MODULE 4 NOTES

CHAPTER 7

CHAPTER 8-REFER ASSIGNMENT QUESTIONS

CHAPTER 9- FROM 9.4.2- 9.5.3 REFER TEXT BOOK

FOR TOPIC HEADINGS AND SUBHEADINGS REFER TEXT BOOK.

MULTIPROCESSORS AND MULTICOMPUTERS

MULTIPROCESSOR SYSTEM INTERCONNECTS

A generalized multiprocessor system is depicted in below Fig. 7.1. This architecture combines features from the UMA, NUMA, and COMA models. Each processor P_i is attached to its own local memory and private cache. Multiple processors are connected to shared-memory modules through an interprocessor-memory network (IPMN). The processors share the access of I/O and peripheral devices through a processor I/O network (PION). Both IPMN and PION are necessary in a shared-resource multiprocessor. Direct interprocessor communications are supported by an optional interprocessor communication network (IPCN) instead of through the shared memory.

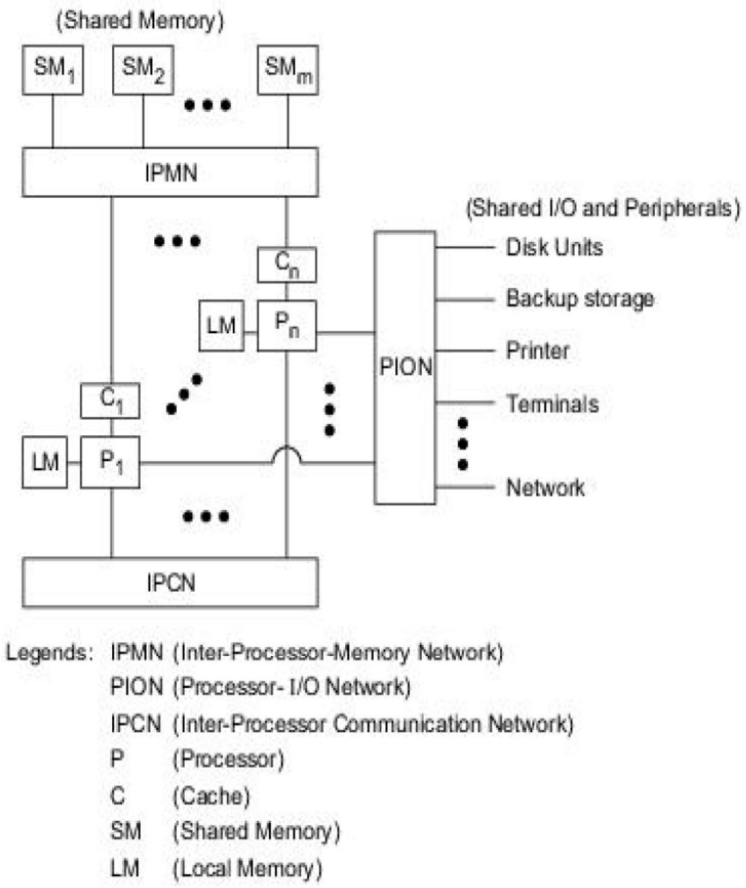


Fig. 7.1 Interconnection structures in a generalized multiprocessor system with local memory, private caches, shared memory, and shared peripherals

Network Characteristic

The interconnection networks choices are based on the topology, timing protocol, switclting method, and control strategy.

Timing, switching, and control are three major operational characteristics of an interconnection network used in multiproccssors.

The timing control can be either synchronous or asynchronous. synchronous networks are controlled by a global clock that synchronizes all network activities. Asynchronous networks use handshaking or interlocking mechanisms to coordinate fast and slow devices requesting use of the same network.

A network can transfer data using either circuit switching or packet switching. In circuit switching, once a device is granted a path in the network, it occupies the path for the entire duration of the data transfer. In packet switching, the information is broken into small packets individually competing for a path in the network.

Network control strategy is classified as centralized or distributed. With centralizcd control, a global controller receives requests from all devices attached to the network and grants the network access to one or more requesters. In a distributed system, requests are handled by local devices independently.

Hierarchical Bus Systems

A bus system consists of a hierarchy of buscs connecting various system and subsystem components in a computer. Each bus is formed with a number of signal, control, and power lines. The hierarchy of bus systems are packaged at different levels as depicted in below Fig. 7.2, including local buses on boards, backplane buses, and I/Obuses.

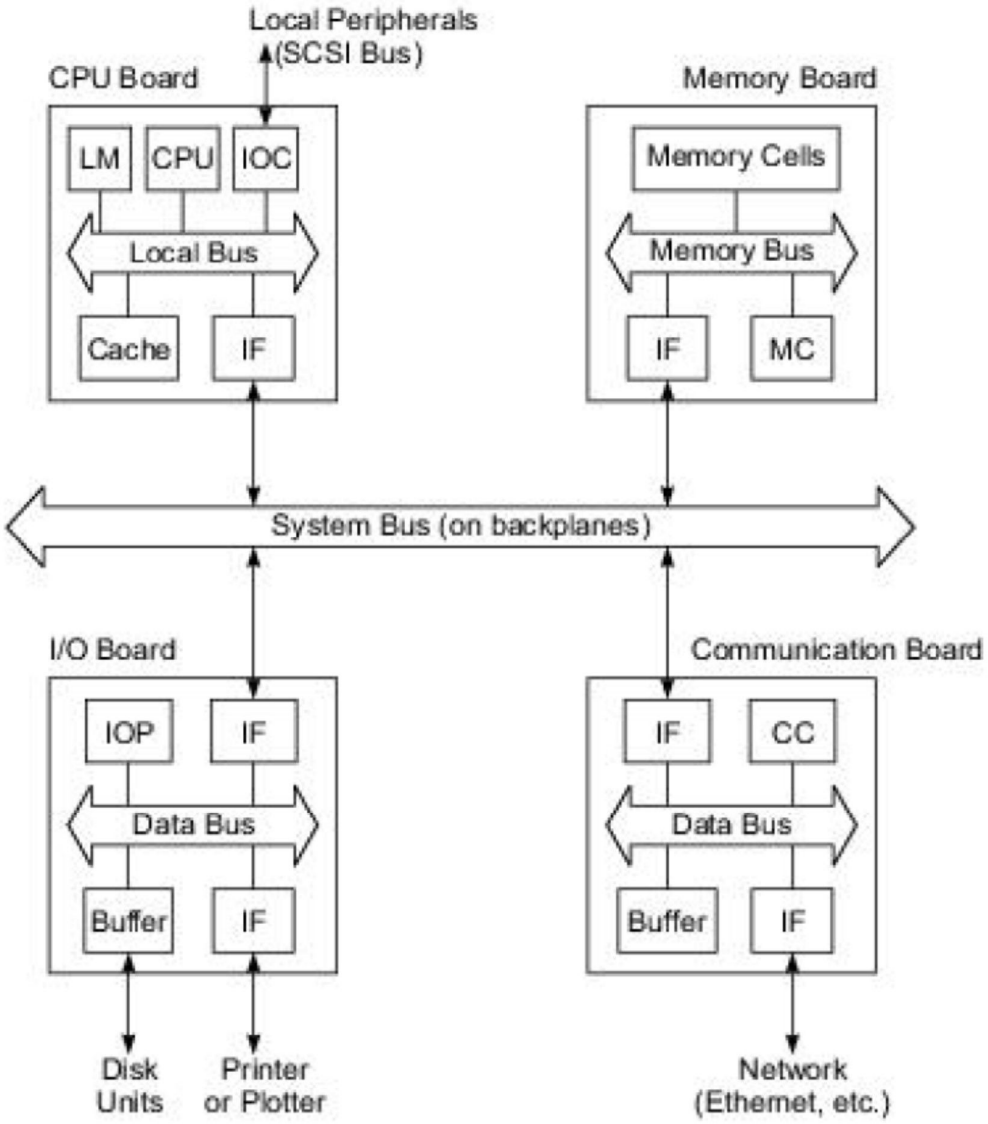


Fig. 7.2 Bus systems at board level, backplane level, and I/O level

Buses implemented within processor chips or on printed circuit boards are called local buses, which provide a common communication path among major cornponents (chips) mounted on the board.

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.

A backkplane bus 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 commmunication path alnong all plug-in boards.Several backplaue bus standards have been developed over time such as the Multibus II and Futurebus+.

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 as shown in Fig. 7.2. Special interface logic is used to connect various board types to the backplane bus.

Hierarchical Buses and Cache

A hierarchical cache bus architecture shown in below Fig. 7.3 was proposed by Wilson in1987. This is a multilevel tree structure in which the leaf nodes are processors and their private caches denoted as 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 interclustcr bus is used to provide communications among the clusters. Second level caches denoted as C2 are used between each cluster bus and the intercluster bus. Snoopy bus coherence protocols can be used to establish comisteney among first-level caches belonging to the same cluster. 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 lntercluster cache coherence is controlled among the second~level caches and the resulting effects are passed to the lower level.

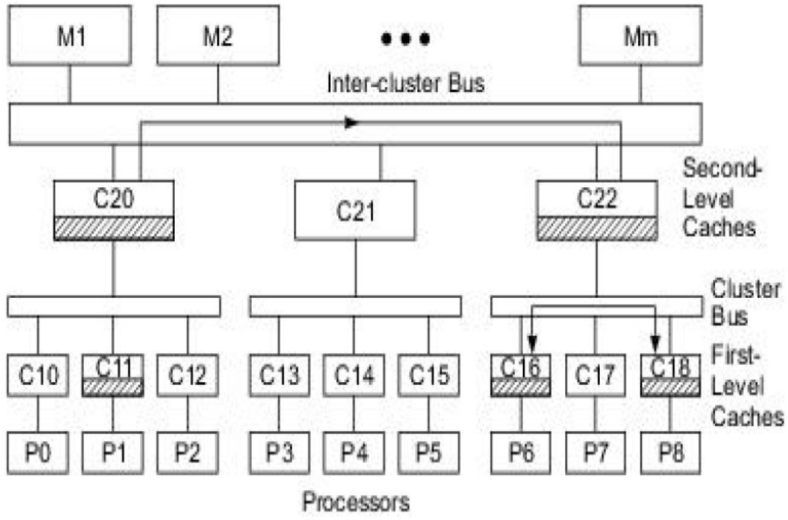


Fig. 7.3 A hierarchical cache/bus architecture for designing a scalable multiprocessor (Courtesy of Wilson; reprinted from *Proc. of Annual Int. Symp. on Computer Architecture, 1987*)

Example: Encore Ultramax multiprocessor architecture

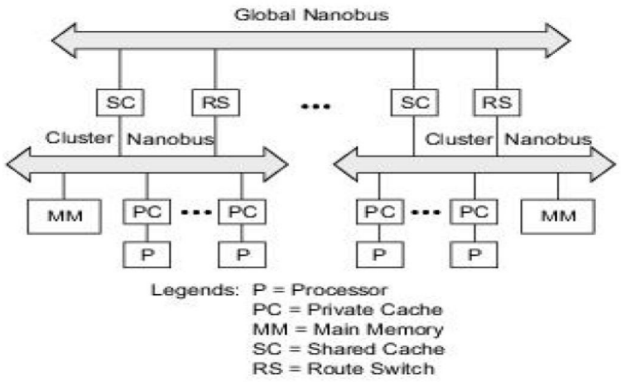


Fig. 7.4 The Ultramax multiprocessor architecture using hierarchical buses with multiple clusters (Courtesy of Encore Computer Corporation, 1987)

The Ultramax had a two-level hierarchical-bus architecture as depicted in above Fig. 7.4. The shared memories were distributed to all clusters instead of being connected to the intercluster bus. The cluster caches formed the second level caches and performed the same filtering and cache coherence control for remote accesses as in Wilson's scheme. When an aceess request reached the top bus, it would be routed down to the cluster memory that matched it with the reference address.

The idea of using bridges between multiprocessor clusters is to allow transactions initiated on a local bus to be completed on a remote bus. The main functions of a bridge include communication protocol conversion, interrupt handling in split transactions, and serving as cache and memory agents. An exampleis shown in below Fig. 7.5, where multiple buses are used to build a very large system consisting of three rnultiproeessor clusters. The bus used in this example is Futurebus+.

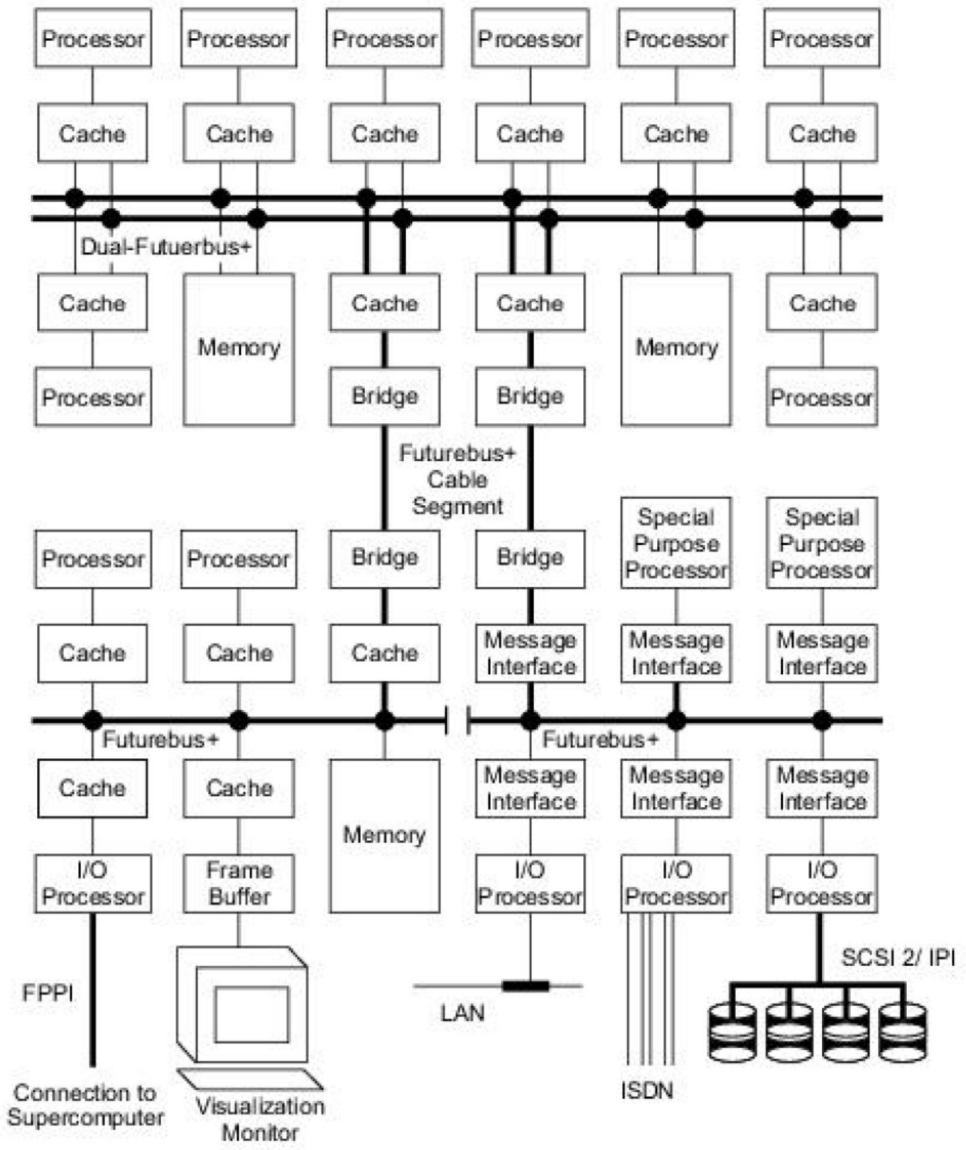


Fig. 7.5 A multiprocessor system using multiple Futurebus+ segments (Reprinted with permission fro
Standard 896.1-1991, copyright © 1991 by IEEE, Inc.)

Crossbar Switch and Multiport Memory

Switched networks provide dynamic interconnections between the inputs and outputs. Major classes of switched networks based on the number of stages and blocking or nonblocking are single stage network, multistage network and crossbar netwoks.

Network Stage

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. 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. The Omega network, Flip network, and Baseline networks are all multistage networks.

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. Examples of blocking networks include the Omega, Baseline and Banyan network 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 perfonn all possible connections between inputs and outputs by rearranging its connections. In such a network, a connection path can always be established between any input-output pair. The Benes networks have such a capability.

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.

Once the data is read from the memory. its value is retumed to the requesting processor along the same crosspoint switch. In general, such a crossbar network requires the use of nxm crosspoint switches.

A crossbar switch network is a single-stage, nonblocking, permutation network. 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 mernory 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.

