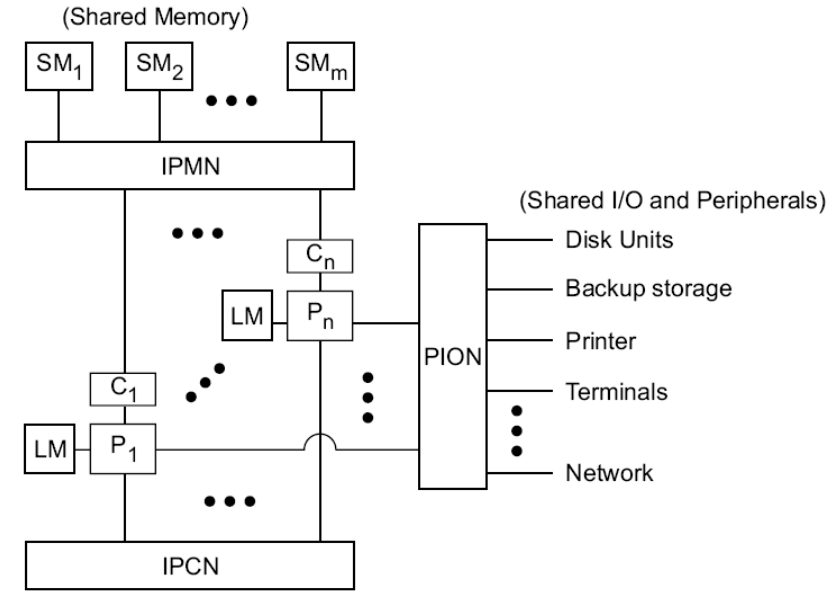
MODULE-IV

Chapter-7 Multiprocessors and Multicomputers

7.1 Multiprocessor system interconnect

- Parallel processing demands the use of efficient system interconnects for fast communication among multiple processors and shared memory, I/O and peripheral devices.
- Hierarchical buses, crossbar switches and multistage networks are often used for this purpose.
- A generalized multiprocessor system is depicted in Fig. 7.1. This architecture combines features from the UMA, NUMA and COMA models.



Legends: IPMN (Inter-Processor-Memory Network)
PION (Processor- I/O Network)
IPCN (Inter-Processor Communication Network)
P (Processor)
C (Cache)
SM (Shared Memory)
LM (Local Memory)

Fig. 7.1 Interconnection structures in a generalized multiprocessor system with local memory, private caches, shared memory, and shared peripherals

- Each processor Pi is attached to its own local memory and private cache.
- These multiple processors connected to share memory through interprocessor memory network (IPMN).

- Processors share the access of I/O and peripheral devices through Processor-I/O Network (PION). Both IPMN and PION are necessary in a shared-resource multiprocessor.
- An optional Interprocessor Communication Network (IPCN) can permit processor communication without using shared memory.

Network Characteristics

The networks are designed with many choices like timing, switching and control strategy like in case of dynamic network the multiprocessors interconnections are under program control.

Timing

- Synchronous – controlled by a global clock which synchronizes all network activity.
- Asynchronous – use handshaking or interlock mechanisms for communication and especially suitable for coordinating devices with different speed.

Switching Method

- Circuit switching – a pair of communicating devices control the path for the entire duration of data transfer
- Packet switching – large data transfers broken into smaller pieces, each of which can compete for use of the path

Network Control

- Centralized – global controller receives and acts on requests
- Distributed – requests handled by local devices independently

7.1.1 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.

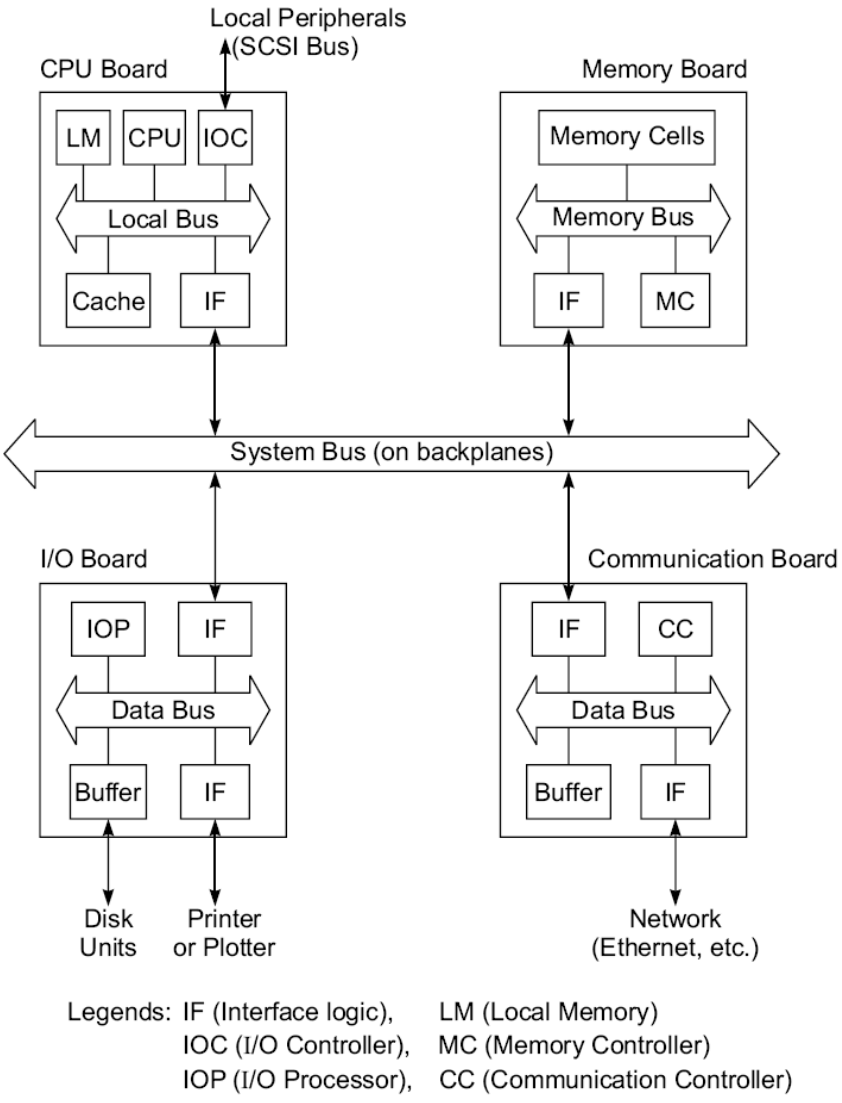


Fig. 7.2 Bus systems at board level, backplane level, and I/O level

- **Local Bus** Buses implemented on *printed-circuit* boards are called *local buses.*
- On a processor board one often finds a local bus which provides a common communication path among major components (chips) mounted on the board.
- A **memory board** uses a *memory bus* to connect the memory with the interface logic.
- An **I/O board** or network interface board uses a *data bus.* Each of these board 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 pathsand utility lines, is built on the backplane.This system bus provides a common communication path among all plug-in boards.

I/O Bus

Input/Output devices are connected to a comuter 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.

Hierarchical Buses and Caches

This is a multilevel tree structure in which the leaf nodes are processors and their private caches (denoted P_j and C_{1j} in Fig. 7.3). These are divided into several clusters, each of which is connected through a cluster bus.

An intercluster bus is used to provide communications among the clusters. Second level caches (denoted as C2i) are used between each cluster bus and the intercluster bus. Each second level cache must have a capacity that is at least an order of magnitude larger than the sum of the capacities of all first-level caches connected beneath it.

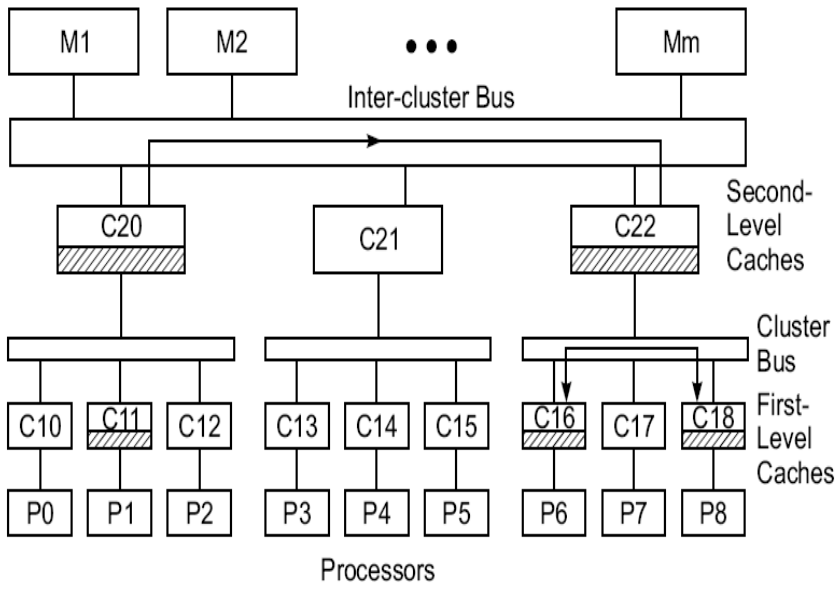


Fig. 7.3 A hierarchical cache/bus architecture for designing a scalable multiprocessor (Courtesy of Wilson; reprinted from *Proc. of Annual Int. Symp. on Computer Architecture, 1987)*

- Each single cluster operates on a single-bus system. Snoopy bus coherence protocols can be used to establish consistency among first level caches belonging to the same cluster.
- Second level caches are used to extend consistency from each local cluster to the upper level.
- The upper level caches form another level of shared memory between each cluster and the main memory modules connected to the intercluster bus.
- Most memory requests should be satisfied at the lower level caches.
- Intercluster cache coherence is controlled among the second-level caches and the resulting effects are passed to the lower level.

7.1.2 Crossbar Switch and Multiport Memory

Single stage networks are sometimes called recirculating networks because data items may have to pass through the single stage many times. The crossbar switch and the multiported memory organization are both single-stage networks.

This is because even if two processors attempted to access the same memory module (or I/O device) at the same time, only one of the requests is serviced at a time.

Multistage Networks

Multistage networks consist of multiple sages of switch boxes, and should be able to connect any input to any output.

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.

A nonblocking multistage network can perform all possible connections between inputs and outputs by rearranging its connections.

Crossbar Networks

Crossbar networks connect every input to every output through a crosspoint switch. A crossbar network is a single stage, non-blocking permutation network.

In an n-processor, m-memory system, n * □m crosspoint switches will be required. Each crosspoint is a unary switch which can be open or closed, providing a point-to-point connection path between the processor and a memory module.

Crosspoint Switch Design

Out of n crosspoint switches in each column of an *n* *□*m* crossbar mesh, only one can be connected at a time.

Crosspoint switches must be designed to handle the potential contention for each memory module. A crossbar switch avoids competition for bandwidth by using $O(N^2)$ switches to connect N inputs to N outputs.

Although highly non-scalable, crossbar switches are a popular mechanism for connecting a small number of workstations, typically 20 or fewer.

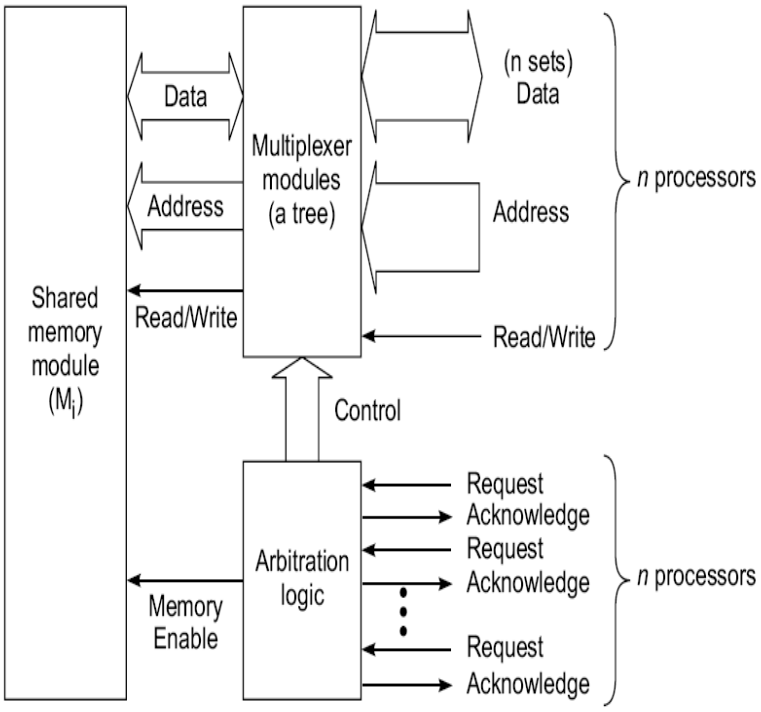


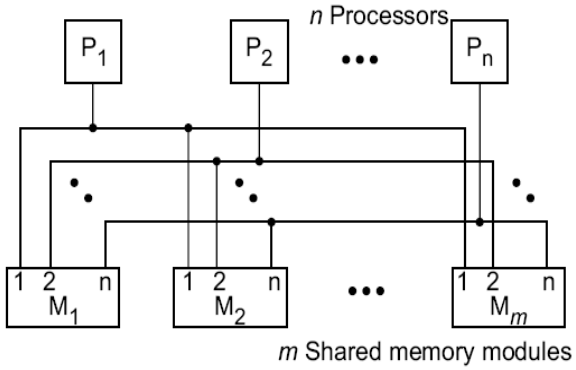
Fig. 7.6 Schematic design of a row of crosspoint switches in a crossbar network

Each processor provides a request line, a read/write line, a set of address lines, and a set of data lines to a crosspoint switch for a single column. The crosspoint switch eventually responds with an acknowledgement when the access has been completed.

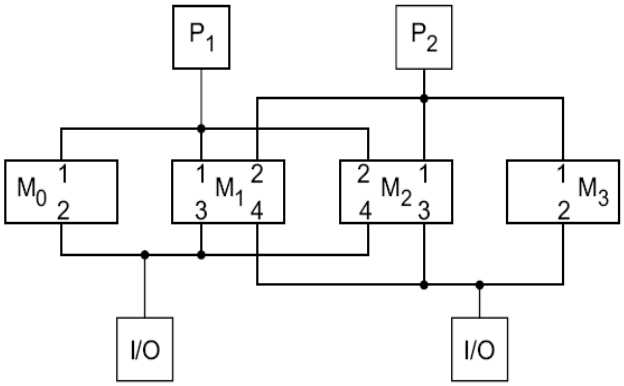
Multiport Memory

Since crossbar switches are expensive and not suitable for systems with many processors or memory modules, multiport memory modules may be used instead.

A multiport memory module has multiple connection points for processors (or I/O devices), and the memory controller in the module handles the arbitration and switching that might otherwise have been accomplished by a crosspoint switch.



(a) *n*-port memory modules used



(b) Memory ports prioritized or privileged in each module by numbers

Fig. 7.7 Multiport memory organizations for multiprocessor systems (Courtesy of P. H. Enslow, *ACM Computing Surveys*, March 1977)

A two function switch can assume only two possible state namely state or exchange states. However a four function switch box can be any of four possible states. A multistage network is capable of connecting any input terminal to any output terminal. Multi-stage networks are basically constructed by so called shuffle-exchange switching element, which is basically a 2 x 2 crossbar. Multiple layers of these elements are connected and form the network.

7.1.3 Multistage and Combining Networks

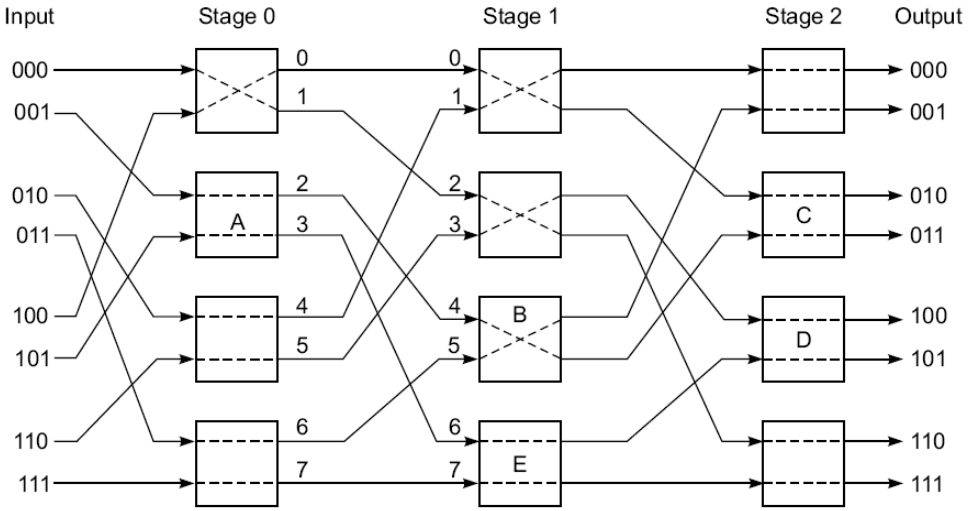
Multistage networks are used to build larger multiprocessor systems. We describe two multistage networks, the Omega network and the Butterfly network, that have been built into commercial machines.

