This should cause a request for a Cache Upgrade, followed by a transition to the M state. However, say that while waiting to get the bus for the Cache Upgrade, some other cache controller acquires the bus and performs a Cache Read&Modify for the same line. This means that the local copy of the line should transition to the I state. In response to this implied transition, the local cache controller should make a Cache Read&Modify request rather than its pending Cache Upgrade request.

The cache controller must deal with many situations such as the one just described, and it is these situations that make implementations of coherence protocols complicated. The cache controller is itself a finite state machine whose responsibility is to implement the coherence protocol's state table. *But*, these are, in reality two different state machines. The coherence protocol table reflects the assumption that state transitions are performed instantaneously and atomically. To reconcile this assumption with the real-world, the controller implements a number of transient states that are not in the coherence specification. A portion of the state table that a controller actually implements for the MESI protocol is illustrated in Table 6. When the processor misses in the cache (finds an invalid line) the cache controller first requests the bus to perform a Cache Read or Cache Read&M. It then goes into either the IR or IW state while it waits for the bus to be granted. Eventually, when the bus is granted, it places its request on the bus, and then it goes into either the IRA or IWB state while it waits for the bus response. Finally, when it gets the response, it writes the newly acquired line into the cache and updates the cache state; in the case of a read, this depends on the value of the shared signal in the response.

If the cache line is in the S state and the thread running on the processor attempts a store, then the cache controller requests the bus and enters the IWR state. Then, if another cache controller places a Read&M request on the bus, the cache controller recognizes this situation and transitions to the IWR state. That is, after another cache controller performs a Cache Read&M, the local copy of the line is no longer valid, and the cache controller must now act as if the local thread had missed in the cache.

There are a number of other issues that can further complicate the cache controller. One is the handling of evictions, another is the overlap of multiple requests from the same thread. These typically can be coalesced, but the coalescing must be correct with respect to the protocol. Combining a memory consistency model with the coherence implementation is another complication. The bottom line is that the coherence/consistency implementation is one of the more error-prone parts of a modern multiprocessor design, and significant effort is applied to validating coherence protocols.

Table 6. Controller state table (partial) for implementing the MESI protocol.

	Action and Next State						
Current State	***Processor Read***	***Processor Write***	***Bus Grant***	***Bus Response***	***Cache Read***	***Cache Read&M***	***Cache Up-grade***
I	Request Bus → IR	Request Bus → IW			No Action → I	No Action → I	No Action → I
S	No Action → S	Request Bus → SW			Respond Shared: → S	No Action → I	No Action → I
E	No Action → E	No Action → M			Respond Shared; → S	No Action → I	
M	No Action → M	No Action → M			Respond dirty; Write back data; → S	Respond dirty; Write back data; → I	
IR			*Cache Read* → IRR				
IW			*Cache Read&M* → IWR				
IRR				If no sharers: → E If sharers: → S Load line into cache			
IWR				→ M Load line into cache			
SW			*Cache Up-grade* → M		Respond Shared: → SW	No Action → IW	No Action → IW

4.3.2 Directory Coherence

In the bus-based protocols, the bus is the ordering point through which cache controllers can communicate their intentions in a sequential fashion. Another natural ordering point is at the memory controller(s) because this is where requests for a particular memory address are logically sent. And, because every system has memory controllers, even if the system doesn't have a centralized bus, coherence protocols managed by the memory controllers can be applied to a wider range of systems than the bus-based snooping protocols. A given memory controller is responsible for the coherence of all lines in the memory that it manages. Consequently, another advantage of enforcing coherence at the

memory controllers is that the number of memory controllers tends to scale with the size of the system, thereby leading to a scalable coherence mechanism.

In a *directory* coherence protocol, the memory controller maintains a repository, or directory, that contains a fairly complete global picture of memory line status. Meanwhile the local caches contain a simplified summary of the global state as with the invalidation protocols. Then, when needed, a local cache controller can invoke the memory controller to coordinate coherence activity, by using the global directory information. It is the memory controller that is in charge. This is in contrast to the bus-based protocols where the shared bus allows the cache controllers snoop and "compare notes" to coordinate coherence activity, and the memory controller is only a passive participant.

One can develop a directory protocol based on virtually any of the bus-based protocols discussed in the previous section. For illustration, we will use a rather generic directory protocol based on an MSI protocol. For the directory protocol, following are the local cache states maintained by the cache controllers. As with the bus-based protocols, these local states are a summary of global state information.

Local Cache States:

Invalid (I): <0,X,X,X....X> -- The local cache does not have a valid copy.

Shared (S): <1,X,X,X,…,1> -- The local cache has a valid copy, main memory has a valid copy, and other caches may or may not have valid copies

Modified(M): <1,0,0,..0,…0> -- The local cache has the only valid copy; in particular, main memory does not have a valid copy.

As noted above, in a directory protocol, the memory directory also contains global state information regarding memory lines. It does this by keeping a separate directory entry for every memory line. In a straightforward implementation, a directory entry literally contains the entire global state vector for the line. Moreover, for implementation simplicity, it often keeps this information in a slightly redundant form. First, the directory entry contains a summary of the global state for each memory line:

Memory Directory States:

Modified (M): <0,..1,0,0,..0,…0> -- One of the caches has a copy of the line; the main memory copy is out of date.

Shared (S): <1,X,X,X,…,1> -- At least one cache has a valid copy, and main memory has a valid copy.

Uncached (U): <0,0,….,1> -- None of the caches has a copy of the line.

In addition, the directory entry keeps a list of any valid cached copies of the line, i.e., a list of *sharers*. The sharers list is either an exact list or a superset list that includes all the valid sharers, but may also include some caches that do not have valid copies. Initially, we will consider a memory directory that (mostly) holds an exact list of the sharers (the exception will be discussed later). This is done by maintaining the global state vector <….> with a 0/1 entry for every cache. The entries with a 1 form the list of sharers. Maintaining a global state vector means that the other directory state information (M,S,U) is redundant However, in an implementation, these M,S,U summaries may expedite the response by the memory controller.

To better see how a directory protocol works, an example is given in Figure 26. This example will also be used for introducing terminology. We consider coherence actions triggered by a processor read or write to the *local cache*; the local cache is managed by the *local* cache controller. With respect to this

cache and cache controller, all the others are *remote* caches and controllers. In the global state vector, without loss of generality, we let the first component represent the local cache's state. Say that initially a cache line is in the global state <0,1,0,….0>. That is, neither the local cache nor main memory has a valid copy. The only valid copy is in one of the remote caches (cache 1 in this example); under these circumstances, the remote cache that has the one valid copy is also referred to as the *owner* cache, and its controller is the *owner* controller. Note that an owner cache is also a remote cache.

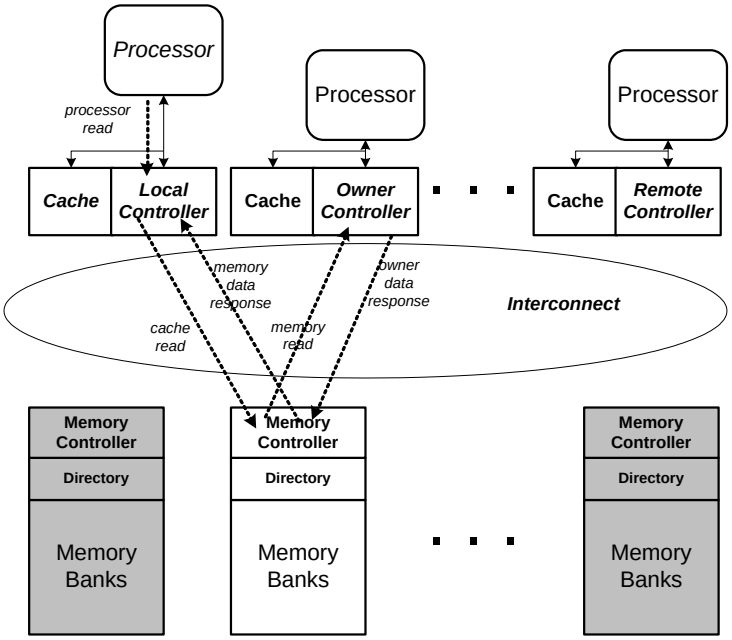


Figure 26. Example of directory protocol.

