7.2 Cache Coherence and Synchronization Mechanisms

Cache coherence protocols for coping with the multicache inconsistency problem are considered below. Snoopy protocols are designed for bus-connected systems. Directory-based protocols apply to network-connected systems. Finally, we study hardware support for fast synchronization. Software-implemented synchronization will be discussed in Chapter 11.

7.2.1 The Cache Coherence Problem

In a memory hierarchy for a multiprocessor system, data inconsistency may occur between adjacent levels or within the same level. For example, the cache and main memory may contain inconsistent copies of the same data object. Multiple caches may possess different copies of the same memory block because multiple processors operate asynchronously and independently.

Caches in a multiprocessing environment introduce the *cache coherence problem*. When multiple processors maintain locally cached copies of a unique shared-memory location, any local modification of the location can result in a globally inconsistent view of memory. Cache coherence schemes prevent this problem by maintaining a uniform state for each cached block of data. Cache inconsistencies caused by data sharing, process migration, or I/O are explained below.

auses:

Inconsistency in Data Sharing The cache inconsistency problem occurs only when multiple private caches are used. In general, three sources of the problem are identified: *sharing of writable data*, *process migration*, and *I/O activity*. Figure 7.12 illustrates the problems caused by the first two sources. Consider a multiprocessor with two processors, each using a private cache and both sharing the main memory. Let X be a shared data element which has been referenced by both processors. Before update, the three copies of X are consistent.

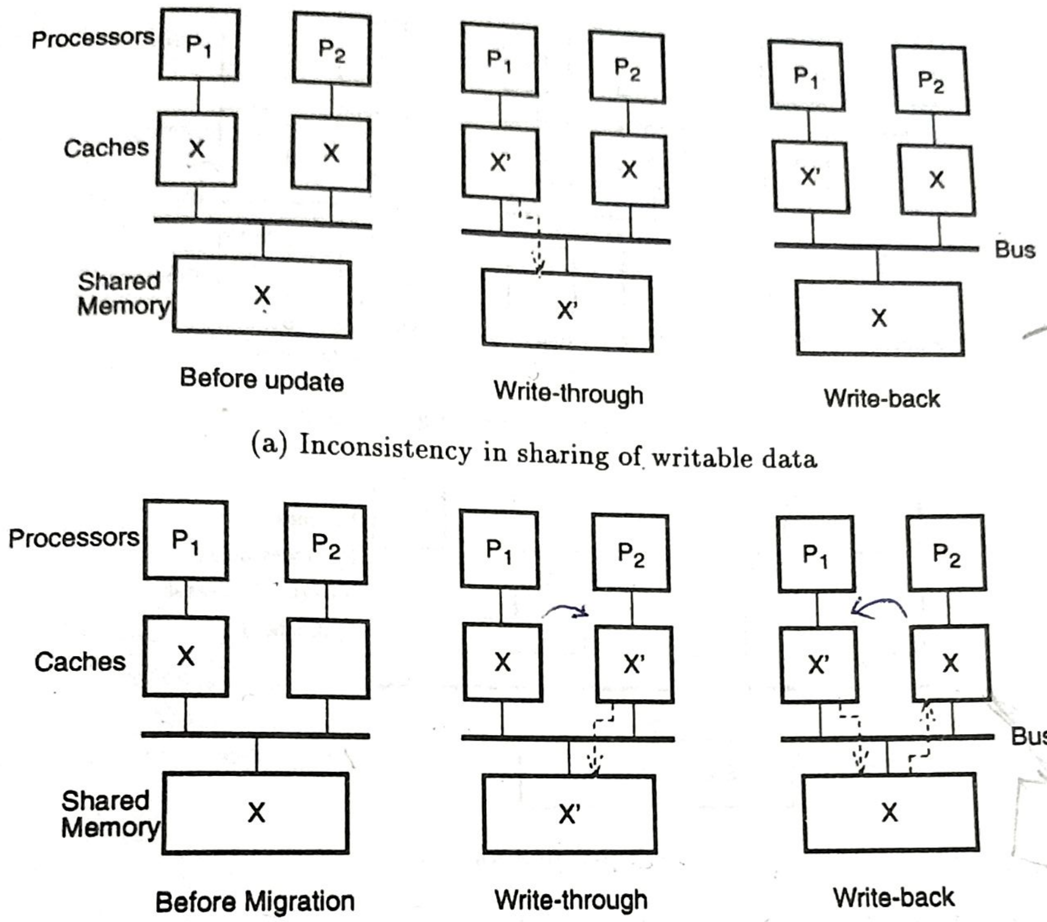
If processor P_1 *writes* new data X' into the cache, the same copy will be written immediately into the shared memory using a *write-through* policy. In this case, inconsistency occurs between the two copies (X' and X) in the two caches (Fig. 7.12a).

