MODULE 3
MULTIPROCESSOR SYSTEM INTERCONNECTS

3.1 Hierarchical Bus Systems

- A bus system consists of a hierarchy of buses connecting various system and subsystem components in a computer.
- Each bus is formed with a number of signal, control, and power lines.
- Different buses are used to perform different interconnection functions.
- In general, the hierarchy of bus systems are packaged at different levels as depicted in Fig. 7.2, including local buses on boards, backplane buses, and I/O buses.

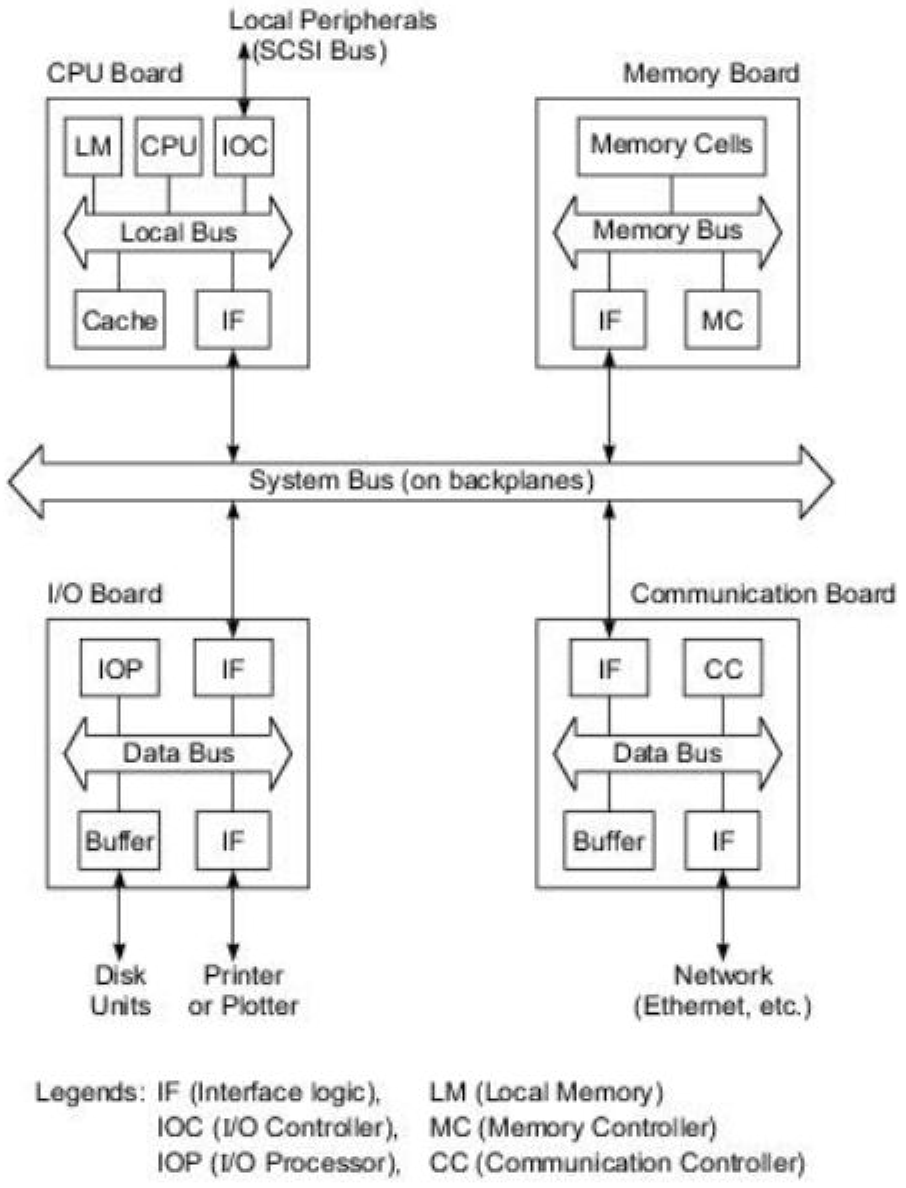

Local Peripherals (SCSI Bus)
CPU Board
LM
CPU
IOC
Local Bus
Cache
IF
Memory Board
Memory Cells
Memory Bus
IF
MC
System Bus (on backplanes)
I/O Board
IOP
IF
Data Bus
Buffer
IF
Disk Units
Printer or Plotter
Communication Board
IF
CC
Data Bus
Buffer
IF
Network (Ethernet, etc.)
Legends: IF (Interface logic), LM (Local Memory)
IOC (I/O Controller), MC (Memory Controller)
IOP (I/O Processor), CC (Communication Controller)

Fig. 7.2 Bus systems at board level, backplane level, and I/O level

Local Bus

- Buses implemented within processor chips or on printed circuit boards are called *local buses*.
- A memory board uses a *memory bus* to connect the memory with the interface logic.
- An I/O or network interface chip or board uses a *data bus*.
- Each of these local buses consists of signal and utility lines.

Backplane Bus

- A backplane is a printed circuit on which many connectors are used to plug in functional boards.
- A system bus consisting of shared signal paths and utility lines, is built on the backplane. This system bus provides a common communication path along all plug-in boards.

I/O bus

- Input /output devices are connected to a computer system through an I/O bus such as the SCSI (Small Computer Systems Interface) bus.
- This bus is made of coaxial cables with taps connecting disks, printer, and other devices to a processor through an I/O controller
- Special interface logic is used to connect various board types to the backplane bus.

3.2 Crossbar Switch and Multiport Memory

Network Stages

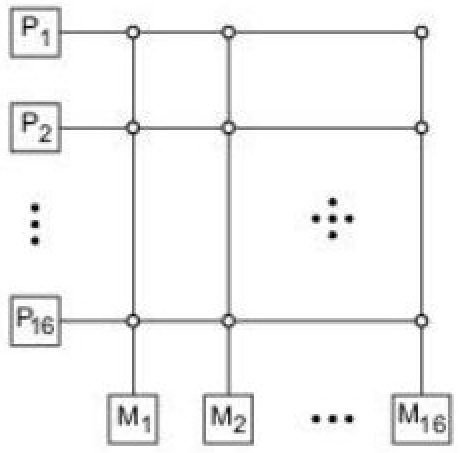
- Depending on the interstage connections used, a *single-stage network* is also called a *recirculating network* because data items may have to recirculate through the single stage many times before reaching their destination.
- A single-stage network is cheaper to build, but multiple passes may be needed to establish certain connections.
- The crossbar switch and multiport memory organization are both single-stage networks.
- A multistage network consists of more than one stage of switch boxes.
- Such a network should be able to connect from any input to any output.