Crosspoint Switch Design (crosspoint switching operations)

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 he designed with extra hardware.

For an n x m crossbar network, n^2 sets of crosspoint switches and a large number of lines are needed which amounts to requiring extensive hardware when n is very large. So far only relatively small crossbar networks with n<=16 to have been built into commercial machines.

Figure 7.6 below shows the schematic design of a row of crosspoint switches in a single crossbar network. Multiplexer modules are used to select one of the 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.

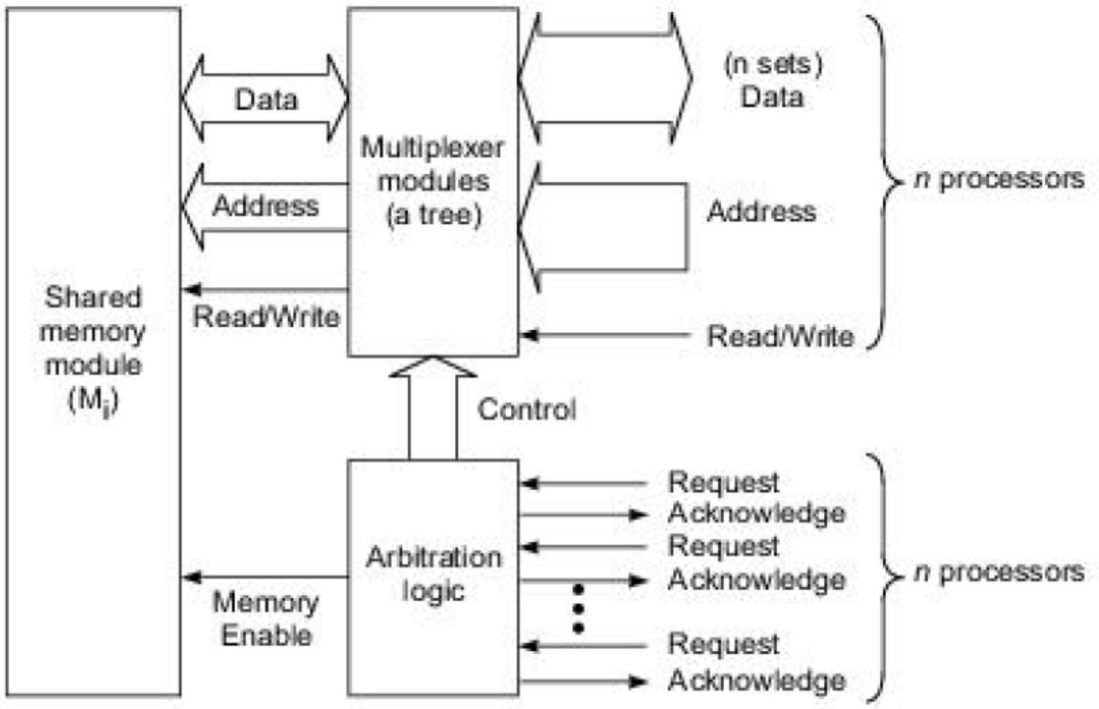


Fig. 7.6 Schematic design of a row of crosspoint switches in a crossbar network

For example, a 4-bit control signal will be generated for n = 16 processors. Based on the control signal received, only one out of n sets of information lines is selected as the output of the multiplexer tree. The memory address is entered for both read and write access. In the case of read, the data fetched from memory are retumed to the selected processor in the reverse direction using the data path established. In case of write, thc data on the data path are stored in memory. Acknowledge signals are used to indicate the arbitration result to all requesting processors. These signals initiate data transfer and are used to avoid conflicts.

Crossbar network Advantages and Limitations

The crossbar network offers the highest bandwidth of n data transfers per cycle, as compared with only one data transfer per bus cycle.

Since all necessary switching and conflict resolution logic are built into the crosspoint switch, the processor interface and memory port logic are much simplified and cheaper.

A crossbar network is cost-effective only for small multiproccssors with a few processors accessing a few memory modules.

A single-stage crossbar network is not expandable once it is built.

For an n x n crossbar network, at most n memory words can be delivered to at most n processors in each cycle.

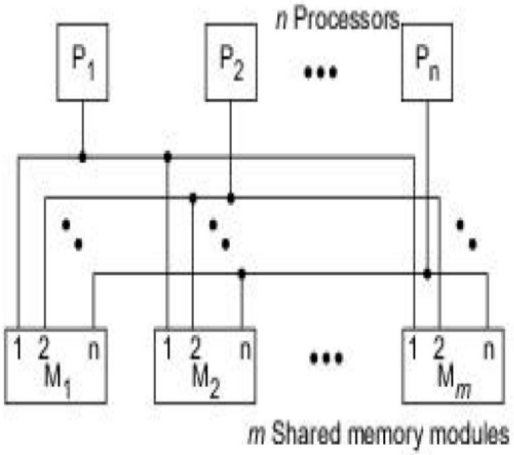
Multiport Memory

As building a crossbar network into a large system is cost prohibitive, some mainframe multiprocessors used a multiport memory organization. The idea is to move all crosspoint arbitration and switching functions associated with each memory module into the memory controller. Thus the memory module becomes more expensive due to the added access ports and associated logic as demonstrated in below Fig 7.7.a. The circles in the diagram represent n switches tied to n input ports of a memory module. Only one of n processor requests can be honored at a time.

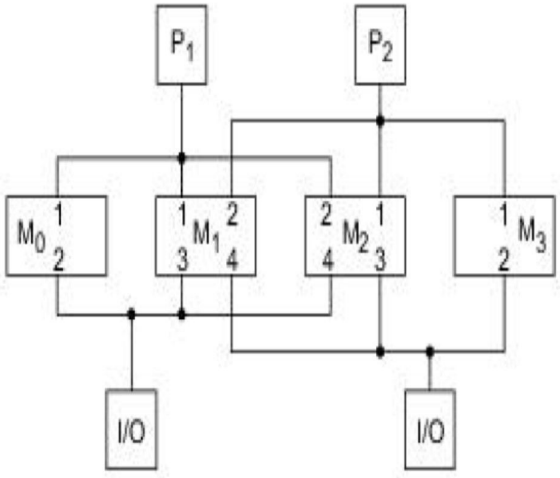
The multiport memory organization is a compromise solution between a low-cost, low-performance bus system and a high-cost, high-bandwidth crossbar system. The contention bus is time-shared by all processors and device modules attached. The multiport memory must resolve conflicts among processors.

A multiport memory multiprocessor is not scalable because once the ports are fitted, no more processors can be added without redesigning the memory controller.

Another drawback is the need for a large number of interconnection cables and connectors when the configuration becomes large. The ports of each memory module in Fig. 7.7b are prioritized. Some of the processors are CPUs, some are I/O processors, and some are connected to dedicated processors.



(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)

Multistage and Combining Networks

Multistage networks arc used to build larger multiprocessor systems. the two multistage networks, Omega network and the Butterfly network, that have been built into commercial machines. A special class of multistage networks, called combining networks, are used for resolving access conflicts automatically through the network.

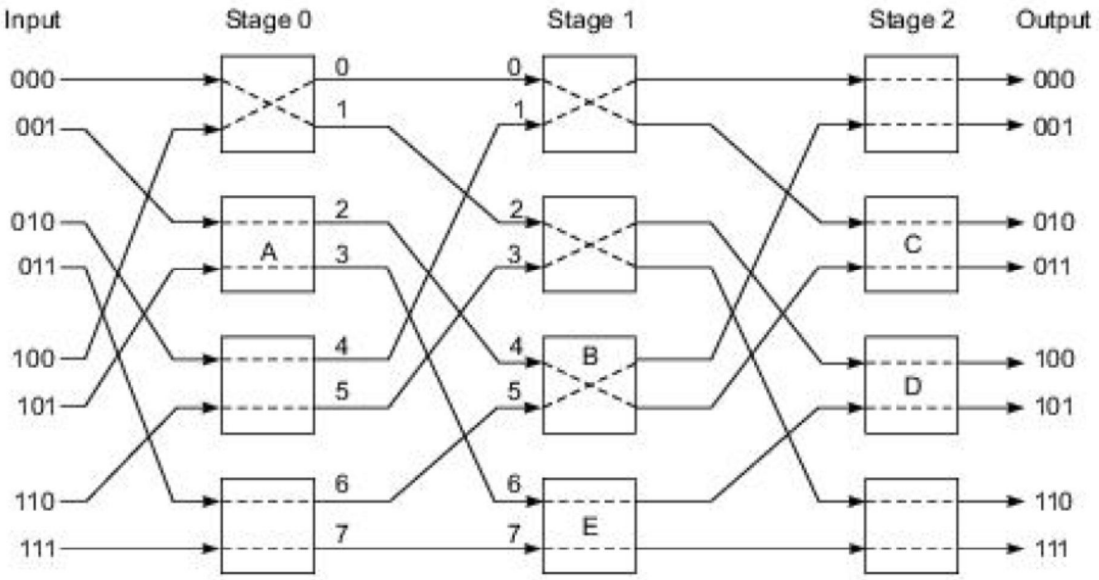
Routing in Omega Network

An 8-input Omega network is shown in below Fig 7.8. In general, an n-input Omega network has $\log_2 n$ stages. The stages are labeled from 0 to $\log_2$ n-l from the input end to the output end. Data routing is controlled by inspecting the destination code in binary. When thc i^{th} 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.

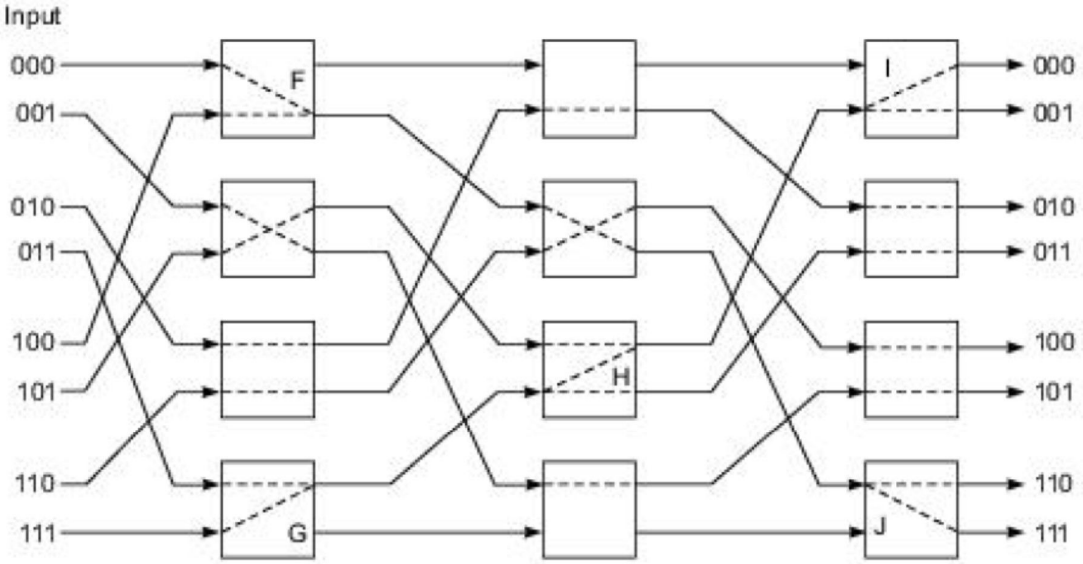
Two switch settings are shown in Figs. 7.8a and b with respect to permutations Π_1 = (0, 7, 6, 4,2) (1, 3) (5) and Π_2 = (0, 6, 4, 7, 3) (1, 5) (2), respectively. The switch settings in Fig. 7.8a are for thc implementation of Π_1, which maps

$$0 \rightarrow 7,\ 7 \rightarrow 6,\ 6 \rightarrow 4,\ 4 \rightarrow 2,\ 2 \rightarrow 0,\ 1 \rightarrow \bar{3},\ 3 \rightarrow \bar{1},\ 5 \rightarrow 5.$$

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 011 is connected to the uppcr 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.



(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

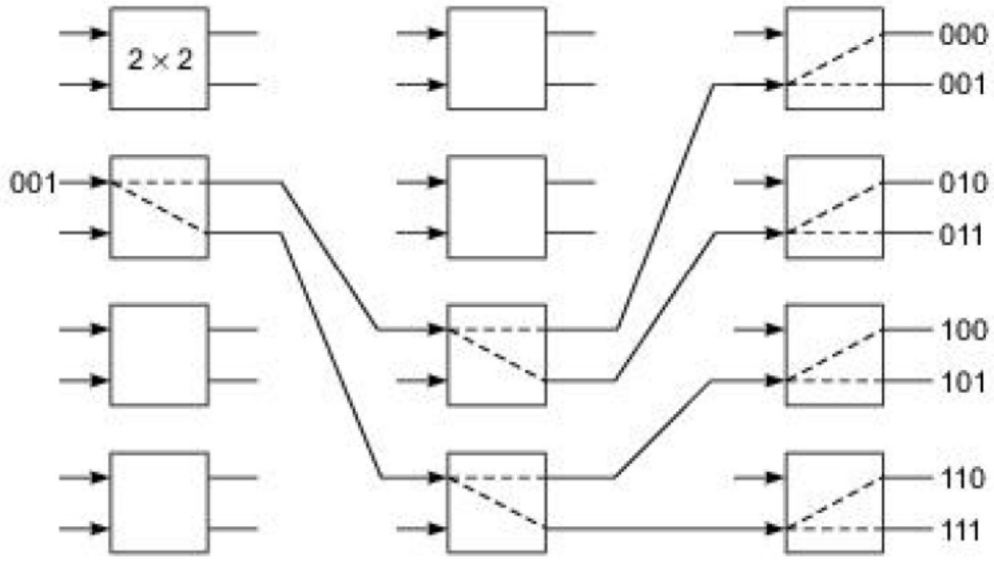
Now consider implementing the permutation π_2 in the 8-input Omega network (Fig. 7.8b). Conflicts in switch settings do exist in three switches identified as F, G, and H. The conflicts occurring at F are caused by the desired routings $000 \rightarrow 110$ and $100 \rightarrow 111$. Since both destination 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, conflicts at switch G between $011 \rightarrow 000$ and $111 \rightarrow 011$, and at switch H between $101 \rightarrow 001$ and $011 \rightarrow 000$. At switches I and J, broadcast is used from one input to two outputs.

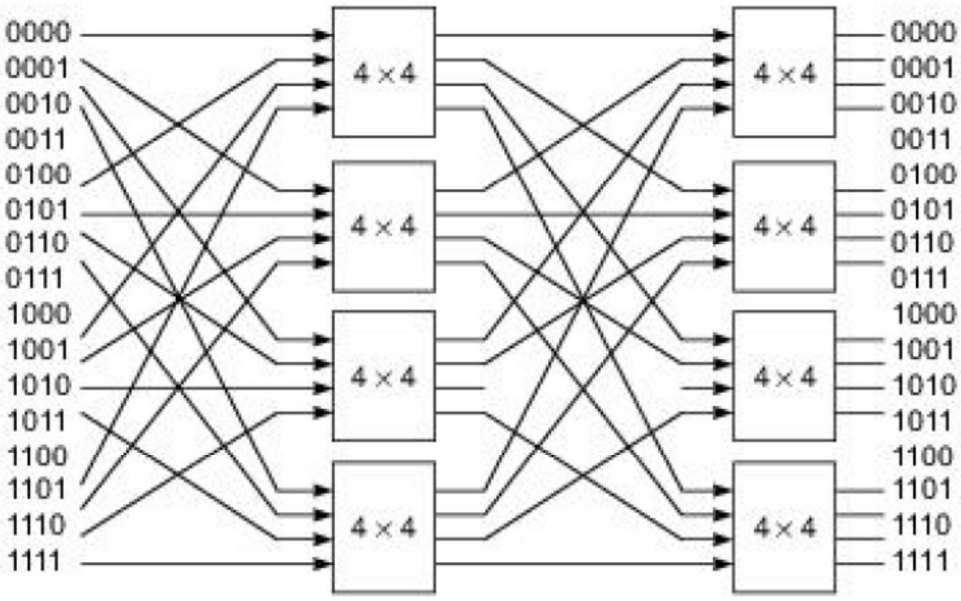
The Omega network is a blocking network. In case of blocking, one can establish the conflicting connections in several passes. For the example π_2, we can connect $000 \rightarrow 110$, $001 \rightarrow 101$, $010 \rightarrow 010$, $101 \rightarrow 001$, $110 \rightarrow 100$ in the first pass and $011 \rightarrow 000$, $100 \rightarrow 111$, $111 \rightarrow 011$ in the second pass. In general, if 2×2 switch boxes are used, an n-input Omega network can implement $n^{n/2}$ permutations in a single pass. There are $n!$ permutations in total.