On the other hand, inconsistency may also occur when a *write-back* policy is used, as shown on the right in Fig. 7.12a. The main memory will be eventually updated when the modified data in the cache are replaced or invalidated.

Process Migration and I/O Figure 7.12b shows the occurrence of inconsistency after a process containing a shared variable X migrates from processor 1 to processor 2 using the write-back cache on the right. In the middle, a process migrates from processor 2 to processor 1 when using write-through caches.

In both cases, inconsistency appears between the two cache copies, labeled X and X'. Special precautions must be exercised to avoid such inconsistencies. A coherence protocol must be established before processes can safely migrate from one processor to another.

Inconsistency problems may occur during I/O operations that bypass the caches.



(a) Inconsistency in sharing of writable data

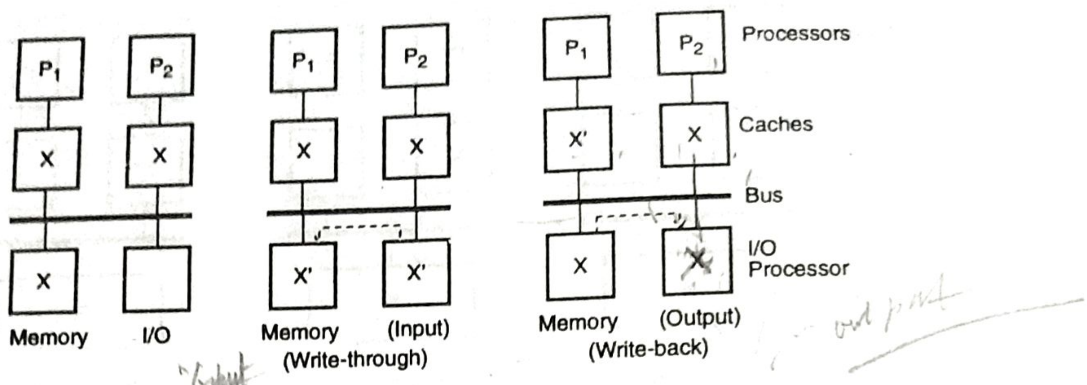
(b) Inconsistency after process migration

Figure 7.12 Cache coherence problems in data sharing and in process migration. (Adapted from Dubois, Scheurich, and Briggs 1988)

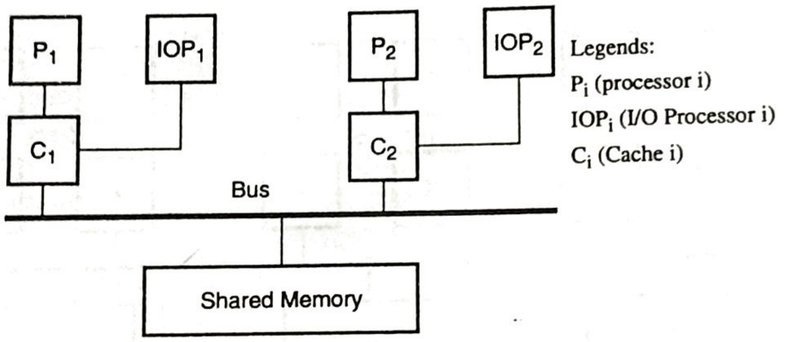
When the I/O processor *loads* a new data X' into the main memory, bypassing the write-through caches (middle diagram in Fig. 7.13a), inconsistency occurs between cache 1 and the shared memory. When outputting a data directly from the shared memory (bypassing the caches), the write-back caches also create inconsistency.

One possible solution to the I/O inconsistency problem is to attach the I/O processors (IOP_1 and IOP_2) to the private caches (C_1 and C_2), respectively, as shown in Fig. 7.13b. This way I/O processors share caches with the CPU. The I/O consistency can be maintained if cache-to-cache consistency is maintained via the bus. An obvious shortcoming of this scheme is the likely increase in cache perturbations and the poor locality of I/O data, which may result in higher miss ratios.