Blocking versus Nonblocking Networks

- A multistage network is called blocking if the simultaneous connections of some multiple input-output pairs may result in conflicts in the use of switches or communication links.
- In a blocking network,multiple passes through the network may be needed to achieve certain input-output connections.
- A multistage network is called nonblocking if it can perform all possible connections between inputs and outputs by rearranging its connections.

Crossbar Networks

- In a crossbar network every input port is connected to a free output port through a crosspoint switch without blocking.
- A crossbar network is a single-stage network built with unary switches at the crosspoints.
- Each crosspoint in a crossbar network is a unary switch which can be set open or closed, providing a point-to-point connection path between the source and destination.
- All processors can send memory requests independently and asynchronously.
- This poses the problem of multiple requests destined for the same memory module at the same time. In such cases, only one of the requests is serviced at a time.



(a) Interprocessor-memory crossbar network built in the C.mmp multiprocessor at Carnegie-Mellon University (1972)

Crosspoint Switch Design

- Out of n crosspoint switches in each column of an n x m crossbar mesh, only one can be connected at a time.
- To resolve the contention for each memory module, each crosspoint switch must be designed with extra hardware.
- Multiplexer modules are used to select one of n read or write requests for service.
- Each processor sends in an independent request, and the arbitration logic makes the selection based on certain fairness or priority rules.

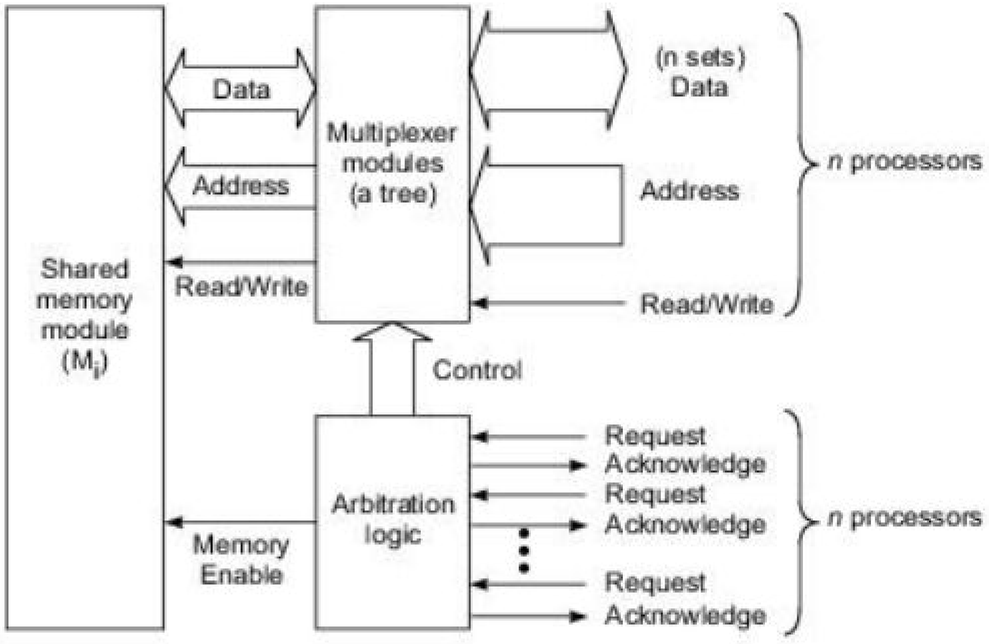
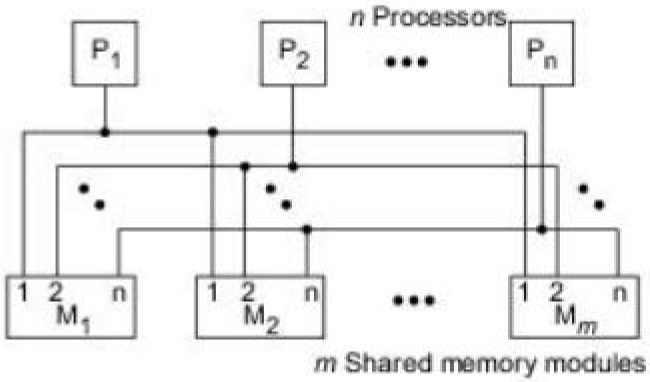


Fig. 7.6 Schematic design of a row of crosspoint switches in a crossbar network

Multiport Memory

- A multiport memory system employs separate buses between cache memory module & each cpu.
- The memory module must have internal control logic to determine which port will have access to memory at a given time.
- Memory access conflicts are resolved by assigning fixed priorities to each memory port..
- Advantage: High transfer rate can be achieved because of multiple paths.
- Drawback is the need for a large number of interconnection cables and connectors when the configuration becomes large.



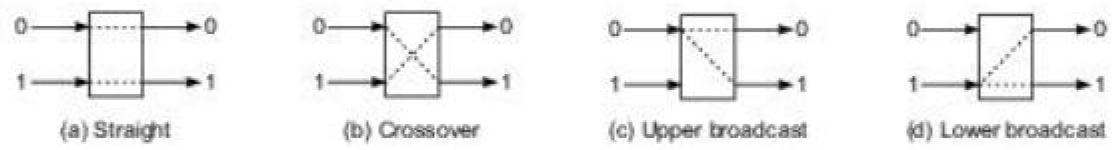
(a) n-port memory modules used

3.3 Multistage and Combining Networks

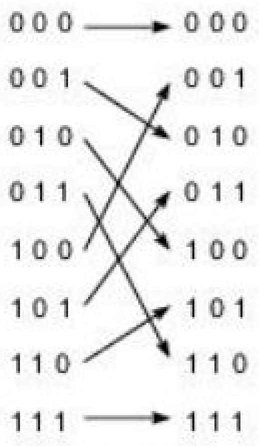
- Multistage networks are used to build larger multiprocessor systems.
- We describe two multistage networks, the Omega network and the Butterfly network,

Routing in Omega Network

- In general, an n-input Omega network has $log_2 n$ stages.
- Each stage requires $\frac{n}{2}$ switch modules.
- The stages are labeled from 0 to $log_2 n - 1$ from the input end to the output end.
- There are four types of switch connections: *straight, crossover, upper broadcast and lower broadcast*.



