7.8.1 Cache coherence

Caches, as we have described in Part I, are used in single processor systems to alleviate the speed differential between the main memory and the processor, a differential which is widening as techniques such as superscalar operation increase the speed of operation of processors. The differential in speeds of processors and shared memory also exists in a shared memory multiprocessor, and it is quite natural to apply caches to a shared memory multiprocessor system. Using cache memory is a natural extension of using a cache in a single processor. However, there are significant additional factors to consider in using cache memory in a multiprocessor environment, in particular maintaining accurate copies of data in the multiple caches in the system. In a single processor system with a cache, it is generally necessary to maintain the data in the cache the same as that in the main memory because the main memory may be accessed by an input/output processor directly and would expect to obtain the most recent values. Of course if the processor is the only device that can access the data, the copy in the main memory could be left inconsistent, or the data not even held in the main memory at all when in the cache. In a multiprocessor system with more than one cache, it is possible that copies of the same data are held in more than one cache. Hence in addition to inconsistency between the main memory and the caches, there could be inconsistency between caches. In all cases, reading different copies of the same data does not generally cause a problem. A complication only exists if individual processors alter their copies of data, because shared data copies should generally be kept identical for correct operation. Maintaining the data in all the caches the same is known as *cache coherence*. The phrase *cache consistency* is also used.

Why would different processors wish to access the same data and why this must data be kept the same? First of course, the parallel algorithm might call for operations to be performed on different data items by different processors. For example, a sorting algorithm might call for swapping different pairs of items at different times by different processors. Second, and closely related to shared data access, a shared item might be accessed by different processors to maintain synchronization (locks and semaphore variables).

All shared memory multiprocessor systems can be described by a set of processors connecting to a set of memory modules through an interconnection network – different systems are characterized by the choice of interconnection network (single bus, multiple bus, multistage interconnection network, hierarchical interconnection network, etc.) An appropriate place for each cache is attached to a processor, as shown in Figure 7.36. It would also be possible to place the caches in front of each memory module, but this arrangement would not decrease the interconnection traffic and contention, and would in effect share each cache with those processors wanting accessing data held in the corresponding main memory. In addition, the delay of the interconnection network would be included in the cache access time.

As we have seen in Chapter 3, there may be two levels of caches, a first level within the processor and a larger second level external cache. In the following we assume that the caches are external second level caches. If the first and second caches are maintained such that all the items in the first level is contained in the second level and the values are

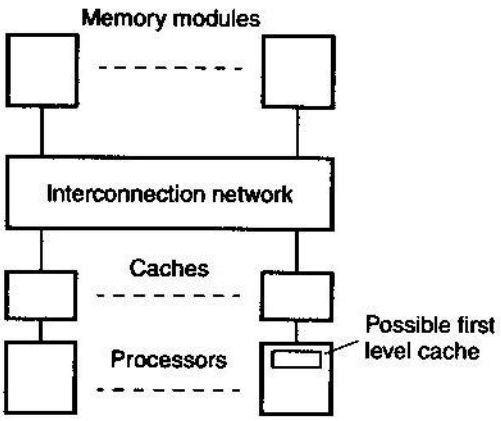

Figure 7.36 Shared memory multiprocessor system with caches

kept the same, then maintaining cache coherence between the second level caches in the system should be sufficient.

7.8.2 Write policy

Write-through is not sufficient, or even necessary, for maintaining cache coherence, as more than one processor writing-through the cache does not keep all the values the same and up to date. Figure 7.37 shows the effect of write-through. Initially a shared variable, x, is stored in the shared memory. Suppose processor 0 reads variable x. A copy of x moves into the cache of processor 0. Suppose processor 1 now reads variable x. A copy of x moves into the cache of processor 1 and the situation is as Figure 7.37(a). Now suppose processor 1 writes to location x. We obtain Figure 7.37(b). After write-through, we obtain Figure 7.37(c) and still inconsistency between caches. Now even though the main memory and the cache of processor 1 are consistent, the cache of processor 0 still has the old value for x.