Routing in Omega Networks

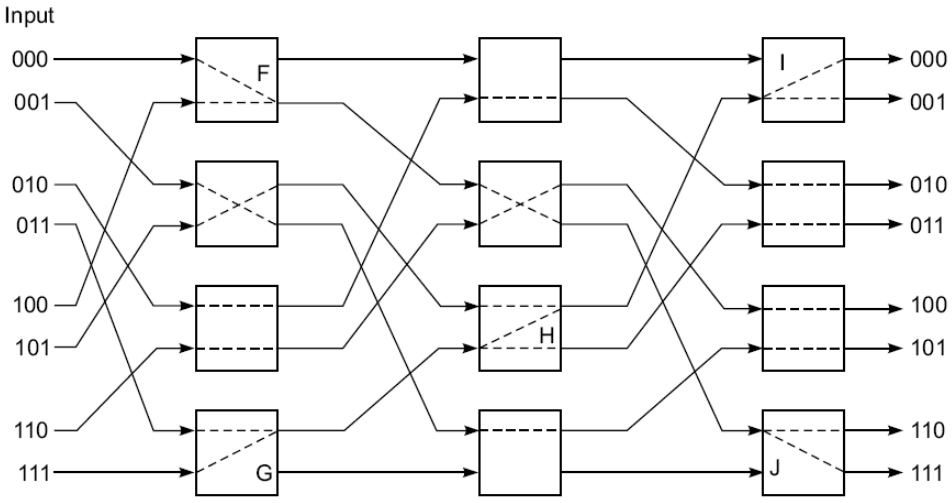
An 8-input Omega network is shown in Fig. 7.8.

In general, an n-input Omega network has l**og$_2$n** stages. The stages are labeled from 0 to log$_2$n — 1 from the input end to the output end.

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.



(a) Permutation π_1 = (0, 7, 6, 4, 2) (1, 3) (5) implemented on an Omega network without blocking



(b) Permutation π_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

- Two switch settings are shown in Figs. 7.8a and b with respect to permutations Π_1 = (0,7,6,4,2) (1,3)(5) and Π_2= (0,6,4,7,3) (1,5)(2), respectively.
- The switch settings in Fig. 7.8a are for the implementation of Π_1, which maps 0 →7, 7 →6, 6→4, 4→2, 2 →0, 1 →3, 3 →1, 5 →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 of the destination 011 is a "zero," switch A must be set straight so that the input 001 is connected to the upper output (labeled 2).

- 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 011 is a "one," implying a flat connection in switch C.
- Similarly, the switches A, E, and D are set for routing a message from input 101 to output 101. There exists no conflict in all the switch settings needed to implement the permutation Π_1 in Fig. 7.8a.

- Now consider implementing the permutation Π_2 in the 8-input Omega network (Fig. 7.8b0. 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.
- 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 as shown in fig. 2.24a.
- The above example indicates the fact that not all permutations can be implemented in one pass through the Omega network.

Routing in Butterfly Networks

- This class of networks is constructed with crossbar switches as building blocks. Fig. 7.10 shows two Butterfly networks of different sizes.
- Fig. 10a shows a 64-input Butterfly network built with two stages ($2=\log_8 64$) of 8X8 crossbar switches.
- The eight-way shuffle function is used to establish the interstage connections between stage 0 and stage 1.
- In Fig. 7.10b, a three-stage Butterfly network is constructed for 512 inputs, again with 8X8 crossbar switches.
- Each of the 64X64 boxes in Fig. 7.10b is identical to the two-stage Butterfly network in Fig. 7.10a.

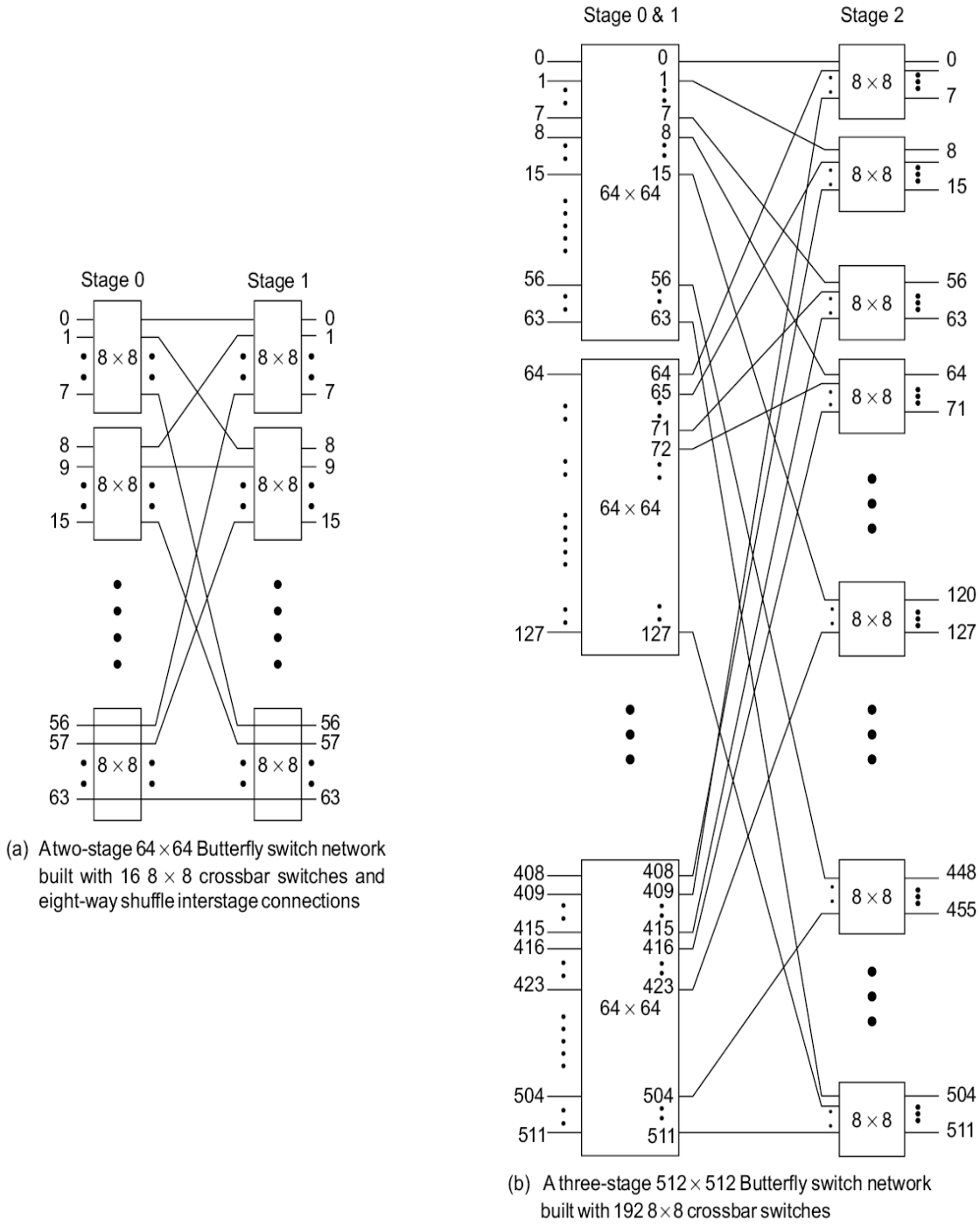


Fig. 7.10 Modular construction of Butterfly switch networks with 8 × 8 crossbar switches (Courtesy of BBN Advanced Computers, Inc., 1990)

- In total, sixteen 8x8 crossbar switches are used in Fig. 7.10a and 16 x 8+8 x 8 = 192 are used in Fig. 7.10b. Larger Butterfly networks can be modularly constructed using more stages.
- Note that no broadcast connections are allowed in a Butterfly network, making these networks a restricted subclass of Omega networks.

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.
- For example, a semaphore variable being used as a synchronization barrier may become a hot spot since it is shared by many processors.
- Hot spots may degrade the network performance significantly. In the NYU Ultracomputer and the IBM RP3 multiprocessor, a combining mechanism has been added to the Omega network.
- The purpose was to combine multiple requests heading for the same destination at switch points where conflicts are taking place.
- An atomic read-modify-write primitive Fetch&Add(x,e), has been developed to perform parallel memory updates using the combining network.

Fectch&Add

- This atomic memory operation is effective in implementing an N-way synchronization with a complexity independent of N.
- In a Fetch&Add(x, e) operation, i is an integer variable in shared memory and e is an integer increment.
- When a single processor executes this operation, the semantics is

```
Fetch&Add(x, e)
{   temp ← x;
    x ← temp + e;                                   (7.1)
    return temp   }
```

- When N processes attempt to Fetch&Add(x, e) the same memory word simultaneously, the memory is updated only once following a serialization principle.
- The sum of the N increments, $e_1 + e_2 + \bullet \bullet \bullet + e_N$, is produced in any arbitrary serialization of the N requests.
- This sum is added to the memory word x, resulting in a new value $x + e_1 + e_2 + \bullet \bullet \bullet + e_N$
- The values returned to the N requests are all unique, depending on the serialization order followed.
- The net result is similar to a sequential execution of N Fetch&Adds but is performed in one indivisible operation.
- Two simultaneous requests are combined in a switch as illustrated in Fig. 7.11.

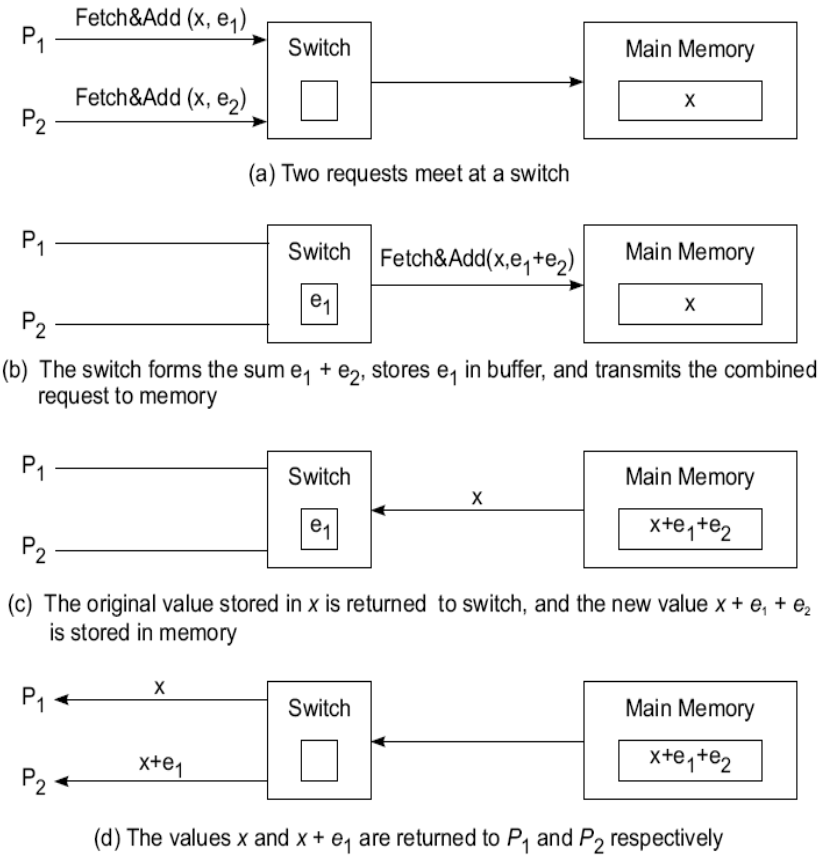


Fig. 7.11 Two Fetch&Add operations are combined to access a shared variable simultaneously via a combining network

One of the following operations will be performed if processor P1 executes **Ans_1 ← Fetch&Add(x,e_1)** and P_2 executes **Ans_2 ← Fetch&Add(x,e_2)** simultaneously on the shared variable x.

If the request from P_1 is executed ahead of that from P_2, the following values are returned:

$Ans_1 \leftarrow x$

$Ans_2 \leftarrow x + e_1$ (7.2)

If the execution order is reversed, the following values arc returned:

$Ans_1 \leftarrow x + e_2$

$Ans_2 \leftarrow x$

Regardless of the executing order, the value **$x + e_1 + e_2$** is stored in memory.

It is the responsibility of the switch box to form the sum **$e_1 + e_2$**, transmit the combined request **Fetch&Add(x, $e_1 + e_2$)**, store the value **e1 (or e2)** in a wait buffer of the switch and return the values x and **$x + e_1$** to satisfy the original requests **Fetch&Add(x, e_1)** and **Fetch&Add(x, e_2)** respectively, as shown in fig. 7.11 in four steps.

7.2 Cache Coherence and Synchronization Mechanisms

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.

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.

- 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 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 (X and X') in the two caches.
- On the other hand, inconsistency may also occur when a write back policy is used, as shown on the right.
- The main memory will be eventually updated when the modified data in the cache are replaced or invalidated.

Process Migration and I/O

The figure 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 processor1 when using write-through caches.

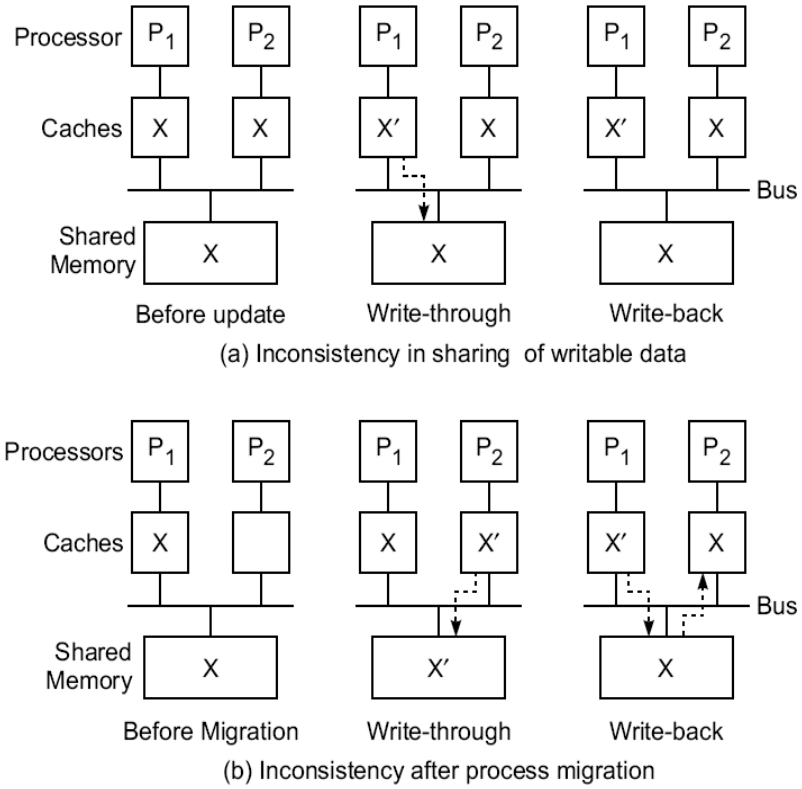


Fig. 7.12 Cache coherence problems in data sharing and in process migration (Adapted from Dubois, Scheurich, and Briggs 1988)

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 rnigrate from one processor to another.

Two Protocol Approaches for Cache Coherence

- Many of the early commercially available multiprocessors used 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 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 links 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.

Protocol Approaches for Cache Coherence:

1. Snoopy Bus Protocol
2. Directory Based Protocol

1. Snoopy Bus Protocol

- Snoopy protocols achieve data consistency among the caches and shared memory through a bus watching mechanism.
- In the following diagram, two snoopy bus protocols create different results. Consider 3 processors (P_1, P_2, P_n) maintaining consistent copies of block X in their local caches (Fig. 7.14a) and in the shared memory module marked X.