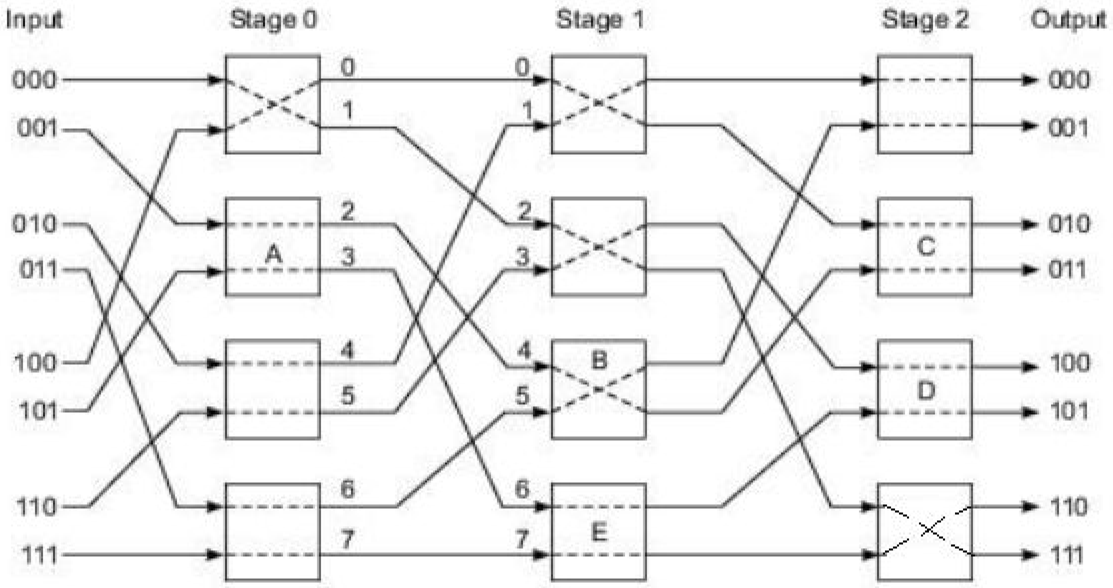
- The interstage connection pattern is the perfect shuffle over 8 objects. The perfect shuffle is obtained by shifting 1 bit to the left and wrapping around the most significant to the least significant position.



(a) Perfect shuffle

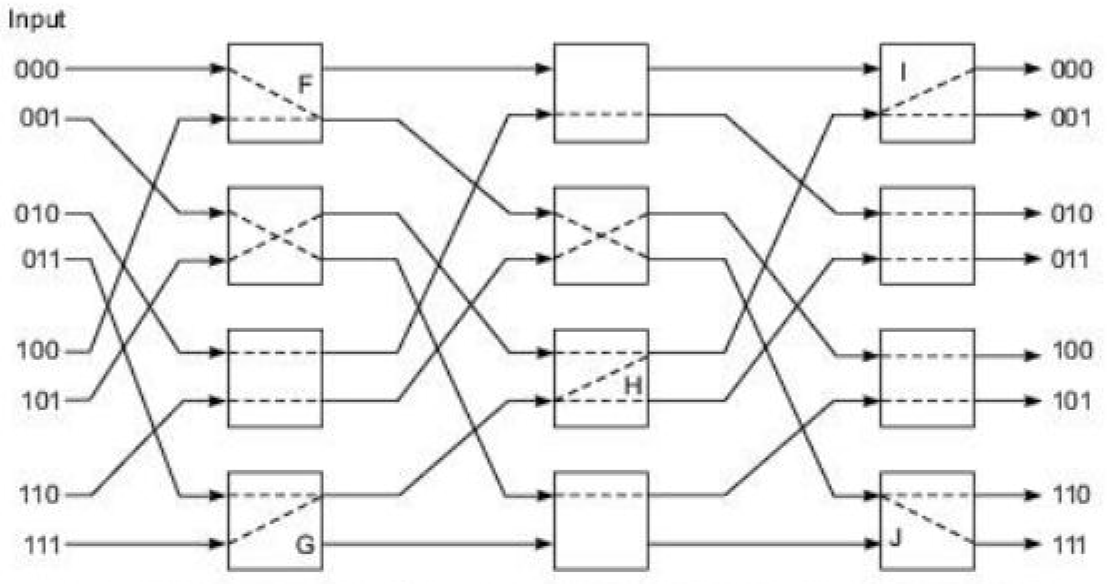
- Data routing is controlled by inspecting the destination code in binary.
- When the ith high-order bit of the destination code is a 0, a 2 x 2 switch at stage i connects the input to the upper output. Otherwise, the input is directed to the lower output.
- Consider the permutation $\pi_1 = (0,7,6,4,2)(1,3)(5)$
    - ❖ Which maps $0 \to 7, 7 \to 6, 6 \to 4, 4 \to 2, 2 \to 0, 1 \to 3, 3 \to 1, 5 \to 5$
    - ❖ Consider the routing of a message from input 001 to output 011
    - ❖ This involves the use of switches A, B, and C.
    - ❖ Since the *most significant bit* ofthe destination 011 is a "zero“, switch A must be set straight so that the input 001 is connected to the upper output.
    - ❖ The *middle bit* in 011 is a “one”, thus input 4 to switch B is connected to the lower output with a “crossover” connection.
    - ❖ The *least significant bit* in fill is a "one", implying a flat connection in switch C.
    - ❖ Similarly the remaining connections are done.

- There exists no conflict in all the switch settings needed to implement the permutation π_1



(a) Permutation $\pi_1 = (0, 7, 6, 4, 2)\ (1, 3)\ (5)$ implemented on an Omega network without blocking

- Consider the permutation $\pi_2 = (0, 6, 4, 7, 3)(1, 5)(2)$
    - Which maps $0 \rightarrow 6, 6 \rightarrow 4, 4 \rightarrow 7, 7 \rightarrow 3, 3 \rightarrow 0, 1 \rightarrow 5, 5 \rightarrow 1, 2 \rightarrow 2$
    - Conflicts in switch settings do exist in three switches identified as F, G, and H.
    - The conflicts occurring at F are caused by the desired routings 000 —> 110 and 100 —> 111.
    -



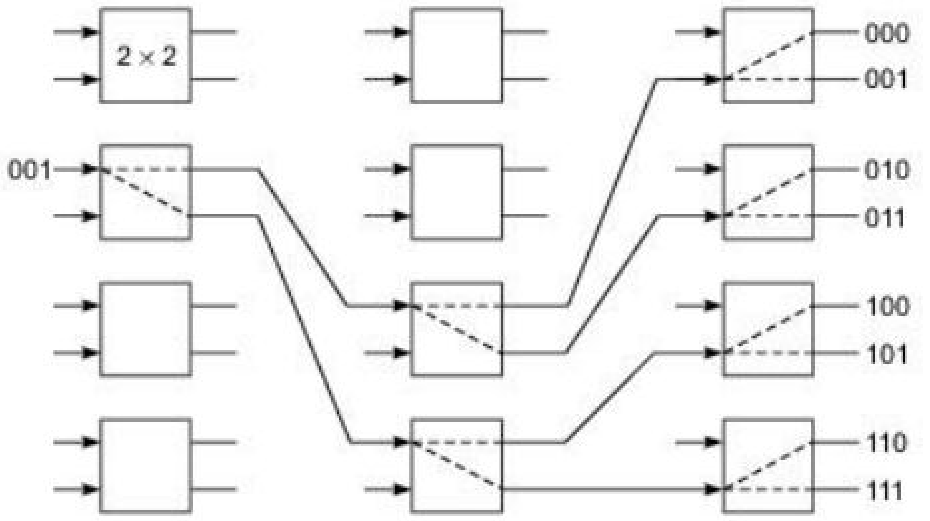
(b) Permutation $\pi_2 = (0, 6, 4, 7, 3)\ (1, 5)\ (2)$ blocked at switches marked F, G, and H