Returning to the example in Figure 26, given the initial global state of <0,1,0,…0> , the local processor performs a read to the cache line. There is a miss in the local cache, and a *cache read* message is issued by the local cache controller, and the message is sent through the interconnection network to the memory controller. We will use the message naming convention that the originator of a message is appended to the type of request contained in the message; in this case the originator is a cache (actually the cache controller) and read request is to get a copy of the line for the local cache.

The cache read command, along with the address of the line, is routed through an interconnection network to the memory controller that manages the memory bank holding the addressed line. This is referred to as the *home memory controller*; it is the memory home of the cache line. Continuing with the example, the home controller accesses its directory and determines that it does not have a valid copy, and it identifies the owner cache which has the only valid copy of the line. The home memory controller then issues a *memory read* message to the home controller. This message is routed through the interconnection network to the owner cache controller. In response, the owner cache controller retrieves the line from its cache and sends it back to the memory controller as part of an *owner data* message. In addition, the owner sets the local state of the line to S, recognizing the fact that it no longer has the only cache holding a copy of the line. When the owner data reaches the memory controller, the memory controller writes the line into the appropriate memory bank, changes its state to S, updates its sharer list to indicate that the local cache has a copy, and then it sends the line to the local cache controller via a *memory data* message. When the memory data reaches the local cache controller, it writes the line into its cache and changes the line's local state to S.

In the example just given a total of four messages pass through the network. After we describe this protocol in more detail, we will discuss message passing alternatives that are intended to optimize the process. To describe the complete directory protocol, we use two state tables, one table for the cache controllers (Table 7), and one table for the memory controller(s) (Table 8). Each of these tables is divided into a left half and a right half. The left side of the controller table is for actions initiated by the local processor (or controller in the case of an eviction). The right side of the controller table is for actions initiated by the memory controller. The left side of the memory controller table is for actions initiated by a local controller and the right side is for actions initiated by a remote controller.

Table 7. Cache controller state diagram for directory protocol.

	Cache Controller Actions and Next States							
	from Processor Side			from Memory Side				
Current State	***Processor Read***	***Processor Write***	***Eviction***	***Memory Read***	***Memory Read&M***	***Memory Invalidate***	***Memory Upgrade***	***Memory Data***
I	*Cache Read* → I'	*Cache Read&M* → I"				No Action → I		
S	No Action → S	*Cache Upgrade* → S'	No Action* → I			Invalidate Frame; *Cache ACK*; → I		
M	No Action → M	No Action → M	*Cache Write-back* → I	*Owner Data;* → S	*Owner Data;* → I	Invalidate Frame; *Cache ACK*; → I		
I'								Fill Cache → S
I"								Fill Cache → M
S'							No Action → M	

The left side of Table 7 is essentially the same as the left side of the MSI bus-based protocol table (Table 1). In this table, for clarity, we add some intermediate, pending states marked with primes (') . A controller waits in a pending state until it gets the eventual response to a message that it issues to the memory controller. For example, if there is a processor read and a cache miss, the controller enters the I' state while it waits for the accessed line to be returned by the memory system.

Another possible difference with the MSI bus-based protocol occurs when a line in the S state is evicted by the cache controller. If no external action is taken (as in bus-based protocol), then the state table is the same as the processor side of Table 1. However, a design option is to notify the memory controller that the line is being evicted. This will allow the memory controller to update its sharer list to keep it exact. If this is not done, then the sharer list will no longer be exact; it will be a superset list of sharers. However, not notifying the memory controller of such an eviction saves some interconnec-

tion network and memory controller activity. We assume that No Action is taken in the protocol described here.

Table 8. Memory controller state table for directory protocol.

	Memory Controller Actions and Next States					
	command from Local Cache Controller			response from Remote Cache Controller		
Current Directory State	***Cache Read***	***Cache Read&M***	***Cache Upgrade***	***Data Write-back***	***Cache ACK***	***Owner Data***
U	*Memory Data*; Add Requestor to Sharers; → S	*Memory Data*; Add Requestor to Sharers; → M				
S	*Memory Data*; Add Requestor to Sharers; → S	*Memory Invalidate* All Sharers; → M'	*Memory Upgrade* All Sharers; → M"	No Action → I		
M	*Memory Read* from Owner; → S'	*Memory Read&M*; to Owner → M'		Make Sharers Empty; → U		
S'						*Memory Data* to Requestor; Write memory; Add Requestor to Sharers; → S
M'					When all ACKS then *Memory Data*; → M	*Memory Data* to Requestor; → M
M"					When all ACKS then → M	

Sequences of coherence actions are initiated by a local processor (or the local controller evicting a line). Let's first consider the actions initiated by processor read (refer to the left side of Table 7). If it hits in the cache, then no further action is needed and the local state of the line is unchanged. If the read misses in the cache, then the cache controller sends a Cache Read to the memory controller and enters the pending state I'. When the memory controller receives the message (left side of Table 8) and the directory state is U or S, then main memory contains a valid copy of the line. The memory controller reads the line from memory and sends it to the requesting local cache (via a Memory Data message) and adds the local cache to the list of sharers. If the directory state is M, then the sequence illustrated earlier in Figure 26 is followed. The memory controller issues a Memory Read to the owner. The owner controller (right side of Table 7) sends the data back to memory via the Owner Data message,

and changes its state to S (it must have been M previously). The memory controller receives this message (right side of Table 8), and passes the data on to the local controller (via a Memory Data message). It also writes a copy of the line into memory, adds the sharer to the list of sharers, and changes its state to S. Finally, the local controller receives the line (right side of Table 7); it writes the line into the local cache, changes the local state to S, and then the data is returned to the processor to satisfy its original read request.

If the local processor performs a write and misses in the cache, then the local controller issues a Cache Read&M message, indicating it wants a copy of the line, and it plans to modify it. If the memory controller finds the directory state to be U, then it can immediately send a copy of the line to the requesting, local cache, change the directory state to M, and place the local cache in the list of sharers (it is the only sharer). More complex situations occur if the memory controller finds the directory state to be S or M. These two cases are covered in the next two paragraphs.

If the memory directory finds the line in the S state, then it can send a copy to the local controller, but it must first invalidate all the shared copies. It does this by going through the sharer's list and sending a Memory Invalidate command to each, along with the address of the cache line. It then waits for the remote caches to respond (it enters the M' state). After all the acknowledgements are received, it then sends a copy of the line to the local cache (via a Memory Data command) and places the directory entry in state M. This sequence is illustrated in Figure 27.

If the Read&M message reaches the memory controller and the directory state is M, then the memory controller requests a copy from the owner (Memory Read&M). The owner will respond with the line (Memory Data), and the memory controller passes the line back to the local controller (Memory Data). The directory state remains at M.