The Omega network can also be used to broadcast data from one source to many destinations, as shown in below in Fig. 7.9a, using the upper broadcast or lower broadcast switch settings. In Fig. 7.9a, the message at input 001 is being broadcast to all eight outputs through a binary tree connection.

The two way shuffle interstage connections can be replaced by four-way shuffle interstage connections when 4 x 4 switch boxes are used as building blocks, as shown in below in Fig. 7.9b for an I6-input Omega network with $\log_4 16 = 2$ stages.



(a) Broadcast connections

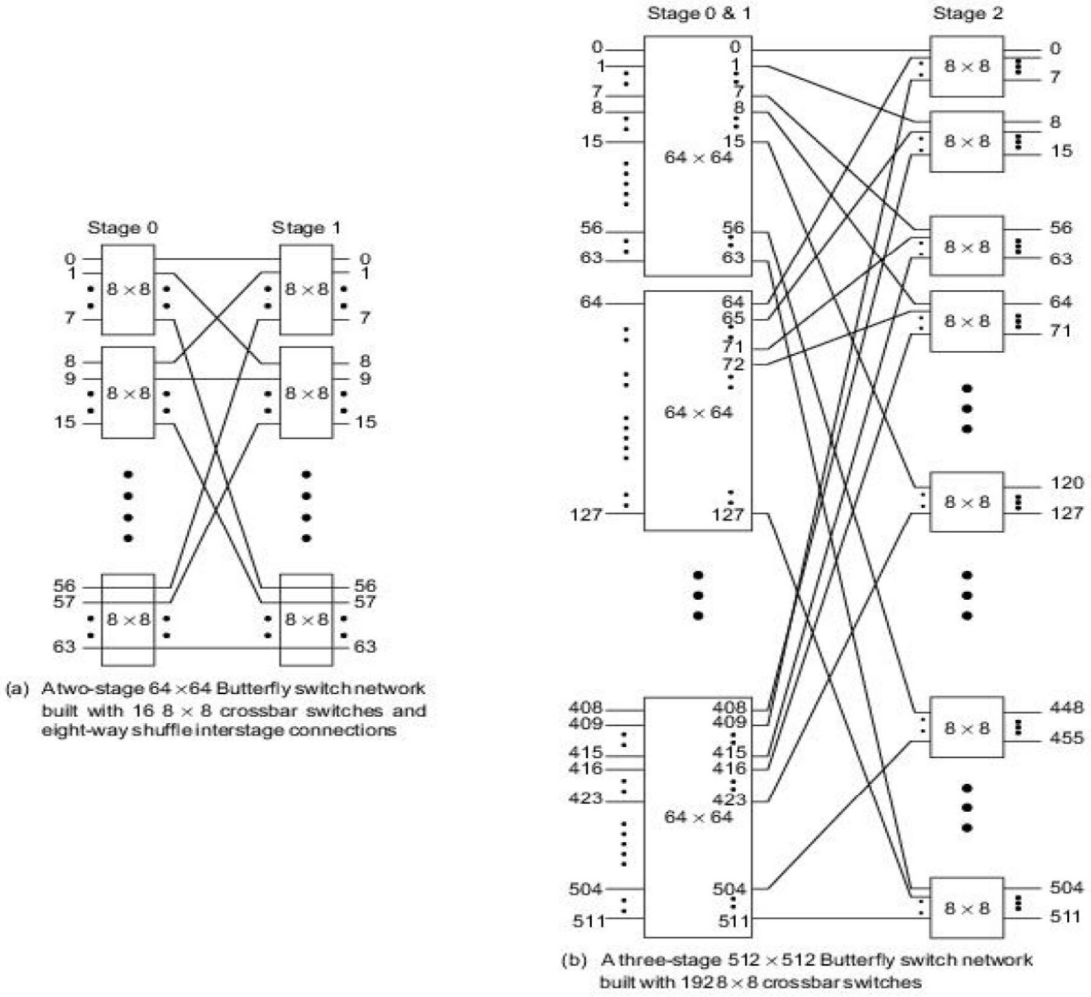


(b) Using four-way shuffle interstage connections

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

Routing In Butterfly Networks

This class of network is constructed with crossbar switches as building blocks. Figure 7.10 below shows two Butterfly networks of different sizes. Figure 7.10a shows 64-input Butterfly network built with two stages of 8 x 8 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 8 x 8 crossbar switches. In total, sixteen 8 x 8 crossbar switches are used in Fig. 7.10a and 16 x 8 + 8 x 8 = 192 are used in Fig.7.10b. 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

(b) A three-stage 512 × 512 Butterfly switch network built with 192 8 × 8 crossbar switches

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

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 a combining mechanism has been added to the Omega network to combine multiple requests heading for the same destination at switch points where conflicts are taking place.

An atomic read-modify-write primitive Fetchf&Add(x, e), has been developed to perform parallel memory updates using the combining network.

Fetch&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, x 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 Fetch&Add(x,e) at the same memory word simultaneously, the memory is updated only once following a.serialization principle. The sum of the N increments, $e_1 + e_2 + ... + e_n$, is produced in any arbitrary serialization of the N requests. This sum is added to the memory word x, resulting in a new value $x + e_1 + e_2 + ... + e_n$. Two simultaneous requests are combined in a switch as illustrated in below Fig. 7.11.

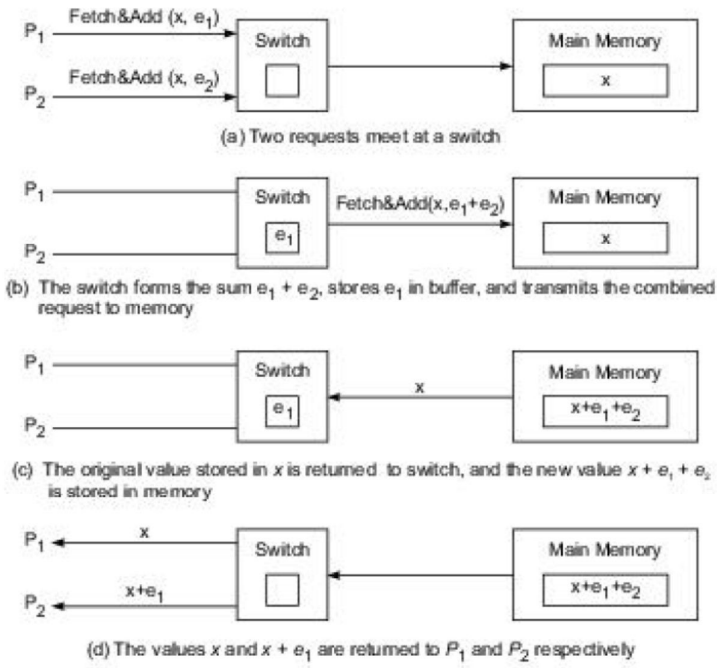


Fig. 7.11 Two Fetch&Add operations are combined to access a shared variable simultaneously via a combining network

One of the following operations will be performed if processor P_1 executes $\text{Ans}_1 \leftarrow \text{Fetch\&Add}(x, e_1)$ and P_2 executes $\text{Ans}_2 \leftarrow \text{Fetch\&Add}(x, e_2)$ 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:

$$\begin{aligned} \text{Ans}_1 &\leftarrow x \\ \text{Ans}_2 &\leftarrow x + e_1 \end{aligned} \tag{7.2}$$

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

$$\begin{aligned} \text{Ans}_1 &\leftarrow x + e_2 \\ \text{Ans}_2 &\leftarrow x \end{aligned} \tag{7.3}$$

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 $\text{Fetch\&Add}(x, e_1 + e_2)$, store the value e_1 (or e_2) in a wait buffer of the switch, and return the values x and $x + e_1$ to satisfy the original requests $\text{Fetch\&Add}(x, e_1)$ and $\text{Fetch\&Add}(x, e_2)$, respectively, as illustrated in Fig. 7.11 in four steps.

Applications and Drawbacks The Fetch&Add primitive is very effective in accessing sequentially allocated queue structures in parallel, or in forking out parallel processes with identical code that operate on different data sets.

Consider the parallel execution of N independent iterations of the following Do loop by p processors:

Doall $N = 1$ to 100
    <Code using N>
Endall

Each processor executes a Fetch&Add on N before working on a specific iteration of the loop. In this case, a unique value of N is returned to each processor, which is used in the code segment. The code for each processor is written as follows, with N being initialized as 1:

$n \leftarrow$ Fetch&Add (N, 1)
While ($n \leq 100$) **Doall**
    {Code using n}
    $n \leftarrow$ Fetch&Add(N, 1)
Endall

The advantage of using a combining network to implement the Fetch&Add operation is achieved at a significant increase in network cost. According to NYU Ultracomputer experience, message queueing and combining in each bidirectional 2×2 switch box increased the network cost by a factor of at least 6 or more.

Additional switch cycles are also needed to make the entire operation an atomic memory operation. This may increase the network latency significantly. Multistage combining networks have the potential of supporting large-scale multiprocessors with thousands of processors. The problem of increased cost and latency may be alleviated with the use of faster and cheaper switching technology in the future.

Multistage Networks in Real System

The IBM RP3 was designed to include 512 processors using a highspeed Omega network for reads or writes and a combining network for synchronization using Fetch&Adds. A 128 port Omega network in the RP3 had a bandwidth 13 Gbytes/s using a 50-MHz clock.

Multistage Omega networks were also built into the Cedar multiprocessor and in the Ultracomputer.

The BBN Butterfly processor used 8 x 8 crossbar switch modules to build a two-stage 64 x 64 Butterfly network for a 64-processor system, and a three-stage 512 x 512 Butterfly switch for a 512-processor system in the TC2000 Series.

The Cray Y-MP multiprocessor used 64, 128, or 256 way interleaved memory banks, each of which could be accessed via four ports. Thc Alliant FXF2800 used crossbar interconnects between seven four-processor (i860) boards plus one I/O board and eight shared interleaved cache boards which were connected to the physical memory via a memory bus.

CACHE COHERENCE AND SYNCHRDNIZATION MECHANISMS

In a multiprocessing environrnment, 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 this leads to cache coherence problem. Cache coherence schemes prevent this problem by maintaining a uniform state for each cached block of data. Cache inconsistencies are caused by data sharing, process migration, or I/O.

Inconsistency in Data Sharing:

The cache inconsistency problem occurs only when multiple private caches are used. The three sources of the problem are sharing of writable data, Process migration and I/O activities.

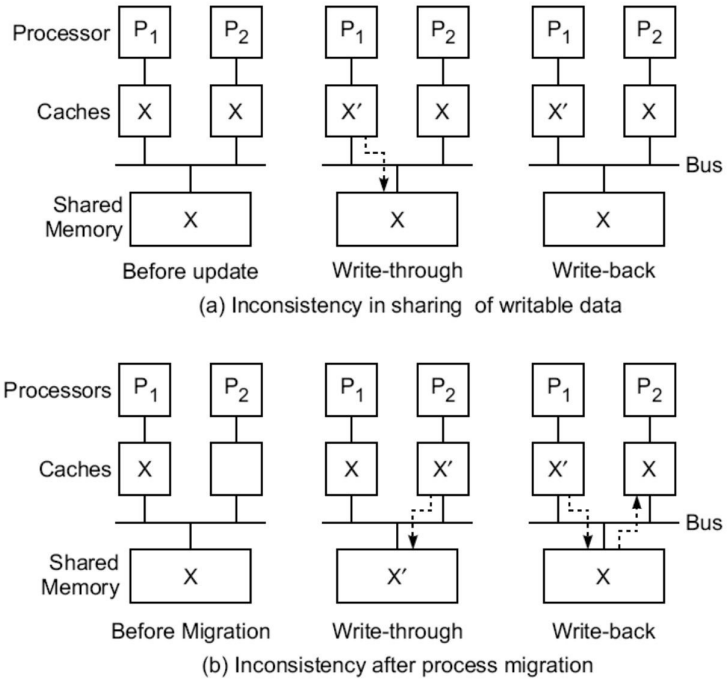


Fig. 7.12 Cache coherence problems in data sharing and in process migration (Adapted from Dubois, Scheurich, and Briggs 1988)

Above Fig7.12 illustrates the problems caused by the sharing of writable data and Process migration . 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 as shown in Fig.7.l2a.

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

Process Migration and I/O

Above Fig 7.12b shows the occurrence of inconsistency after a process containing a shared variable X migrates from processor 1 to processor 2 using the write-back cache on the right. In the middle, a process migrates from processor 2 to processor1 when using write-through caches. In both cases, inconsistency appears between the two cache copies, labeled X and X'. Special precautions must be exercised to avoid such inconsistencies. A coherence protocol must be established before processes can safely rnigrate from one processor to another.

Inconsistency problems may occur during I/O operations that bypass the caches. when the I/O processor loads a new data X' into the main memory, bypassing the write through caches as shown in middle diagram in Fig. 7.13a, inconsistency occurs between cache1 and the shared memory. When outputting a data directly from the shared memory, the write-hack 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 C1), respectively, as shown in Fig. 7.l3b. This way I/O processors share caches with the CPU. The I/O consistency can be maintained if cache-to-cache consistency is maintained via the bus. An obvious shortcoming of this scheme is the likely increase in cache perturbations and the poor locality of data, which may result in higher miss ratios.

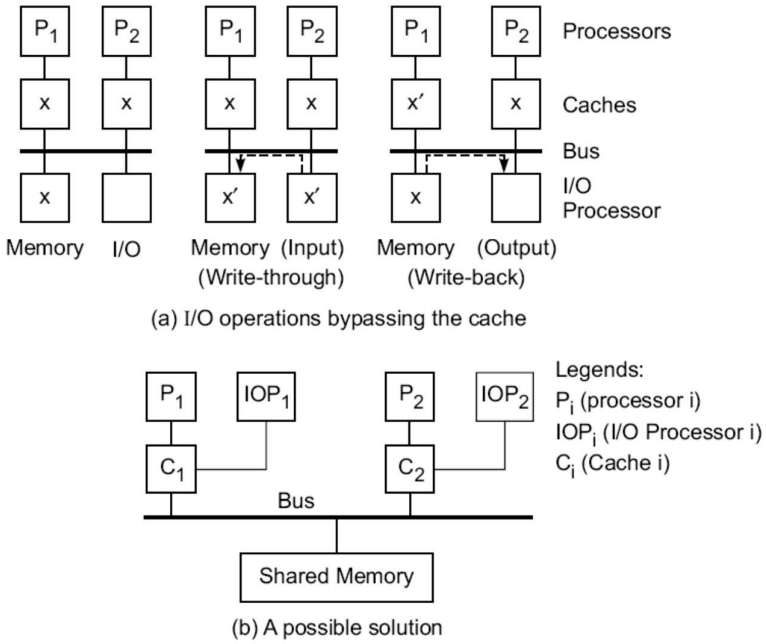


Fig. 7.13 Cache inconsistency after an I/O operation and a possible solution (Adapted from Dubois, Scheurich, and Briggs, 1988)

Two Protocol Approaches

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

In scalable multiprocessor systems interconnect processors using short point-to-point links in direct or multistage networks the cache coherence problem can be solved using some variant of directory schemes.

in general. a cache coherence protocol consists of the set of possible states in the local caches, the state in the shared memory, and the state transitions caused by the messages transported through the interconnection nctwork to keep memory coherent.

Snoopy Bus Protocols

Snoopy protocols achieve data consistency among the caches and shared memory through a bus watching mechanism. As illustrated in below Fig.7.14, two snoopy bus protocols create different results. Consider three processors (Pl, P2, and Pn) maintaining consistent copies of block X in their local caches (Fig. 7. l 4a) and in the shared-memory module marked X. Using a write-in validate protocol, the processor Pl modifies (writes) its cache from X to X', and all other copies are invalidated via the bus denoted I in Fig. 7.l4b, Invalidated blocks are also called dirty, meaning they should not be used. The write-update protocol (Fig.7.l4c) 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.