Fig. 7.8 Two switch settings of an 8 × 8 Omega network built with 2 × 2 switches

- ❖ Since both destination addresses have a leading bit 1, both inputs to switch F must be connected to the lower output. To resolve the conflicts, one request must be blocked.
- ❖ Similarly, we see conflicts at switch G between 011—>000 and 111 —> 011, and at switch H between 101 -—> 001 and 011 —> 000.
- ❖ At switches I and J, broadcast is used from one input to two outputs, which is allowed if the hardware is built to have four legitimate states.
- ❖ This example indicates the fact that not all permutations can be implemented in one pass through the Omega network.
- ❖ The Omega network is a blocking network.
- ❖ In case of blocking, one can establish the conflicting connections in several passes.
- ❖ For the example π_2, we can connect 000 —> 110, 001 —> 101, 010 —> 010, 101 —> 001, 110 —> 100 in the first pass and 011—>000, 100 —> 111, 111 → 011 in the second pass.
- ❖ In general, if 2 x 2 switch boxes are used, an n-input Omega network can implement $n^{\frac{n}{2}}$ permutations in a single pass.
- ❖ There are n! permutations in total.
- ❖ In general, a maximum of $log_2 n$ passes are needed for an n-input Omega.

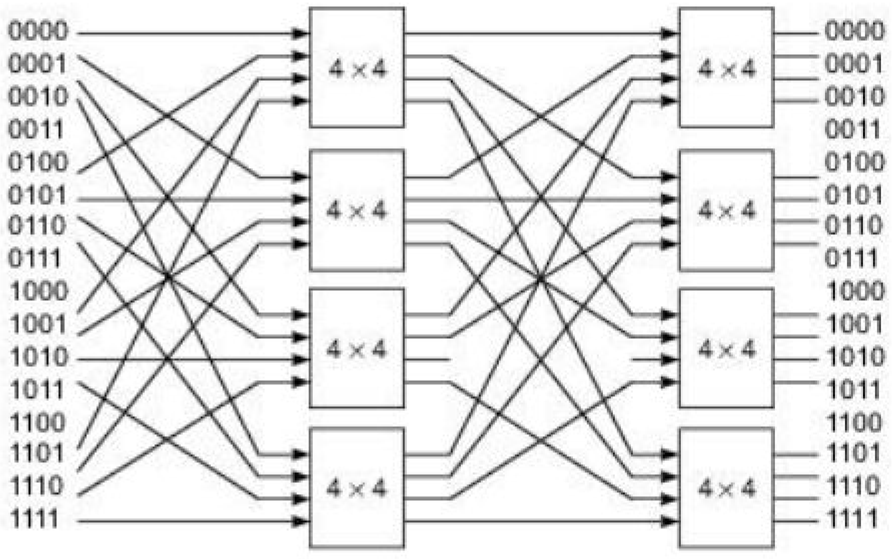
- The Omega network can also be used to broadcast data from one source to many destinations, as exemplified in Fig.a, using the upper broadcast or lower broadcast switch settings.

- In Fig. a, the message at input 001 is being broadcast to all eight outputs through a binary tree connection.



(a) Broadcast connections

- The two way shuffle interstage connections can be replaced by four-way shuffie interstage connections when 4 x 4 switch boxes are used as building blocks, as exemplified in Fig. 7.9b for a I6-input Omega network with $log_4 16$ = 2 stages.
- Note that a four-way shuffle corresponds to dividing the I6 inputs into four equal subsets and then shuffling them evenly among the four subsets.



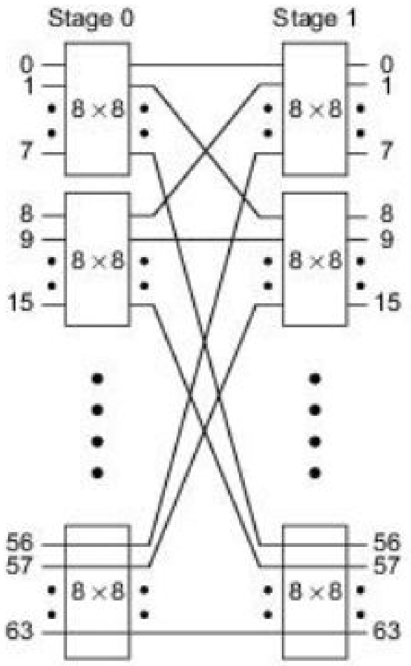
(b) Using four-way shuffle interstage connections

Fig. 7.9 Broadcast capability of an Omega network built with 4 × 4 switches

Routing In Butterfly Networks

- This class of networks is constructed with crossbar switches as building blocks.

- Note that no broadcast connections are allowed in a Butterfly network, making these networks a restricted subclass of Omega networks.



(a) A two-stage 64 × 64 Butterfly switch network built with 16 8 × 8 crossbar switches and eight-way shuffle interstage connections

The Hot-Spot Problem

- When the network traffic is nonuniform, a hot spot may appear corresponding to a certain memory module being excessively accessed by many processors at the same time.
- Hot spots may degrade the network performance significantly.
- The purpose was to combine multiple requests heading for the same destination at switch points where conflicts are taking place.

3.4 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.

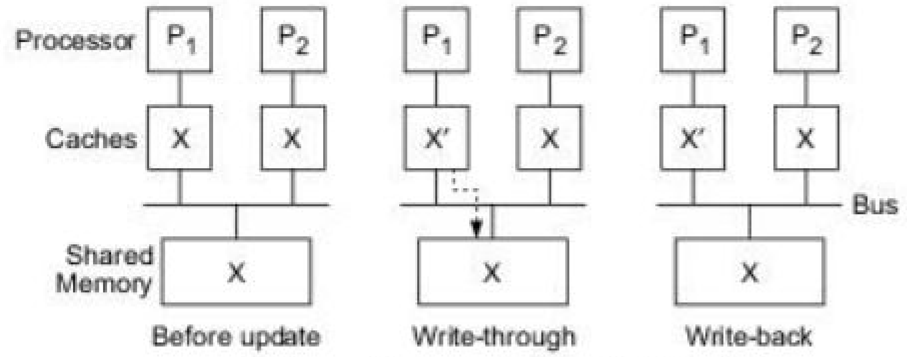
3.4.1 The Cache Coherence Problem

In a memory hierarchy for a multiprocessor system, data inconsistency may occur between adjacent levels or within the same level.