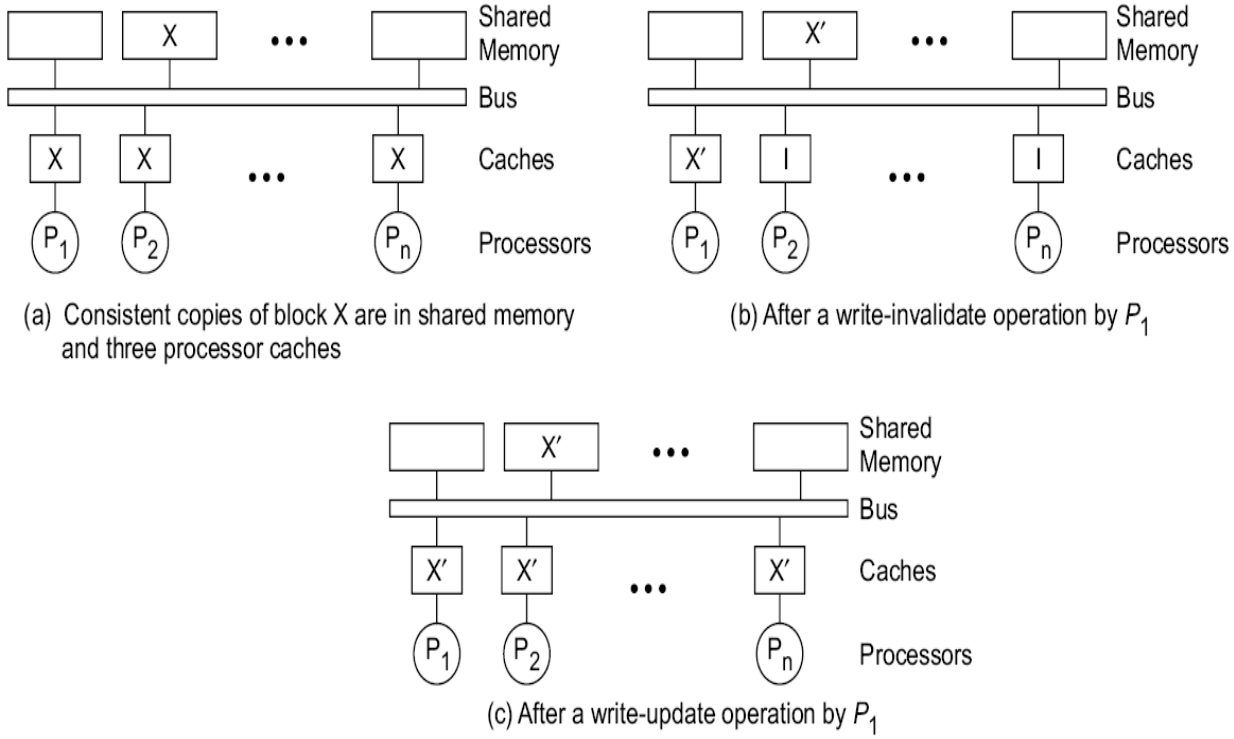
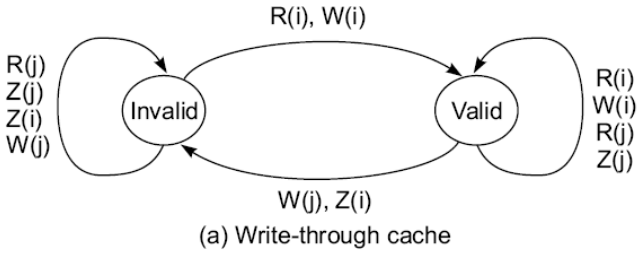


Fig. 7.14 Write-invalidate and write-update coherence protocols for write through caches (1: invalidate)

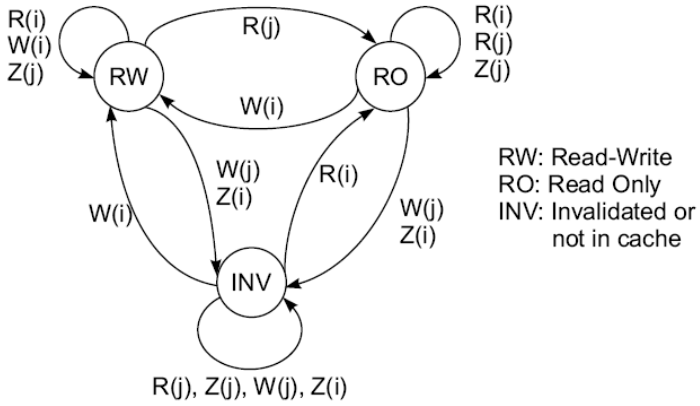
- Using a write-invalidate protocol, the processor P_1 modifies (writes) its cache from X to X', and all other copies are invalidated via the bus (denoted I in Fig. 7.14b). Invalidated blocks are called dirty, meaning they should not be used.
- The write-update protocol (Fig. 7.14c) demands the new block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



(a) Write-through cache



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 $j \neq i$.
R(j) = Read block copy in cache *j* by processor $j \neq i$.
Z(j) = Replace block copy in cache $j \neq i$.

(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)

- 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)).

Write Back Caches:

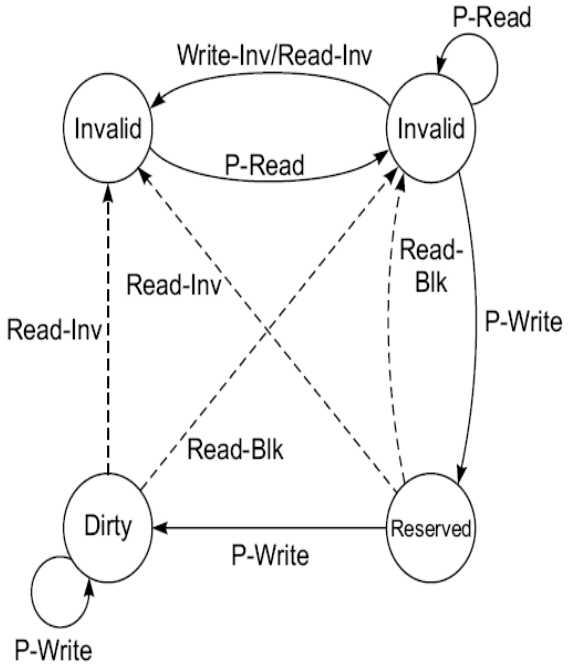
- The valid state of a write-back cache can be further split into two cache states, **Labeled RW(read-write)** and **RO(read-only)** as shown in Fig.7.15b.
- The INV (invalidated or not-in-cache) cache state is equivalent to the invalid state mentioned before. This three-state coherence scheme corresponds 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)) 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:

- *Valid:* The cache block, which is consistent with the memory copy, has been *read* from shared memory and has not been modified.
- *Invalid:* The block is not found in the cache or is inconsistent with the memory copy.
- *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.

Dirty: The 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).



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)

2. Directory Based Protocol

A write-invalidate protocol may lead to heavy bus traffic caused by *read-misses*, resulting from the processor updating a variable and other processors trying to read the same variable. On the other hand, the write-update protocol may update data items in remote caches which will never be used by other processors. In fact, these problems pose additional limitations in using buses to build large multiprocessors.

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. Various directory-based protocols differ mainly in how the directory maintains information and what information it stores.

Tang (1976) proposed 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.

A distributed-directory scheme was proposed by Censier and Feautrier (1978). Each memory module maintains a separate directory which records the state and presence information for each memory block. The state information is local, but the presence information indicates which caches have a copy of the block.

In Fig. 7.17, a *read-miss* (thin lines) in cache 2 results in a request sent to the memory module. The memory controller retransmits the request to the dirty copy in cache 1. This cache *writes back* its copy. The memory module can supply a copy to the requesting cache. In the case of a *write-hit* at cache 1 (bold lines), a command is sent to the memory controller, which sends invalidations to all caches (cache 2) marked in the presence vector residing in the directory D_1.

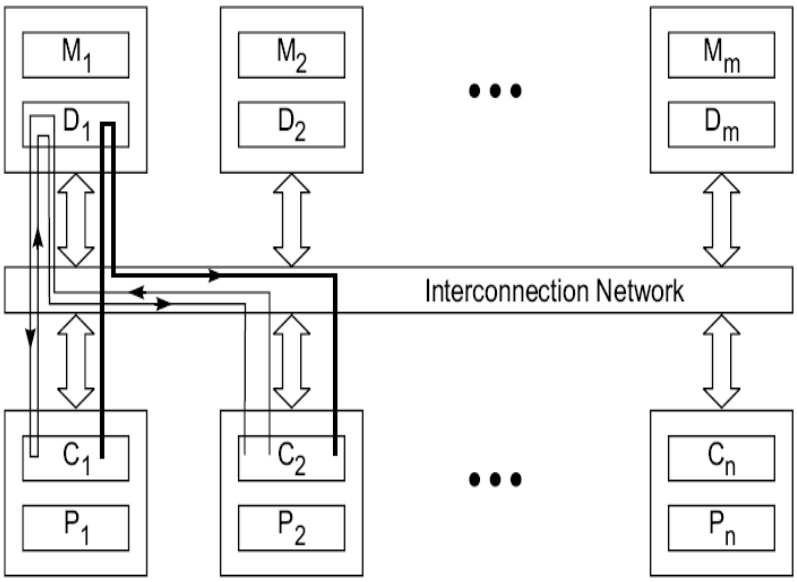


Fig. 7.17 Basic concept of a directory-based cache coherence scheme (Courtesy of Censier and Feautrier, *IEEE Trans. Computers,* Dec. 1978)

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*, and *chained 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.

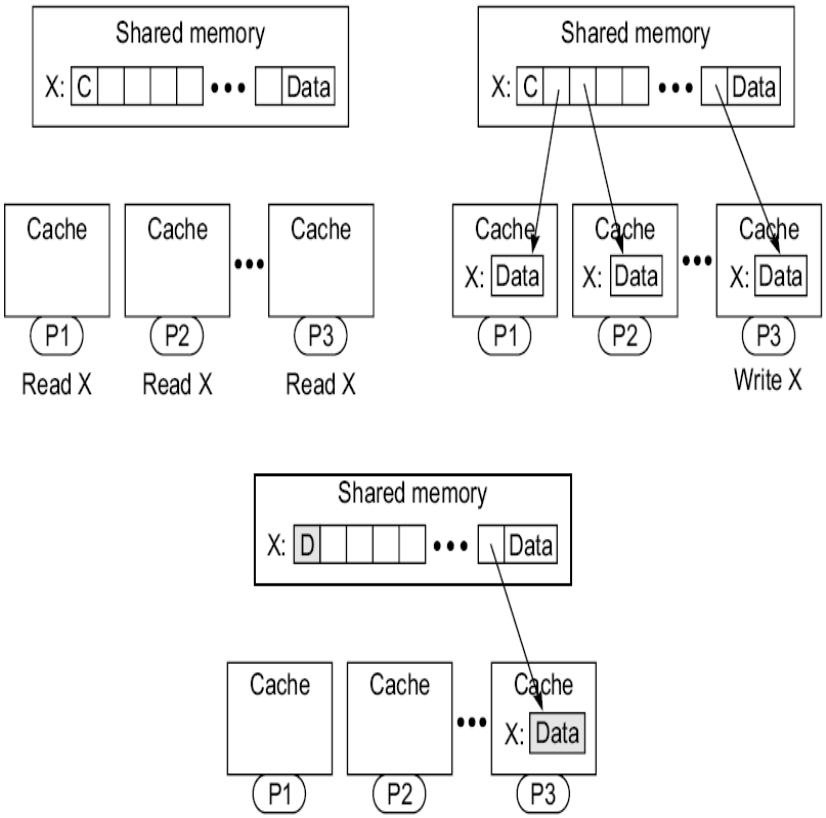
Full-Map Directories The full-map protocol implements directory entries with one bit per processor and a dirty bit. Each bit represents the status of the block in the corresponding processor's cache (present or absent). If the dirty bit is set, then one and only one processor's bit is set and that processor can write into the block.

A cache maintains two bits of state per block. One bit indicates whether a block is valid, and the other indicates whether a valid block may be written. The cache coherence protocol must keep the state bits in the memory directory and those in the cache consistent.

Figure 7.18a illustrates three different 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 (C1, 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:

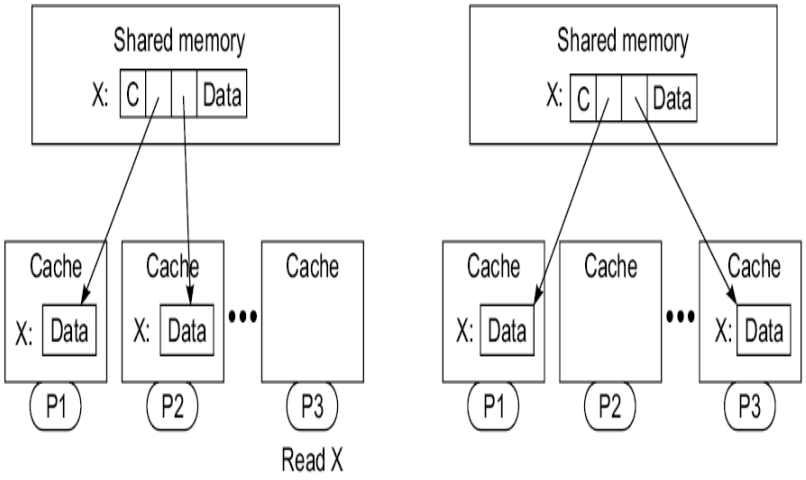
(1) Cache C3 detects that the block containing location X is valid but that the processor does not have permission to write to the block, indicated by the block's write-permission bit in the cache.
(2) Cache C3 issues a write request to the memory module containing location X and stalls processor P3.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



(a) Three states of a full-map directory

Limited Directories Limited directory protocols are designed to solve the directory size problem. Restricting the number of simultaneously cached copies of any particular block of data limits the growth of the directory to a constant factor.

A directory protocol can be classified as $Dir_i\ X$ using the notation from Agarwal et al (1988). 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 $Dir_N\ NB$. A limited directory protocol that uses $i < N$ pointers is denoted $Dir_i\ NB$. The limited directory protocol is similar to the full-map directory, except in the case when more than i caches request read copies of a particular block of data.



(b) Eviction in a limited directory

Figure 7.18b shows the situation when three caches request read copies in a memory system with a $Dir_2 NB$ 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 Cl 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.

If the multiprocessor exhibits processor locality in the sense that in any given interval of time only a small subset of all the processors access a given memory word, then a limited directory is sufficient to capture this small worker set of processors.

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. 7.18c). 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.

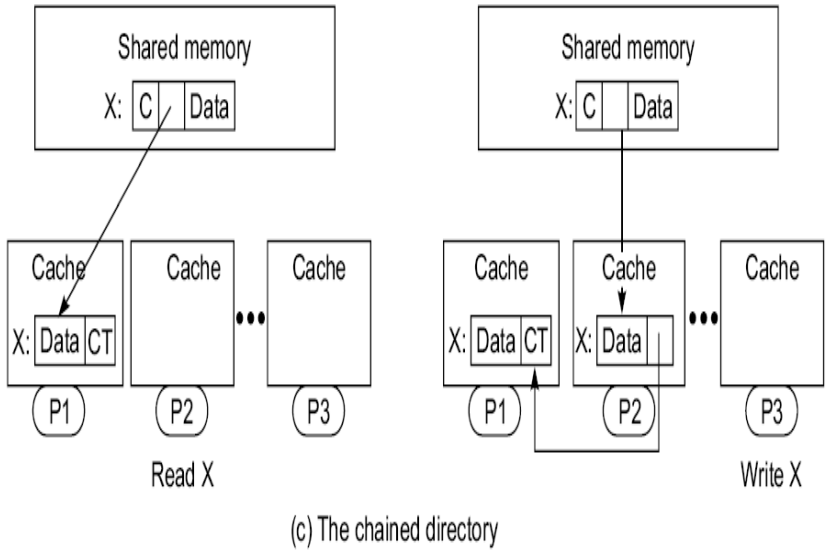
By repeating the above step, all of the caches can cache a copy of the location X. If processor P3 writes to location X, it is necessary to send a data invalidation message down the chain. To ensure sequential consistency, the memory module denies processor P3 write permission until the processor with 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.