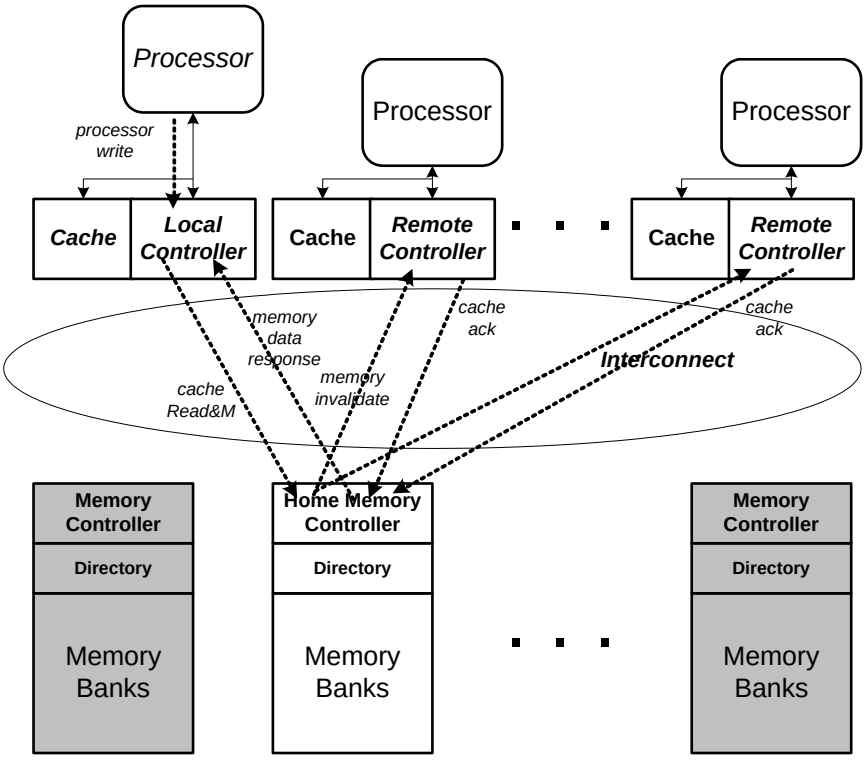


Figure 27. Example of directory protocol; the local processor writes to a line that is shared in remote caches.

An example sequence of thread reads and writes is in Figure 28. The first read by T0 gets the line from memory in a fairly straightforward manner. When the same thread makes a write request, then the controller needs to upgrade the line. In this case, the memory controller would normally send upgrade commands to remote caches; however, there is only one sharer, C0, so the memory controller does not need to issue the Memory Upgrade messages (the reason for the * in the example). When T1 reads the line, then it issues a Cache Read, the memory controller issues a Memory Read to get the only valid copy of the line from C0, and then passes it onto C1 via an MD command. Finally, when T1 performs a write, the memory controller must issue Memory Invalidate messages to the sharers (C0 and C2). After they acknowledge via Cache Acknowledge (CA) messages, then the memory controller can pass the data to C1 via a Memory Data message. As an optimization, the memory controller may pass the data to C1 prior to getting the acknowledgements, but as the ordering point, the Memory Controller should wait for the acknowledgements before it can handle any subsequent requests regarding the given cache line.

Thread Event	Controller Actions	Data From	global state	local states: C0	C1	C2
0. Initially:			<0,0,0,1>	I	I	I
1. T0 read→	CR,MD	Memory	<1,0,0,1>	S	I	I
2. T0 write→	CU, MU*,MD		<1,0,0,0>	M	I	I
3. T2 read→	CR,MR,MD	C0	<1,0,1,1>	S	I	S
4. T1 write→	CRM,MI,CA,MD	Memory	<0,1,0,0>	I	M	I

Figure 28. Example of Directory coherence protocol.

For some coherence actions, the directory protocol just described requires four sequential passes or "hops" through the interconnection network. A prime example is given in Figure 26; the same example is illustrated in Figure 29a using an abbreviated diagram. Each hop through the network is time consuming, and performance will be improved if this latency can be reduced. This leads to a three hop protocol illustrated in Figure 29b. In this protocol, the memory controller includes in its message to the owner controller the name of the local cache that needs a copy of the line, the owner cache can then send the requested line directly to the local cache. To maintain proper ordering, the owner controller concurrently sends an acknowledgement (and/or a copy of the line) to the Memory Controller indicating that it has done so.

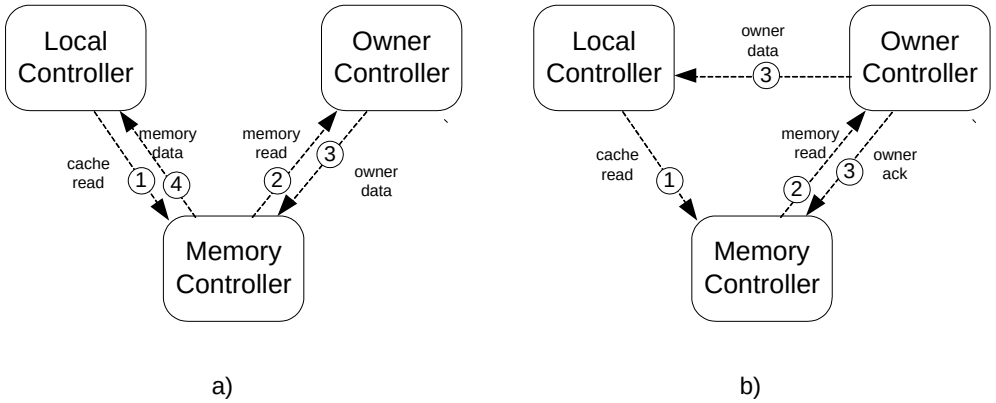


Figure 29. Four hop (a) and Three hop (b) protocols implementing Directory coherence.

The state table definition of a directory protocol presents a much simpler picture than that of a real design. A similar situation was noted above for bus-based protocols (nasty realities), but in the case of directory protocols the realities are probably nastier. The reason is that the protocol is largely distributed with both cache controllers and the memory controller actively participating. And, because there is no centralized bus, coherence-related actions of one controller are not immediately observable by the others. This means that at any point in time, a local controller has no way of knowing the coherence messages that may be directed to it but which have not yet arrived.

Let's consider a simple example. Say that a processor has decided to evict a cache line that is in the M state. If it simply writes back the line and immediately re-uses the cache frame for some other line (as is suggested by Table 7), then there is a problem if a message requesting a copy of the evicted line arrives shortly thereafter. This message may have been in transit at the time the eviction decision was made by the local controller. At this point, however, the local cache no longer has a copy of the line with which it can respond. One option is for the local controller to then re-direct the message back to the memory controller (i.e., negative acknowledgement). Another option is for the memory controller to acknowledge the completion of all line write-back messages. It is the responsibility of the local controller to retain a copy of the evicted back line until it gets the write-back acknowledgement. Then if it gets a memory read message for a recently evicted line, it can respond with the saved version of the line. This is just one example. In a realistic protocol there are a myriad of complicating possibilities, so we won't dwell on the details of this particular protocol – it is more important to give the flavor of the problem.

Two features of the basic directory protocol just described are 1) there is a directory entry for every memory line and 2) the directory entry keeps a complete global state vector. If, for example, the global state information consists of two bytes (16 processors in the system), and the line size is 128 bytes, then the directory overhead is at least 1.5% (this ignores the state summary information M,S,U). However, if the number of processors is increased to 128, for example, then the overhead increases to about 12%. This overhead starts to become significant, and for very large scale systems (say 1024 processors) it can become prohibitive. Consequently there have been a number of proposals to reduce this overhead.