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 IMO are explained below.

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, I/O activity.

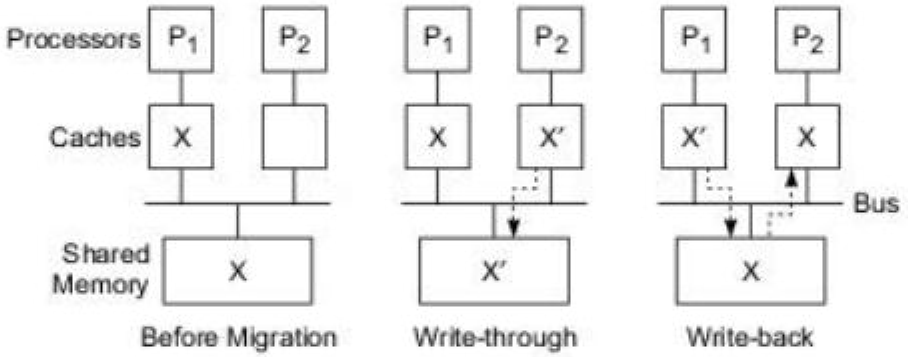
- Let X be a shared data element which has been referenced by both processors. Before update, the three copies of X are consistent.
- If processor P1 writes new data X' into the cache, the same copy will be written immediately into the shared memory under a write through policy. In this case. inconsistency occurs between the two copies in the two caches
- On the other hand, inconsistency may also occur when a write back policy is used.



(a) Inconsistency in sharing of writable data

Process Migration and I/O

- Figure b 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 Xand X'.



(b) Inconsistency after process migration

Inconsistency due to I/O operations

- 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 (IOP1 and IOP2) to the private caches (C1 and C2), respectively.

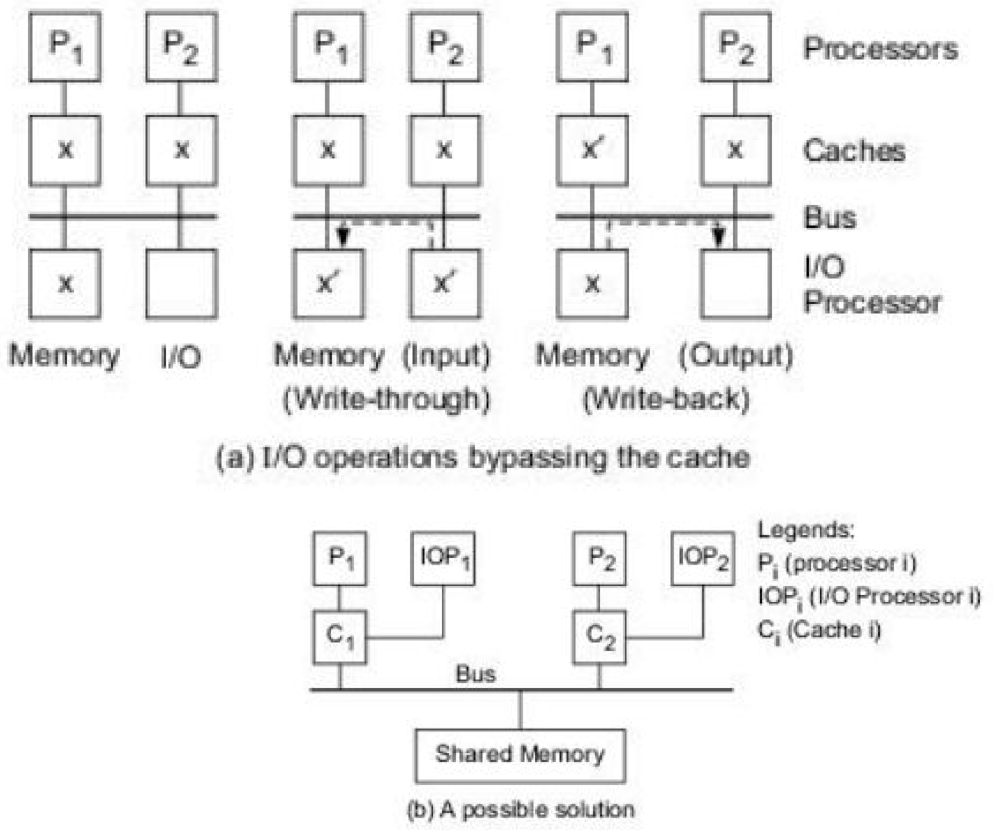
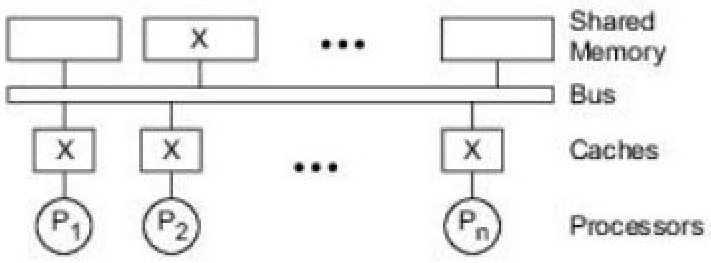


Fig. 7.13 Cache inconsistency after an I/O operation and a possible solution (Adapted from Dubois, Scheurich, and Briggs, 1988)

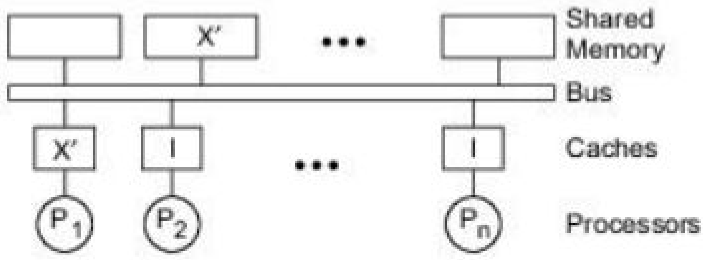
3.4.2 Snoopy Bus Protocols

- 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
- 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.
- Consider three processors (PI, P2, and Pn) maintaining consistent copies of block X in their local caches and in the shared-memory module marked X.



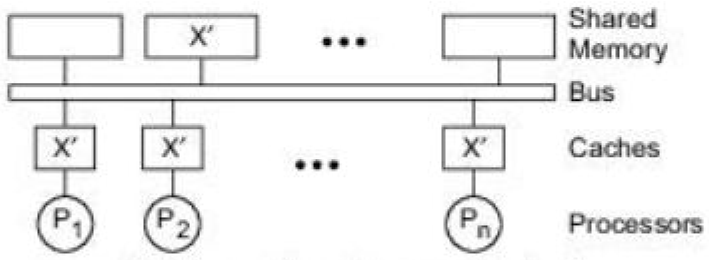
(a) Consistent copies of block X are in shared memory and three processor caches

- Using a write-invalidate protocol, the processor P1 modifies (writes) its cache from X to X', and all other copies are invalidated via the bus (denoted I in Fig.b]. invalidated blocks are sometimes called dirty, meaning they should not be used.



