direct-mapping cache memory. Another reason is that the difference between the miss ratios of direct-mapping and set-associative cache memories diminishes as the size of the cache memories increases. Therefore, set-associative mapping is preferred for small cache memories, while direct-mapping is preferred for large cache memories.

4.7.8. Cache Memory Coherence

In shared-bus microprocessor systems, cache memories have an important role in reducing the traffic on the shared system bus. Each processor has its own cache memory with one or two levels, which forms a local memory and allows the processor to access instructions and data without using the system bus. With an independent cache memory for each processor, there is a possibility for two or more cache memories to contain different versions of the same information in the same time. This represents the *cache-memory coherence problem*.

This problem cannot be solved entirely by using the write-through policy. As discussed in Section 4.7.3, this policy causes both the cache and main memory to be updated whenever a memory write operation occurs. Consider, for example, that one processor updates variable X in both its cache and main memory. If another processor changes the variable X, the new value of X will be written into main memory, but the two cache memories will contain different values for X. Subsequent read operations from these cache memories can lead to inconsistent results. To assure coherence, a mechanism is needed that informs each cache memory about changes of the shared data stored in other cache memories.

The cache-memory coherence problem can be solved with either hardware or software methods. One software method is to mark information during program compilation to indicate that they either can or cannot be stored in cache memory (cacheable or non-cacheable, respectively). All shared items which can be written are marked as non-cacheable, meaning that they can be accessed only from main memory. The coherence can then be assured by a write-through policy that requires a processor to mark a shared data X in its cache memory as invalid whenever the processor writes into X. When the processor accesses X again, it is forced to access the main memory instead of the cache memory, thereby always obtaining the most recent version of X. This solution can significantly degrade system performance. Invalidation also forces the removal of the data from the cache memory, which increases the main-memory traffic.

Hardware methods of maintaining cache-memory coherence offer the advantages of higher speed and program transparency, but they are more expensive. One possible solution is for a processor to broadcast its write operations to all cache memories and the global memory via the shared bus. Every cache-memory controller in the system then examines its assigned addresses to determine if the requested data resides presently in its cache memory. If it is, the corresponding cache memory block is either updated or marked as modified. The disadvantage of this technique is that every write operation into a cache memory forces all cache memories to check if they contain the particular data.

A less costly hardware method is called *cache-memory snooping*. With this method, each processor is equipped with circuits to continuously monitor the system-bus activity in order to detect references by other processors to memory addresses currently in its cache memory. The processor can also signal other processors that it has a copy of the referenced data and, when necessary, can delay the other processor's accesses to main memory. If processor P_2 attempts to read the data with an address that is currently assigned to P_1's cache memory, P_1 detects this attempt, called *snoop read hit;* similarly, there is a *snoop write hit*. On detecting a snoop hit, P_1 determines whether real or potential incoherence exists and then performs appropriate operations to eliminate it. The following situations are typical:

- Suppose that processor P_1 detects a snoop read hit when the copy in its cache memory of the requested word is modified, and P_1 has not yet updated the main memory (this situation can only occur when the write-back policy is used). Processor P_1 signals P_2 to suspend its read request while P_1 updates main memory by writing back the block containing the requested word. Then processor P_1 signals P_2 to complete its memory read operation.

- If processor P_1 detects a snoop write hit, it knows that the copy in its cache memory of the requested word is about to be modified. Processor P_1 therefore marks the copy as modified. The next time processor P_1 tries to read the same word, a cache miss occurs that forces P_1 to read a valid copy from main memory.

Another response to a snoop write hit by processor P_1 is to capture the new data on the system bus as processor P_2 writes it to global memory. Processor P_1 can then use the captured data to update its cache memory.

To maintain consistency in a multiprocessor system, or in a uniprocessor system with independent I/O processors, a cache-memory controller must keep track the state of each cache-memory block under its control. For this purpose, the controller attaches a few state bits to every block stored in the cache data memory, and processes the states according to some coherence algorithm or protocol. Processors such as Pentium and PowerPC employ a standard cache-memory coherence protocol called the MESI *coherency protocol*. This protocol is based on the following four states:

- *M* (*Modified*): The block has been modified by a recent write hit to the cache memory.

- *E* (*Exclusive*): The block has not been modified, so it is the same as the copy in main memory, and no other processor has a copy of the block.

- *S* (*Shared*): The block has not been modified, but other processors may have a copy of the block.

- *I* (*Invalid*): The data in the block is not valid.

Figure 4.48 presents a simplified version of the MESI protocol, showing how the states of a cache-memory block change in response to various read and write conditions, assuming that a write-back policy and a cache-memory snooping mechanism are used. A one-level cache memory is assumed, although this protocol also works well with multiple cache-memory levels.