One approach is based on the observation that most lines have either a very small number of sharers, or they have a large number of sharers. Consequently, the directory tracks a small number of sharers (say two or three) by keeping their processor identifiers. Then, if the number of sharers exceeds this number, a flag in the directory is set, and it is assumed that all the processors are sharers. This then

requires a broadcast when the copies of a line have to be invalidated in all the sharers. Alternatively, the sharing list (vector) can be defined to have a coarser granularity than individual processors. For example, a single bit in the vector may represent four or eight processors. When invalidation is required, all four or eight are sent invalidation messages.

Another way to reduce overhead is to keep a directory entry only for the memory lines that are actually cached. This is based on the observation that at any given time, most memory lines are uncached. With this approach, the directory itself is a form of cache. The directory access becomes an associative lookup. If the lookup hits, then the directory information of a cached line is available. If the lookup misses, then there are no cached copies (the line is in the U state). In some cases, the configuration of the directory cache and line access patterns may force the eviction of an entry from the directory cache (which, in turn causes the memory controller to evict cached copies). However, with appropriately sized and configured directory caches, this can be made a rare event.

4.4 Memory Consistency

Memory coherency involves ordering among accesses to one memory location; with memory consistency, the problem expand to the ordering among memory accesses to all locations, and, in many respects, this is a more challenging problem.

In Chapter 2, we discussed consistency models as an architecture issue, and described both sequential consistency and weakly-ordered consistency models. However, we have deferred details regarding memory ordering models and their implementation to this chapter because they are both closely tied to implementation issues regarding memory hierarchy implementations (described in this chapter) and load/store buffering (described in Chapter 3).

Following are a number of implementation features, intended to improve performance, which will all give correct results for single-threaded applications, but which may break sequential consistency when multiple threads are running.

1) Load instructions can pass other load instructions. This can happen either because the processor issues load instructions out-of-order, or because one load can pass another as it goes through the cache access path. For example, in a system that allows overlapped cache handling, one load may miss in the data cache, and a following load may hit in the cache.

2) Loads instructions can pass store instructions. This could happen due to aggressive, speculative out-of-order issue, or it could happen even if loads and stores issue in-order as soon as their addresses are available. In this latter case, a store may be held in the store buffer, waiting for data, while a following load (to a different address) goes ahead.

3) Store instructions may pass other store instructions. One way this can happen is with coalescing store buffers. Because stores occurring at different times may be placed into the same store buffer, store reordering is a byproduct.

4) Store buffering with forwarding may let a load from a given processor see the result of a store from the same processor before it is observed by other processors. Write-through caches cause similar problems; the cache is like a large store buffer and a processor may read its own data before other processors do.

5) Some coherence protocols may cause stores to be visible at different processors at different times. Consider a directory protocol where the memory controller sends out invalidations. The different cache controllers may see invalidations at different times; so, following a store instruction, some processors may read the old value, while others read new value.

There are two main aspects to a memory ordering model; one is *write atomicity*, and the other is *load/store reordering*. These two aspects cannot always be strictly separated, but for our purposes, it is useful to discuss them separately.

4.4.1 Write Atomicity

Because most systems have cache memories associated with each processor, a given logical memory location may be replicated in one or more caches, in store buffers, and in main memory. This means that each of the threads running on the processors has its own "view" of the contents of memory. Theoretically, all these views should be the same; however, if a thread performs a store instruction to a given memory location, all the replicas may not be updated at the same time, so the different threads may have inconsistent views of memory.

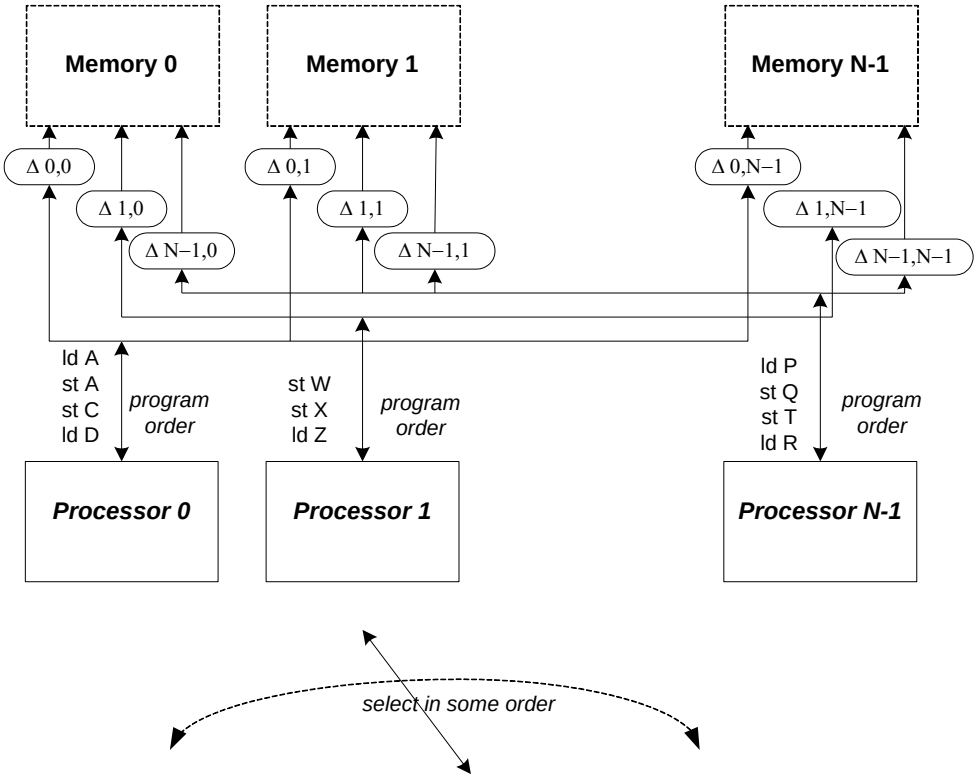


Figure 30. Writes by processors to memory observable by other processors may experience different delays.

This is illustrated in Figure 30. This figure is patterned after the classic sequential consistency model, but the major difference is that each thread essentially observes its own version of memory (in this figure the different memories are not memory banks, rather they are the per thread images of the entire memory). In the figure there are multiple paths to memory from the different threads with different delays; Δ i, j represents the delay from thread i to thread j's memory; in general, these delays do not have to be the same for all accesses. Consequently, in general, a write by one thread may be observed

by the other threads at different times. Another important feature, not illustrated in the figure, is that memory coherence is also assumed to hold; that is, updates to the each individual memory location take place in the same sequence across all the memories.

There are three cases of interest. One is where all the Δ i, j are equal; this condition is called *Write Atomicity*, and it is present in a sequentially consistent system (although this alone is not sufficient for sequential consistency). A second case is where, for a given thread i, all the Δ i, j i≠j are equal but do not necessarily equal Δ i,i; Δ i,i is usually less than Δ i,j . With respect to different writes from the same thread, however, the Δ i, j can be different. In this case, a given thread "sees" its own writes before the other threads, and all the other threads see the given thread's writes in the same order. This can happen, for example, if there is a store buffer with forwarding (however, this implementation feature by itself does not necessarily violate architected write atomicity). Finally, in the third case, the Δ i, j delays can be arbitrary. This could happen in a directory protocol, for example, where a memory controller sends invalidation messages on a store, but the invalidation messages arrive at caches at different times.