(b) After a write-invalidate operation by P_1

- The write-update protocol (Fig.c) 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.



(c) After a write-update operation by P_1

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 ≠ 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.

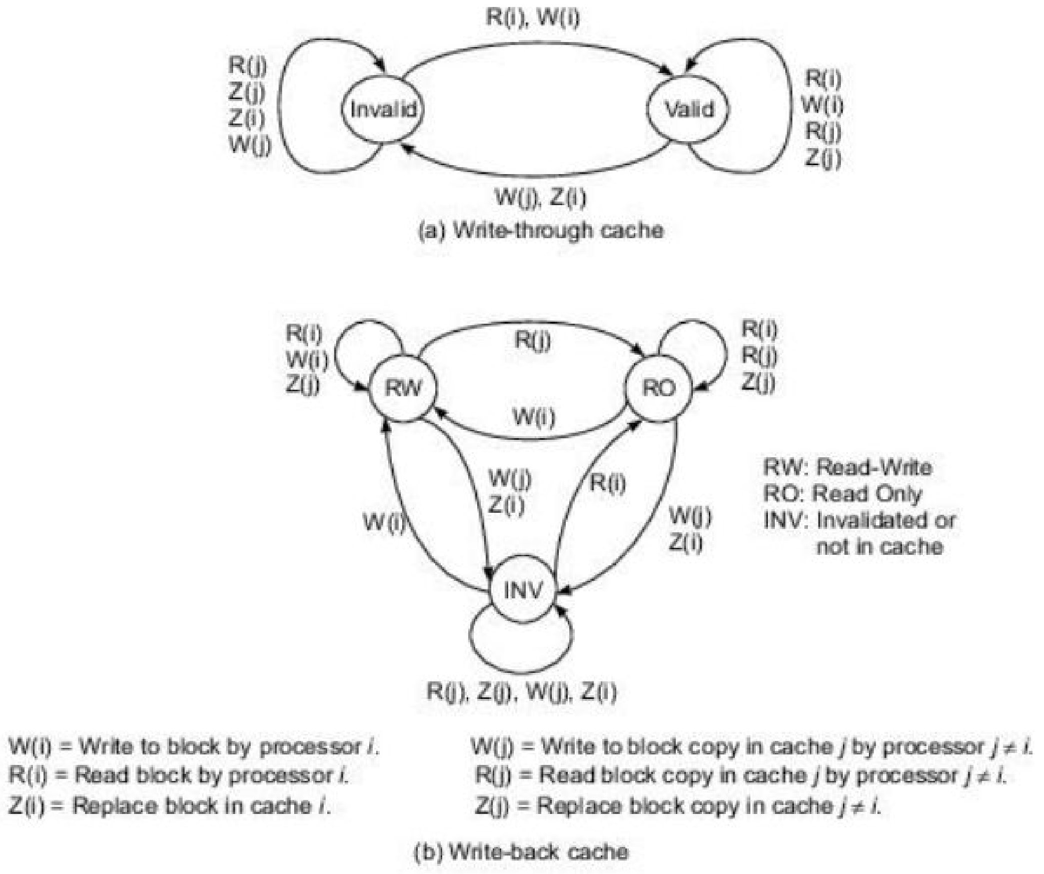

R(i), W(i)
R(j)
Z(j)
Z(i)
W(j)
Invalid
Valid
R(i)
W(i)
R(j)
Z(j)
W(j), Z(i)
(a) Write-through cache
R(i)
W(i)
Z(j)
RW
R(j)
RO
W(i)
R(i)
R(j)
Z(j)
W(j)
Z(i)
R(i)
W(i)
W(j)
Z(i)
INV
RW: Read-Write
RO: Read Only
INV: Invalidated or not in cache
R(j), Z(j), W(j), Z(i)
W(i) = Write to block by processor *i*.
R(i) = Read block by processor *i*.
Z(i) = Replace block in cache *i*.
W(j) = Write to block copy in cache *j* by processor *j* ≠ *i*.
R(j) = Read block copy in cache *j* by processor *j* ≠ *i*.
Z(j) = Replace block copy in cache *j* ≠ *i*.
(b) Write-back cache

Fig. 7.15 Two state-transition graphs for a cache block using write-invalidate snoopy protocols (Adapted from Dubois, Scheurich, and Briggs, 1988)

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*) as shown in Fig. 7.15b.
- The INV (invalidated or not-in-cache) cache state is equivalent to the invalid state mentioned before.
- This three-state coherence scheme corresponds to an *ownership protocol.*

- When the memory owns a block, caches can contain only the RO copies of the block. In other words, multiple copies may exist in the RO state and every processor having a copy (called a keeper of the copy) can read (R(i), R(j) the copy safely.
- The INV state is entered whenever a remote processor writes (W(j)) its local copy or the local processor replaces (Z(i)) its own block copy.
- The RW state corresponds to only one cache copy existing in the entire system owned by the local processor i. Read (R(i)) and write (W(i)) can be safely performed in the RW state.
- From either the RO state or the INV state, the cache block becomes uniquely owned when a local write (W(i)) takes place.

Write-once Protocol

- James Goodman (1983) proposed a cache coherence protocol for bus-based multiprocessors.
- This scheme combines the advantages of both write-through and write-back invalidations.
- In order to reduce bus traffic, the very first write of a cache block uses a write-through policy.
- This will result in a consistent memory copy while all other cache copies are invalidated.
- After the first write, shared memory is updated using a write-back policy. This scheme can be described by the four-state transition graph shown in Fig. 7.16.
- The four cache states are defined below:
    1. *Valid*: The cache block, which is consistent with the memory copy, has been read from shared memory and has not been modified.
    2. *Invalid:* The block is not found in the cache or is inconsistent with the memory copy.
    3. *Reserved:* Data has been written exactly *once* since being *read* from shared memory. The cache copy is consistent with the memory copy, which is the only other copy.
    4. Dirty Thc cache block has been modified (written) more than once, and the cache copy is the only one in the system (thus inconsistent with all other copies).
- The solid lines in Fig. 7.16 correspond to access commands issued by a local processor labeled *read-miss, write-hit*, and *write-miss*.
- Whenever a read-miss occurs, the valid state is entered.
- The first write-hit leads to the *reserved state*.
- The second write-hit leads to the dirty state, and all future write-hits stay in the *dirty* state.
- Whenever a *write-miss* occurs, the eache block enters the dirty state.
- The dashed lines correspond to invalidation commands issued by remote processors via the snoopy bus.
- The *read-invalidate* command reads a block and invalidates all other copies.
- The *write-invalidate* command invalidates all other copies of a block. The bus-read command corresponds to a normal memory read by a remote processor via the bus