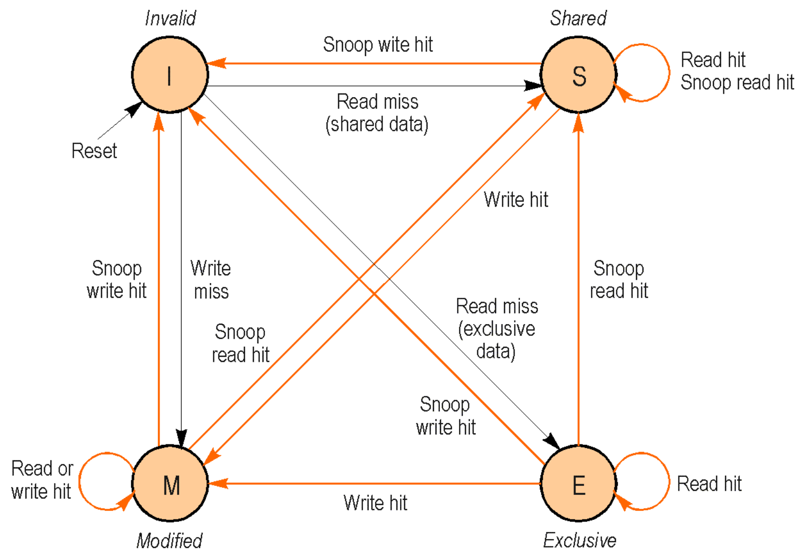

Figure 4.48. Simplified state-transition graph for a cache-memory block using the MESI coherence protocol.

First consider the effect of read operations on the state of a cache-memory block. Read hits to the block do not change its state, as opposed to read misses. When a processor P_1 initially tries to read the cache memory, the cache-memory controller changes all block states to *I* (invalid) and forwards the read request to main memory. Thus *I* acts as a reset state that triggers a block transfer to the cache memory; the loaded block's state is set to *E* (exclusive) if no other processor has a copy of the same block. If during P_1's read operation a snooping processor P_2 signals via the shared bus that its cache memory has a copy of the same block which has not been modified, the state of the block in P_1's cache memory is set to *S* (shared) instead of *E*. If, on the other hand, processor P_2 signals that its cache memory has a modified copy of the same block, the cache memories are no longer coherent. To resolve this incoherence, the signal from processor P_2 causes processor P_1 to postpone its memory read and to relinquish the system bus. Processor P_2 then assumes the role of bus master and writes its modified block into main memory. Processor P_2 also changes the state of its copy of the cache-memory block from *E* to *S* because the block is now shared. This state change is specified by the transition from *E* to *S* marked "snoop read hit" in Figure 4.48. Finally, the first processor P_1 repeats its main-memory read request and obtains a copy of the block, which it marks as S.

In the case of a write hit to P_1's cache memory, if the target block is in either of the states S or E, the block's state changes to M (modified). In the case of state S, processor P_1 signals to other processors that it is performing a write operation to a shared block, and they respond by marking their copies of the shared block as I (invalid). The modified block remains in the M state in P_1's cache memory during subsequent read and write operations.

A write miss to P_1's cache memory triggers a main-memory read operation that replaces the target block in the cache memory, where it is marked with the M state. If some other processor P_2 has a copy of the same block which has not been modified (it is in the state S or E), processor P_2 changes the state of its copy to I. If processor P_2 has a modified copy of the block, P_2 sends a signal to processor P_1, causing the latter to delay its memory read. Processor P_2 then takes control of the system bus and writes its modified block into main memory; processor P_2 also changes the state of its copy, since it knows that the shared block in main memory is about to be changed by processor P_1. Control of the bus is then returned to processor P_1, which completes its block transfer.

4.8. Virtual Memory

4.8.1. Principle of Virtual Memory

Virtual memory allows to use a much larger memory than the actual physical memory. In a virtual memory system, the main and secondary memories appear to a user program like a single, large, and directly addressable memory.

Prior to the appearance of virtual memory, if a program's address space exceeded the size of the available memory, the programmer was responsible for breaking up the program into smaller fragments called *overlays*. Each overlay than could fit in main memory. All these overlays were stored in secondary memory, such as on a disk, and individual overlays were loaded into main memory as they were needed. This process required knowledge of where the overlays were to be stored on disk, knowledge of input/output operations required for accessing the overlays, and keeping track of the entire overlay process. This was a very complex process that made the complexity of programming a computer even more difficult.

The concept of virtual memory was created primarily to relieve the programmer of this task. Virtual memory allows the user to write programs that exceeds the bounds of physical memory and still execute properly. It also allows for multiprogramming, by which main memory is shared among many users on a dynamic basis. With multiprogramming, portions of several programs are placed in the main memory at the same time, and the processor switches its time among these programs. The processor executes one program for a brief period of time (called a *quantum* or *time-*