There are two solutions to this situation, either:

1. Invalidate copy in the cache of processor 0, or
2. Update copy in the cache of processor 0

both of which require access to the cache of processor 0 as shown in Figure 7.37(d). Invalidation of the copy of x held in the cache of processor 0 is done by resetting the valid bit associated with x in the cache. Now processor 0 must access the main memory if it references x again, to bring a new copy of x back into its cache. If copies existed in caches apart from the cache of processor 1, these copies would also need to be invalidated. The alternative to invalidation would be to update all cached copies with the new

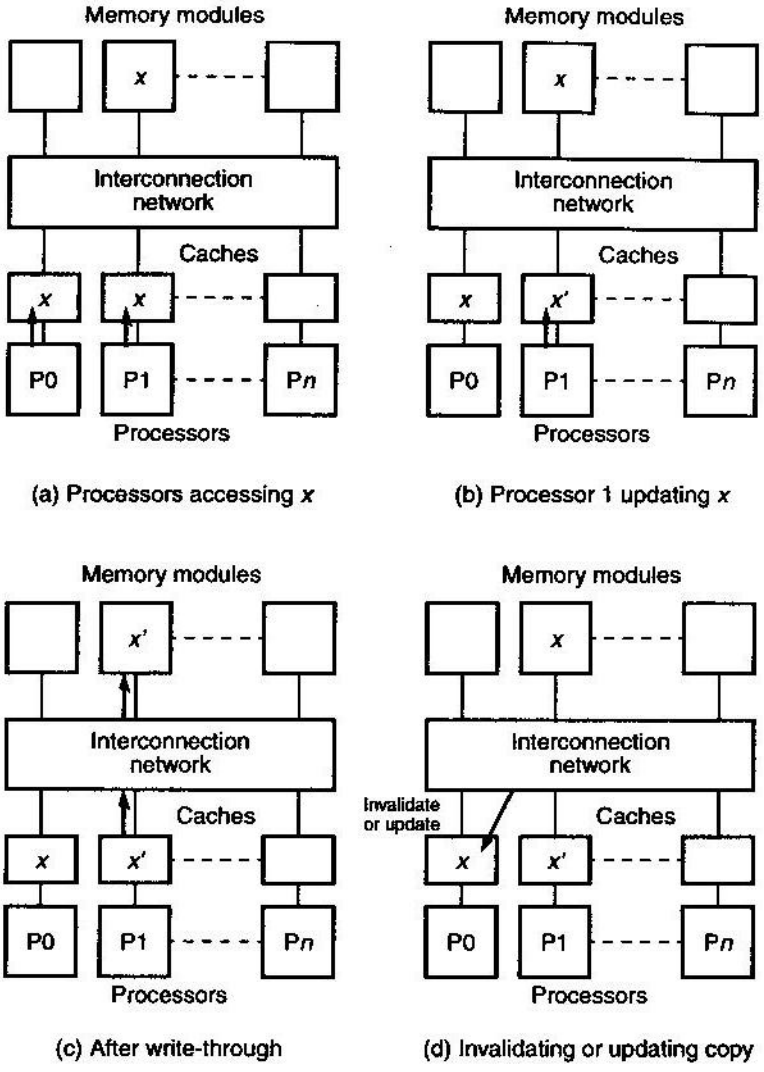

Figure 7.37 Inconsistency with write through policy copy

value of x. This option is not usually implemented because of the overhead of the update. In any event, it may be not completely necessary because not all processors may access the location again. With invalidation, write back may be practiced rather than write-through to reduce the memory traffic. Then there is only one valid copy in one cache, and one processor has *ownership* of this copy.

There are numerous variations of invalidate and update protocols developed in the research community, see Flynn (1995) for further details. We will review one protocol, MESI used with bus-based microprocessor systems, and directory methods used also with other types of interconnection networks later.

False sharing in multiprocessors with caches

In all protocols, access to one location in the line of a cache is considered access to the line. *False sharing* is the term used to describe the situation when more than one processor accesses different parts of a line but not the actual data items. False sharing can result in significant reduction in performance because, in maintaining cache coherence, the smallest unit considered is the line. False sharing can be reduced by distributing the data into different lines if sharing is expected. This would be a task for the compiler, and requires both knowledge of the use of the data and the architectural arrangements of the caches.