Consider an example where sequential consistency is violated because write atomicity is not present. In Figure 31, three threads operate on variables A and B. It is assumed that all three threads present loads and stores to the memory system in program order. However, because write atomicity is not enforced the Store to A by thread 0 takes longer to reach Thread 2's copy of memory location A than it takes to reach Thread 1's copy of location A. Consequently, the test by Thread 1 is true, and the store to B takes place. If this update is observed at Thread 2's copy of location B before it sees Thread 0's store to A, then Thread 2 will see a value (A=0) that violates sequential consistency.

```
                    Initially A=B=0

Thread0:            Thread1:            Thread2:
Store A←1

                    if (A==1)
                    Store B←1
                                        if (B==1)
                                        Load A=0
```

Figure 31. Sequential consistency can be violated when write atomicity is not present.

4.4.2 Load/Store Reordering

Now, we consider the second feature of an ordering model, and, as with write atomicity, we consider the implications of certain hardware implementations. For high performance, it is advantageous if a processor can issue and execute instructions out of the original program order. Most superscalar processors are designed to operate this way. And, when done in a uniprocessor, the design is carefully constructed so that results of a computation are the same as if the instructions had been executed in program order. In an out-of-order processor, one place that instructions are reordered is in the path to memory. Often it is the case that a store instruction is blocked because the store data is not yet available, yet a following load to a different address is ready. In this case, the load may be allowed to issue and pass the store instruction. In other cases, one load may be ready to issue (its address has been computed), but a preceding load in program order may not yet be ready. In these cases, the one load may pass the other. In general, any of the load or store instructions can pass another (provided they are to different memory addresses).

As noted above, in a uniprocessor, the hardware is designed so that the eventual results of instructions are the same as if the loads and stores had been executed in program order. As we have already discussed, however, in a multiprocessor the reordering of loads and stores can lead to systems that are not sequentially consistent.

To put it all together, consider Figure 32. We see that, generally speaking, a relaxed memory ordering model may allow load/store reordering (which conceptually happens at the processor side of the system), and relaxation of write atomicity (which conceptually is related to the memory system). Here we emphasize "conceptually", because in real designs, the processor and memory system are tightly integrated so that write atomicity and read/write reordering cannot be so easily compartmentalized.

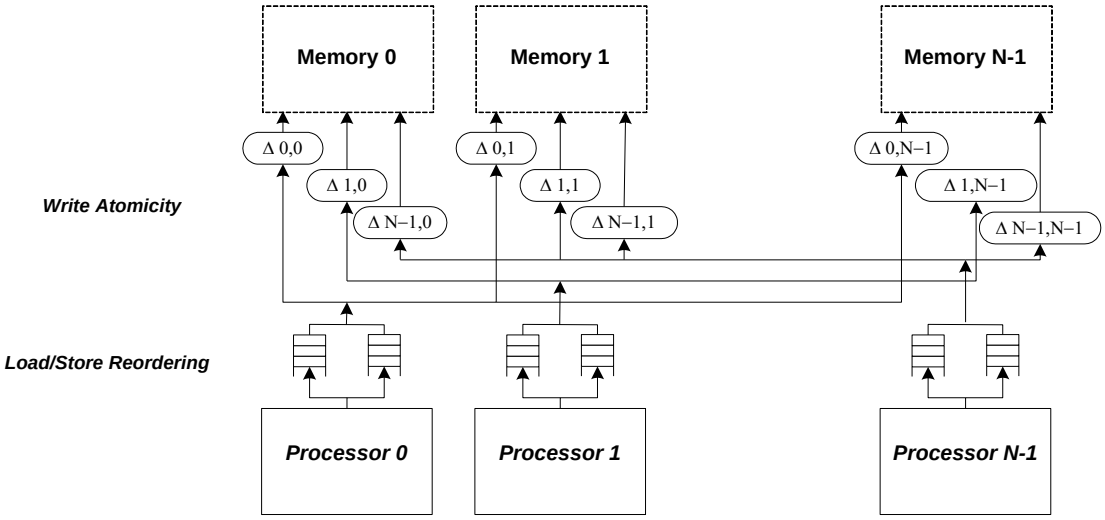


Figure 32. In general, relaxed memory ordering depends on write atomicity and load/store reordering. The ordering selector has been deleted from the figure, but conceptually, there is one that selects from each processor.

4.4.3 Implementations of Relaxed Ordering Models

The major ISAs, from IBM, Sun, Intel, and AMD, implement a variety of memory ordering models – and, because some ISAs specify more than one ordering model, there are probably more models than ISAs! For some of the older ISAs, designers sometimes implemented shared memory in a way that optimized performance for their particular design, then retrofitted an architected specification into later designs for compatibility. That is, the ordering model was essentially defined according to an early implementation. This approach, unfortunately, has led to a wide variety of complex models that are largely incompatible among different ISAs. Newer ISAs, developed with multiprocessing in mind, either have multiple, selectable ordering models, which gives the hardware designer/software programmer some alternatives to work with, and/or they have a relaxed baseline model with a variety of memory barrier operations to selectively enforce memory ordering when needed.

Probably the three most important relaxations of sequential consistency, from a performance perspective, are: 1) to let loads pass loads 2) to let loads pass stores 3) to allow a store being held in a store buffer to forward data to a later load. The reason is that loads are typically on performance critical paths, and it is important to execute load instructions as soon as possible.

The x86 architecture implements a form relaxed ordering that is called *processor consistency*. Here, there is write atomicity, but all load/store reordering is allowed, except stores are not allowed to pass stores.

With this model, the producer/consumer example given earlier (Figure 18 of Chapter 2) would work. Because stores cannot pass stores, and write atomicity is implemented, the write of A in the example will always appear to precede the write to Flag.

The Sun SPARC ISA offers a more modern style of architected memory ordering models. It defines three primary ordering models, with instructions that enforce stricter ordering among specific types of operations. The strongest of the three ordering models is Total Store Ordering (TSO). With TSO, loads can pass stores (to different addresses), but other load/store ordering is preserved. The key point is that all the stores performed by a given thread are made visible to all the other threads at the same time, and in order. However, a load is able to read a value written to the same address by its own thread before it is made visible to the other threads, (thereby enabling store buffer forwarding). With TSO most cases of inter-thread communication will work as expected. For example, our earlier producer/consumer example will work properly. Also the example in Figure 31 will work properly (it will be sequential consistency).

The SPARC ISA also supports a weaker model Partial Store Ordering (PSO) where stores are allowed to pass stores. With PSO, the example in Figure 31 might fail. The advantage of PSO is that it in a typical implementation it allows more overlapping of store instructions. Then, when operating in PSO mode, it is possible to force stores to execute in order by inserting a STBAR (store barrier) instruction between the stores that should be ordered.

The weakest model supported by the SPARC ISA is Relaxed Memory Ordering (RMO). With this model, all varieties of load/store reordering are permitted. A MEMBAR instruction, with a four bit encoding, can then be used to enforce any combination to load/store reordering on a per situation basis.

Note, however, in the hardware, designers sometimes do not implement all the models allowed by the ISA; a weaker model does not have to be implemented as long as a stronger model is implemented. For example, the designers of a SPARC processor might decide to implement only TSO and RMO. When PSO mode is required, the implementation would simply use the TSO mode.

In recent years, greater efforts have been made toward implementing sequential consistency in high performance ways; these are discussed below. The complexity of multithreaded code (regardless of the model) has also led to a re-thinking in the way that memory ordering is defined in the first place. This leads into Transactional Memory.