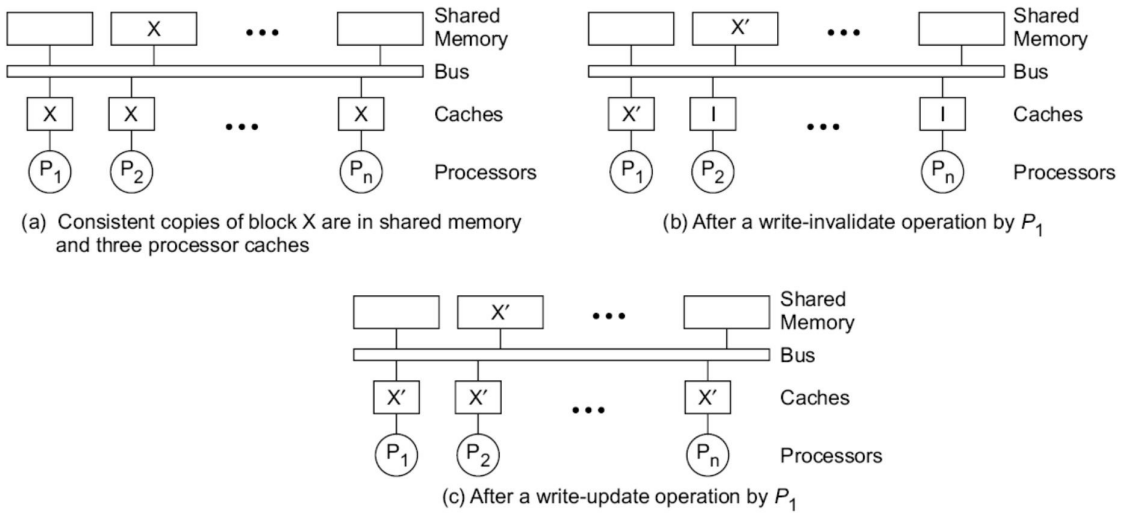


Fig. 7.14 Write-invalidate and write-update coherence protocols for write through caches (1: invalidate)

Write-trough cache

The states of a cache block copy change with respect to read, write, and replacement operations in thc cache. **Figure 7.15 shows the state transitions for two basic write-invalidate snoopy protocols** 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, wherej ≠ i. For each of the two cache states, six possible events may take place. In a valid state (Fig. 7.15a), all processors can read (R(i),R(j)) safely. Local processor i can also write (W (i)) safely in a valid state. The invalid state corresponds to the case of the block either being invalidated or being replaced (Z(i) or Z(j)). Wherever a remote processor writes W(j) into its cache copy, all other cache copies become invalidated. The cache block in cache i becomes valid whenever a successful read (R(i)) or write (W(i)) is carried out by a local processor i. The cache directory can be made in dual copies or dual-ported to filter out most invalidations. In case locks are cached, an atomic Test&Set must be enforced.

Write-Back Cache

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. l5b. The INV (invalidated or not-in-cache) cache state is equivalent to the invalid state. This three-state coherence scheme corresponds to an ownership protocol. 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.

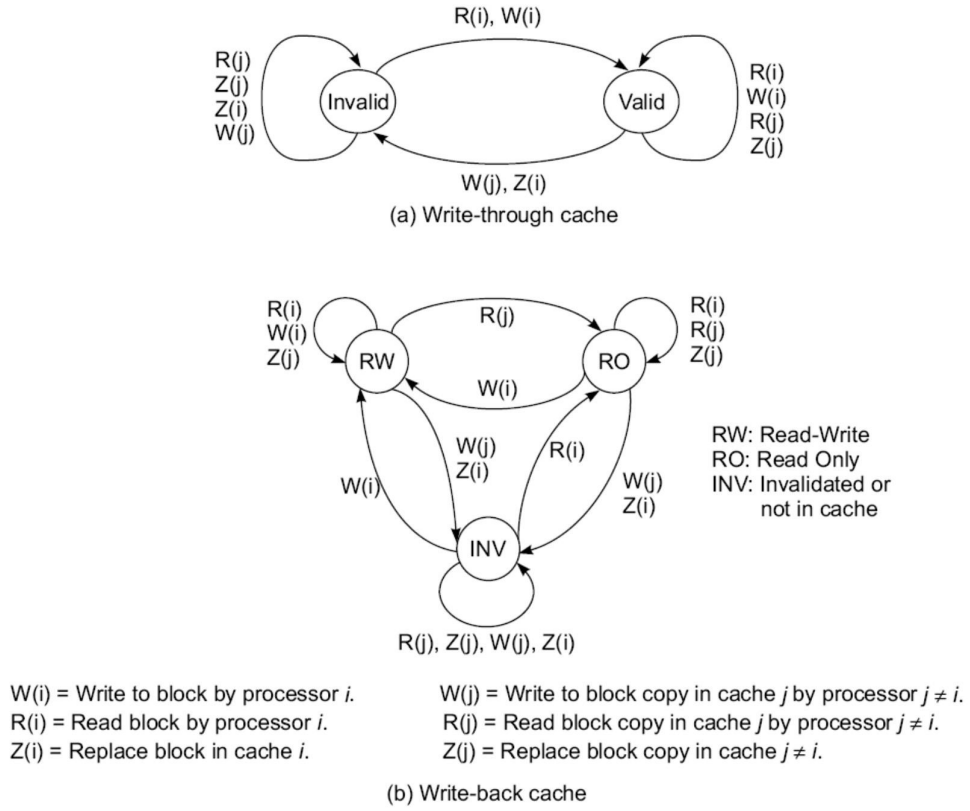


Fig. 7.15 Two state-transition graphs for a cache block using write-invalidate snoopy protocols (Adapted from Dubois, Scheurich, and Briggs, 1988)

Write-once Protocol

James Goodman 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 he described by the four-state transition graph shown in belowFig. 7.16.

The four cache states are

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 thc 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- Thc cachc block has been modified more than once, and the cache copy is the only one in the system.

To maintain consistency, the protocol requires two different sets of commands. 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 cache 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.

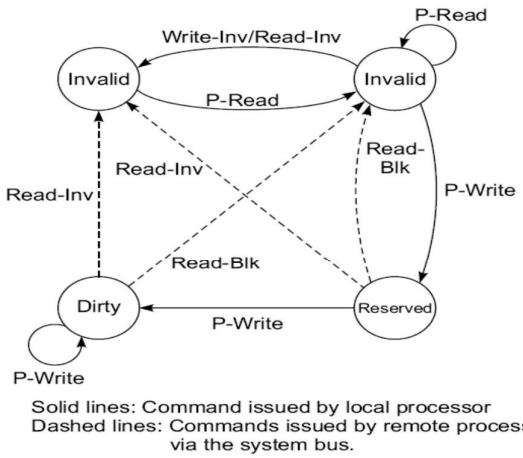


Fig. 7.16 Goodman's write-once cache coherence protocol using the write invalidate policy on write-back caches (Adapted from James Goodman 1983, reprinted from Stenstrom, *IEEE Computer*, June 1990)

Cache Event and Action

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 cachc. If a dirty copy does exist in a rcmote cache, that cache will inhibit main memory and send a copy to the requesting cache. In all cases, thc cache copy will enter the valid state after a read-miss.

Write-hit- If the copy is in thc dirty or reserved state, thc write can be carried out locally and the ncw state is dirty. If thc 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 resrved after this first write.

Write-miss- When a processor fails to write in a local cache, thc 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 cachc 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 thc snoopy bus for invalidation.

Block Replacement- If a copy is dirty it has to be written back to main memory by block rcplaccmcnt. If the copy is clean no block replacement will take place.

Multilevel Cache Coherence

To maintain consistency among cache copies at various levels. Wilson proposed an extension to the write-invalidate protocol used on a single bus. Consistency of caches at different levels is illustrated in Fig.7.3 An invalidation must propagate vertically up and down in order to invalidate all copies in the shared caches at level 2. Suppose processor P1, issues a write request. The write request propagates up to the highest level and invalidates copies in C20, C22, C16, and C18, as shown by the arrows to all the shaded copies. High-level caches such as C20 keep track of dirty blocks beneath them. A subsequent read request issued by P7 will propagate up the hierarchy because no copies exist. Wben it reaches the top level, cache C20 issucs a flush request down to cache C11 and the dirty copy is supplied to the private cache associated with processor P7.

Protocol Performance Issues

The performance of any snoopy protocol depends heavily on the workload patterns and implementation efficiency. The main motivation for using the snooping mechanism is to reduce bus traffic, with a secondary goal of reducing the effective memory-access time.

The block size is very sensitive to cache performance in write-invalidate protocols. but not in write-update protocols. For a uniprocessor system, bus traffic and memory-access time are mainly contributed by cache misses. The miss ratio decreases when block size increases.

For a system requiring extensive process migration or synchronization. the write-invalidate protocol will perform better. Write-invalidate also facilitates the implementation of synchronization primitives.

DIRECTOY-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 protocaols for network-connected multiprocessors.

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. 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 Cl, C2, and C3 requesting copies of location X. Three pointers 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, 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, and there is a single pointer to the block of data in cache C3.

Limited Directories

Limited directory protocols are designed to solve the directory size problem. A directory protocol can be classified as Dir_i X notation 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.

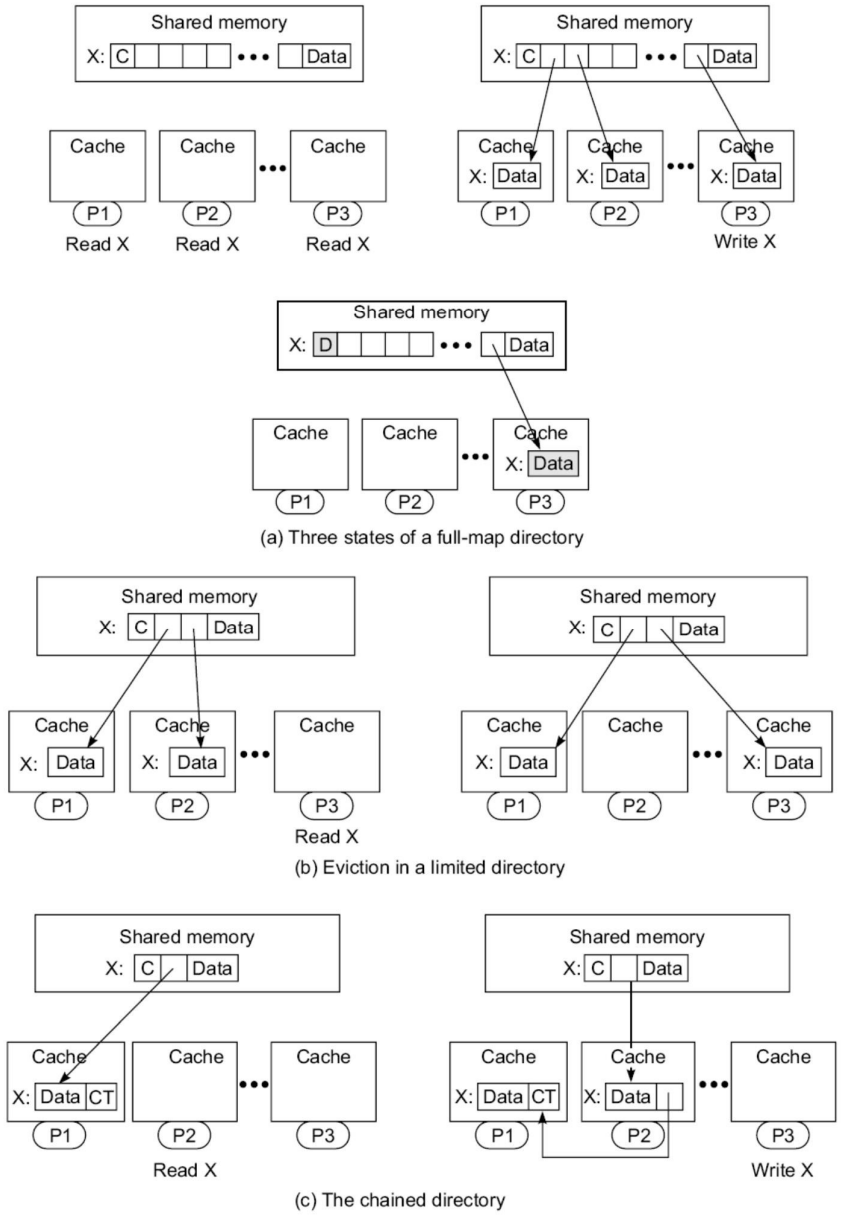


Fig. 7.18 Three types of cache directory protocols (Courtesy of Chaiken et al., *IEEE Computer*, June 1990)

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

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 Pl reads location X, the memory sends a copy to cache Cl, along with a chain termination(CT) pointer.The memory also keeps a pointer to cache Cl. 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 ensurue sequential consistency, the memory module denies processor P3 write permission until the processor with the chain termination pointer acknowledges the invalidation of the chain.

Cache Design Alternative

Shared Cache is an alternative approach to maintaining cache coherence is to completely eliminate the problem by using shared caches attached to shared-memory modules. No private caches are allowed in this ease. This approach will reduce the main memory access time but contributes very little to reducing the overall memory-access time and to resolving access conflicts.

Non-cacheable Data Another approach is not to cache shared writable data. Shared data are noncacheable and only instructions or private data are cacheable in local caches. Shared data include locks, process queues and any other data structures protected by critical sections. The compiler must tag data as either cacheable or noncacheable.

Cache Flushing A mini approach is to use cache flushing every time a synchronization primitive is executed. This may work well with transaction processing multiprocessor systems. Cache flushes are slow unless special hardware is used. This approach does not solve I/O and process migration problems.

Hardware Synchronization Mechanisms

Synchronization is a special form of communication in which control infomiation is exchanged instead of data between communicating processm residing in the same or different processors. Synchronization can be implemented in software, firmware, and hardware through controlled sharing of data and control information in memory.

Multiprocessor systems use hardware mechanisms to implement low-level or primitive synchronization operations or use software level synchronization mechanisms such as Semaphores or monitors.

Atomic Operation

Most multiproccssors are equipped with hardware mechanisms for enforcing atomic operations such as memory read, write, or read-modify-write operations which can be used to implement some synchronization primitives.

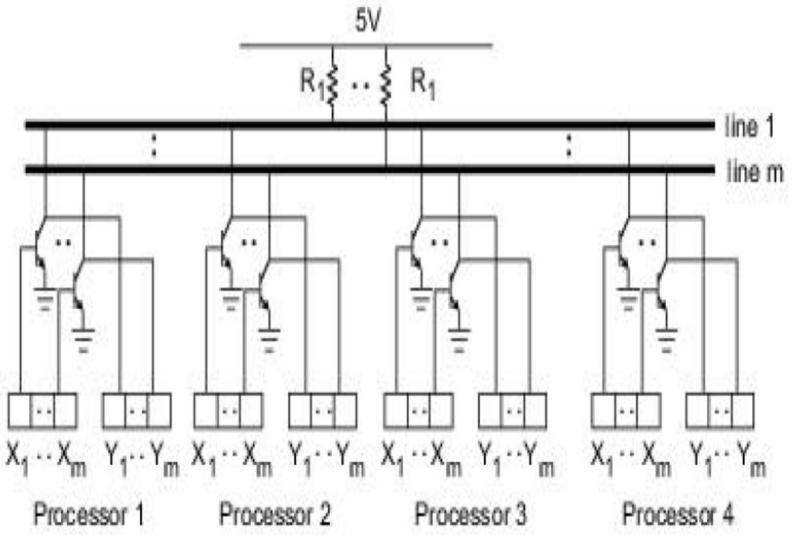
For example, the synchronization primitives, Test&Set(lock) and Reset (lock), are used for synchronization purpose as illustrated below

```
Test&Set   (lock)
           temp ← lock;    lock ← 1;
           return temp
Reset      (lock)
           lock ← 0
```

Test&Set is implemented with atomic read-modify-write memory operations. To synchronize concurrent processes, the software may repeat Test&Set until the returned value (temp) becomes 0. This synchronization primitive may tie up some bus cycles while a processor enters busy-waiting on the spin lock. A lock tied to an interrupt is called a suspended lock. Whenever the process fails to open the lock, it records its status and disables all interrupts aiming at the lock. When the lock is open, it signals all waiting processors through an interrupt. A similar primitive. Compare&Swap, was implemented in IBM mainframes. Concurrent processes residing in different processors can be synchronized using barriers. A barrier can be impirrmcnted by a shared-memory word which keeps counting the number of processes reaching the barrier. After all processes have updated the barrier counter, the synchronization point has been reached. No processor can execute beyond the barrier until the synchronization process is complete.

Wired Barrier Synchronization

A wired-NOR logic is shovm in Fig. 7.19 for implernenting a barrier mechanism for fast synchronization. Each processor uses a dedicated control vector X = (X1, ,..., Xm) and accesses a common monitor vector Y = (Y1, Y2, ...,Ym) in shared memory, where m corresponds to thc barrier lines used. Each barrier line is wired-NOR to n transistors from n processors. Whenever bit X_i is raised to high (1), the corresponding transistor is closed, pulling down the level of barrier line i. The wired-NOR connection implies that line i will be high (1) only if control bits X_i, from all processors are low (0). When all processes finish their jobs, the X_i bits of the participating processors are all set to 0 and the barrier line is then raised to high (1), signaling the synchronization barrier has been crossed. Multiple barrier lines can be used simultaneously to monitor several synchnonization points. Figtue 7.19 shows the synchronization of four processes residing on four processors using one barrier line.