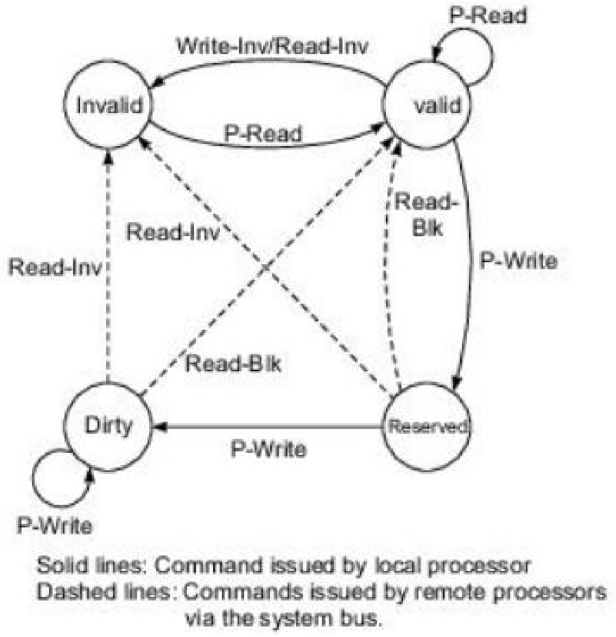

P-Read
Write-Inv/Read-Inv
Invalid
valid
P-Read
Read-Inv
Read-Blk
Read-Inv
P-Write
Read-Blk
Dirty
Reserved
P-Write
P-Write
Solid lines: Command issued by local processor
Dashed lines: Commands issued by remote processors via the system bus.

Fig. 7.16 Goodman's write-once cache coherence protocol using the write invalidate policy on write-back caches (Adapted from James Goodman 1983, reprinted from Stenstrom, *IEEE Computer*, June 1990)

Cache Events and Actions The memory-access and invalidation commands trigger the following events and actions:

- ***Read-miss***: When a processor wants to read a block that is not in the cache, a *read-miss* occurs. A *bus-read* operation will be initiated. If no *dirty* copy exists, then main memory has a consistent copy and supplies a copy to the requesting cache. If a dirty copy does exist in a remote cache, that cache will inhibit the main memory and send a copy to the requesting cache. In all cases, the cache copy will enter the *valid* state after a *read-miss*.
- ***Write-hit:*** If the copy is in the *dirty* or reserved state, the write can be carried out locally and the new state is dirty. If the new state is valid, a *write-invalidate* command is broadcast to all caches, invalidating their copies. The shared memory is *written through*, and the resulting state is *reserved* after this first write.
- ***Write-miss***: When a processor fails to write in a local cache, the copy must come either from the main memory or from a remote cache with a dirty block. This is accomplished by sending a read-invalidate command which will invalidate all cache copies. The local copy is thus updated and ends up in a dirty state.
- ***Read-hit***: Read-hits can always be performed in a local cache without causing a state transition or using the snoopy bus for invalidation.
- ***Block Replacement***: If a copy is dirty, it has to be written back to main memory by block replacement. If thc copy is clean (i.e., in either the valid, reserved, or invalid state), no replacement will take place.

3.4.3 Directory-Based Protocols

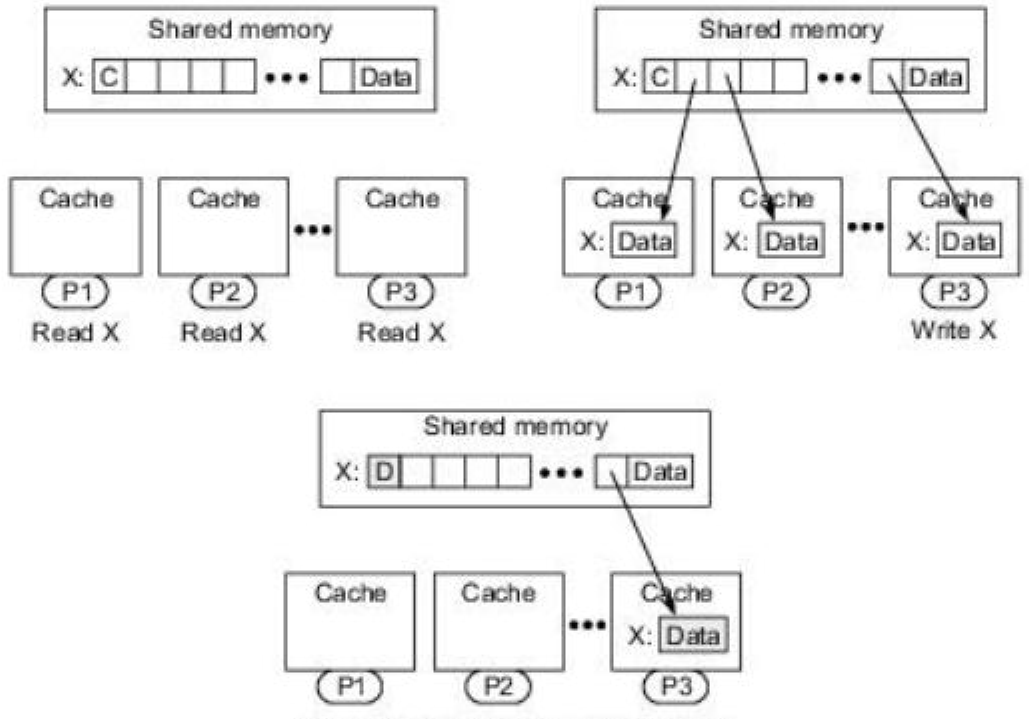
When a multistage or packet switched network is used to build a large multiprocessor with hundreds of processors, the snoopy cache protocols must be modified to suit the network capabilities. Since broadcasting is expensive to perform in such a network, consistency commands will be sent only to those caches that keep a copy of the block. This leads to directory based protocols for network-connected multiprocessors.

Directory Structures

- In a multistage or packet switched network, cache coherence is supported by using cache directories to store information on where copies of cache blocks reside.
- The first directory scheme, which used a *central directory* containing duplicates of all cache directories. This central directory, providing all the information needed to enforce consistency, is usually very large and must be associatively searched, like the individual cache directories. Contention and long search times are two drawbacks in using a central directory for a large multiprocessor
- In a distributed-directory scheme each memory module maintains a separate directory which records the state and presence information for each memory block.
- A cache-coherence protocol that does not use broadcasts must store the locations of all cached copies of each block of shared data. This list of cached locations, whether centralized or distributed, is called a *cache directory*.
- A directory entry for each block of data contains a number of pointers to specify the locations of copies of the block.
- Each directory entry also contains a *dirty bit* to specify whether a particular cache has permission to Write the associated block of data.
- Different types of directory protocols fall under three primary categories: *full map directories, limited directories, chained directories.*

Full-Map Directories

- Full-map directories store enough data associated with each block in global memory so that every cache in the system can simultaneously store a copy of any block of data. That is each directory entry contains N pointers, where N is the number of processors in the system.
- The full-map protocol implements directory entries with one bit per processor and a dirty bit. Each bit represents the status of the block in the corresponding processor's cache (present or absent).
- If the dirty bit is set, then one and only one processor's bit is set and that processor can write into the block.