4.4.4 Implementing Sequential Consistency

The simplest (but slowest performing) way of implementing sequential consistency is to perform loads and stores in program order, and to wait for an acknowledgement from the memory system for each store. The memory system acknowledgement indicates that the store is now visible to all the other processors (i.e., if they perform a load, this is the value they will see). Furthermore, the memory system, which generates the acknowledgements, handles all store instructions in the order received and completes one before beginning the next. This may require waiting for any cache invalidations to take place, (and to be acknowledged), for example. For most implementations, especially those with deep cache hierarchies, this would be ponderously slow. The problem of implementing sequential consistency is somewhat simplified in a bus-based systems, and, through speculation, high performance implementations can be built.

BUS-BASED IMPLEMENTATIONS

In a bus-based system, the bus provides a central ordering point not only for memory coherence but for memory consistency. By relying on this ordering point, implementing sequential consistency in a bus-based system with snooping is fairly straightforward. First, each processor requests the bus (for loads and stores that require the bus) in program order. Second, each cache controller responds to bus requests (for performing local cache invalidations, write-backs, and updates) in the same order as they are received over the bus. Then, a simple sequential consistency implementation performs all loads and stores in program order and, when a load or store needs to use the bus, the processor waits for the bus to be granted before proceeding to the next load or store. The bus grant signal (because of the two implementation features) is essentially equivalent to an acknowledgement for a store instruction. This observation will speed things up a little, say as compared with waiting for an acknowledgement from the memory controller.

However, going beyond the simple implementation, some performance optimizations are possible. To analyze one class of optimizations, consider an implementation where stores that require the bus (for an invalidation, upgrade, or update) are presented to the bus in program order. These instructions are not considered to be "committed" until they have acquired the bus. The bus acquisition points are places where one processor is notifying the others that the data for a memory location for which they may have a local copy is being changed (or may be changed potentially). Then, a processor can execute its load instructions out of order, as long as the value that such a load instruction receives does change by the time it commits (and all preceding stores have committed). The only way a value could change is if some other processor notifies the others that it is potentially changing the value via the bus.

For example, Figure 33a contains an expanded version of the producer-consumer example given earlier. In the sequence for Thread 0, there is an additional load instruction from a new variable B, and an additional load of Flag is added to the sequence at the bottom of the sequence. The sequentially consistent interleaving of the two sequences is shown, and the loads and stores that require bus operations are marked with a " * ". In this example (Figure 33a), the hardware implementation could have performed the Load of B out-of-order with respect to the Store to Flag, and the value of Flag could have been forwarded to the following Load Flag (while the Store was still in the store buffer, prior to acquiring the bus). These two implementation optimizations do not affect the observable results that satisfy the sequentially consistent sequence. The results will be exactly the same as a strictly in-order implementation.

However, we just performed and analyzed the suggested optimizations in retrospect, because we were able to look at the eventual sequentially consistent interleaving to observe that the results would be the same after the optimizations in Thread 0's execution. In practice, at the time the Load B is ready to execute in an out-of-order processor the Store to Flag is waiting in the store buffer, there is no way for the processor to know in advance that there won't be a bus transaction involving B after the Load B is executed. If, as is shown in Figure 33b, there should happen to be a Store B on the bus after the Load B, but before Store Flag acquires the bus, then sequential consistency would be violated.

If this should happen, then the Load to B should not have been allowed to execute out-of-order. One approach to dealing with this situation is to let the Load B pass the Store Flag speculatively, and then, if a bus coherence operation involving B should subsequently occur before the Store Flag gets the bus, the Load B instruction is squashed, the pipeline is flushed (using the ROB mechanism) and the Load B is restarted. In most cases letting the Load B go out-of-order would be a safe speculation, and there would be a net performance gain.

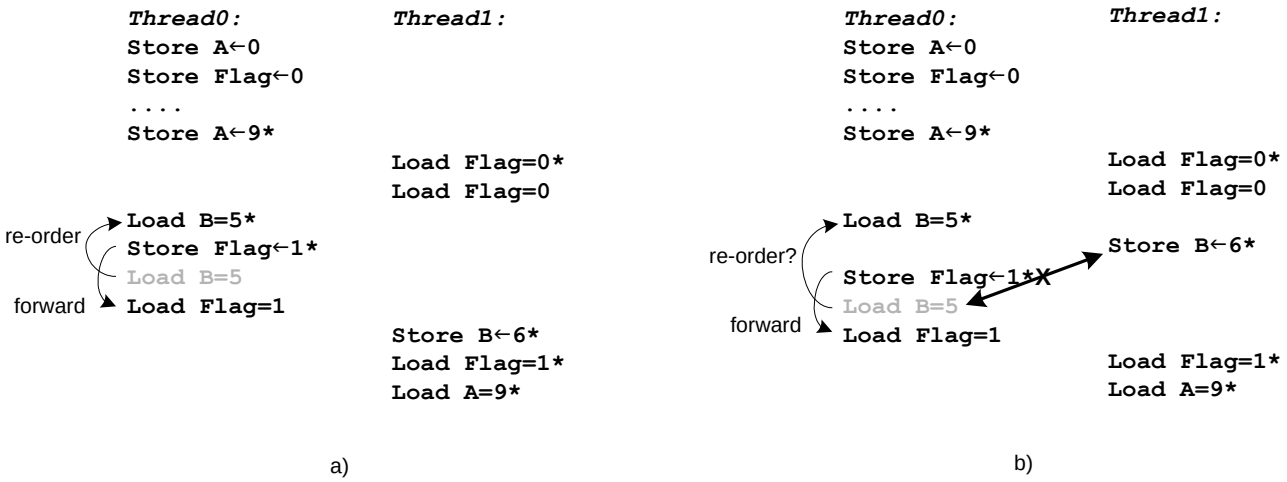


Figure 33. Reordering in an SC implementation a) The Load B instruction passes the Store Flag and Load Flag gets forwarded value from Store Flag.; b) A Store B uses the bus before the Store Flag, causing sequential consistency to be violated.

The above discussion leads us to high performance implementations of sequential consistency that relies on speculation. One such implementation is in Figure 34. In the figure, completed load instructions are shown in a queue; although completed, these loads have not yet retired. These loads are marked as speculative if they have taken load values from the local L1 data cache or from the store data buffer. This load address queue may be implemented separately, as shown in the, but its functionality may be integrated with other load address buffering. In any case, all speculative loads are marked and addresses in the completed load address queue are checked with store addresses (for invalidates, upgrades, or updates) that appear on the bus. When this happens, it may mean that there is a violation of sequential consistency, so the load is marked as "flush". When the load gets to the head of the ROB, the flush bit triggers a pipeline flush, rollback, and restart beginning with the load instruction (see Chapter 3).