(a) Barrier lines and interface logic

Step 1: Forking (use of one barrier line)

	Processor 1	Processor 2	Processor 3	Processor 4
Line	1			
X	1	1	1	1
Y	0	0	0	0

Step 2: Process 1 and Process 3 reach the synchronization point

	Process 1	Process 2	Process 3	Process 4
X	0	1	0	1
Y	0	0	0	0

Step 3: All processes reach the synchronization point

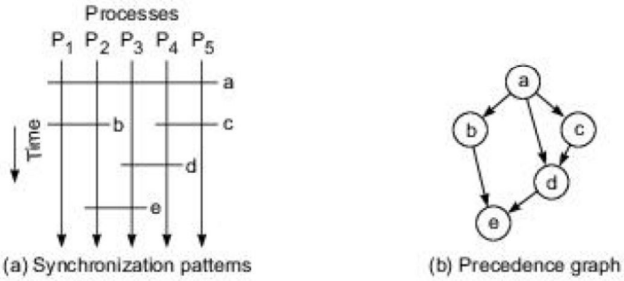
	Process 1	Process 2	Process 3	Process 4
X	0	0	0	0
Y	1	1	1	1

(b) Synchronization steps

Fig. 7.19 The synchronization of four independent processes on four processors using one wired-NOR barrier line (Adapted from Hwang and Shang, *Proc. Int. Conf. Parallel Processing*, 1991)

Example 7.2 Wired barrier synchronization of five partially ordered processes (Hwang and Shang, 1991)

If the synchronization pattern is predicted after compile time, then one can follow the precedence graph of a partially ordered set of processes to perform multiple synchronization as demonstrated in Fig. 7.20.



(a) Synchronization patterns (b) Precedence graph

Step 0: Initializing the control vectors (use 5 barrier lines)

	Processor 1	Processor 2	Processor 3	Processor 4	Processor 5
X	1 1 0 0 0	1 1 0 0 1	1 0 0 1 1	1 0 1 1 0	1 0 1 0 0
Y	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0
Step 1: Synchronization at barrier a					
X	0 1 0 0 0	0 1 0 0 1	0 0 0 1 1	0 0 1 1 0	0 0 1 0 0
Y	1 0 0 0 0	1 0 0 0 0	1 0 0 0 0	1 0 0 0 0	1 0 0 0 0
Step 2a: Synchronization at barrier b					
X	0 0 0 0 0	0 0 0 0 1	0 0 0 1 1	0 0 1 1 0	0 0 1 0 0
Y	1 1 0 0 0	1 1 0 0 0	1 1 0 0 0	1 1 0 0 0	1 1 0 0 0
Step 2b: Synchronization at barrier c					
X	0 0 0 0 0	0 0 0 0 1	0 0 0 1 1	0 0 0 1 0	0 0 0 0 0
Y	1 1 1 0 0	1 1 1 0 0	1 1 1 0 0	1 1 1 0 0	1 1 1 0 0
Step 3: Synchronization at barrier d					
X	0 0 0 0 0	0 0 0 0 1	0 0 0 0 1	0 0 0 0 0	0 0 0 0 0
Y	1 1 1 1 0	1 1 1 1 0	1 1 1 1 0	1 1 1 1 0	1 1 1 1 0
Step 4: Synchronization at barrier e					
X	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0
Y	1 1 1 1 1	1 1 1 1 1	1 1 1 1 1	1 1 1 1 1	1 1 1 1 1

(c) Synchronization steps

Fig. 7.20 The synchronization of five partially ordered processes using wired-NOR barrier lines (Adapted from Hwang and Shang, *Proc. Int. Conf. Parallel Processing*, 1991)

Here five processes (P_1, P_2, …, P_5) are synchronized by snooping on five barrier lines corresponding to five synchronization points labeled a, b, c, d, e. At step 0 the control vectors need to be initialized. All five processes are synchronized at point a. The crossing of barrier a is signaled by monitor bit Y_1, which is observable by all processors.

Barriers b and c can be monitored simultaneously using two lines as shown in steps $2a$ and $2b$. Only four steps are needed to complete the entire process. Note that only one copy of the monitor vector Y is maintained in the shared memory. The bus interface logic of each processor module has a copy of Y for local monitoring purposes as shown in Fig. 7.20c.

THREE GENERATIONS OF MULTICOMPUTERS

Design Choices in the Past

As illustrated in Fig. 7.21, the choices made involve the selection of processors, memory structure, interconnection schemes,and control strategy.

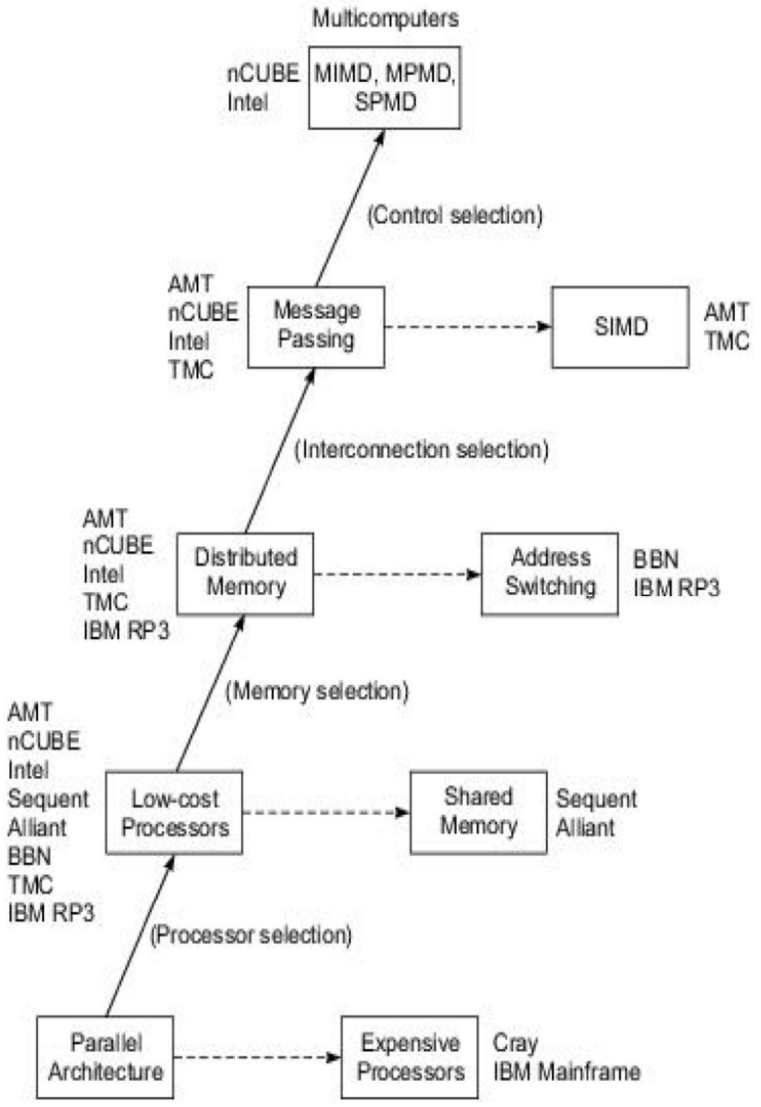


Fig. 7.21 Design choices made in the past for developing message-passing multicomputers compared to those made for other parallel computers (Courtesy of Intel Scientific Computers, 1988)

First Generation

Caltech's Cosmic Cube (Seitz, 1933) was the first of the first generation multicomputers. The lntel iPSC/1, Ametek S/14, and nCUBE/l0 were various evolutions of the original Cosmic Cube. For example, the iPSC/1 used i80286 processors with 512 Kbytes of local memory per node. each node was implemented on a single printed-circuit board with eight I/O port.

Table 7.1 *Three Early Generations of Multicomputer Development*

Generation	*First*	*Second*	*Third*
Years	*1983–87*	*1988–92*	*1993–97*
Typical node			
MIPS	1	10	100
Mflops scalar	0.1	2	40
Mflops vector	10	40	200
Memory (Mbytes)	0.5	4	32
Typical system			
N (nodes)	64	256	1024
MIPS	64	2560	l00K
Mflops scalar	6.4	512	40K
Mflops vector	640	10K	200K
Memory (Mbytes)	32	lK	32K
Communication latency (100-byte message)			
Neighbor (microseconds)	2000	5	0.5
Nonlocal (microseconds)	6000	5	0.5

The Second Generation

A major improvement of the second generation included the use of better processors. such as i386 in the iPSS/2 and i860 in the iPSC/860 and in the Delta. The nCUBE/2 implemented 64 custom-designed VLSI processors on a single PC board. The memory per node was also increased to 10 times that of the first generation. Most importantly, hardware supported routing, such as warmhole routing, reduced the communication latency significantly from 6000 μs to less than 5 μs.

The architecture of a typical second-generation multicomputer is shown in Fig. 7.22. This corresponds to a l6-node mesh-connected architecture. Mesh routing chips (MRCs) are used to establish the four-neighbor mesh network. All the mesh communication channels and MRCs are built on a backplane.

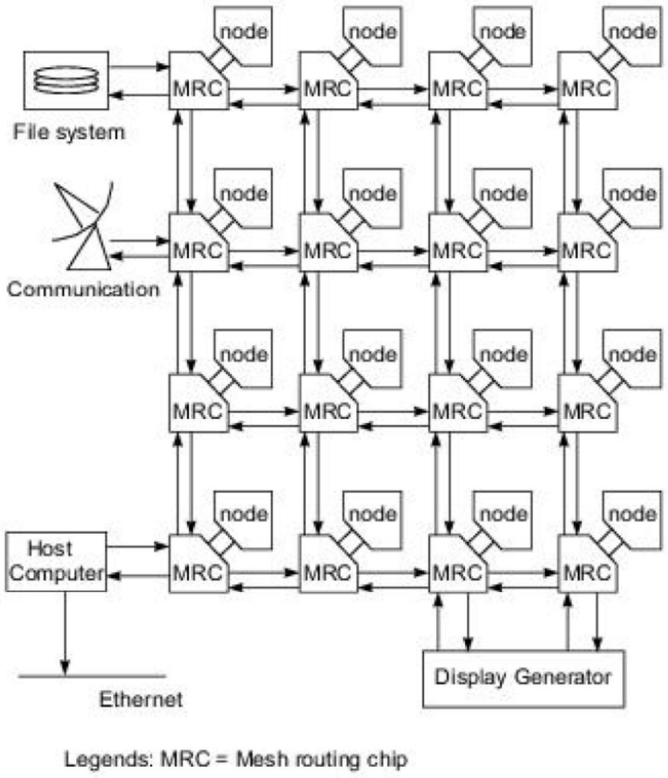


Fig. 7.22 The architecture of a second-generation multicomputer using a hardware-routed mesh interconnect (Courtesy of Charles Seitz; reprinted with permission from "Concurrent Architectures", *VLSI and Parallel Computation*, edited by Suaya and Birtwistle, Morgan Kaufmann Publishers, 1990)

The Third Generation

These designs laid the foimdation for the current generation of multicomputers. Caltoch had the Mosaic C project designed to use VLSI implemented nodes, each containing a 14-MIPS processor, 20-Mbytes routing channels, and 16 Kbytes of RAM integrated on a single chip.MIT built the J-machine which it planned to extend to a 65K-nodc multicomputer with VLSI nodes interconnected by a three-dimensional mesh network.

The first two generations of multicomputers have been called medium grain systems, With a significant reduction in communication latency, the third generation systems may be called fine-grain multicompurters.The fine-grain system may require block-level cache communications. This fine-grain and shared virtual memory approach can in theory combine the relative merits of multiproccssors and multicomputers in a hetrogeneous processing (HP) environment.

The lntel Paragon System

The hypercube multicomputers were made with homogeneous nodes this limited the I/O bandwidth, and thus these computers could not be used in solving large-scale problems with efiiciency or high throughput. The lntel Paragon was designed to overcome this difficulty. The usage model tumed the multicomputer into an applications server with multiuser access in a network environment. The architecture of the lntel Paragon system is shown in Fig. 7.23. The mesh architecture of the Paragon is divided into three sections. The middle section, called the compute partition, is a mesh of numeric nodes implemented with lntel i860XP microprocessors. This array bad an aggregate of 8.8 Gbytes of distributed memory. The system had a potential performance of 5 to 300 Gflops collectively. All I/O was handled by the two disk I/O columns at the left and right edges of the mesh. Each column was a 16 X 1 array of l6 disk nodes. The prooessors used in the I/O columns were Intel i386's which supervised the massive data transfers between the disk arrays and the computational array during I/O operations. The system I/O column was made up of six service nodes, two tape nodes, two Ethernet nodes, and a HIPPI node.

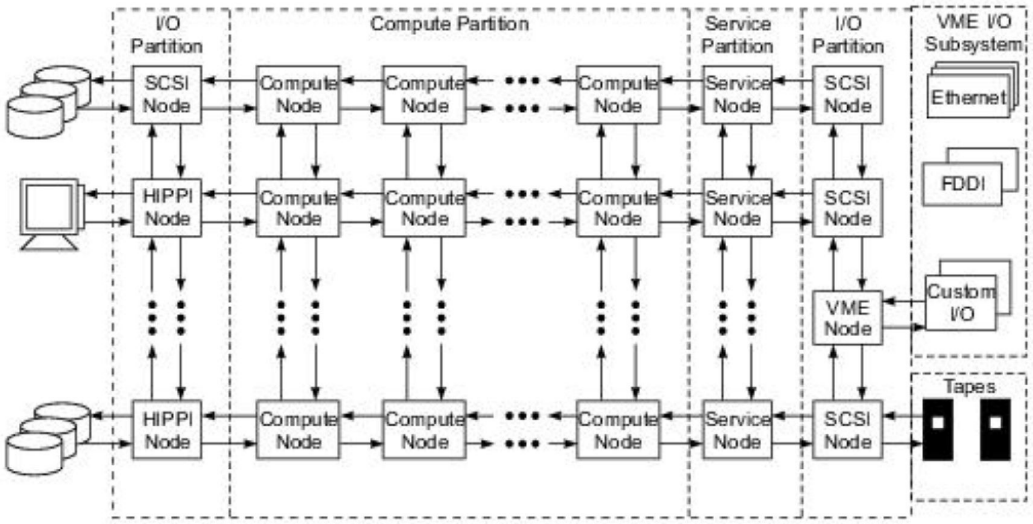


Fig. 7.23 The Intel Paragon system architecture (Courtesy of Intel Supercomputer Systems Division, 1991)

Node and Router Architecture

The Paragon was designed as an experimental system. The typical node architecture is shown in Fig. 7.24. Each node was on a separate board. For numeric nodes, the processor and floating-point units were on the same i860 chip. The local memory look up most of the board space. The external I/O interface was implemented only on the boundary nodes with a computational array. The message I/O interface was required for message passing between local nodes and the mesh network. The mesh-connected router is shown in Fig.7.25.

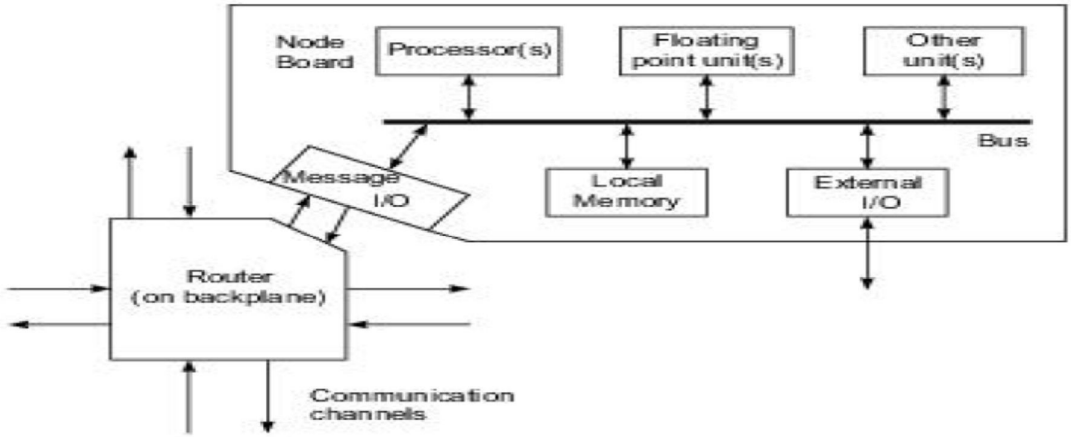


Fig. 7.24 Node architecture of the Paragon multicomputer

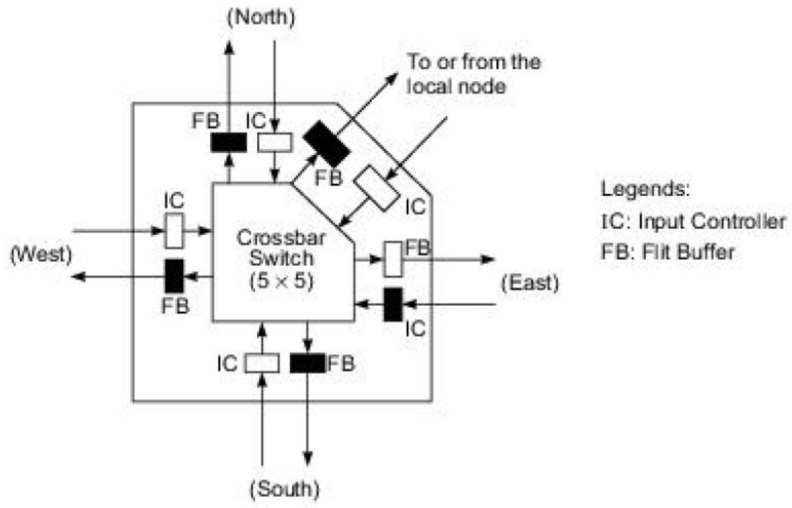


Fig. 7.25 The structure of a mesh-connected router with four pairs of I/O channels connected to neighboring routers

Flow Control Digits (flits) buffers were used at the end of input channels to hold thc incoming flits. All the I/O channels shown in Figs. 7.24 and 7.25 are Physical channels which allow only one message(flit) to pass at a time.