Two Protocol Approaches Many of today's commercially available multiprocessors use bus-based memory systems. A bus is a convenient device for ensuring cache coherence because it allows all processors in the system to observe ongoing memory



(a) I/O operations bypassing the cache



(b) A possible solution

Figure 7.13 Cache inconsistency after an I/O operation and a possible solution. (Adapted from Dubois, Scheurich, and Briggs, 1988)

transactions. If a bus transaction threatens the consistent state of a locally cached object, the cache controller can take appropriate actions to invalidate the local copy. Protocols using this mechanism to ensure coherence are called *snoopy protocols* because each cache snoops on the transactions of other caches.

On the other hand, scalable multiprocessor systems interconnect processors using short point-to-point wires in direct or multistage networks. Unlike the situation in buses, the bandwidth of these networks increases as more processors are added to the system. However, such networks do not have a convenient snooping mechanism and do not provide an efficient broadcast capability. In such systems, the cache coherence problem can be solved using some variant of directory schemes.

In general, a cache coherence protocol consists of the set of possible states in the local caches, the state in the shared memory, and the state transitions caused by the messages transported through the interconnection network to keep memory coherent. In what follows, we first describe the snoopy protocols and then the directory-based protocols. These protocols rely on software, hardware, or a combination of both for implementation. Cache coherence can also be enforced in the TLB or assisted by the compiler. Other approaches to designing a scalable cache coherence interface will be studied in Chapter 9.

7.2.2 Snoopy Bus Protocols

In using private caches associated with processors tied to a common bus, two approaches have been practiced for maintaining cache consistency: *write-invalidate* and *write-update policies*. Essentially, the write-invalidate policy will invalidate all remote copies when a local cache block is updated. The write-update policy will broadcast the new data block to all caches containing a copy of the block.

Snoopy protocols achieve data consistency among the caches and shared memory through a bus watching mechanism. As illustrated in Fig. 7.14, two snoopy bus protocols create different results. Consider three processors (P_1, P_2, and P_n) maintaining consistent copies of block X in their local caches (Fig. 7.14a) and in the shared-memory module marked X.

Using a *write-invalidate protocol*, the processor P_1 modifies (writes) its cache from X to X', and all other copies are invalidated via the bus (denoted I in Fig. 7.14b). Invalidated blocks are sometimes called *dirty*, meaning they should not be used. The *write-update protocol* (Fig. 7.14c) demands the new block content X' be broadcast to all cache copies via the bus. The memory copy is also updated if write-through caches are used. In using write-back caches, the memory copy is updated later at block replacement time.

Write-Through Caches The states of a cache block copy change with respect to *read*, *write*, and *replacement* operations in the cache. Figure 7.15 shows the state transitions for two basic write-invalidate snoopy protocols developed for write-through and write-back caches, respectively. A block copy of a write-through cache i attached to processor i can assume one of two possible cache states: *valid* or *invalid* (Fig. 7.15a).

A remote processor is denoted j, where $j \neq i$. For each of the two cache states, six possible events may take place. Note that all cache copies of the same block use the same transition graph in making state changes.

In a *valid* state (Fig. 7.15a), all processors can *read* ($R(i), R(j)$) safely. Local processor i can also *write* ($W(i)$) safely in a *valid* state. The *invalid* state corresponds to the case of the block either being invalidated or being replaced ($Z(i)$ or $Z(j)$).

Wherever a remote processor *writes* ($W(j)$) into its cache copy, all other cache copies become invalidated. The cache block in cache i becomes *valid* whenever a successful read ($R(i)$) or write ($W(i)$) is carried out by a local processor i.

The fraction of *write cycles* on the bus is higher than the fraction of *read cycles* in a write-through cache, due to the need for request invalidations. The *cache directory* (registration of cache states) can be made in dual copies or dual-ported to filter out most invalidations. In case *locks* are cached, an atomic Test&Set must be enforced.

Write-Back Caches The *valid* state of a write-back cache can be further split into two cache states, labeled RW (*read-write*) and RO (*read-only*) in Fig. 7.15b. The INV (invalidated or not-in-cache) cache state is equivalent to the *invalid* state mentioned