Ordering of references

Usually we would expect that when a processor reads a shared variable, it obtains the value produced by the most recent write to the shared variable, irrespective of the processor that did the write operation. This is known as *strict consistency*. We always assume that instructions are executed in program order.[1] Given more than one program executed by separate processors, we generally also assume that the overall effect of the parallel programs is not changed by any arbitrary interleaving of instruction execution in time. This assumption is know as *sequential consistency* (Lamport, 1979). Most shared memory multiprocessor systems and their caches conform to the sequential consistency model, and most parallel programming languages have this implicit assumption. With additional constraints on the programming, it is possible to relax these consistency assumptions and potentially improve the performance of the system. Since access to shared data items is usually controlled through critical sections and other synchronization constructs, it is reasonable to insist on using such constructs for accessing shared data. Only the access to shared variables (locks and semaphore variables) need be sequentially consistent if we force access to all other shared variables through critical sections. This is known as *weak ordering* (Dubois, Scheurich and Briggs, 1988[2]; Adve and Hill, 1990). In weak ordering, accesses to global synchronization variables are sequentially consistent. No access to a synchronization variable is issued before all previous global data accesses have been performed, and no access to global data is issued before a previous access to a synchronization variable has been performed.

---

[1] This restriction is relaxed within a superscalar processor for internal functions or read/write operations on internal registers – with significant hardware complexity to detect data dependencies. Detecting data dependencies on data stored externally in main/cache memory is impractical – hence our assumption is reasonable and not restrictive.

[2] See this reference for a formal definition of *strong ordering* which is similar to sequential consistency.

It is necessary to ensure that during any cache coherence protocol (invalidate or update) processors are not allowed to access locations before coherence has been established, i.e. sequential consistency must be maintained.

7.8.3 Methods of achieving cache coherence

Several possibilities exist to maintain *cache coherence*, in particular:

1. Shared caches.
2. Non-cacheable items.
3. Broadcast write mechanisms.
4. Snoop bus mechanism.
5. Directory methods.

Clearly, a single *shared* cache shared by all processors with the appropriate controls would maintain cache coherence. Also, a shared cache might be feasible for DMA devices accessing the cache directly rather than relying on a write policy. However, with several processors the performance of the system would seriously degrade, due to contention, and a shared cache is not a viable option.

Non-cacheable items

Cache coherence problems only occur on data that can be altered. Such writable data could be maintained only in the shared main memory and not placed in the cache at all (i.e. *non-cacheable* items). It is difficult to make arbitrary memory locations non-cacheable, but it is easy to provide hardware to completely disable a specific cache. It is also possible to make a specific cache line non-cacheable. When the processor accesses a line and gets a miss, the data word is accessed in main memory but the line is not brought into the cache. Of course this approach would cause all locations in the main memory having the same index in the cache to be non-cacheable. Non-cacheable items are probably best done at the virtual memory translation stage, by having specific pages non-cacheable.

Since writing usually only applies to data, caches can be separated into a code cache and data cache as we have seen in Chapter 3. Our cache coherence problems really only need to deal with data, ignoring the possibility of self-modifying code – which unfortunately could not be ignored by Intel in developing compatible upgrades in their 8086 family. Motorola with their MC68000 was in a better position when they clearly separated data addressing and code addressing modes and introduced a *data (alterable) addressing* mode, but still software problems have occurred with the MC68040 (Feldman and Retter, 1994). Note that code need not be self-modifying to cause problem; simply interleaving code and data so that both could exist in the same cache line creates the same situation.

Broadcast writes

In *broadcast writes*, every cache write request is also sent to all other caches in the

system. Such broadcast writes interrogate the caches to discover whether each cache holds the information being altered. The copy is then either updated (*update* write) or invalidated (*invalidate* write) by resetting the valid bit associated with the line as described earlier. The use of invalidating words is generally preferable. In any event, significant additional memory transactions occur with the broadcast method, though it has been implemented on large computer systems.

Snoop bus mechanism

Though broadcast methods would be expensive to implement with a multistage interconnection network, there is one interconnection network which can achieve broadcast fairly efficiently, namely the single bus network found in many microprocessor systems. In the *snoop bus* mechanism, a bus watcher unit with each processor/cache observes the transactions on the bus and in particular monitors all memory write operations. Figure 7.38 shows the general hardware arrangement. If a location in main memory is altered and a copy exists in the cache, the snoop bus controller invalidates the cache copy by resetting the corresponding valid bit in the cache. This action requires the unit to have access not only to the valid bits in the cache, but also to the tags in the cache, or copies of the cache tags, in order to compare the main memory address tag with the cache tag. Alternatively, the cache line with the same index as the main memory location can be invalidated, whether or not the tags correspond. The unit then does not need to access the tags, though access to the valid bits is still necessary. However, the unit might mark as invalid a cache line which does not correspond to an altered main memory word, because the cache location with the same index as the main memory location would be invalidated, irrespective of the values of the tags.