MESSAGE-PASSING MECHANISMS

Message-Routing Schemes

Message Format

Information units used in message routing are specified in Fig. 7.26. A message is the logical unit for internode communication. A packet is the basic unit containing the destination address for routing purposes. 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 fiits are the data elements of a packet. 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 etc.

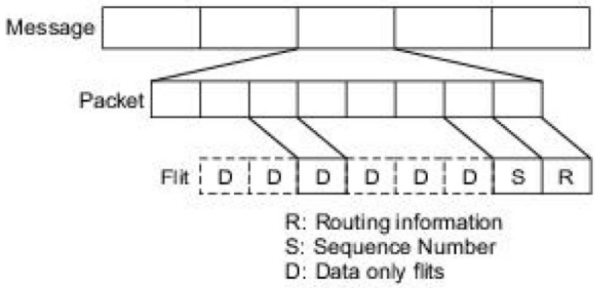


Fig. 7.26 The format of message, packets, and flits (control flow digits) used as information units of communication in a message-passing network

Store-and-Forward Routing

Packets are the basic unit of information flow in a Store-and-Forward network. The concept is illustrated in Fig. 7.27a. 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. The latency in store-and-forward networks is directly proportional to the distance (the number of hops) between the sotuce and the destination,

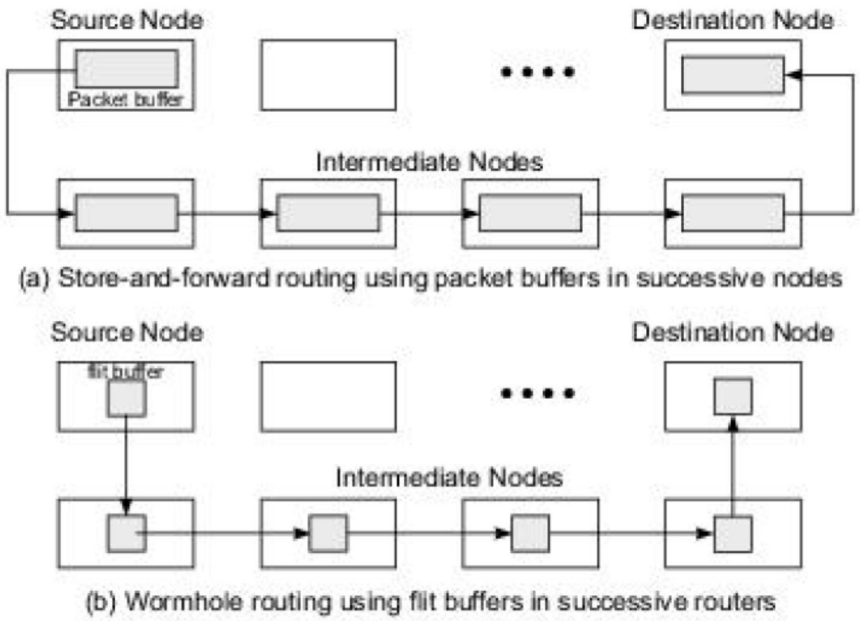


Fig. 7.27 Store-and-forward routing and wormhole routing (Courtesy of Lionel Ni, 1991)

Wormhole Routing

By subdividing the packet into smaller flits, latter generations of multicomputers implemem the wormhole routing scheme, as illustrated in above Fig. 7.27b. 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. The packet can be visualized as a railroad train with an engine car (the header flit) towing a long sequence of box ears (data flits). Only the header flit knows where the train (packet) is going. All the data flits must follow the header flit.

Asynchronous Pipelining

The pipelining of successive flits in a packet is done asynchronously using a handshalring 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 (Fig. 7.23a) to receive a flit (i.e. the flit buffer is available), it pulls the R/A line low. When the sending router (S) is ready (Fig. 7.28b), 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 (i.e. is transmitted to the next node) (Fig 7.28-d), the cycle repeats itself for the transmission of the neat flit i + 1 until the entire packet is transmitted.

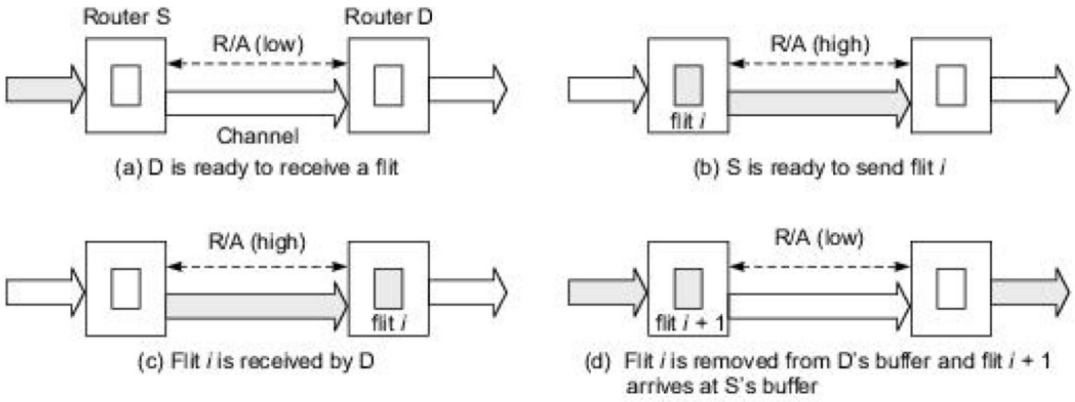


Fig. 7.28 Handshaking protocol between two wormhole routers (Courtesy of Lionel Ni, 1991)

Latency Analysis A time comparison between store-and-forward and wormhole-routed networks is given in Fig. 7.29. Let L be the packet length (in bits), W the channel bandwidth (in bits/s), D the distance (number of nodes traversed minus 1), and F the flit length (in bits).

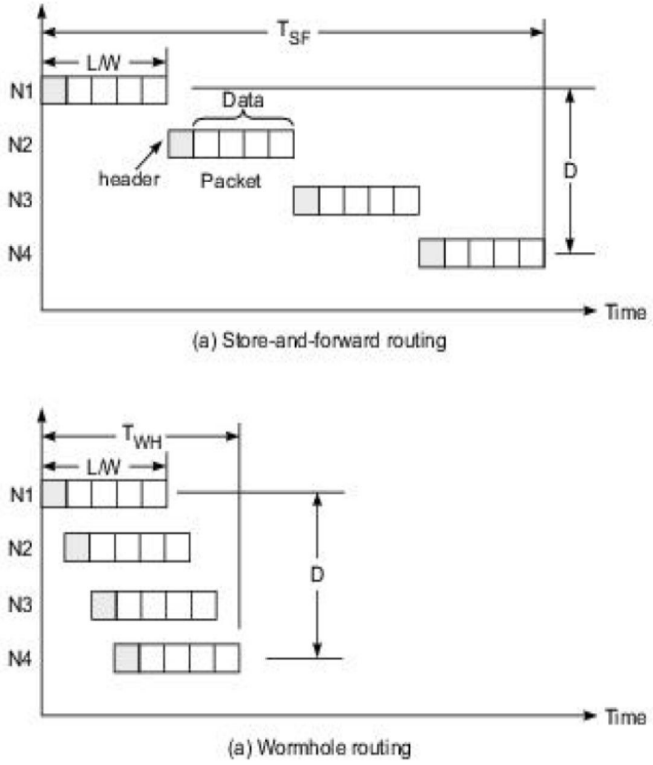


Fig. 7.29 Time comparison between the two routing techniques

The communication latency T_{SF} for a store-and-forward network is expressed by

$$T_{SF} = \frac{L}{W}(D+1) \tag{7.5}$$

The latency T_{WH} for a wormhole-routed network is expressed by

$$T_{WH} = \frac{L}{W} + \frac{F}{W} \times D \tag{7.6}$$

Equation 7.5 implies that T_{SF} is directly proportional to D. In Eq. 7.6, $T_{WH} = L/W$ if $L >> F$. Thus the distance D has a negligible effect on the routing latency.

We have ignored the network startup latency and block time due to resource shortage (such as channels being busy or buffers being full, etc.) The channel propagation delay has also been ignored because it is much smaller than the terms in T_{SF} or T_{WH}.

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 virtual channels.

Virtual Channels A virtual channel is a 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. Figure 7.30 shows the concept of four virtual channels sharing a single physical channel.

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.

In other words, the physical channel is time-shared by all the virtual channels. Besides the buffers and channel involved, some channel states must be identified with different virtual channels. The source buffers hold flits awaiting use of the channel. The receiver buffers hold flits just transmitted over the channel. The channel (wires or fibers) provides a communication medium between them.

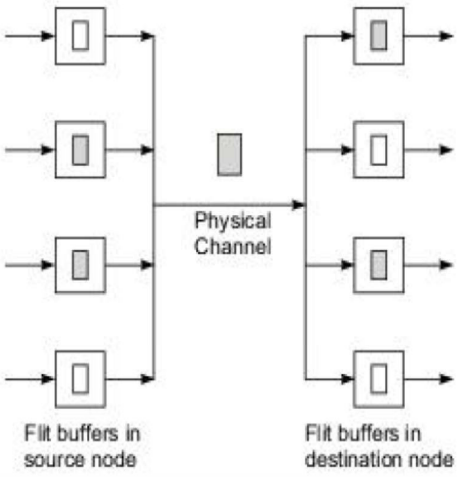
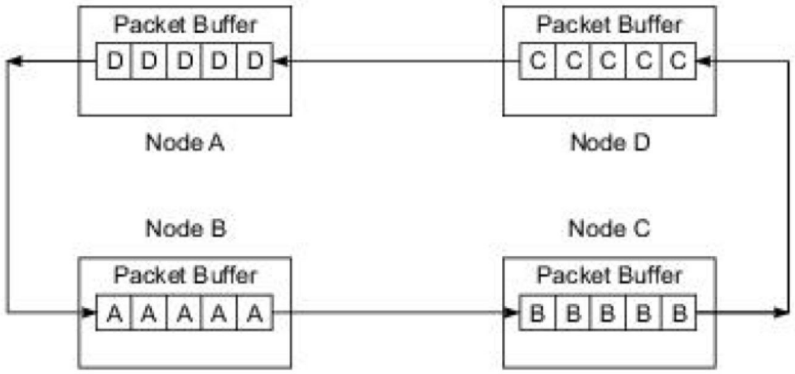


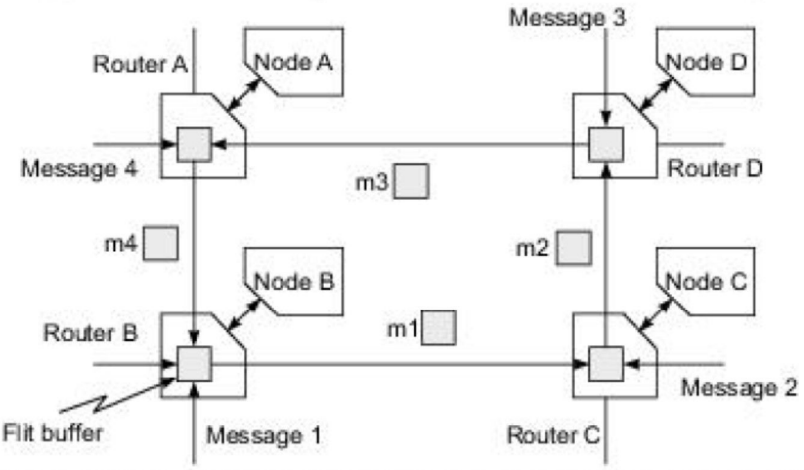
Fig. 7.30 Four virtual channels sharing a physical channel with time multiplexing on a flit-by-flit basis

Example 7.3 The deadlock situations caused by circular waits at buffers or at channels

As illustrated in below Fig. 7.31, two types of deadlock situations are caused by a circular wait at buffer or channels. A buffer deadlock is shown in Fig. 7.31a for a store-and-forward network. A circular wait situation results from four packets occupying Four buffets in four nodes. Unless one packet is discarded or misrouted, the deadlock cannot be broken. In Fig. 7.3lb, a chhanel deadlock results from four messages being simultaneously transmitted along four channels in a mesh-connected network using Wormhole routing. Four flits from four messages occupy the four channels simultaneously. If none of the channels in the cycle is freed, the deadlock situation will continue.



(a) Buffer deadlock among four nodes with store-and-forward routing



(b) Channel deadlock among four nodes with wormhole routing; shaded boxes are flit buffers

Fig. 7.31 Deadlock situations caused by a circular wait at buffers or at communication channels

Deadlock Avoidance

Adding two virtual channels, V3 and V4 in Fig. 7.32.c, will break the deadlock cycle. A modified channel-dependence graph is obtained by using the virtual channels V3 and V4, after the use of channel C1, instead of reusing C3 and C4. 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 unidireetioml channels or bidirectional channels.

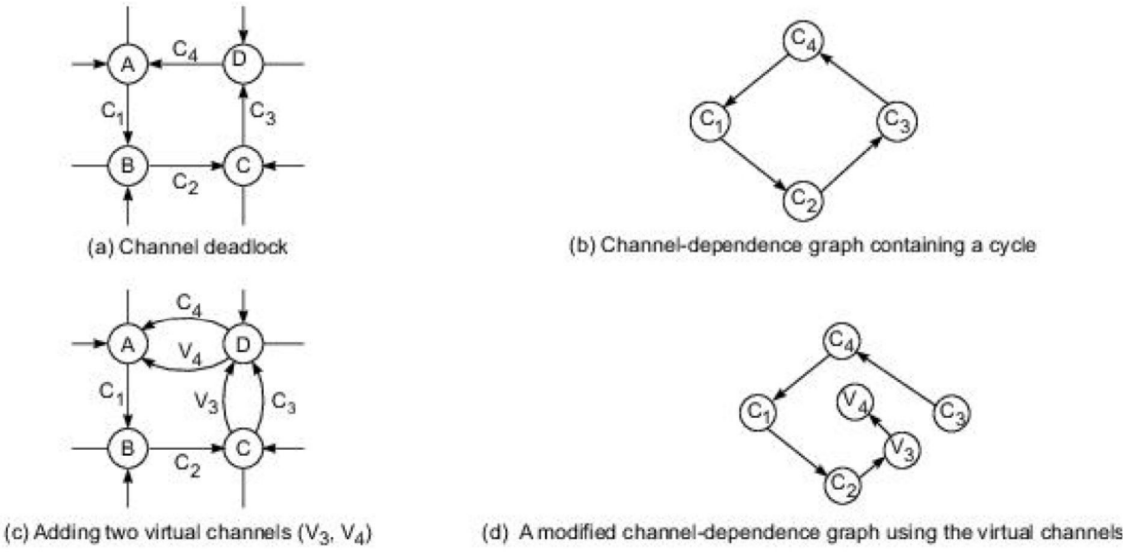


Fig. 7.32 Deadlock avoidance using virtual channels to convert a cycle to a spiral on a channel-dependence graph

Flow Control Strategies

When two or more packets collide at a node when competing for buffer or channel resources, policies must be set regarding how to resolve the conflict. Based on these policies, deterministic and adaptive routing algorithms are developed for one-to-one or unicast communication.

Pocket Collision Resolution

ln order to move a flit between adjacent nodes in a pipeline of channels, three elements must he present: 1) the source buffer holding the flit, (2) the channel being allocated, and (3) the receiver buffer accepting the flit. When two packets reach the same node, they may request the same receiver buffer or the same outgoing channel. Two arbitration decisions must be made: (i) Which packet will be allocated the channel? and (ii) What will he done with the packet being denied the channel?.