The possibility of cache block replacement complicates chained-directory protocols.

Suppose that caches C1 through CN all have copies of location X and that location X and location Y map to the same (direct-mapped) cache line. If processor P_i reads location Y, it must first evict location X from its cache with the following possibilities:

(1) Send a message down the chain to cache C_{i-1} with a pointer to cache C_{i+1} and splice C_i out of the chain, or

(2) Invalidate location X in cache C_{i+1} through cache C_N.



(c) The chained directory

Fig. 7.18 Three types of cache directory protocols (Courtesy of Chaiken et al., *IEEE Computer*, June 1990)

7.4 Message – Passing Mechanisms

Message passing in a multicomputer network demands special hardware and software support. In this section, we study the store-and-forward and wormhole routing schemes and analyze their communication latencies. We introduce the concept of virtual channels. Deadlock situations in a message-passing network are examined. We show how to avoid deadlocks using virtual channels.

7.4.1 Message-Routing Schemes

Message formats are introduced below. Refined formats led to the improvement from store-and-forward to wormhole routing in two generations of multicomputers. A handshaking protocol is described for asynchronous pipelining of successive routers along a communication path. Finally, latency analysis is conducted to show the time difference between the two routing schemes presented.

Message Formats Information units used in message routing are specified in Fig. 7.26. A *message* is the logical unit for internode communication. It is often assembled from an arbitrary number of fixed-length packets, thus it may have a variable length.

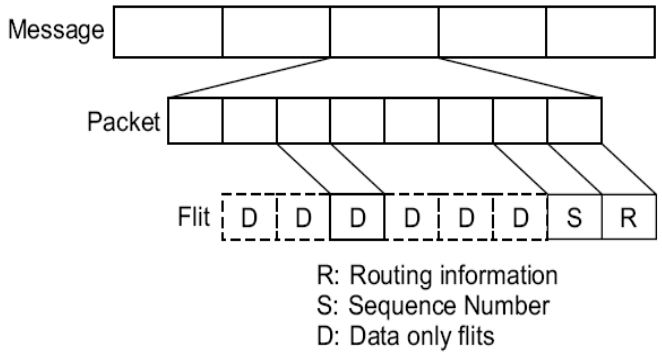


Fig.7.26 The format of message, packets, and flits (control flow digits) used as information units of communication in a message-passing network

A *packet* is the basic unit containing the destination address for routing purposes. Because different packets may arrive at the destination asynchronously, a sequence number is needed in each packet to allow reassembly of the message transmitted.

A packet can be further divided into a number of fixed-length *flits* (flow control digits). Routing information (destination) and sequence number occupy the header flits. The remaining flits are the data elements of a packet.

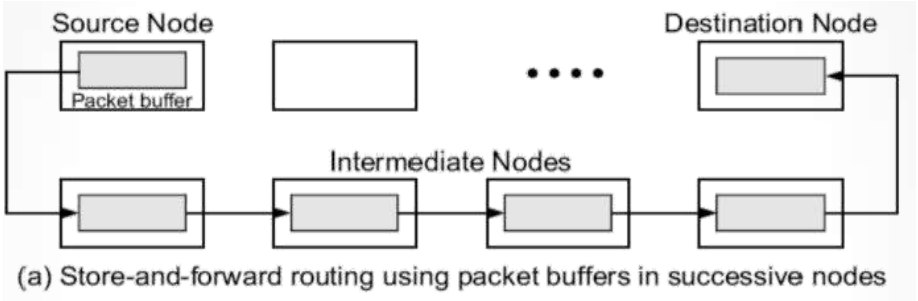
In multicomputers with store-and-forward routing, packets are the smallest unit of information transmission. In wormhole-routed networks, packets are further subdivided into flits. The flit length is often affected by the network size.

The packet length is determined by the routing scheme and network implementation. Typical packet lengths range from 64 to 512 bits. The sequence number may occupy one to two flits depending on the message length. Other factors affecting the choice of packet and flit sizes include channel bandwidth, router design, network traffic intensity, etc.

Two Message Passing mechanisms are:

1. Store and Forward Routing
2. Wormhole Routing

1. Store and Forward Routing

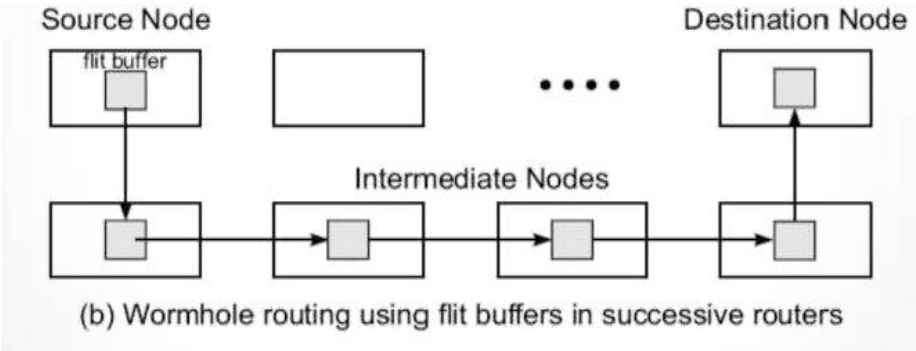


(a) Store-and-forward routing using packet buffers in successive nodes

- Packets are the basic unit of information flow in a store-and-forward network.
- Each node is required to use a packet buffer.

- A packet is transmitted from a source node to a destination node through a sequence of intermediate nodes.
- When a packet reaches an intermediate node, it is first stored in the buffer.
- Then it is forwarded to the next node if the desired output channel and a packet buffer in the receiving node are both available.

2. Wormhole Routing



(b) Wormhole routing using flit buffers in successive routers

- Packets are subdivided into smaller flits. Flit buffers are used in the hardware routers attached to nodes.
- The transmission from the source node to the destination node is done through a sequence of routers.
- All the flits in the same packet are transmitted in order as inseparable companions in a pipelined fashion.
- Only the header flit knows where the packet is going.
- All the data flits must follow the header flit.
- Flits from different packets cannot be mixed up. Otherwise they may be towed to the wrong destination.

Asynchronous Pipelining

- The pipelining of successive flits in a packet is done asynchronously using a handshaking protocol as shown in Fig. 7.28. Along the path, a 1-bit ready/request (R/A) line is used between adjacent routers.

- When the receiving router (D) is ready (7.28a) to receive a flit (ie., a flit buffer is available), it pulls the R/A line low. When the sending router (S) is ready (Fig. 2.8b), it raises the line high and transmits flit I through the channel.
- While the flit is being received by D (Fig. 7.28c), the R/A line is kept high. After flit I is removed from D's buffer (ie., transmitted to the next node) (Fig. 7.28d), the cycle repeats itself for the transmission of the next flit i+1 until the entire packet is transmitted.

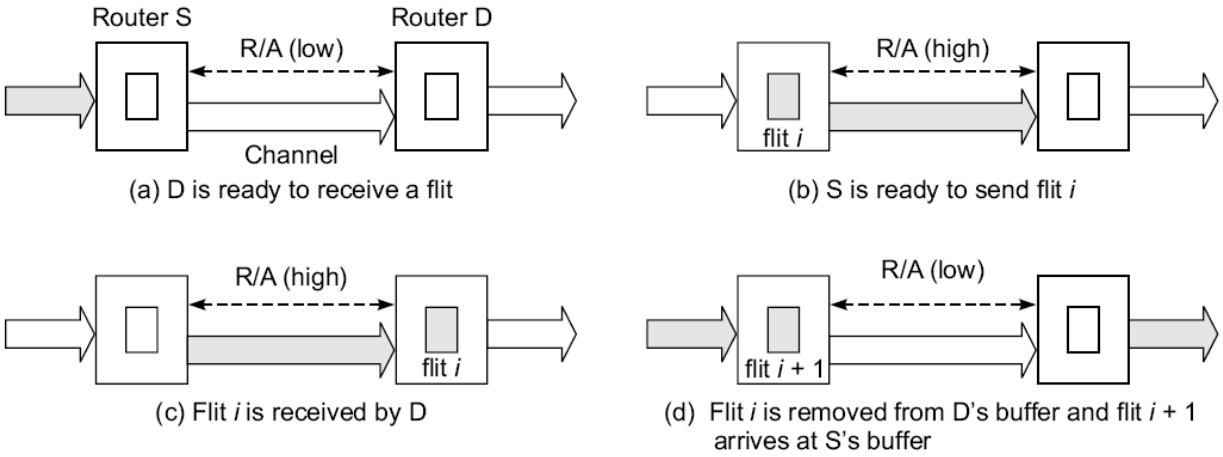
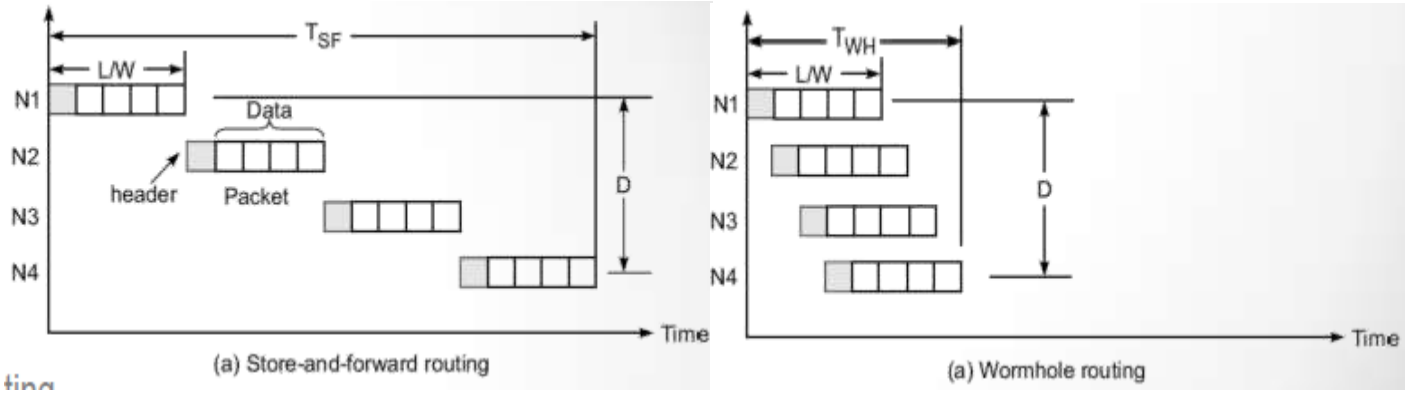


Fig. 7.28 Handshaking protocol between two wormhole routers (Courtesy of Lionel Ni, 1991)

Advantages:

- Very efficient
- Faster clock

Latency Analysis:



- The communication latency in store-and-forward networks is directly proportional to the distance (the number of hops) between the source and the destination.

  $\mathbf{T_{SF} = L\ (D + 1)\ /\ W}$

- Wormhole Routing has a latency almost independent of the distance between the source and the destination

  $\mathbf{T_{WH} = L\ /\ W + F\ D\ /\ W}$

  where, L: Packet length (in bits)
  W: Channel Bandwidth (in bits per second)
  D: Distance (number of nodes traversed minus 1)
  F: Flit length (in bits)

7.4.2 Deadlock and Virtual channels

The communication channels between nodes in a wormhole-routed multicomputer network are actually shared by many possible source and destination pairs. The sharing of a physical channel leads to the concept of virtual channels.

Virtual channels

- A virtual channel is logical link between two nodes. It is formed by a flit buffer in the source node, a physical channel between them and a flit buffer in the receiver node.
- Four flit buffers are used at the source node and receiver node respectively. One source buffer is paired with one receiver buffer to form a virtual channel when the physical channel is allocated for the pair.
- Thus the physical channel is time shared by all the virtual channels. By adding the virtual channel the channel dependence graph can be modified and one can break the deadlock cycle.
- Here the cycle can be converted to spiral thus avoiding a deadlock. Virtual channel can be implemented with either unidirectional channel or bidirectional channels.
- However a special arbitration line is needed between adjacent nodes interconnected by bidirectional channel. This line determines the direction of information flow.
- The virtual channel may reduce the effective channel bandwidth available to each request.

- There exists a tradeoff between network throughput and communication latency in determining the degree of using virtual channels.

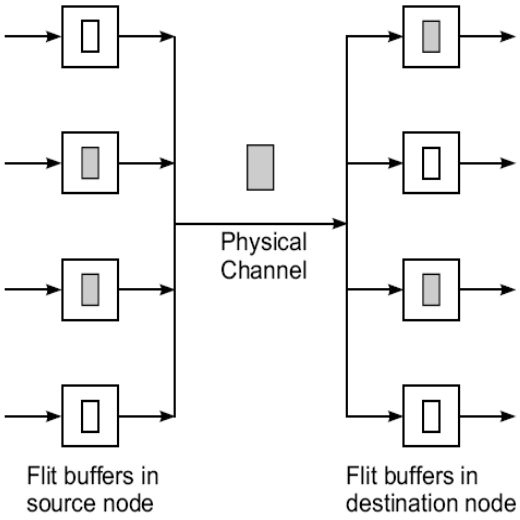


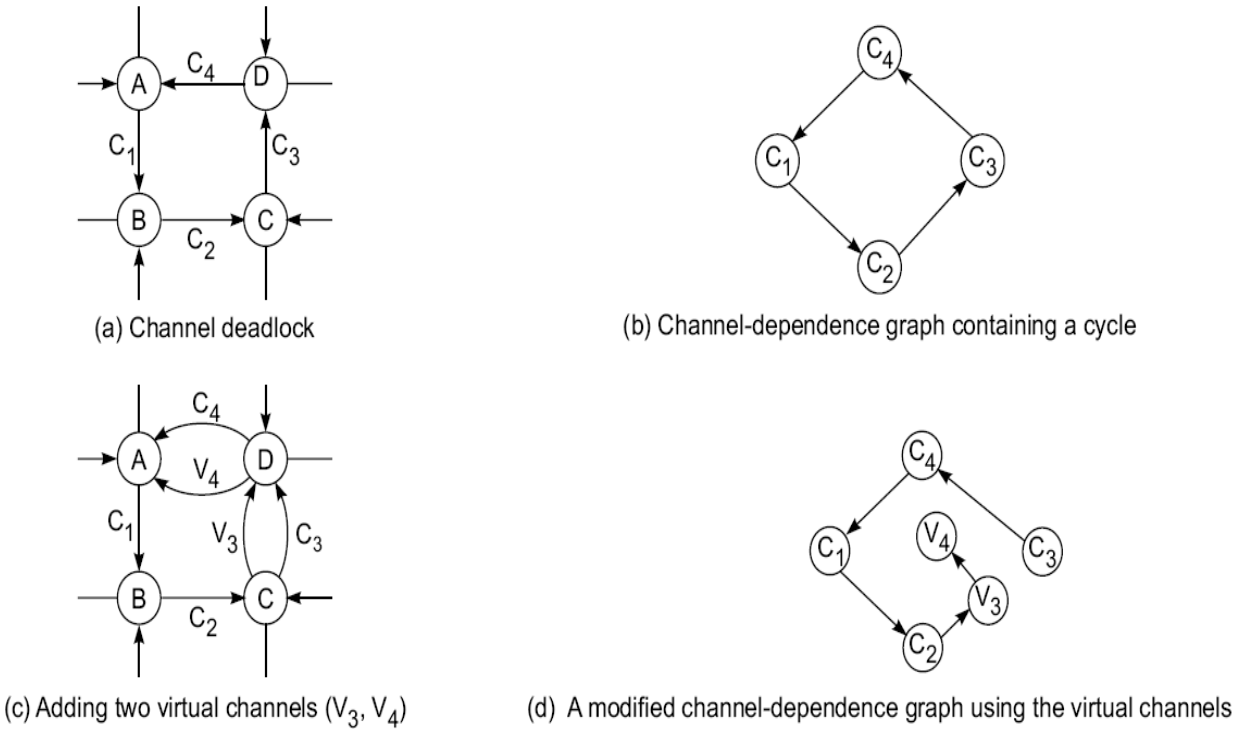
Fig. 7.30 Four virtual channels sharing a physical channel with time multiplexing on a flit-by-flit basis

Deadlock Avoidance

By adding two virtual channels, V_3 and V_4 in Fig. 7.32c, one can break the deadlock cycle. A modified channel-dependence graph is obtained by using the virtual channels V_3 and V_4, after the use of channel C_2, instead of reusing C_3 and C_4.