In the snoop bus protocol, sufficient time must elapse for the signals to stabilize along the bus. This might require several bus propagation delays along the bus as the signals settle down. With increasing processor clock frequencies, the bus propagation delay times become more significant, and ultimately limit the speed of the cache coherence protocols.

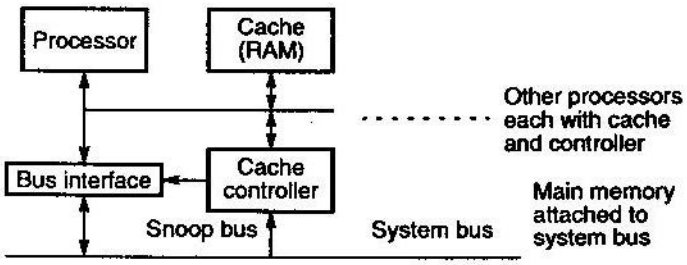

Figure 7.38 Cache with snoop bus controller

Mathematical performance analysis of seven different multiprocessor cache coherence techniques for single bus systems is given by Yang, Bhuyan and Liu (1989). Perhaps the most popular snoop protocol with microprocessor manufacturers is the four-state MESI (Modified/Exclusive/Shared/Invalid) invalidate protocol, which is based upon Goodman's write-once 4-state protocol (Goodman, 1983). The MESI protocol can be found in the internal data cache of Intel Pentium, the second level Pentium cache controller, the Intel 82490 (Intel, 1994c), the Intel i860 processor (Intel, 1992b), and Motorola MC88200 cache controller (Motorola, 1988b), among others. One version of the MESI protocol has a "write-once" transition. We will describe the protocol without this transition as applied to a multiprocessor situation. The variation with the write-once transition is also appropriate for a single processor system to maintain first and second level caches consistent.

In the MESI protocol, each line in the cache can be in one of four states:

1. Modified (exclusive) – The line is only in this cache and has been modified (written) with respect to memory. Copies do not exist in other caches *or in memory*.
2. Exclusive (unmodified) – The line is only in this cache and has not being modified. It is consistent with memory. Other copies do not exist in other caches.
3. Shared (unmodified) – This line potentially exists in other caches. It is consistent with memory. To stay in this state, access to line can only be for reading.
4. Invalid – This line has been invalidated and does not contain valid data.

Two bits can be associated with each line to indicate the state of the line. The modified (exclusive) and exclusive (unmodified) states are used to indicate that the processor has the only copy of the cache line. In the modified (exclusive) state, the processor has altered the contents of the line from that kept in the main memory and hence a valid copy does not even exist in the main memory. It will be necessary to write back the line before any other cache can use the line. Lines enter the invalid state by being invalidated by other processors, i.e. this is an invalidate protocol.

A transition from one state to another state will depend upon the current state, whether the processor is reading the line or writing to the line, a hit or miss occurs, and whether access is to a shared line. The major transitions are shown in the state diagram in Figure 7.39 assuming fetch-on write is practiced. Two types of transition are indicated, those initiated by the local processor, and those initiated by a remote processor's activity on the bus when a shared line is accessed.

Suppose a processor makes a read request for a line not in the cache (read miss). The request will be broadcast along the bus. If no other cache holds a copy of the line, no complaints will be forthcoming from remote processors (actually their snoop bus controllers). The main memory will be accessed to load the line into the cache and the line enters the exclusive state. If a copy of the line did exist elsewhere, a response would be obtained from a remote processor indicating access to a shared line. The line would enter the shared state, and all other copies of the line will enter the shared state irrespective of their initial state (modified, exclusive, or shared).