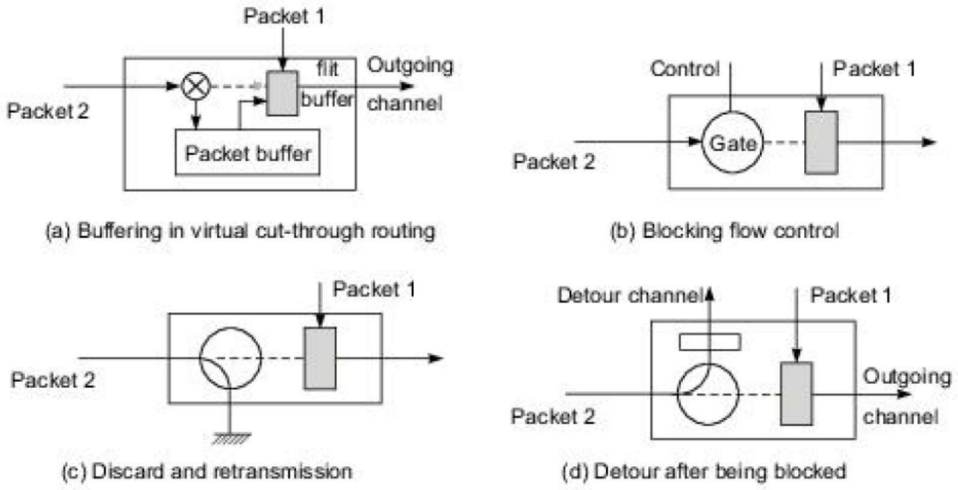


Fig. 7.33 Flow control methods for resolving a collision between two packets requesting the same outgoing channel (packet 1 being allocated the channel and packet 2 being denied)

Figure 7.33 above illustrates four methods for resolving the conflict between two packets competing for the use of the same outgoing channel at an intermediate node. Packet l is being allocated the channel, and packet 2 being denied. A buffering method has been proposed with the virtual cut-through routing in which packet 2 is temporarily stored in a packet buffer. When the channel becomes available later, it will be transmittcd thcn. This buffering approach has the advantage of not wasting the resources already allocated. Pure wormhole routing uses a blocking policy in case of packet collision. as illustrated in Fig. 7.33b. The second packet is being blocked from advancing. Figure 7.33c shows the discard policy, which simply drops the packet being blocked from passing through. The fourth policy is called detour(Fig. 7.33d), The blocked packet is routed to a detour channel. The blocking policy is economical to implement but may result in the idling of resources allocated to the blocked packet. The discard policy may result in a severe waste of resources and it demands packet retransmission and acknowledgment. Detour routing offers more flexibility in packet routing.

Dimension-Order Routing

Packet routing can be conducted deterministically or adaptively. In deterministic routing the communication path is completely detemtined by the source and destination addresses. Adaptive routing may depend on network conditions, and alternate paths are possible.

The follwing two are deterministic routing algorithms based on dimension-order routing

- X-Y routing, since a routing path along the X-dimension is decided first before choosing a path along the Y-dimension.
- E-cube routing.

E-cube Routing on Hypercube Consider an n-cube with $N = 2^n$ nodes. Each node b is binary-coded as $b = b_{n-1}b_{n-2} \ldots b_1b_0$. Thus the source node is $s = s_{n-1} \ldots s_1s_0$ and the destination node is $d = d_{n-1} \ldots d_1d_0$. We want to determine a route from s to d with a minimum number of steps.

We denote the n dimensions as $i = 1,2, \ldots, n$, where the ith dimension corresponds to the $(i-1)$st bit in the node address. Let $v = v_{n-1} \ldots v_1v_0$ be any node along the route. The route is uniquely determined as follows:

1. Compute the direction bit $r_i = s_{i-1} \oplus d_{i-1}$ for all n dimensions $(i = 1, \ldots, n)$. Start the following with dimension $i = 1$ and $v = s$.
2. Route from the current node v to the next node $v \oplus 2^{i-1}$ if $r_i = 1$. Skip this step if $r_i = 0$.
3. Move to dimension $i + 1$ (i.e. $i \leftarrow i + 1$). If $i \leq n$, go to step 2, else done.

Example 7.4 E-cube routing on a four-dimensional hypercube

The above E-cube routing algorithm is illustrated with the example in Fig. 7.34. Now $n = 4$, $s = 0110$, and $d = 1101$. Thus $r = r_4r_3r_2r_1 = 1011$. Route from s to $s \oplus 2^0 = 0111$ since $r_1 = 0 \oplus 1 = 1$. Route from $v = 0111$ to $v \oplus 2^1 = 0101$ since $r_2 = 1 \oplus 0 = 1$. Skip dimension $i = 3$ because $r_3 = 1 \oplus 1 = 0$. Route from $v = 0101$ to $v \oplus 2^3 = 1101 = d$ since $r_4 = 1$.

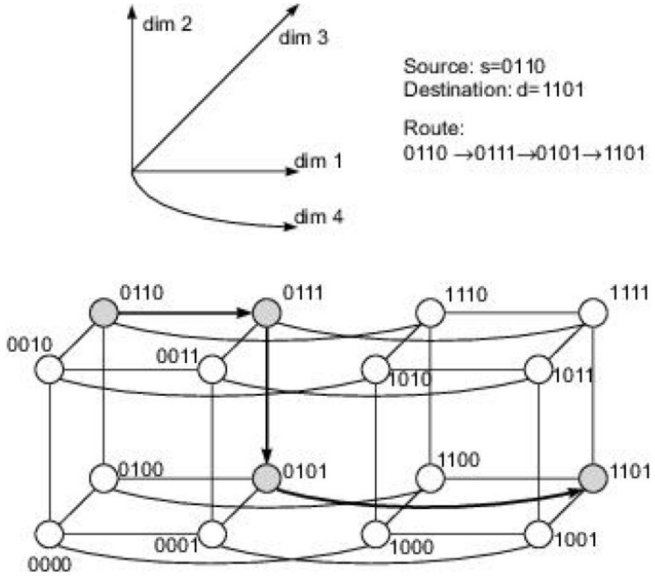


Fig. 7.34 E-cube routing on a hypercube computer with 16 nodes

The route selected is shown in Fig. 7.34 by arrows. Note that the route is determined from dimension 1 to dimension 4 in order. If the ith bit of s and d agree, no routing is needed along dimension i. Otherwise, move from the current node to the other node along the same dimension. The procedure is repeated until the destination is reached.

Example 7.5 X-Y routing on a 2D mesh-connected multicomputer

Four (source, destination) pairs are shown in below Fig. 7.35 to illustrate the four possible routing patterns on a two-dimensional mesh. An east-north route is needed fiom node (2,1) to node (7.6). An east-south route is set up from node (0,7) to node (4,2). A west-south route is needed from node (5,4) to (2,0). The fourth route is west-north bound from node (6,3) to node (1,5). If the X dimension is always routed first and then the Y-dimension, a deadlock or circular wait situation will not exist.

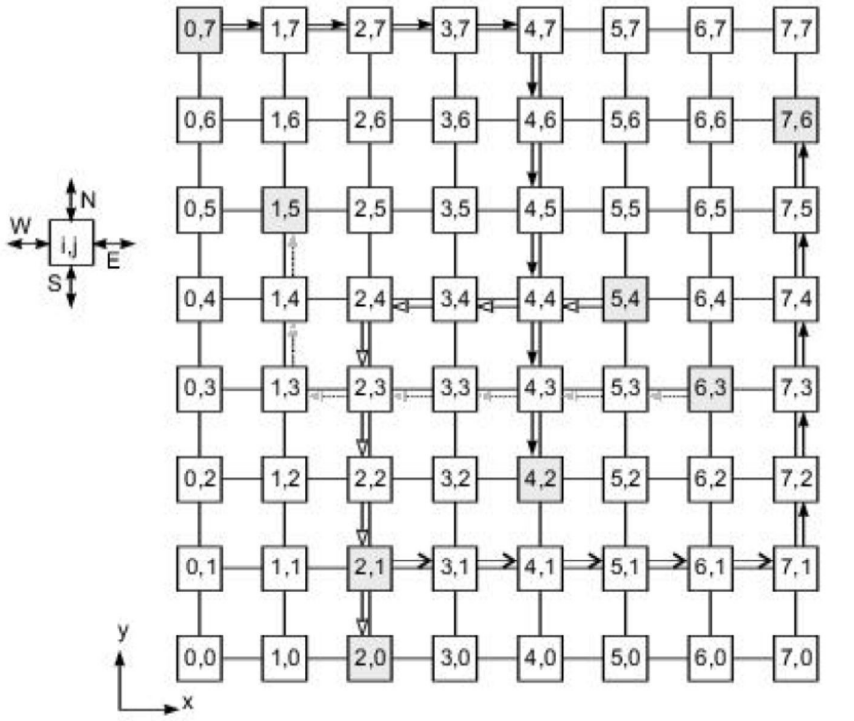


Fig. 7.35 X-Y routing on a 2D mesh computer with 8 × 8 = 64 nodes

Adaptive Routing

The main purpose of using adaptive routing is to achieve efficiency and avoid deadlock. The idea can be further extended by having virtual channels in all connections along the same dimension of a mesh-connected network as shown in below Fig. 7.36. An example of X-Y routing which uses two pairs of virtual channels in the Y-dimension of a mesh is shown in below fig 7.36. For westbound traffic, thc virtual nenvork in Fig. 7.36c can be used to avoid deadlock because all eastbound X-channels are not in use. Similarly, the virtual network in Fig 7.36d supports only eastbound traffic using a different set of virtual Y-channels.

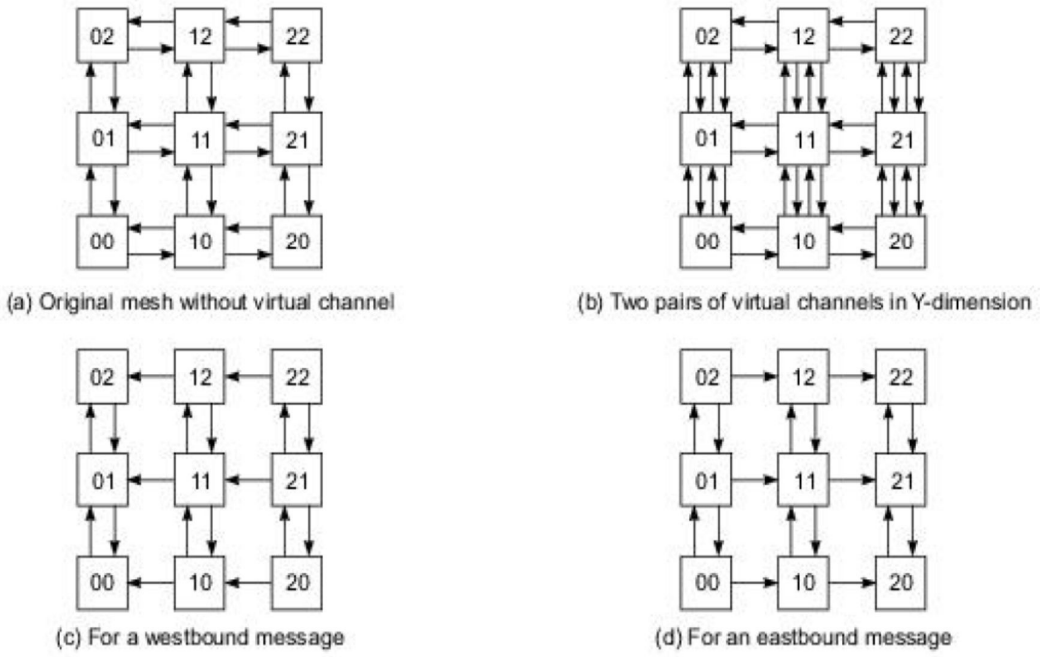


Fig. 7.36 Adaptive X-Y routing using virtual channels to avoid deadlock; only westbound and eastbound traffic are deadlock-free (Courtesy of Lionel Ni, 1991)

Multicast Routing Algorithms

Communication Pattern

one-to-one unicast pattern with one source and one destination.

A multicast pattem corresponds to one-to-many communication in which one source sends the same message to multiple destinations.

A broadcast pattern corresponds to the case of one-to-all communication.

The most generalized pattern is the many-to-many conference communication.

Routing Efficiency

Two commnly used efficiency parameters are channel bandwidth and communication latency. The channel bandwidth at any time instant (or during any time period) indicates the effective data transmission rate achieved to deliver the messages. The latency is indicated by the packet transmission delay involved.

Example 7.7 Multicast and broadcast on a mesh-connected computer

Multicast routing is implemented on a 3 × 3 mesh in Fig. 7.37. The source node is identified as S, which transmits a packet to five destinations labeled D_i for $i = 1, 2, ..., 5$.

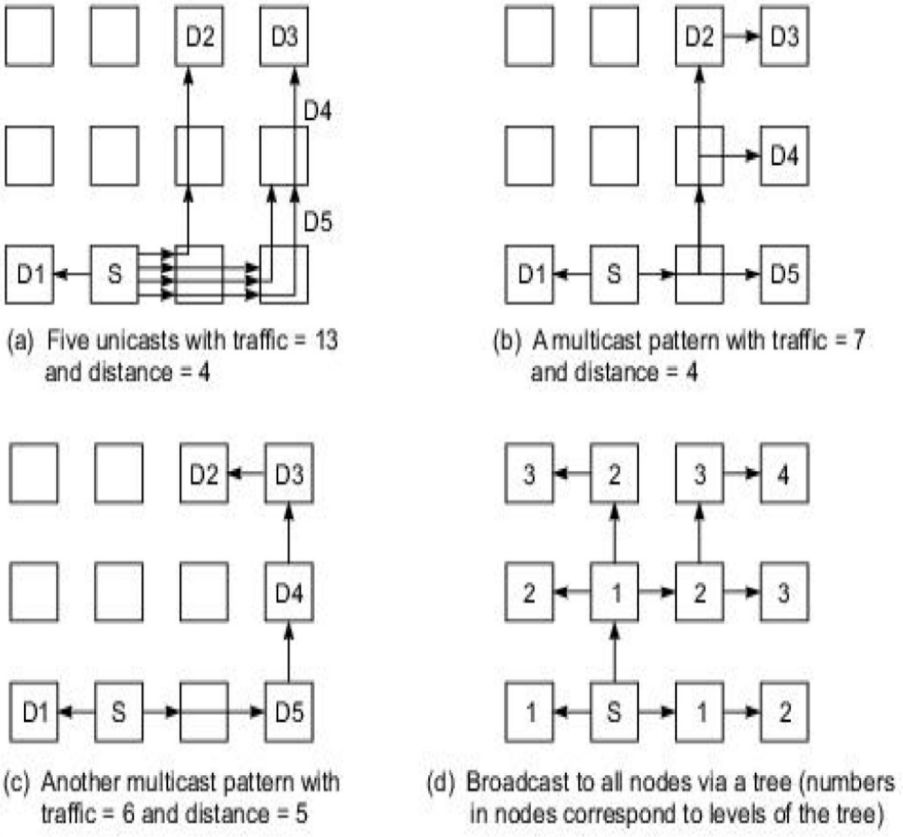
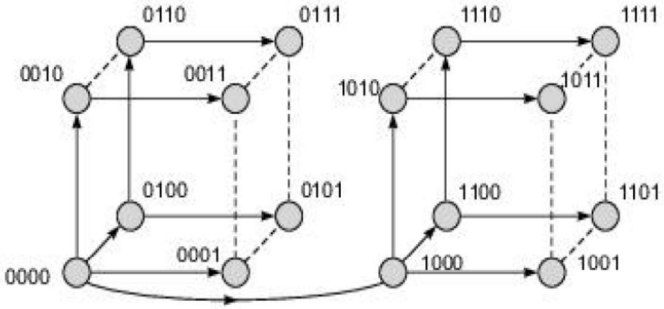


Fig. 7.37 Multiple unicasts, multicast patterns, and a broadcast tree on a 3 × 4 mesh computer

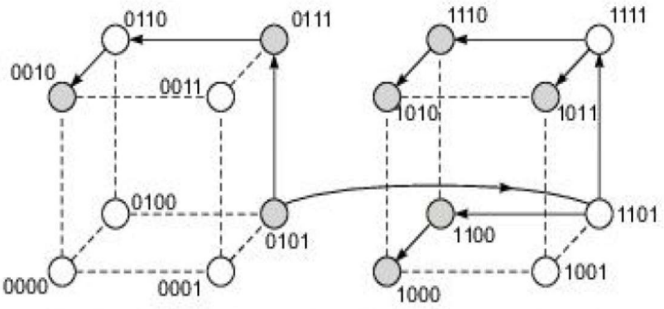
This five destinalion multicast can be implemented by five unicasts, as shown in Fig. 7.37a. The X-Y routing trafiic requires the use of 1 + 3 + 4 + 3 + 2 = 13 channels, and the latency is 4 for the longest path leading to D3. Two Multi-cast routes are given in Fig.7.37b and 7.37c. resulting in traffic of 7 and 6. respectively. On a wormhole-routed network, thc multicast route in Fig. 7.37c is better. For a store-and-forward network, thc route in Fig.7.37b is better and has a shorter latency.