The cycle in Fig. 7.32b is being converted to a spiral, thus avoiding a deadlock. Channel multiplexing can be done at the flit level or at the packet level if the packet length is sufficiently short.

Virtual channels can be implemented with either unidirectional channels or bidirectional channels.



(a) Channel deadlock

(b) Channel-dependence graph containing a cycle

(c) Adding two virtual channels (V_3, V_4)

(d) A modified channel-dependence graph using the virtual channels

Fig. 7.32 Deadlock avoidance using virtual channels to convert a cycle to a spiral on a channel-dependence graph

Chapter-8 Multivector and SIMD Computers

8.1 Vector Processing Principles

Vector Processing Definitions

Vector: A vector is a set of scalar data items, all of the same type, stored in memory. Usually, the vector elements are ordered to have a fixed addressing increment between successive elements called the stride.

Vector Processor: A vector processor is an ensemble of hardware resources, including vector registers, functional pipelines, processing elements, and register counters, for performing vector operations.

Vector Processing: Vector processing occurs when arithmetic or logical operations are applied to vectors. It is distinguished from scalar processing which operates on one or one pair of data.

Vector processing is faster and more efficient than scalar processing.

Vectorization: The conversion from scalar code to vector code is called vectorization.

Vectorizing Compiler: A compiler capable of vectorization is called a Vectorizing Compiler (vectorizer).

8.1.1 Vector Instruction Types

There are six types of vector instructions. These are defined by mathematical mappings between their working registers or memory where vector operands are stored.

1. Vector - Vector instructions
2. Vector - Scalar instructions
3. Vector - Memory instructions
4. Vector reduction instructions
5. Gather and scatter instructions
6. Masking instructions

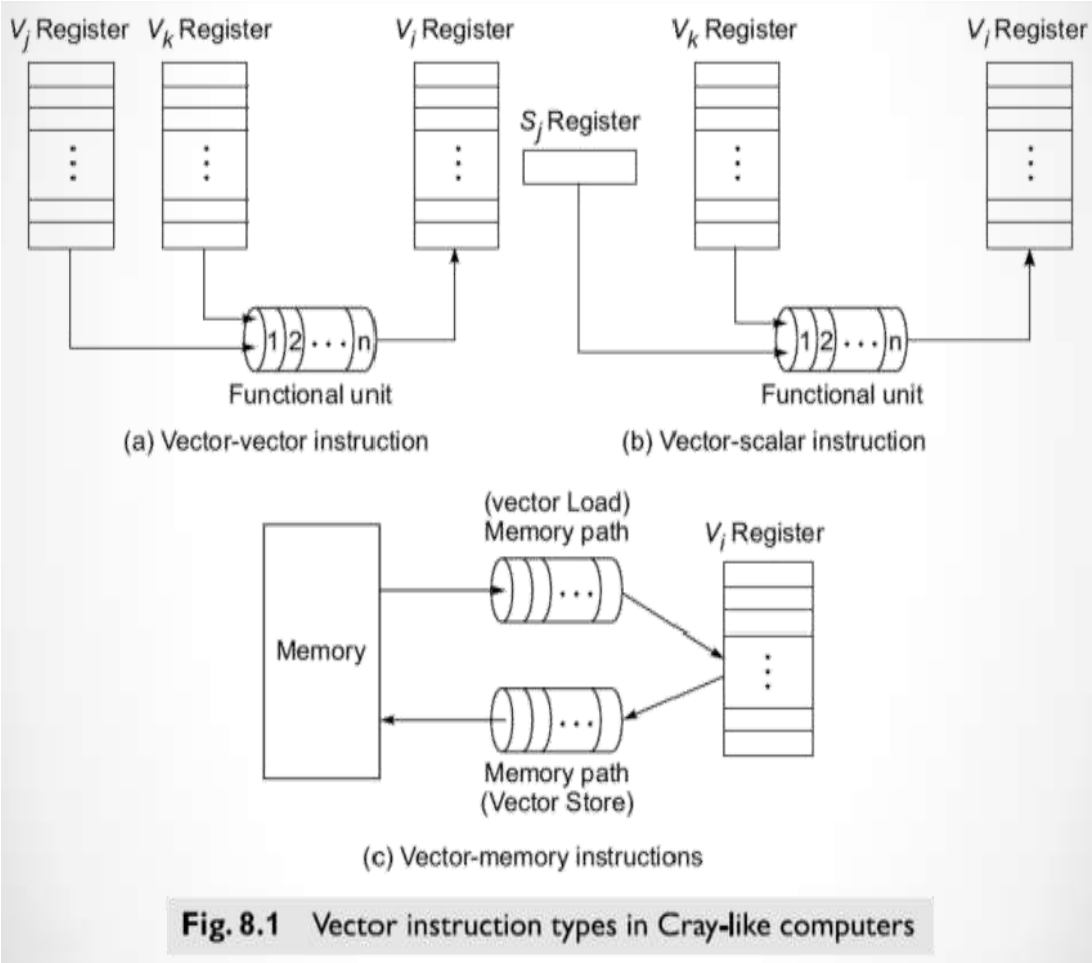


Fig. 8.1 Vector instruction types in Cray-like computers

1. **Vector - Vector instructions:** One or two vector operands are fetched form the respective vector registers, enter through a functional pipeline unit, and produce result in another vector register.

   F1: Vi → Vj

   F2: Vi x Vj →Vk

Examples: V1 = sin(V2), V3 = V1+ V2

2. **Vector - Scalar instructions**

   Elements of vector register are multiplied by a scalar value.

   F3: s x Vi → Vj

   Examples: V2 = 6 + V1

3. **Vector - Memory instructions:** This corresponds to Store-load of vector registers (V) and the Memory (M).

   F4: M → V (Vector Load)

   F5: V → M (Vector Store)

   Examples: X = V1 V2 = Y

4. **Vector reduction instructions:** include maximum, minimum, sum, mean value.

   F6: Vi → s

   F7: Vi x Vj → s

5. **Gather and scatter instructions** Two instruction registers are used to gather or scatter vector elements randomly throughout the memory corresponding to the following mappings

   F8: M → Vi x Vj (Gather)

   F9: Vi x Vj → M (Scatter)

   Gather is an operation that fetches from memory the nonzero elements of a sparse vector using indices.

   Scatter does the opposite, storing into memory a vector in a sparse vector whose nonzero entries are indexed.

6. **Masking instructions** The Mask vector is used to compress or to expand a vector to a shorter or longer index vector (bit per index correspondence).

F10: Vi x Vm → Vj (Vm is a ***binary vector***)

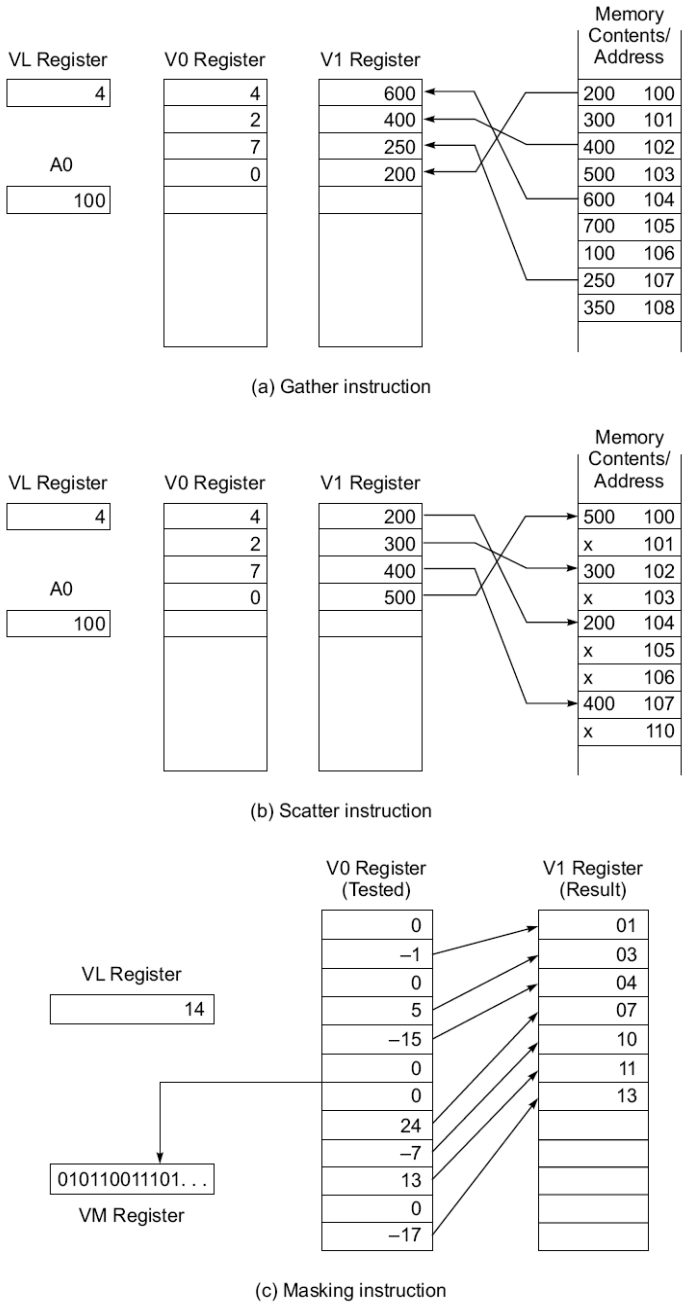


Fig. 8.2 Gather, scatter and masking operations on the Cray Y-MP (Courtesy of Cray Research, 1990)

- The *gather*, *scatter*, and *masking* instructions are very useful in handling sparse vectors or sparse matrices often encountered in practical vector processing applications.
- Sparse matrices are those in which most of the entries arc zeros.
- Advanced vector processors implement these instructions directly in hardware.

8.1.2 Vector-Access Memory Schemes

The flow of vector operands between the main memory and vector registers is usually pipelined with multiple access paths.

Vector Operand Specifications

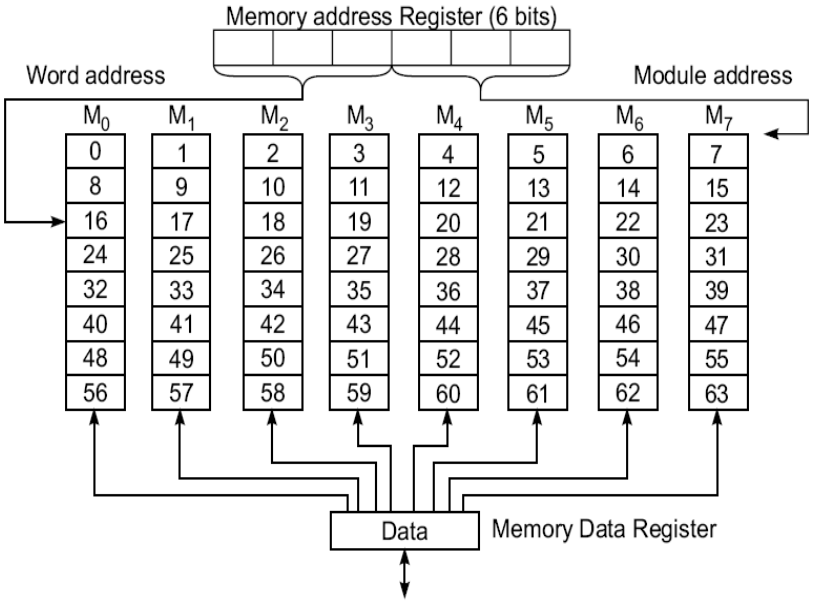
- Vector operands may have arbitrary length.
- Vector elements are not necessarily stored in contiguous memory locations.
- To access a vector a memory, one must specify its base, stride, and length.
- Since each vector register has fixed length, only a segment of the vector can be loaded into a vector register.
- Vector operands should be stored in memory to allow pipelined and parallel access. Access itself should be pipelined.

Three types of Vector-access memory organization schemes

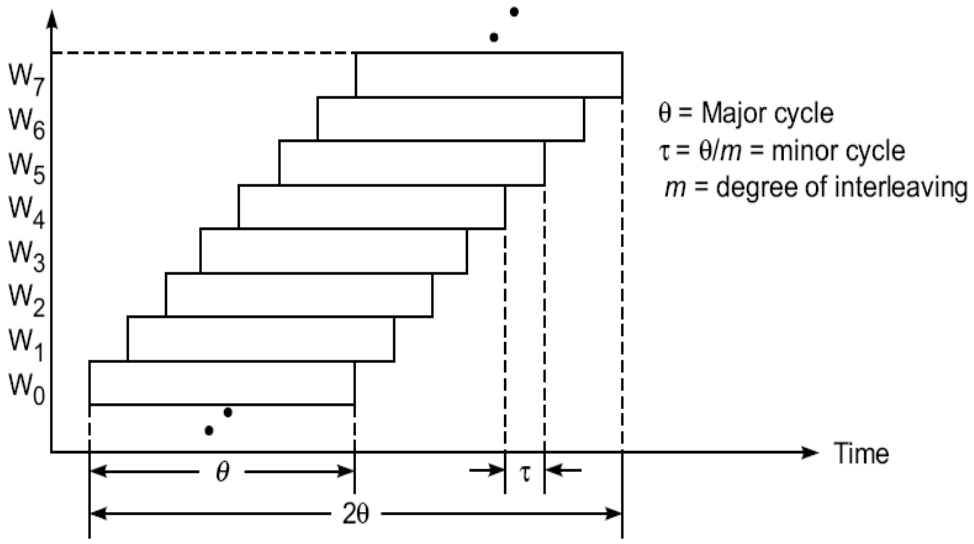
1. C-Access memory organization

The m-way low-order memory structure, allows **m** words to be accessed concurrently and overlapped.

The access cycles in different memory modules are staggered. The low-order **a** bits select the modules, and the high-order **b** bits select the word within each module, where $\mathbf{m=2^a}$ and **a+b = n** is the address length.