If the processor had made a write request for a line not in the cache (write miss), the line has to be read from the main memory and modified before loading into the cache. (It

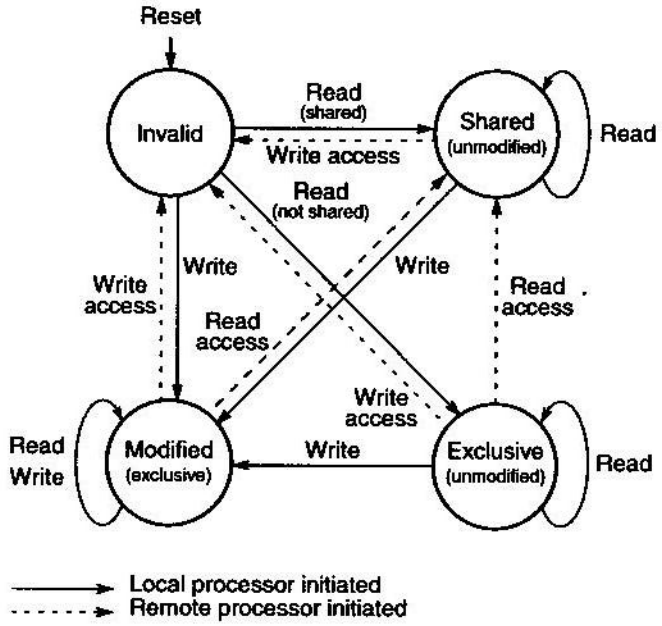

Figure 7.39 MESI protocol – major transitions without write-once

may be that only part of the line is altered, say a single byte.) Before the processor can load the line, it has to establish that the main memory holds a valid copy, which it may not because a copy may exist in a remote cache in the modified state. A generated sequence called "read with intent to modify, RWITM" is made. If a copy of the line does exist in another cache in the modified state, a remote processor will interrupt the RWITM sequence and write the line back to the main memory. The state of the remote copy will be changed to the invalid state. Then the the RWITM sequence can resume and memory accessed again to obtain the updated line. The final state of the line is the modified state in which no other copy exists even in the main memory. If a copy of the line had existed in another cache but not in the modified state, this copy is simply invalidated and the memory access can continue immediately.

All read hits do not alter the state of the line. If a processor makes a write reference to an existing line in its shared state, it will broadcast the request on the bus to inform the other caches, update its line, and change the state of the line to the modified state. All other existing copies will enter the invalid state. If a processor makes a write reference to a line in its exclusive state, all it needs to do is update the line and change its state to the modified state since there are no other copies of the line.

Figure 7.40 shows a typical sequence of events between two processors, 1 and 2, making requests for a location x. (Access to any location in a line in the cache is considered access to the whole line and it does not matter which location in the line is accessed;