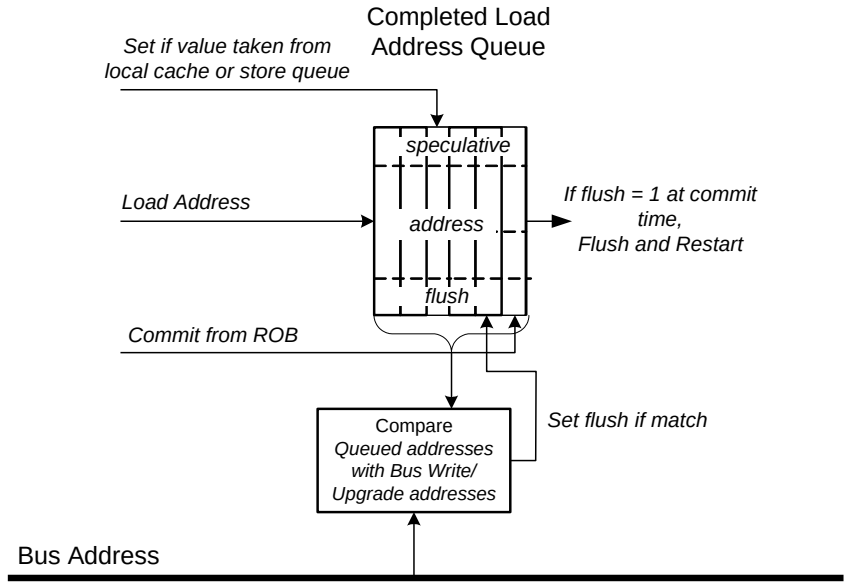


Figure 34. A high performance implementation of sequential consistency.

Returning to the example of Figure 33b, the Load B would be allowed to execute, but would be placed into the Completed Load Address Queue. Then, when the Store B comes onto the bus, its address would match the address of the Load B instruction in the Completed Load Address Queue. This would cause the "flush" bit to be set. Then, at the time the Load B is ready to retire, the flush bit would trigger a flush and restart with the Load B. Then, the second time it executes, it would get the correct (recently stored) value.

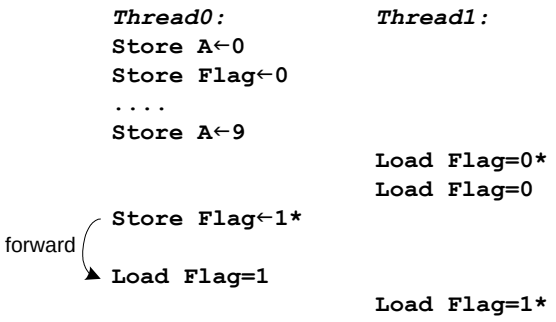


Figure 35. Example of non-speculative forwarding from the store buffer which does not violate sequential consistency.

It isn't always necessary to use speculation to optimize with respect to sequential consistency. In the example shown in Figure 35, where the Load Flag takes its result from the store buffer while the Store Flag is queued up, will always be a safe optimization. That is, the forward can always be done non-speculatively without violating sequential consistency. The reason is that whenever the Store Flag gets the bus, it will get a copy of the corresponding cache line and will write the same value that has been forwarded to the Load Flag. Because the Load Flag is the very next memory instruction after the Store Flag, it can always be placed right after the Store Flag in a sequentially consistent ordering; furthermore, if this is done, the load does not require the bus (it hits in the cache) and therefore is not externally visible. Whether the load literally happens immediately after the store is irrelevant. There is a sequentially consistent interleaving where it *appears* to occur immediately after the store to the rest of the system, and it is the *appearance* that determines whether sequential consistency is satisfied. Note that we are not relaxing sequential consistency in any way; rather this is a case where an *implementation* feature (forwarding) does not affect sequential consistency (which is an *architecture* feature).

DIRECTORY IMPLEMENTATIONS

In a directory system, the distributed nature and the possiblity of multiple memory controllers makes the problem of implementing sequential consistency much more difficult than with a centralized bus. The problem is that there is no single ordering point (outside of the processor's themselves). A straightforward solution, then, is to have a memory controller send an acknowledge to a processor performing a store, and forcing the processor wait for the acknowledge before proceeding with any other loads or stores. When a processor performs a store that requires the memory controller to perform invalidations, then the controller should send out the invalidations and then wait for the acknowledgements before send an acknowledgement to the processor. An optimization is to have the invalidating processors send there acknowledges directly to the waiting processor.

In practice, there have been relatively few systems implementing directory coherence, and as far as the author knows, none of these have implemented sequential consistency. An interesting proposal for implementing sequential consistency is due to Martin and Hill [reference]. In this approach, token coherence, each memory line is allocated a fixed number of tokens and tokens are neither created nor destroyed. A local cache holding a copy of the line may be read from the line only if the cache has

possession of one of the tokens. A line may be written only if the local cache possesses all of the tokens. Then, an implementation of the protocol governs the collection and distribution of tokens as required for a processor load or store. The token coherence protocol is implemented in a manner that also assure sequential consistency.

4.5 Multi-level Cache Hierarchies

Thus far, we have focused more-or-less on single level caches; that is, the discussion has typically assumed that caches communicate directly with the coherence mechanism, whether it is a bus or a memory controller. With multi-level caches, however, this may not be the case. A multi-level cache can have caches associated with the memory side or the processor side. In a sense, the row buffers in DRAM are memory side caches. Here, we are more interested in processor side caches where with memory rather than an L2 cache. An example of such a system is illustrated in Figure X. Here, the L2 caches are directly connected to the coherence bus, and the L1 caches communicate with the bus only indirectly.

Now, maintaining coherence at the L2 caches will not assure coherence among the L1 caches.

Snoop at the level 2 caches. Then, all the accesses must be checked at both the caches. On a heavily loaded bus, this may consume a lot of L1 cache bandwidth. One solution is to use a second set of tags. (duplicates – in essence the tag array becomes two-ported). Another solution is to force the inclusion property. That is, any line the L1 must be in the L2 (or in general, any line in level i must be in level i+1. This means that if there is a snoop miss in the L2, there is not need to even look in the L1's directory. In effect, the L2 filters snoop requests fro the L1.

For this reason many cache hierarchies support the inclusion property. To implement inclusion, the caches must have a certain property. A straightforward implementation then contains back pointers that point from the L2 back to the L1 – give discussion, figure, and example.

Actually multi-level coherence is what is important here.

4.6 Transactional Memory

NOTE

Describe transactional memory with an example, and runtime-based solution in the software chapter. In the ISA chapter observe that there are hardware solutions, hybrid solutions and hardware accelerated solutions. Summarize each and give the jist of it without going into too much detail.

Point out that transactional memory semantics are very unusual at the assembly language level. That is there is an inter-instruction semantic, in effect – like brackets. It is unusual because it implies some implied state. The big problem with hardware implementations, then, is that the instructions between the brackets is essentially unbounded, while all buffering resources within the processor are very much limited.

Give poster child example of a hash table – should probably have been done in the architecture section. And then do the implementation here.

The advantages of transactional memory are clear in examples that use hash table data structures. A very simple example of this type is in Figure X.

```
#define n   10000
int buckets[100], data[n];
main()
{
```

```
input(data);
call histogram (1, n);
}


histogram (m, n);
int m, n;
{
for (i=0; i<n; i++) {
  buckets[a[i+m]]++;
return;
}


#define n   10000
int buckets[100], data[n];
lock_type bucket_lock;
main()
int n, inc, is, incr;
{
if (procid == 0)
     {
     input(data);
     lock_init bucket_lock;
     }
inc = (n+npocs-1)/nprocs;
is = inc*procid;
incr = min( inc, n-is);
call histogram (is+1, incr);
}


histogram (m, n);
int m, n;
{
for (i=0; i<n; i++) {
            lock bucket_lock;
            buckets[a[i+m]]++;
            unlock bucket_lock;
return;
}
```

The most straightforward solution for doing this with multiple threads would put a lock on the bucket, increment the proper bucket,

Cover a simplified version of software transactional memory – ignore object-oriented aspects, GC, nested transactions, etc. Describe data structures and go through an example. The basic idea is that the version of data is recorded when a read is done – the version should not be changed by some other transaction before this one completes. And, the write data is logged for backup – and ownership of the object is taken.