(a) Eight-way low-order interleaving (absolute address shown in each memory word)



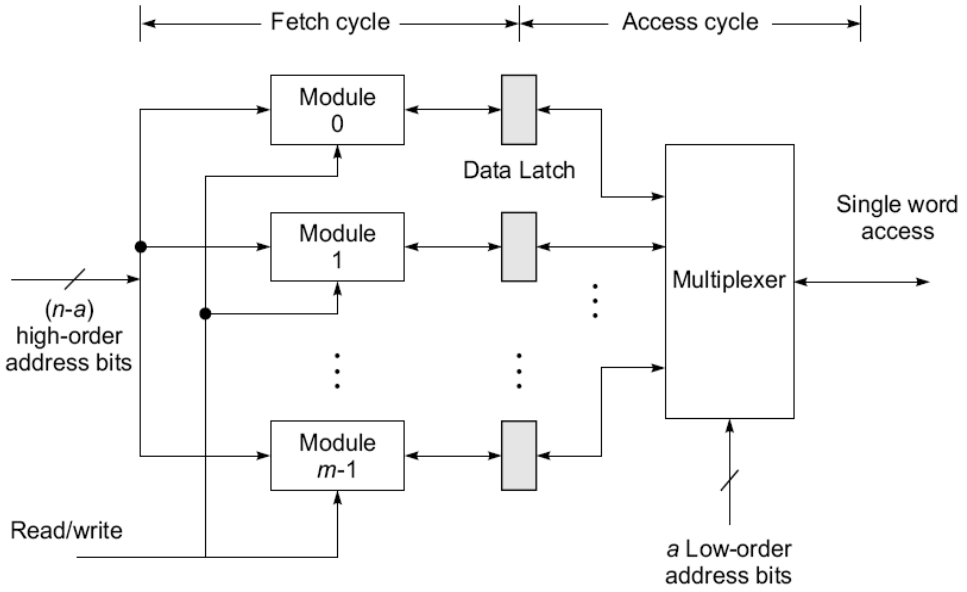
(b) Pipelined access of eight consecutive words in a C-access memory

Fig. 5.16 Multiway interleaved memory organization and the C-access timing chart

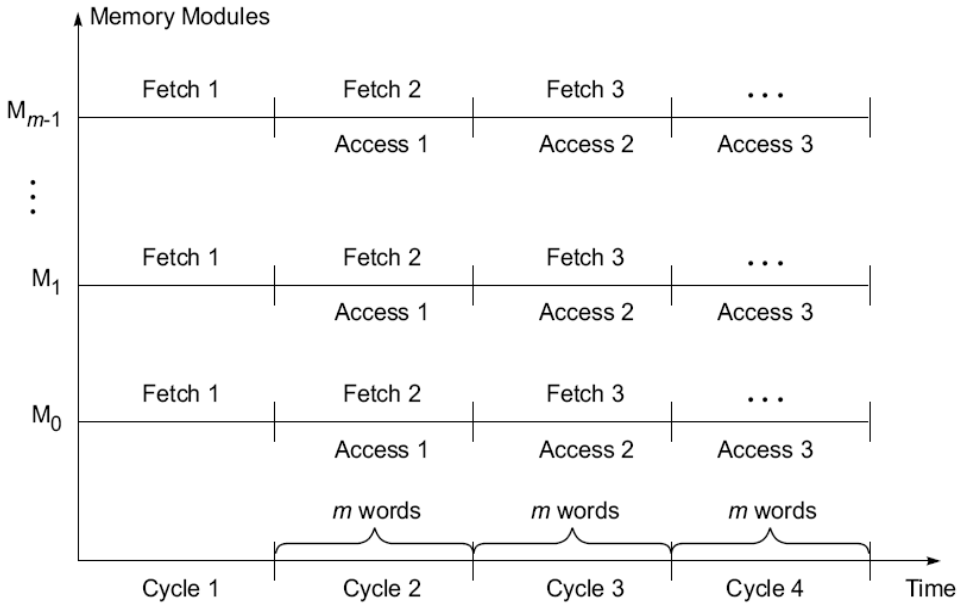
- To access a vector with a stride of 1, successive addresses are latched in the address buffer at the rate of one per cycle.
- Effectively it takes m **minor** cycles to fetch m words, which equals one **(major) memory** cycle as stated in Fig. 5.16b.
- If the stride is 2, the successive accesses must be separated by two minor cycles in order to avoid access conflicts. This reduces the memory throughput by one-half.
- If the stride is 3, there is no module conflict and the maximum throughput (m words) results.
- In general, C-access will yield the maximum throughput of m words per memory cycle if the stride is relatively prime to m, the number of interleaved memory modules.

2. S-Access memory organization

All memory modules are accessed simultaneously in a synchronized manner. The high order **(n-a)** bits select the same offset word from each module.



(a) S-access organization for an *m*-way interleaved memory



(b) Successive vector accesses using overlapped fetch and access cycles

Fig. 8.3 The S-access interleaved memory for vector operands access

At the end of each memory cycle (Fig. 8.3b), $\mathbf{m = 2^a}$ consecutive words are latched. If the stride is greater than 1, then the throughput decreases, roughly proportionally to the stride.

3. C/S-Access memory organization

- Here C-access and S-access are combined.
- **n** access buses are used with **m** interleaved memory modules attached to each bus.

- The **m** modules on each bus are **m-way** interleaved to allow C-access.
- In each memory cycle, at most **m.n** words are fetched if the **n** buses are fully used with pipelined memory accesses

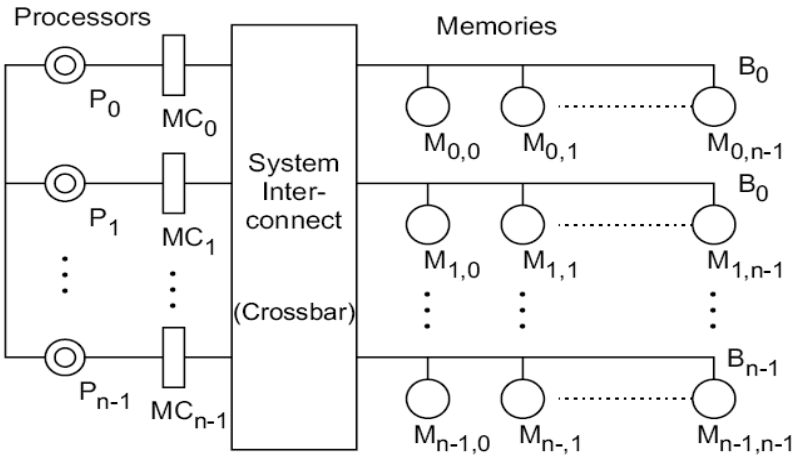


Fig. 8.4 The C/S memory organization with *m* = *n*. (Courtesy of D.K. Panda, 1990)

- The C/S-access memory is suitable for use in vector multiprocessor configurations.
- It provides parallel pipelined access of a vector data set with high bandwidth.
- A special vector cache design is needed within each processor in order to guarantee smooth data movement between the memory and multiple vector processors.

8.3 Compound Vector Processing

A compound vector function (CVF) is defined as a composite function of vector operations converted from a looping structure of linked scalar operations.

```
Do 10 I=1,N
Load R1, X(I)
Load R2, Y(I)
Multiply R1, S
Add R2, R1
Store Y(I), R2
10 Continue
```

where X(I) and Y(I), I=1, 2,…. N, are two source vectors originally residing in the memory. After the computation, the resulting vector is stored back to the memory. S is an immediate constant supplied to the multiply instruction.

After vectorization, the above scalar SAXPY code is converted to a sequence of five vector instructions:

M(x : x + N-1) → V1	Vector Load
M(y : y + N-1) → V2	Vector Load
S X V1 → V1	Vector Multiply
V2 X V1 → V2	Vector Add
V2 → M(y : y + N-1)	Vector Store

X and y are starting memory addresses of the X and Y vectors, respectively; V1 and V2 are two N-element vector registers in the vector processor.

CVF: Y(1:N) = S X(1:N) + Y(1:N) or Y(I) = S X(I) + Y(I)

where Index I implies that all vector operations involve N elements.

- Typical CVF for one-dimensional arrays are load, store, multiply, divide, logical and shifting operations.
- The number of available vector registers and functional pipelines impose some restrictions on how many CVFs can be executed simultaneously.
- **Chaining**:

  Chaining is an extension of technique of internal data forwarding practiced in scalar processors. Chaining is limited by the small number of functional pipelines available in a vector processor.
- **Strip-mining:**

  When a vector has a length greater than that of the vector registers, segmentation of the long vector into fixed-length segments is necessary. One vector segment is processed at a time (in Cray computers segment is 64 elements).
- **Recurrence**:

  The special case of vector loops in which the output of a functional pipeline may feed back into one of its own source vector registers

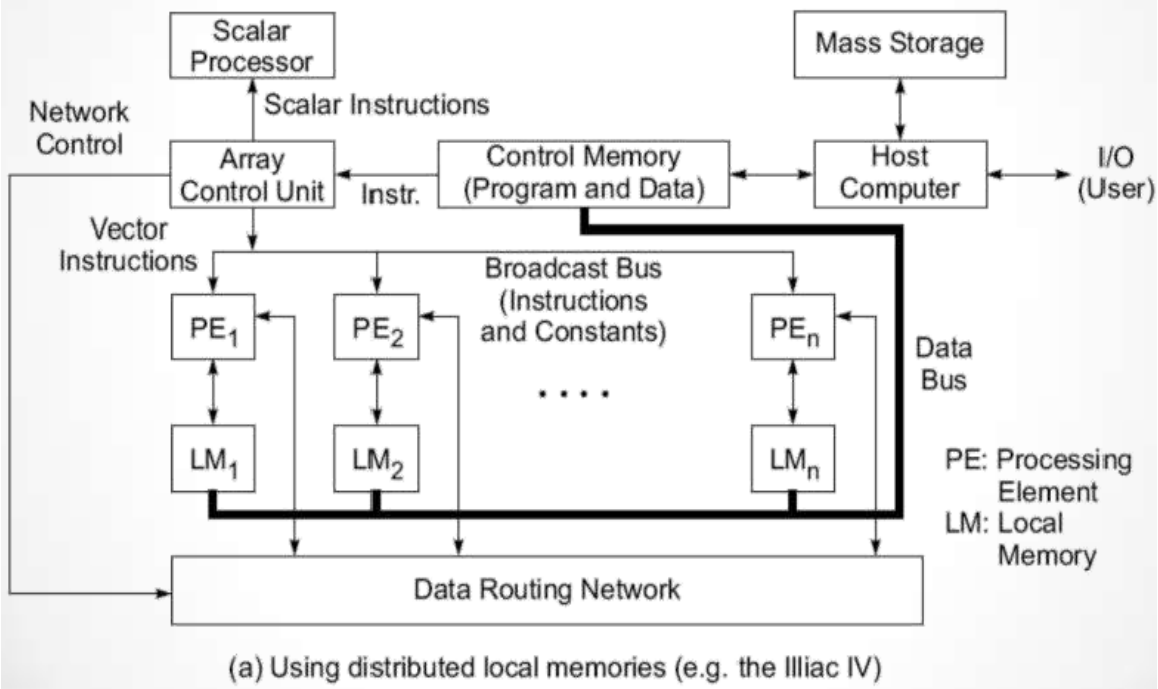
8.4 SIMD Computer Organizations

SIMD Implementation Models OR (Two models for constructing SIMD Super Computers)

SIMD models differentiates on base of memory distribution and addressing scheme used.

Most SIMD computers use a single control unit and distributed memories, except for a few that use associative memories.

1. Distributed memory model



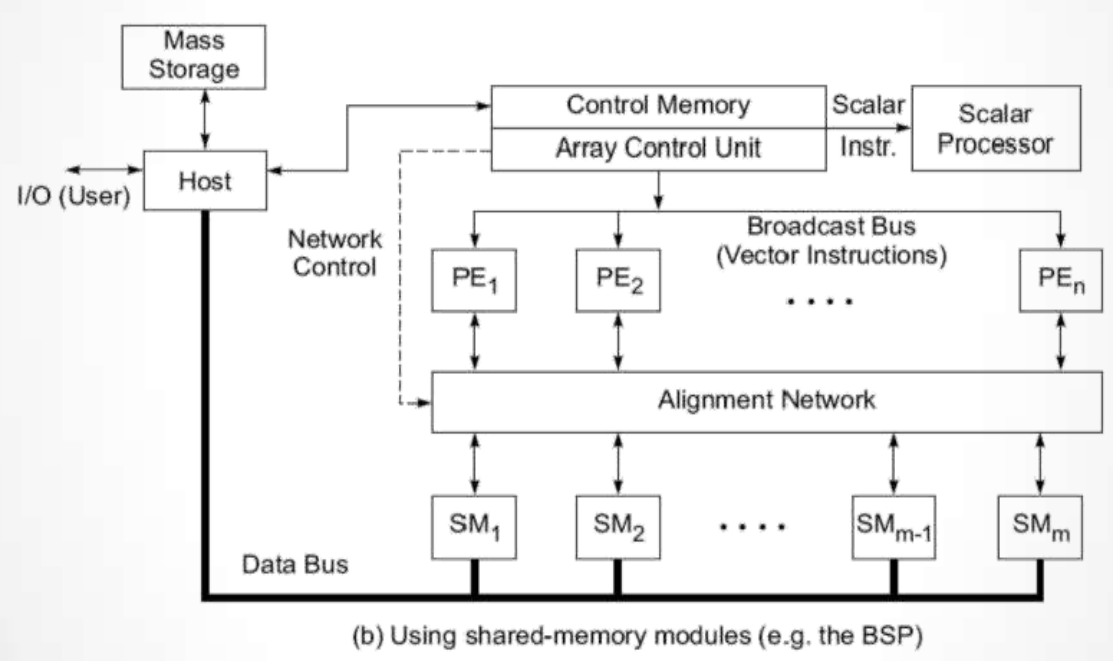
(a) Using distributed local memories (e.g. the Illiac IV)

- Spatial parallelism is exploited among the PEs.
- A distributed memory SIMD consists of an array of PEs (supplied with local memory) which are controlled by the array control unit.
- Program and data are loaded into the control memory through the host computer and distributed from there to PEs local memories.
- An instruction is sent to the control unit for decoding. If it is a scalar or program control operation, it will be directly executed by a scalar processor attached to the control unit.
- If the decoded instruction is a vector operation, it will be broadcast to all the PEs for parallel execution.
- Partitioned data sets are distributed to all the local memories attached to the PEs trough a vector data bus.

- PEs are interconnected by a data routing network which performs inter-PE data communications such as shifting, permutation and other routing operations.

2. Shared Memory Model

- An alignment network is used as the inter-PE memory communication network. This network is controlled by control unit.
- The alignment network must be properly set to avoid access conflicts.
- Figure below shows a variation of the SIMD computer using shared memory among the PEs.
- Most SIMD computers were built with distributed memories.



(b) Using shared-memory modules (e.g. the BSP)

8.4.2 CM-2 Architecture

The Connection Machine CM-2 produced by Thinking Machines Corporation was a fine-grain MPP computer using thousands of bit-slice PEs in parallel to achieve a peak processing speed of above 10 Gflops.

Program Execution Paradigm

All programs started execution on a front-end, which issued microinstructions to the back-end processing array when data-parallel operations were desired. The sequencer broke down these microinstructions and broadcast them to all data processors in the array.

Data sets and results could be exchanged between the front-end and the processing array in one of three ways as shown in the figure:

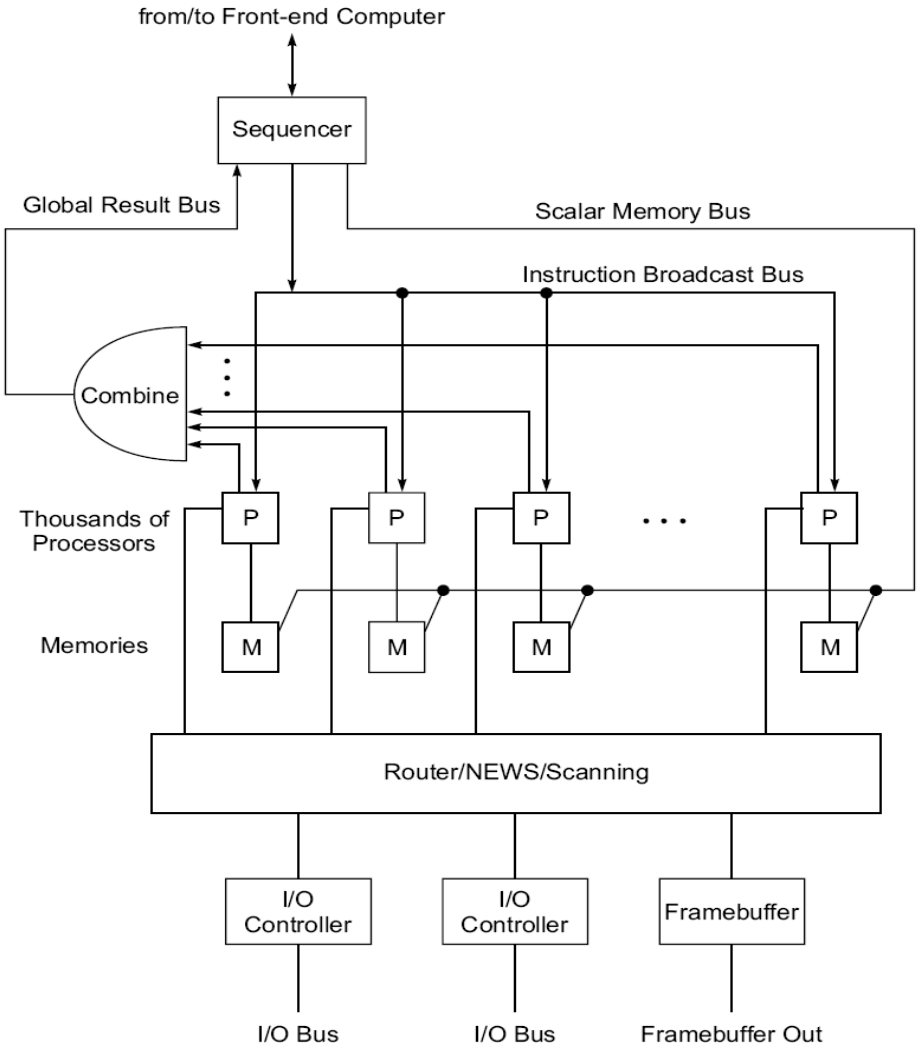


Fig. 8.23 The architecture of the Connection Machine CM-2 (Courtesy of Thinking Machines Corporation, 1990)

- **Broadcasting:** Broadcasting was carried out through the broadcast bus to all data processors at once.
- **Global combining:** Global combining allowed the front-end to obtain the sum, largest value, logical OR etc, of values one from each processor.
- **Scalar memory bus:** Scalar bus allowed the front-end to read or to write one 32-bit value at a time from or to the memories attached to the data processors.