(a) Three states of a full-map directory

- In the first state, location X is missing in all of the caches in the system.
- The second state results from three caches (Cl, C2, and C3) requesting copies of location X.
- Three pointers (processor bits) are set in the entry to indicate the caches that have copies of the block of data.
- In the first two states, the dirty bit on the left side of the directory entry is set to clean (C), indicating that no processor has permission to write to the block of data.
- The third state results from cache C3 requesting write permission for the block.
- In the final state, the dirty bit is set to dirty (D), and there is a single pointer to the block of data in cache C3.

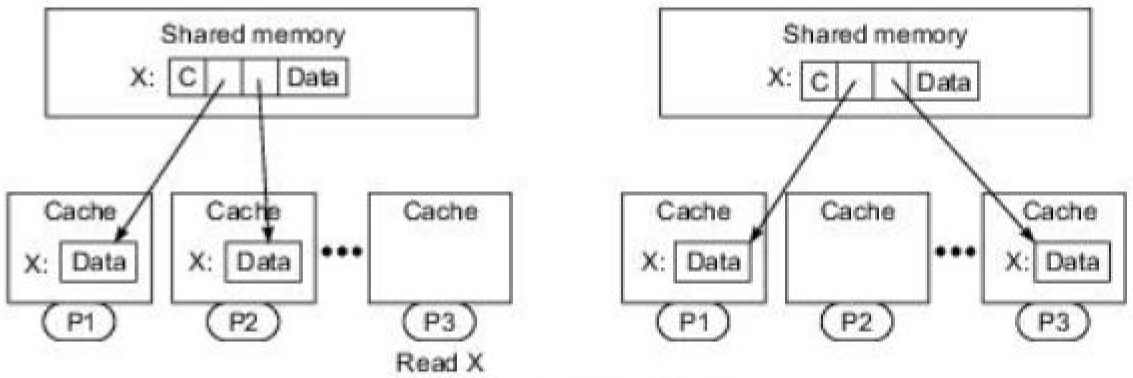
Let us examine the transition from the second state to the third state in more detail. Once processor P3 issues the write to cache C3, the following events will take place:

1. Cache C3 detects that the block containing location X is valid but that the processor does not have permission to write to the block, indicated by the block‘s write-permission bit in thc cache.
2. Cache C3 issues a write request to the memory module containing location X and stalls processor P3.
3. The memory module issues invalidate requests to caches C1 and C2.
4. Caches C1 and C2 receive thc invalidate requests, set the appropriate bit to indicate that the block containing location X is invalid and send acknowledgements back to the memory module.
5. The memory module receives the acknowledgements, sets the dirty bit, clears the pointers to caches C1 and C2, and sends write permission to cache C3.

6. Cache C3 receives the write permission message, updates the state in the cache, and reactivates processor P3.

Limited Directories

- Limited directory protocols are designed to solve the directory size problem.
- A directory protocol can be classified as Dir_iX .
- The symbol i stands for the number of pointers, and X is NB for a scheme with no broadcast.
- A full-map scheme without broadcast is represented as Dir_NNB .
- A limited directory protocol that uses i<N pointers is denoted Dir_iNB .

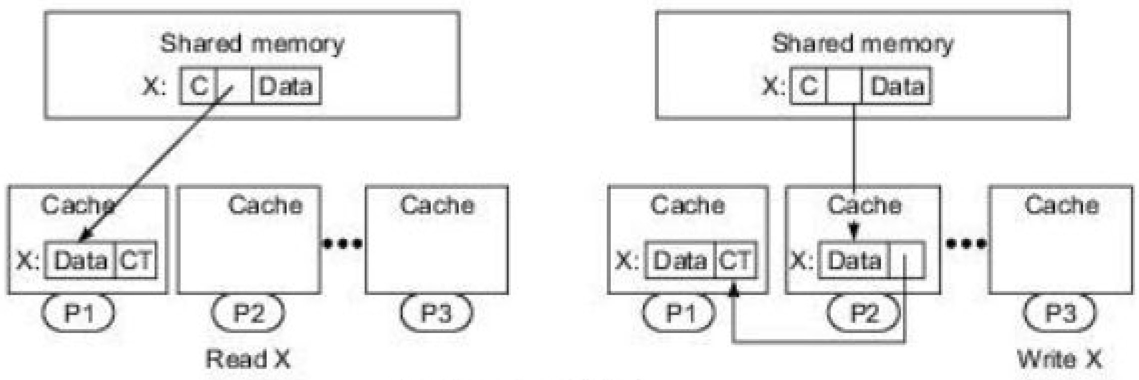


(b) Eviction in a limited directory

- Figure b shows the situation when three caches request read copies in a memory system with a Dir_2NB protocol.
- In this case, we can view the two-pointer directory as a two-way set-associative cache of pointers to shared copies.
- When cache C3 requests a copy of location X, the memory module must invalidate the copy in either cache C1 or cache C2. This process of pointer replacement is called *eviction*.
- Since the directory acts as a set-associative cache, it must have a pointer replacement policy.
- Dir_iB protocols allow more than i copies of each block of data to exist, but they resort to a broadcast mechanism when more than i cached copies of a block need to be invalidated.

Chained Directories

- Chained directories realize the scalability of limited directories without restricting the number of shared copies of data blocks.
- This type of cache coherence scheme is called a *chained* scheme because it keeps track of shared copies of data by maintaining a chain of directory pointers.
- The simpler of the two schemes implements a singly linked chain, which is best described by example (Fig.c).



(c) The chained directory

- Suppose there are no shared copies of location X. If processor P1 reads location X, the memory sends a copy to cache C1, along with a *chain termination* (CT) pointer. The memory also keeps a pointer to cache C1.
- Subsequently, when processor P2 reads location X, the memory sends a copy to cache C2, along with the pointer to cache C1. The memory then keeps a pointer to cache C2.
- By repeating the above step, all of the caches can cache a copy of the location X.
- If processor P3 writes to location X, it is necessary to send a data invalidation message down the chain.
- To ensure sequential consistency, the memory module denies processor P3 write permission until the processor with the chain termination pointer acknowledges the invalidation of the chain.
- Perhaps this scheme should be called a *gossip protocol* (as opposed to a snoopy protocol) because information is passed from individual to individual rather than being spread by covert observation.