To open for read:

Get transaction record

If transaction record owned by this transaction do nothing

If transaction record owned by another transaction, call contention manager (exponential backoff, abort)

Else -- the record is shared -- log (original) version# into read set

To open for write:

Get transaction record

If unowned, take ownership

Else call contention manager

Log (original) version# into write set

Put copy of original value into undo log

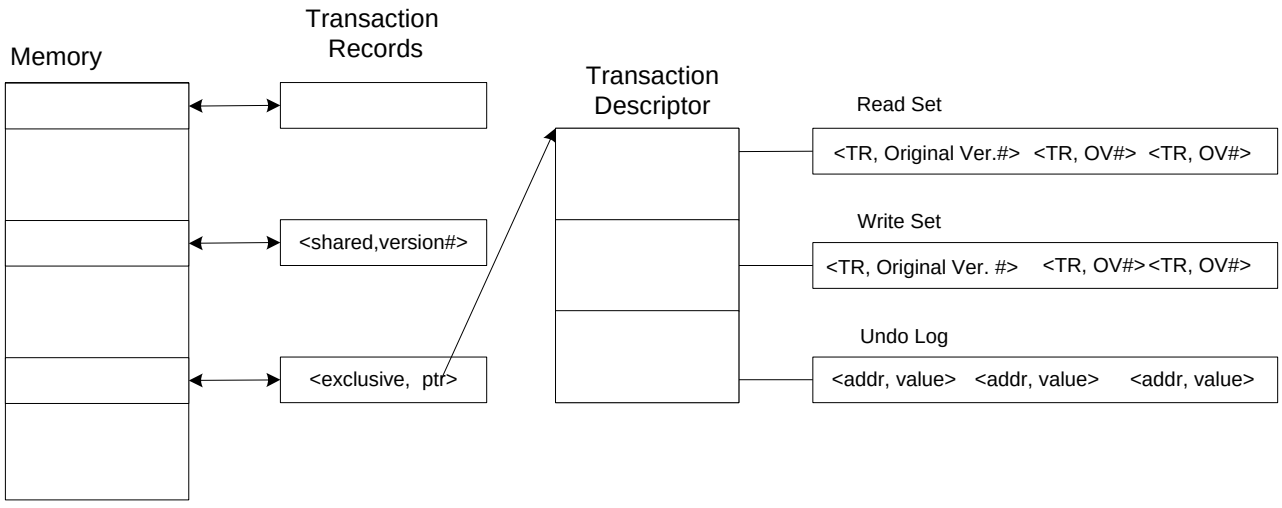
On commit:

Validate read set – check version numbers

If validation fails, then abort (backup with undo log)

Else -- validation succeeds -- update version numbers, release owned transaction records

Notify contention manager



First, look at a simple implementation that buffers – this is similar to the SC implementation given earlier.

Look at software implementation with undo log, read set, write set, versions.

4.7 Virtual Memory and TLB Coherence

4.8 References

1. Adve, S. V. and K. Gharachorloo, Shared Memory Consistency Models: A Tutorial, IEEE Computer, 29(12):66-76, December 1996.
2. Archibald, J., and J.-L. Baer, "Cache coherence protocols: Evaluation using a multiprocessor simulation model," *ACM Trans. Comp. Systems*, pp. 273-298,1986.
3. Atkinson, R. A. and E. M. McCreight, "The Dragon Processor," Proc. 2nd Int. Conf. on Architectural Support for Programming Languages and Operating Systems, October 1987, pp. 65-69.
4. Blundell, C. et al., "Deconstructing Transactional Sematics: The Subtleties of Atomicity," *Fourth Annual Workshop on Duplicating, Deconstructing, and Debunking*, June 2005.
5. Cuppu, V., B. Jacob, T. Mudge, "A performance comparison of contemporary DRAM architectures," 26th Int. Symp. Computer Architecture, May 1999, pp. 222-233.
6. Gharachorloo, K. et al., Specifying System Requirements for Memory Consistency Models, Computer Sciences Technical Report #1199, University of Wisconsin, Madison, December 1993. Also available as Technical Report #CSL-TR-93-594, Stanford University.
7. Gupta, A. and Wolf-Dietrich Weber, Cache Invalidation Patterns in. Shared-Memory Multiprocessors, I*EEE Transactions on Computers*, July 1992.
8. Gupta, A., W.-D. Weber, and T. Mowry, Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes, *Proc. 1990 Int. Conf. on Parallel Processing*. pp. 1:312-321, 1990
9. Hammond, L. et al., "Programming with Transactional Coherence and Consistency (TCC)", *Proc. ASPLOS,* October 2004.
10. Hammond, L. et al., "Transactional Memory Coherence and Consistency," *Proc. International Symposium on Computer Architecture,* June 2004.
11. Hill, M. D., "Multiprocessors Should Support Simple Memory Consistency Models," *IEEE Computer*, Aug. 1998, pp. 28-34.
12. Hur, I. and C. Lin, "Adaptive History-Based Memory Schedulers for Modern Processors," IEEE Micro, Jan.-Feb. 2006, pp. 22-29.
13. Jacob, B. and D. Wang, "DRAM: Architectures, Interfaces, and Systems," DRAM Tutorial, 2002 Int. Symp. on Computer Architecture, June 2002.
14. Lamport, L., "How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs," *IEEE Trans. on Computers*, September 1979, pp. 690-691.
15. Lenoski, D. et al. The Stanford Dash Multiprocessor, *IEEE Computer*, pp. 63--79, March 1992.
16. Marty, M. R. and M. D. Hill, "Coherence Ordering for Ring-based Chip Multiprocessors," 39th Int. Symp. on Microarchitecture, Dec. 2006, pp. 309-320.

17. Natarajan, C. et al., A Study of Performance Impact of Memory Controller Features in Multi-Processor Server Environment. WMPI 2004, pp. 80-87.

18. Natarajan, C., B. Christenson, and F. Briggs, "A Study of Performance Impact of Memory Controller Features in Multi-Processor Server Environment," Proc. of the 3rd Workshop on Memory Perf. Issues, June 2004 pp. 80-87.

19. Nesbit, K. J. et al., "Fair Queuing Memory Systems," 39th Int. Symp. Microarchitecture, Dec. 2006, pp. 208-222.

20. Rajwar, R. et al., "Virtualizing Transactional Memory," *Proc. International Symposium on Computer Architecture,* June 2005.

21. Rixner, S., "Memory Controller Optimizations for Web Servers," 37th Int. Symp. on Microarchitecture, Dec. 2004, pp. 355 – 366.

22. Rixner, S., et al., "Memory Access Scheduling," 27th Int. Symp. on Comp. Arch., June 2000, pp. 128 – 138.

23. Saha, B., A.-R. Adl-Tabatabai, Q. Jacobson, "Architectural Support for Software Transactional," 39th Int. Symp. on Microarchitecture, Dec. 2006, pp. 185-196.

24. Steinberg, U., Parallel Architectures: Memory Consistency and Cache Coherency, Dresden Technical University, 2005.

25. Sweazey, P. and A. J. Smith, A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus, *Proc. Thirteenth International Symposium on Computer Architecture,* June 1986.

26. Thacker, C. P., L. C. Stewart, and E. H. Satterthwaite Jr., " Firefly: A Multiprocessor Workstation," IEEE Transactions on Computers, Aug. 1988, pp. 909-920.

27. Black, D. L., et al., "Translation Lookaside Buffer Consistency: a Software Approach," *Proc. ASPLOS-III,* pp. 113-112, 1989.

28.