Processing Array

The processing array contained from 4K to 64K bit-slice data processors(PEs), all of which were controlled by a sequencer.

Processing Nodes

Each data processing node contained 32 bit-slice data processors, an optional floating point accelerator and interfaces for inter processor communication.

Hypercube Routers

The router nodes on all processor chips were wired together to frm a Boolean n-cube. A full configuration of CM-2 had 4096 router nodes on processor chips interconnected as a 12-dimensional hypercube.

Major Applications of CM-2

The CM-2 has been applied in almost all the MPP and grand challenge applications.

- Used in document retrieval using relevance feedback,
- in memory based reasoning as in the medical diagnostic system called QUACK for simulating the
  diagnosis of a disease,
- in bulk processing of natural languages.
- the SPICE-like VLSI circuit analysis and layout,
- computational fluid dynamics,
- signal/image/vision processing and integration,
- neural network simulation and connectionist modeling,
- dynamic programming,
- context free parsing,
- ray tracing graphics,
- computational geometry problems.

8.4.3 MasPar MP-1 Architecture

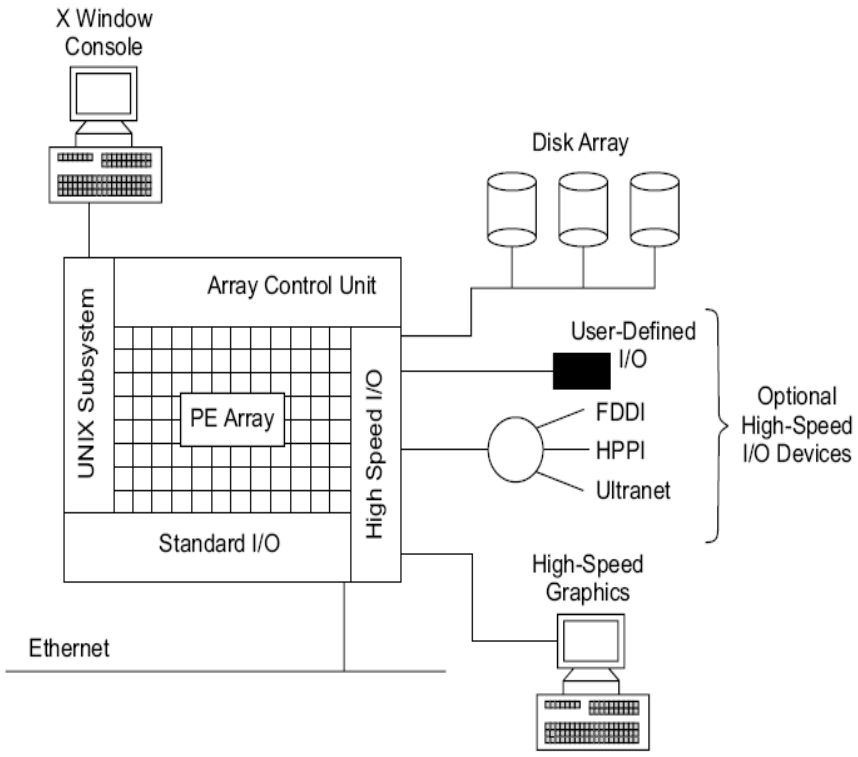
The MP-1 architecture consists of four subsystems:

i) PE array,

ii) Array Control Unit (ACU),

iii) UNIX subsystem with standard I/O,

iv) High-speed I/O subsystem



(a) MP-1 System Block Diagram

- **The UNIX subsystem** handles traditional serial processing.
- **The high-speed I/O**, working together with the **PE array**, handles massively parallel computing.
- The MP-1 family includes configurations with 1024, 4096. and up to 16,384 processors. The peak performance of the 16K-processor configuration is 26,000 MIPS in 32-bit RISC integer operations. The system also has a peak floating-point capability of 1.5 Gfiops in single-precision and 650 Mflops in double-precision operations.
- **Array Control Unit** The ACU is a 14-MIPS scalar RISC processor using a demand paging instruction memory. The ACU fetches and decodes MP-1 instructions, computes addresses and scalar data values, issues control signals to the PE array, and monitors the status of the PE array.

The ACU is microcoded to achieve horizontal control of the PE array. Most scalar ACU instructions execute in one 70-ns clock. The whole ACU is implemented on one PC board.

An implemented functional unit, called a *memory machine,* is used in parallel with the ACU. The memory machine performs PE array load and store operations, while the ACU broadcasts arithmetic, logic, and routing instructions to the PEs for parallel execution.

Chapter 9- Scalable, Multithreaded, and Dataflow Architectures

9.1 Latency Hiding Techniques.

9.1.1 Shared Virtual Memory

Single-address-space multiprocessors/multicomputers must use shared virtual memory.

The Architecture Environment

- The Dash architecture was a large-scale, cache-coherent, NUMA multiprocessor system as depicted in Fig. 9.1.

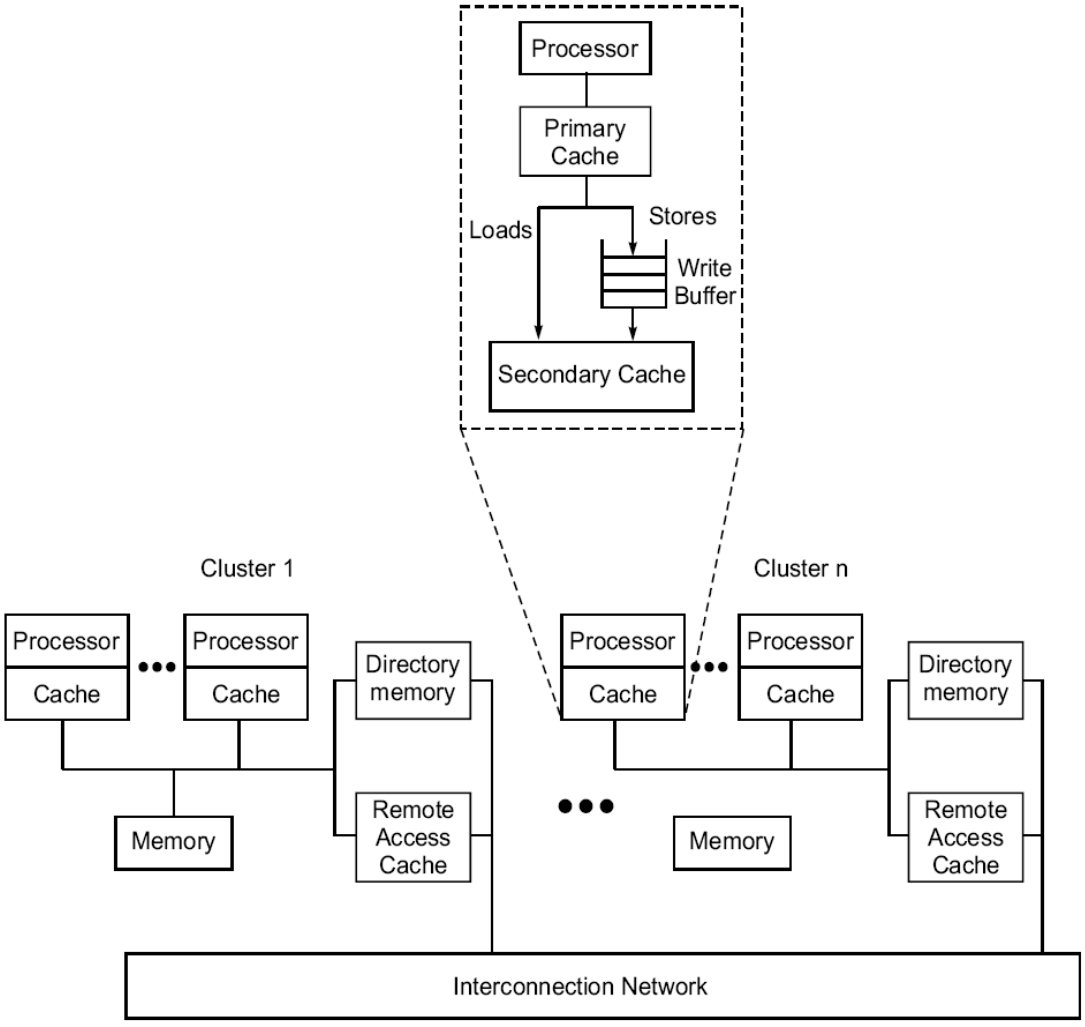


Fig. 9.1 A scalable coherent cache multiprocessor with distributed shared memory modeled after the Stanford Dash (Courtesy of Anoop Gupta et al, *Proc. 1991 Ann. Int. Symp. Computer Arch.*)

- It consisted of multiple multiprocessor clusters connected through a scalable, low latency interconnection network.

- Physical memory was distributed among the processing nodes in various clusters. The distributed memory formed a global address space.
- Cache coherence was maintained using an invalidating, distributed directory-based protocol. For each memory block, the directory kept track of remote nodes caching it.
- When a write occurred, point-to-point messages were sent to invalidate remote copies of the block.
- Acknowledgement messages were used to inform the originating node when an invalidation was completed.

- Two levels of local cache were used per processing node. Loads and writes were separated with the use of write buffers for implementing weaker memory consistency models.
- The main memory was shared by all processing nodes in the same cluster. To facilitate prefetching and the directory-based coherence protocol, directory memory and remote-access caches were used for each cluster.
- The remote-access cache was shared by all processors in the same cluster.

The SVM Concept

- Figure 9.2 shows the structure of a distributed shared memory. A global virtual address space is shared among processors residing at a large number of loosely coupled processing nodes.
- The idea of Shared virtual memory (SVM) is to implement coherent shared memory on a network of processors without physically shared memory.
- The coherent mapping of SVM on a message-passing multicomputer architecture is shown in Fig. 9.2b.

- The system uses virtual addresses instead of physical addresses for memory references.

- Each virtual address space can be as large as a single node can provide and is shared by all nodes in the system.
- The SVM address space is organized in pages which can be accessed by any node in the system. A memory-mapping manager on each node views its local memory as a large cache of pages for its associated processor.

Page Swapping

- A memory reference causes a page fault when the page containing the memory location is not in a processor's local memory.

- When a page fault occurs, the memory manager retrieves the missing page from the memory of another processor.
- If there is a page frame available on the receiving node, the page is moved in.
- Otherwise, the SVM system uses page replacement policies to find an available page frame, swapping its contents to the sending node.
- A hardware MMU can set the access rights (nil, read-only} writable) so that a memory access violating memory coherence will cause a page fault.
- The memory coherence problem is solved in IVY through distributed fault handlers and their servers. To client programs, this mechanism is completely transparent.
- The large virtual address space allows programs to be larger in code and data space than the physical memory on a single node.
- This SVM approach offers the ease of shared-variable programming in a message-passing environment.
- In addition, it improves software portability and enhances system scalability through modular memory growth.

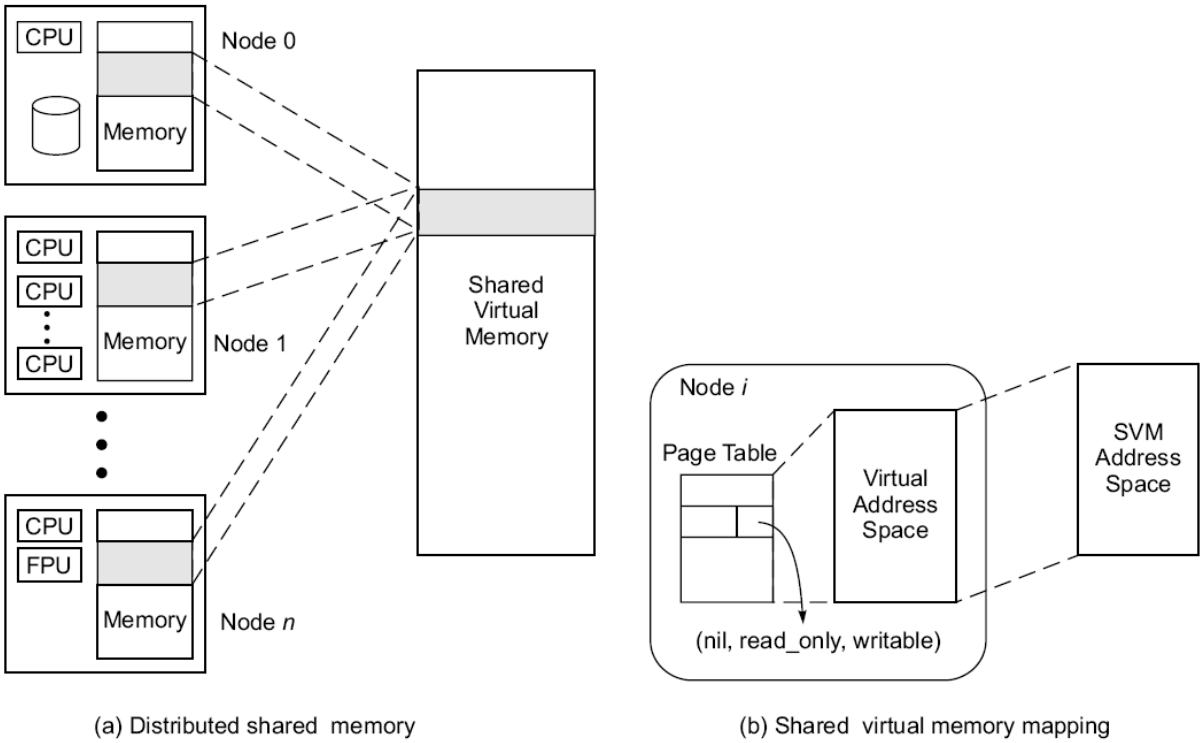


Fig. 9.2 The concept of distributed shared memory with a global virtual address space shared among all processors on loosely coupled processing nodes in a massively parallel architecture (Courtesy of Kai Li, 1992)

Latency hiding can be accomplished through 4 complementary approaches:

i) **Prefetching techniques** which bring instructions or data close to the processor before they are actually needed

ii) **Coherent caches** supported by hardware to reduce cache misses

iii) **Relaxed memory consistency models** by allowing buffering and pipelining of memory references

iv) **Multiple-contexts support** to allow a processor to switch from one context to another when a long latency operation is encountered.

1. Prefetching Techniques

Prefetching uses knowlwdge about the expected misses in a program to move the corresponding data close to the processor before it is actually needed.

Prefetching can be classified based on whether it is

- Binding
- Non binding

or whether it is controlled by

- hardware
- software

- **Binding prefetching** : the value of a later reference (eg, a register load) is bound at the time when the prefetch completes.
- **Non binding prefetching** : brings data close to the processor, but the data remains visible to the cache coherence protocol and is thus kept consistent until the processor actually reads the value.
- **Hardware Controlled Prefetching**: includes schemes such as long cache lines and instruction lookahead.
- **Software Controlled Prefetching**: explicit prefetch instructions are issued. Allows the prefetching to be done selectively and extends the possible interval between prefetch issue and actual reference.

2. Coherent Caches

- While the cache coherence problem is easily solved for small bus-based multiprocessors through the use of snoopy cache coherence protocols, the problem is much more complicated for large scale multiprocessors that use general interconnection networks.

- As a result, some large scale multiprocessors did not provide caches, others provided caches that must be kept coherent by software, and still others provided full hardware support for coherent caches.
- Caching of shared read-write data provided substantial gains in performance. The largest benefit came from a reduction of cycles wasted due to read misses. The cycles wasted due to write misses were also reduced.
- Hardware cache coherence is an effective technique for substantially increasing the performance with no assistance from the compiler or programmer.

3. Relaxed memory consistency models

Some different consistency models can be defined by relaxing one or more requirements in sequential consistency called relaxed consistency models. These consistency models do not provide memory consistency at the hardware level. In fact, the programmers are responsible for implementing the memory consistency by applying synchronization techniques.