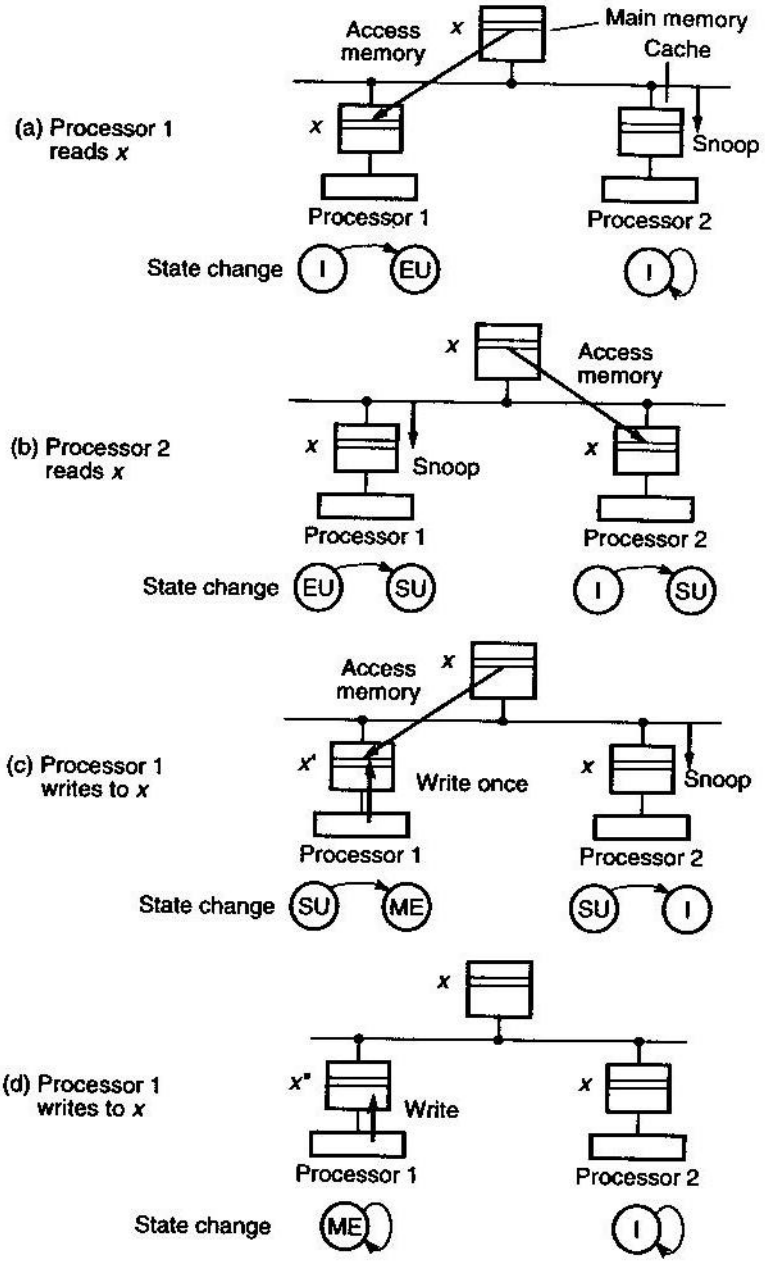

Figure 7.40 A sequence of MESI protocol state changes (I = Invalid, SU = Shared – unmodified, EU = Exclusive – unmodified, ME = Modified – exclusive)

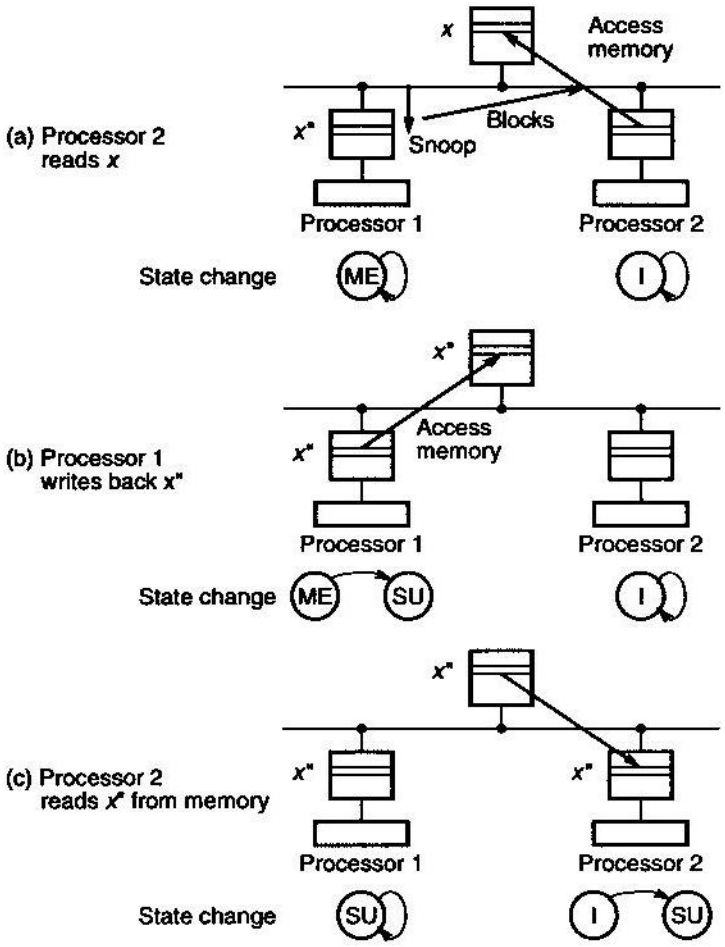

Figure 7.41 MESI protocol state changes from exclusive ownership to shared

the effects are the same.)

Figure 7.41 illustrates the steps when processor 2 tries to read location line x". It obtains a read miss, and attempts to access the main memory. Processor 1 is snooping the bus and recognizes the access to a location which it has exclusive ownership, and blocks processor 2's read operation. Then it writes x" back to the main memory and releases processor 2 to retry accessing the main memory. Processor 2 now obtains x" which it loads into its cache. The state of both cache lines changes to shared state.