There are 4 comparisons to define the relaxed consistency:

- Relaxation
- Synchronizing vs non-synchronizing
- Issue vs View-Based
- Relative Model Strength

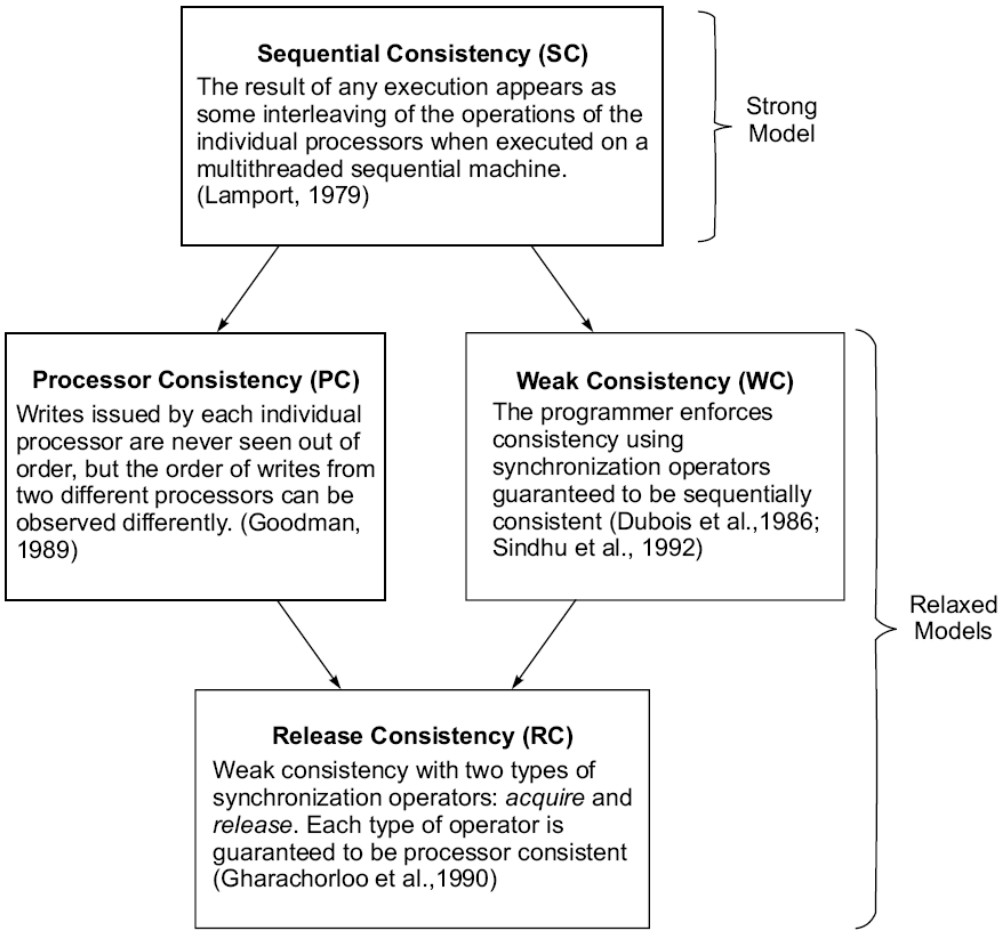


Fig. 9.8 Intuitive definitions of four memory consistency models. The arrows point from strong to relaxed consistencies (Courtesy of Nitzberg and Lo, *IEEE Computer*, August 1991)

9.2 Principles of Multithreading

9.2.1 Multithreading Issues and Solutions

Multithreading demands that the processor be designed to handle multiple contexts simultaneously on a context-switching basis.

Architecture Environment

Multithreading MPP system is modeled by a network of Processor (P) and memory (M) nodes as shown in Fig. 9.11a. The distributed memories form a global address space.

Four machine parameters are defined below to analyze the performance of this network:

1. **The Latency (L):** This is the communication latency on a remote memory access. The value of L includes the network delays, cache-miss penalty and delays caused by contentions in split transactions.

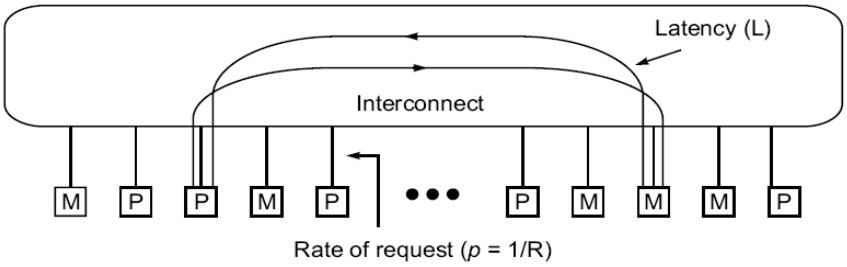
2. **The number of Threads (N):** This is the number of threads that can be interleaved in each processor. A thread is represented by a context consisting of a program counter, a register set and the required context status words.
3. **The context-switching overhead (C):** This refers to the cycles lost in performing context switching in a processor. This time depends on the switch mechanism and the amount of processor states devoted to maintaining active threads.
4. **The interval between switches (R):** This refers to the cycles between switches triggered by remote reference. The inverse p=1/R is called the rate of requests for remote accesses. This reflects a combination of program behavior and memory system design.

In order to increase efficiency, one approach is to reduce the rate of requests by using distributed coherent caches. Another is to eliminate processor waiting through multithreading.

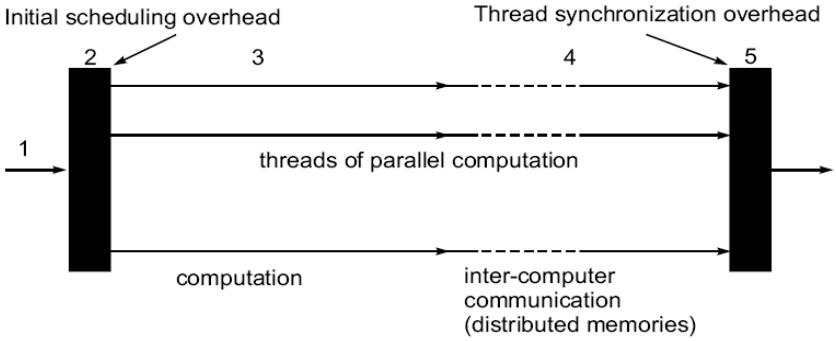
Multithreaded Computations

Fig 9.11b shows the structure of the multithreaded parallel computations model.

The computation starts with a sequential thread (1), followed by supervisory scheduling (2), where the processors begin threads of computation (3), by intercomputer messages that update variables among the nodes when the computer has distributed memory (4), and finally by synchronization prior to beginning the next unit of parallel work (5).



(a) The architecture environment. (Courtesy of Rafael Saavedra, 1992)



(b) Multithreaded computation model. (Courtesy of Gordon Bell, *Commun. ACM*, August 1992)

Fig. 9.11 Multithreaded architecture and its computation model for a massively parallel processing system

The communication overhead period (4) inherent in distributed memory structures is usually distributed throughout the computation and is possibly completely overlapped.

Message passing overhead in multicomputers can be reduced by specialized hardware operating in parallel with computation.

Communication bandwidth limits granularity, since a certain amount of data has to be transferred with other nodes in order to complete a computational grain. Message passing calls (4) and synchronization (5) are nonproductive.

Fast mechanisms to reduce or to hide these delays are therefore needed. Multithreading is not capable of speedup in the execution of single threads, while weak ordering or relaxed consistency models are capable of doing this.

Problems of Asynchrony

Massively parallel processors operate asynchronously in a network environment. The asynchrony triggers two fundamental latency problems:

1. Remote loads
2. Synchronizing loads

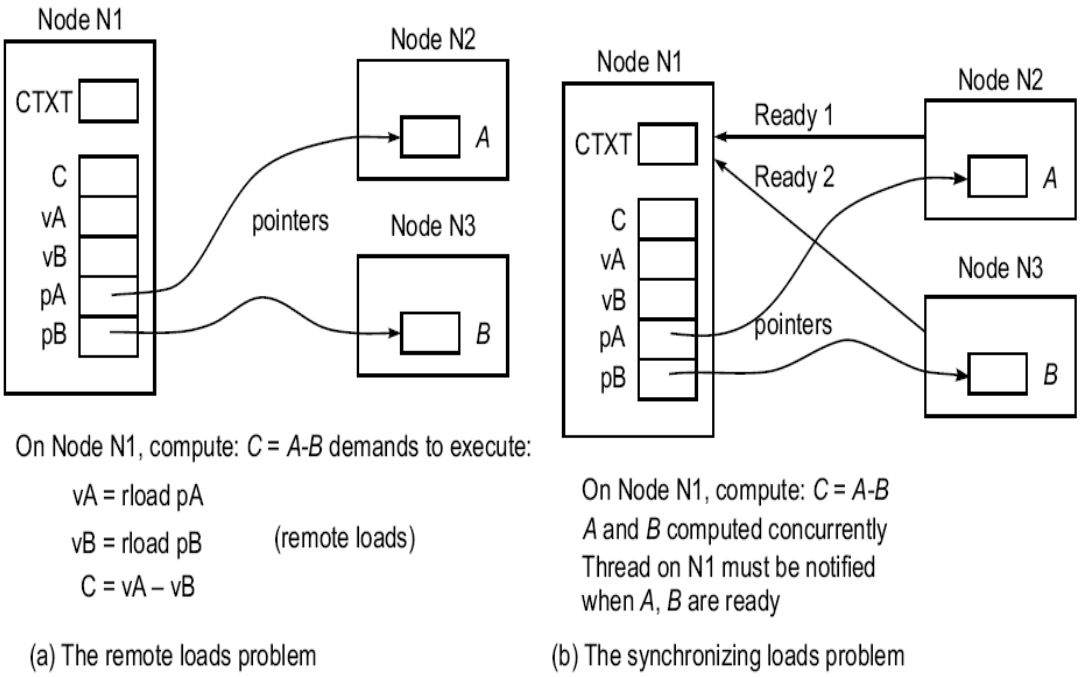
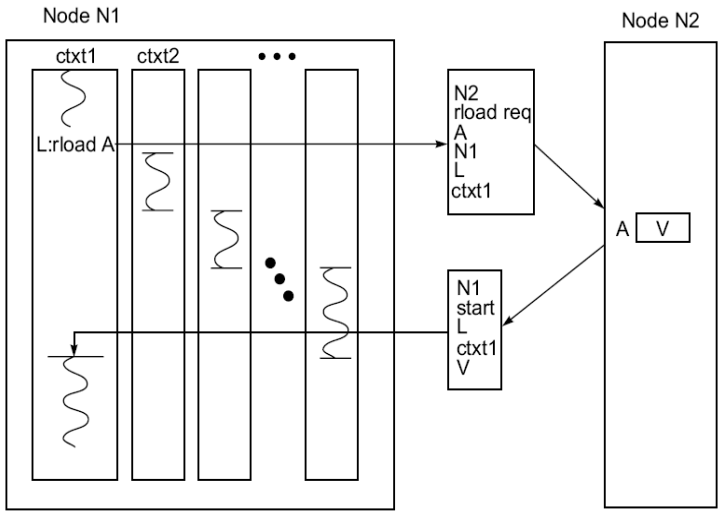


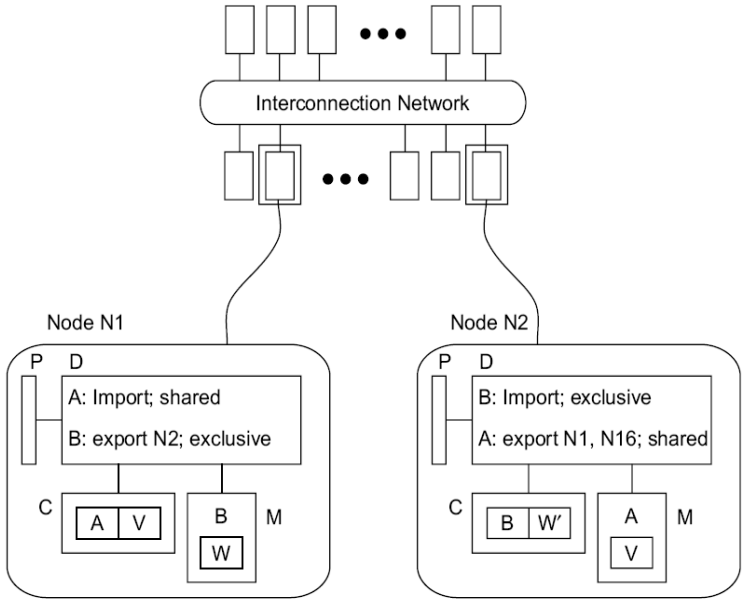
Fig. 9.12 Two common problems caused by asynchrony and communication latency in massively parallel processors (Courtesy of R.S. Nikhil, Digital Equipment Corporation, 1992)

Solutions to Asynchrony Problem

1. Multithreading Solutions
2. Distributed Caching



(a) Multithreading solution



P = Processor; D = Directory; C = Cache; M = Memory

(b) Distributed cacheing

Fig. 9.13 Two solutions for overcoming the asynchrony problems (Courtesy of R. S. Nikhil, Digital Equipment Corporation, 1992)

1. **Multithreading Solutions** – Multiplex among many threads

When one thread issues a remote-load request, the processor begins work on another thread, and so on (Fig. 9.13a).

- Clearly the cost of thread switching should be much smaller than that of the latency of the remote load, or else the processor might as well wait for the remote load's response.
- As the internode latency increases, more threads are needed to hide it effectively. Another concern is to make sure that messages carry continuations. Suppose, after issuing a remote load from thread T1 (Fig 9.13a), we switch to thread T2, which also issues a remote load.
- The responses may not return in the same order. This may be caused by requests traveling different distances, through varying degrees of congestion, to destination nodes whose loads differ greatly, etc.
- One way to cope with the problem is to associate each remote load and response with an identifier for the appropriate thread, so that it can be reenabled on the arrival of a response.

2. Distributed Caching

- The concept of Distributed Caching is shown in Fig. 9.13b. every memory location has an owner node. For example, N1 owns B and N2 owns A.
- The directories are used to contain import-export lists and state whether the data is shared (for reads, many caches may hold copies) or exclusive (for writes, one cache holds the current value).
- The directories multiplex among a small number of contexts to cover the cache loading effects.

- The Distributed Caching offers a solution for the remote-loads problem, but not for the synchronizing-loads problem.
- Multithreading offers a solution for remote loads and possibly for synchronizing loads.
- The two approaches can be combined to solve both types of remote access problems.

9.2.2 Multiple-Context Processors

Multithreaded systems are constructed with multiple-context (multithreaded) processors.

Enhanced Processor Model

- A conventional single-thread processor will wait during a remote reference, it is idle for a period of time L.
- A multithreaded processor, as modeled in Fig. 9.14a, will suspend the current context and switch to another, so after some fixed number of cycles it will again be busy doing useful work, even though the remote reference is outstanding.
- Only if all the contexts are suspended (blocked) will the processor be idle.

The objective is to maximize the fraction of time that the processor is busy, we will use the efficiency of the processor as our performance index, given by:

$$\text{Efficiency} = \frac{\text{busy}}{\text{busy} + \text{switching} + \text{idle}}$$

where busy, switching and idle represent the amount of time, measured over some large interval, that the processor is in the corresponding state.

The basic idea behind a multithreaded machine is to interleave the execution of several contexts on order to dramatically reduce the value of idle, but without overly increasing the magnitude of switching.

Context-Switching Policies

Different multithreaded architectures are distinguished by the context-switching policies adopted.

Four switching policies are:

1. **Switch on Cache miss –** This policy corresponds to the case where a context is preempted when it causes a cache miss.

   In this case, R is taken to be the average interval between misses (in Cycles) and L the time required to satisfy the miss.

   Here, the processor switches contexts only when it is certain that the current one will be delayed for a significant number of cycles.
2. **Switch on every load -** This policy allows switching on every load, independent of whether it will cause a miss or not.

   In this case, R represents the average interval between loads. A general multithreading model assumes that a context is blocked for L cycles after every switch; but in the case of a switch-on-load processor, this happens only if the load causes a cache miss.
3. **Switch on every instruction** – This policy allows switching on every instruction, independent of whether it is a load or not. Successive instructions become independent , which will benefit pipelined execution.
4. **Switch on block of instruction** – Blocks of instructions from different threads are interleaved. This will improve the cache-hit ratio due to locality. It will also benefit single-context performance.