In the write-once version of the MESI protocol, the state transitions are slightly

different to that described in Figure 7.39. All read misses cause a change to the shared state. The next write hit on the line causes a change to the exclusive state (the so-called write-once transition). The next write hit causes a change to the modified state. Full details of this protocol can be found in the description of the Motorola MC88200 cache controller (Motorola, 1988b).

Directory methods

Directory methods are particularly suitable for shared memory multiprocessor systems with multistage or hierarchical interconnection networks where broadcast mechanisms would be expensive to implement. A directory is a list of entries identifying cached copies of shared memory locations. The directory is used after a processor writes to a cached location, to invalidate (or update) copies in other caches.

The shared memory directory has one entry for each shared memory location that can be cached. Each entry has pointers to cached locations (lines). Pointers to the processors/caches are sufficient. In addition, a single bit is provided with each entry to indicate whether the location is "dirty" or "clean" , i.e. whether the location has been modified or not. We share call this bit D (= 1 if dirty). This bit will be used to establish whether any processor can write to the location. Bits are attached to the lines in the caches indicating whether the processor has permission to write to the line (generally whether the line is shared or not). There are three main ways of implementing the directory, the full directory, limited directories, and chained directories:

1. A full directory

In a full directory, the directory is centralized in the main memory, and provides for the capability of pointing to all caches. Hence with n processors, there would be n pointers for each entry. This can be implemented efficiently with a single bit for each processor. Each bit is set to a 1 if the corresponding cache has a copy of the data, otherwise the bit is a 0. Figure 7.42 shows a full directory. In this figure, a copy of location x is held in each cache. Each line in the caches is provided with two "state" bits, a *valid bit* which is set to a 1 if the information in the cache is valid, and a *private bit* which is set to a 1 if the processor is allowed to write to the line.

Suppose processor 2 writes to location x. At this time, processor 2 does not have permission to write to this location. A request to memory is made to write to this location, and processor 2 waits for permission to proceed. An invalidate signal is issued to all caches holding copies of x. Each cache receiving this invalidate signal sets the corresponding valid bit to 0, and sends back an acknowledgement signal to the memory. After the acknowledgement signals are received, the memory module sets the dirty bit to 1, and sends a signal to processor 2 giving it permission to write to location x. At that time, processor 2 may proceed to write to location x in the cache (and write-through to the memory assuming write-through is implemented). This protocol ensures sequential consistency.

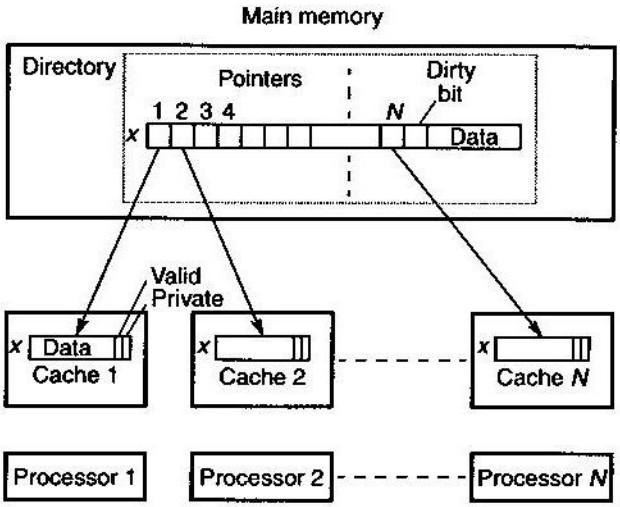

Figure 7.42 Shared memory cache protocol using a full directory

2. Limited directories

A significant problem with the full directory method is the overhead of the directory entries. Each location needs an extra $N + 1$ bits for N processors, and hence the complexity grows linearly with increasing system size. The full directory method allows all shared memory locations to have copies in all caches. However it is quite unlikely that all caches would need a copy of every cached location. It is more likely that only a few copies actually exist at any instant. In the limited directory method, only a fixed number of caches may have copies of any particular location, say n copies. Therefore only n pointers are provided in the directory where $n < N$. Now each pointer must uniquely identify a cache, and $\log_2 N$ bits will be needed for each pointer rather than a single bit, $n \log_2 N$ bits are needed for each entry rather than N bits. Moreover, the limited directory will scale much better than the full directory because the size of the directory grows less than linearly with increasing system size, for any constant value of n.

When more than n copies are requested, a decision has to be made as to which copies are to be maintained, and which are to be invalidated. Pointers are altered to point to those copies which are to be maintained, "evicting" other copies.

Choosing the value for n is analogous to selecting the working set in a paging system. We want a value for n which captures a sufficient number of copies for application programs to execute efficiently. Preferably some mechanism should be in place to cope with more than n copies. One method is to use a broadcast mechanism to invalidate copies when more than n copies are requested. This would be suitable for networks which can support broadcast mechanisms. The reader is directed to Agarwal *et al.* (1988) for more details for such mechanisms.

3. Chained directories

The chained directory approach also attempts to reduce the size of the directory. In the chained directory, a linked list is used to hold the directory entries. Two possibilities exist for implementing the linked list:

a. Single linked list.
b. Double linked list.

A single linked list is shown in Figure 7.43. Here the shared memory directory pointer points to one copy in a cache. A pointer here points to the next copy of the location. The last location in the linked list is terminated with a special symbol. The chained directory allows N copies of a location to be maintained. Whenever a new copy is called for, the linked list must be broken and pointers altered. For example, suppose processor 5 wishes keep a copy of x. The pointer from the shared memory is altered to point to cache 5, and the pointer in cache 5 is set to point to cache 2. The memory must send the pointer to cache 2 to cache 5, together with the data. Only when the whole structure has been established, will processor 5 be allowed to access location x. If a processor writes to a location, an invalidate signal must be sent down the chained path. The chain must also be altered when a location is removed from a cache. A double linked list maintains both backward and forward pointers which enables a location to be inserted and deleted more efficiently although it requires more pointers.

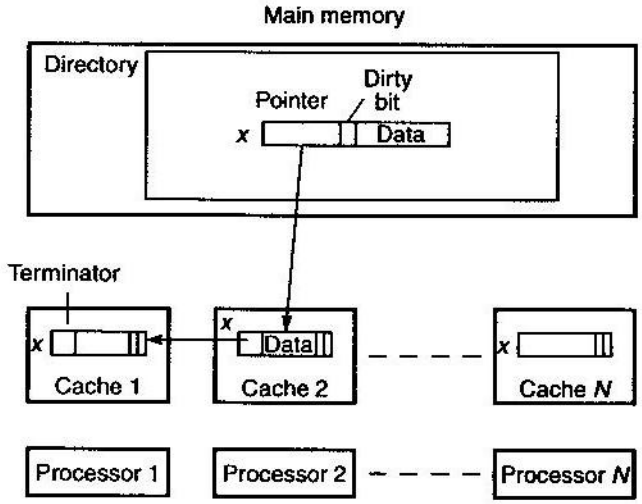

Figure 7.43 Shared memory cache protocol using a chained directory

Apart from hardware solutions to cache coherence, software approaches are possible. The reader is directed to Cheong and Veidenbaum (1990) for further details. We should mention that a shared memory multiprocessor system also is likely to have a paging mechanism, and multiple TLBs (translation look-aside buffers). These are essentially cache memories and need to be maintained consistent. The reader is directed in the first instance to Teller (1990) for details of how multiple TLBs can be maintained consistent.

PROBLEMS

7.1 Prove that the maximum speed-up of a multiprocessor system having n processors, in which each processor uses the bus for the fraction m of every cycle, is given by m.

7.2 Identify the relative advantages of the synchronous bus and the asynchronous bus.

7.3 Identify the relative advantages of parallel arbitration and serial arbitration.

7.4 Identify the relative advantages of centralized arbitration and decentralized arbitration.

7.5 Identify the relative advantages of the daisy chain grant arbitration mechanism and the daisy chain request arbitration mechanism. Which would you choose for a new microprocessor? Why?

7.6 A 3-to-8 line priority encoder is a logic component which accepts eight request inputs and produces a 3-bit number identifying the highest priority input request using fixed priority. A 3-to-8 line decoder accepts a 3-bit number and activates one of its eight outputs, as identified by the input number. Show how these components could be used to implement parallel arbitration. Derive the Boolean expressions for each component and show that these equations correspond to the parallel arbitration expressions given in the text.

7.7 Design a parallel arbitration system using three levels of parallel arbiter parts and determine the arbitration time of the system.

7.8 Suppose a rotating daisy chain priority circuit has the following signals:

BR	Bus request from bus master
BG	Bus grant to bus master