Example 7.8 Multicast and broadcast on a hypercube computer

To broadcast on an n-cube, a similar spanning tree is used to reach all nodes within a latency of n. This is illustrated in Fig. 7.38a for a 4-cube rooted at node 0000. Again, minimum traffic should result with a broadcast tree for a hypercube.



(a) Broadcast tree for a 4-cube rooted at node 0000



(b) A multicast tree from node 0101 to seven destination nodes 1100, 0111, 1010, 1110, 1011, 1000, and 0010

Fig. 7.38 Broadcast tree and multicast tree on a 4-cube using a greedy algorithm (Lan, Esfahnian, and Ni, 1990)

A greedy multicast tree is shown in Fig. 7.38b for sending a packet from node 0101 to seven destination nodes. The greedy multicast algorithm is based on sending the packet through the dimension(s) which can reach the most number of remaining destinations.

Starting from the source node $S = 0101$, there are two destinations via dimension 2 and five destinations via dimension 4. Therefore, the first-level channels used are $0101 \rightarrow 0111$ and $0101 \rightarrow 1101$.

From node 1101, there are three destinations reachable in dimension 2 and four destinations via dimension 1. Thus the second-level channels used include $1101 \rightarrow 1111$, $1101 \rightarrow 1100$, and $0111 \rightarrow 0110$.

Similarly, the remaining destinations can be reached with third-level channels $1111 \rightarrow 1110$, $1111 \rightarrow 1011$, $1100 \rightarrow 1000$, and $0110 \rightarrow 0010$, and fourth-level channel $1110 \rightarrow 1010$.

Virtual Networks Consider a mesh with dual virtual channels along both dimensions as shown in Fig. 7.39a.

These virtual channels can be used to generate four possible virtual networks. For west-north traffic, the virtual network in Fig. 7.39b should be used.

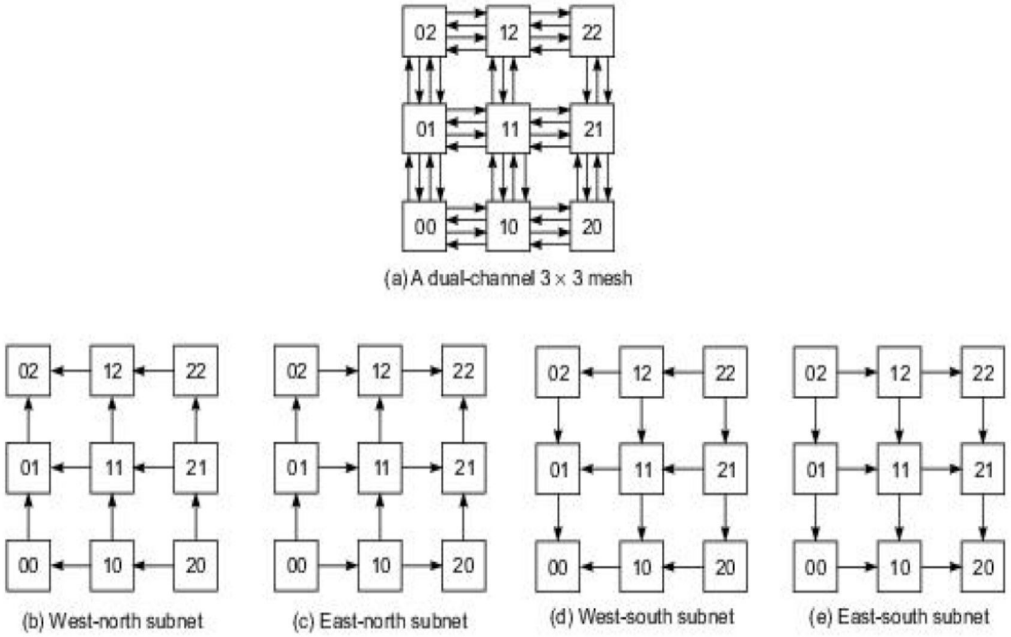


Fig. 7.39 Four virtual networks implementable from a dual-channel mesh

If both pairs between adjacent nodes are physical channels, then any two of the four virtual networks can be simultaneously used without conflict. If only one pair of physical channels is shared by the dual virtual channels between adjacent nodes, then only (b) and (c) or (e) and (d) can be used simultaneously. Other combinations, such as (b) and (c), or (b) and (d), or (c) and (e), or (d) and (c), cannot coexist at the same time due to a shortage of channels.

Network Partitioning

The conept of virtual networks leads to the partitioning of a given physical network into logical subnetworks for multicast communications. The idea is illustrated in below Fig. 7.40. Suppose source node (4, 2) wants to transmit to a ubset ofnodes in the 6 X 8 mesh. The mesh is partitioned into four logical subnets. All traffic heading for east and north uses the subnet at the upper right corner. Similarly, one constructs three other subnets at the remaining corners of the mesh.

Nodes in the fifth column and third row are along the boundary between subnets. Essentially, the traffic is being directed outward from the center node (4, 2). There is no deadlock if an X-Y multicast is performed in this partitioned mesh.

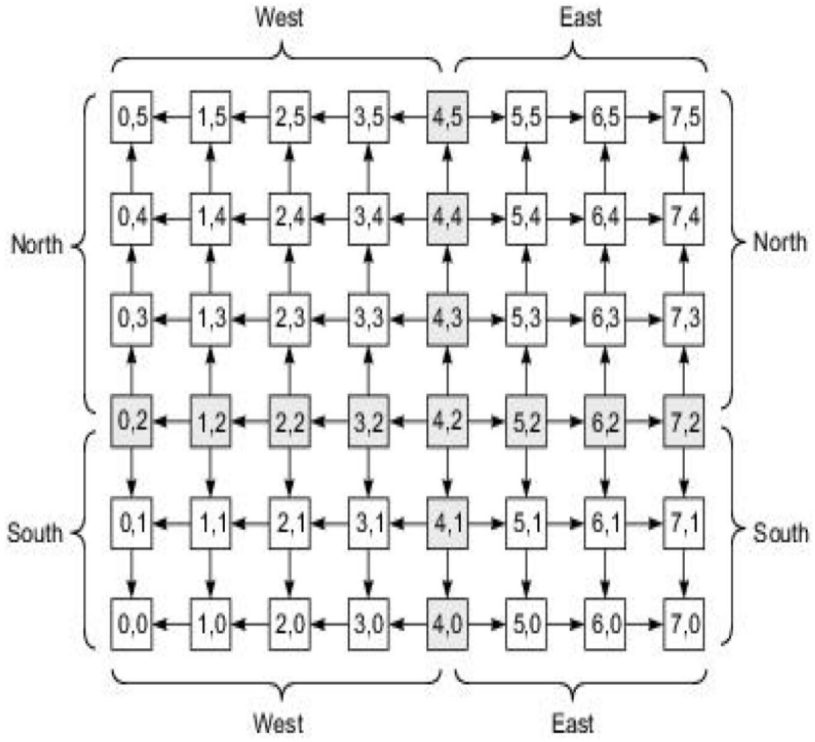


Fig. 7.40 Partitioning of a 6 × 8 mesh into four subnets for a multicast from source node (4,2). Shaded nodes are along the boundary of adjacent subnets (Courtesy of Lin, McKinly, and Ni, 1991)

SCALABLE, MULTITHREADED, AND DATAFLOW ARCHITECTURES

LATENCY HIDING TECHNIQUES

Latency hiding can be accomplished through four complementary approaches:

Using prefetching techniques which bring instnictions or data close to the processor before they are actually needed.

Using Coherent caches supported by hardware to reduce cache misses

Using relaxed consistency memory models by allowing buffering and pipelining of memory references.

Using multiple contexts support to allow a processor to switch from one contest to another when a long-latency operation is encountered.

Shared Virtual Memory

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

The Architecture Environment

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

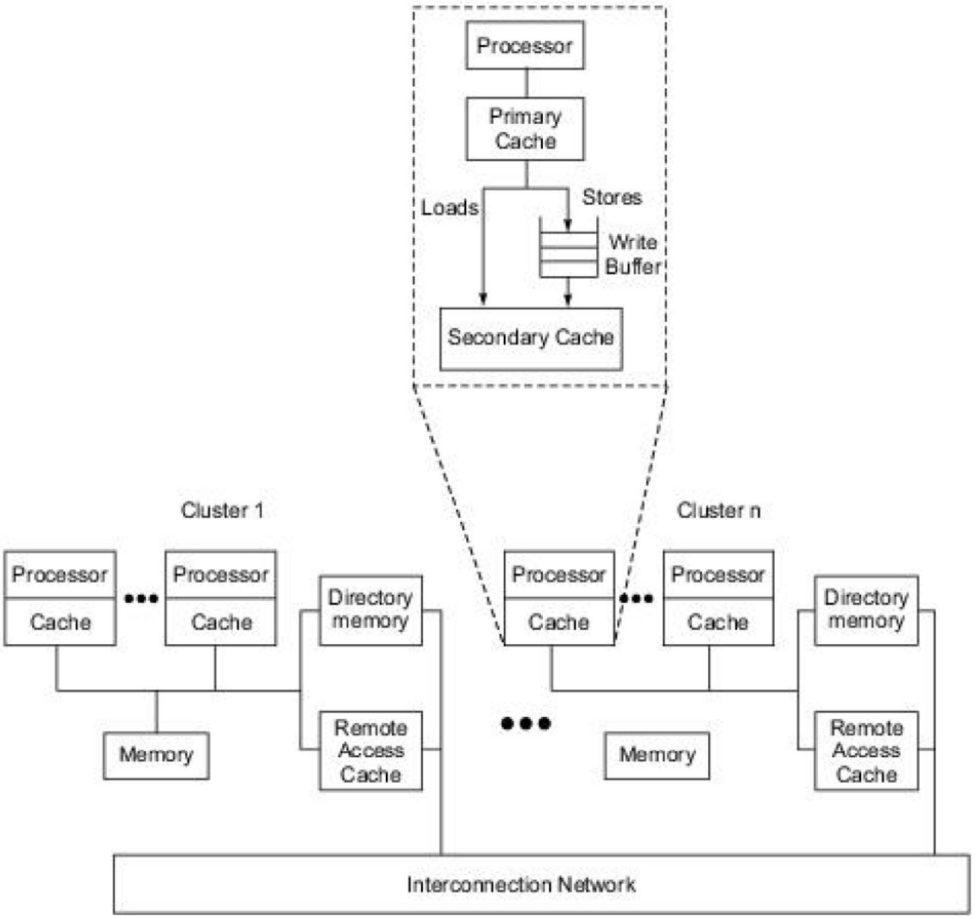


Fig. 9.1 A scalable coherent cache multiprocessor with distributed shared memory modeled after the Stanford Dash (Courtesy of Anoop Gupta et al, *Proc. 1991 Ann. Int. Symp. Computer Arch.*)

The SVM Concept

Figure 9.2 below 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 was 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 ofphysical addresses for memory references.

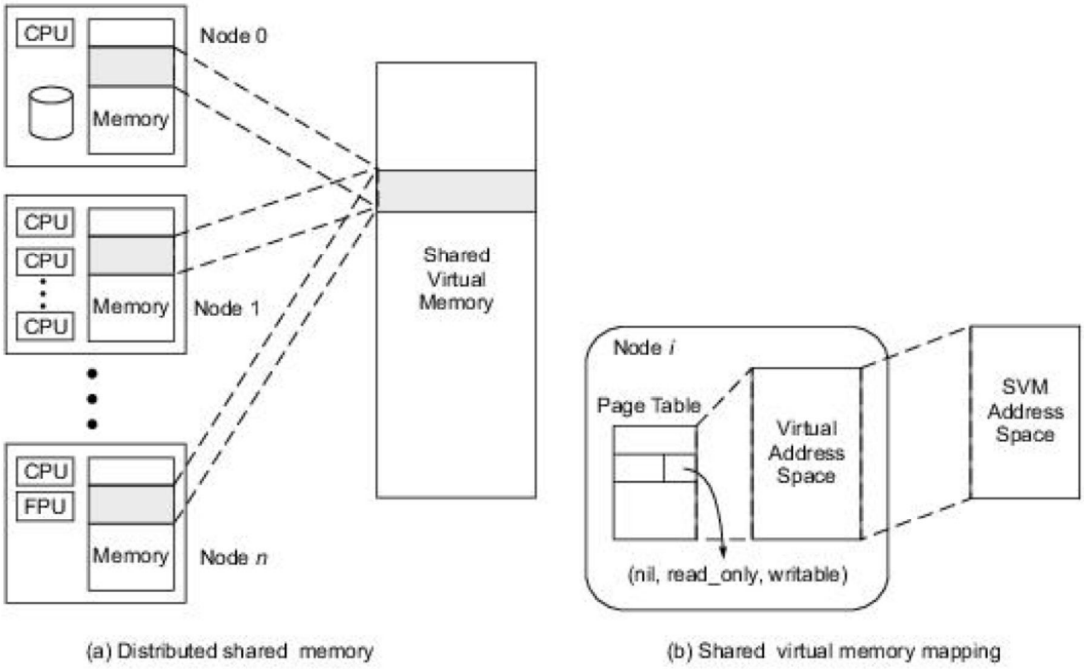


Fig. 9.2 The concept of distributed shared memory with a global virtual address space shared among all processors on loosely coupled processing nodes in a massively parallel architecture (Courtesy of Kai Li, 1992)

Page Swapping

A memory reference causes a page fault when the page containing thc memory location is not in a processors local memory. When a page fault occurs, the memory manager retrieves the missing page from the memory of another processor. lf 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 contentsto 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.

Example SVM System

Nitzberg and Lo (1991) conducted a survey of SVM research systems. Excerpted from their survey, descriptions of four representative SVM systems are suinrnarized in Table 9.1. Dash implemented SVM with a directory-based coherence protocol. Linda ofiercd a shared associative object rnernory with access Functions.

Table 9.1 *Representative SVM Research Systems (Excerpts from Nitzberg and Lo, IEEE Comput., August 199 1)*

System and Developer	*Implementation and Structure*	*Coherence Semantics and Protocols*	*Special Mechanics for Performance and Synchronization*
Stanford Dash (Lenoski, Laudon, Gharachorloo, Gupta, and Hennessy, 1988–).	Mesh-connected network of Silicon Graphics 4D/340 workstations with added hardware for coherent caches and prefetching.	Release memory consistency with write-invalidate protocol.	Relaxed coherence, prefetching, and queued locks for synchronization.
Yale Linda (Carriero and Gelernter, 1982–).	Software-implemented system based on the concepts of tuple space with access functions to achieve coherence via virtual memory management.	Coherence varied with environment; hashing used in associative search; no mutable data.	Linda could be implemented for many languages and machines using C-Linda or Fortran-Linda interfaces.
CMU Plus (Bisiani and Ravishankar, 1988–).	A hardware implementation using MC 88000, Caltech mesh, and Plus kernel.	Used processor consistency, nondemand write-update coherence, delayed operations.	Pages for sharing, words for coherence, complex synchronization instructions.
Princeton Shiva (Li and Schaefer, 1988).	Software-based system for Intel iPSC/2 with a Shiva/native operating system.	Sequential consistency, write-invalidate protocol, 4-Kbyte page swapping.	Used data structure compaction, messages for semaphores and signal-wait, distributed memory as backing store.

Prefetching Techniques

Prefetching uses knowledge 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 or nonbinding, and whether it is controlled by hardware or software.

With binding prefetching, the value of a later reference (e.g. a register load) is bound at the time when the prefetch completes. This places restrictions on when

a binding prcfctch can be issued, since the value will become stale if another processor modifies the same location during the interval between prefetch and reference. Binding prefetching may result in a significant loss in perfomtance due to such limitations.

ln contrast, nonbinding prefetching also brings the 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. The effectiveness of long cache lines is limited by the reduced spatial locality in multiprocessor applications, while instruction lookahead is limited by branches and the finite lookahead bufier size.

With software-controlled prefetching, explicit prefetch instructions are issued. Software control allows the prefctching to be done selectively.The disadvantages of software comrol includc the extra instruction overhead required to generate the prefetches, as well as the need for sophisticated software intervention.

Benefits of Prefetching

The most obvious benefit occurs when a prefetch is issued early enough in thc code so that thc line is already in thc caehe by the time it is referenced.

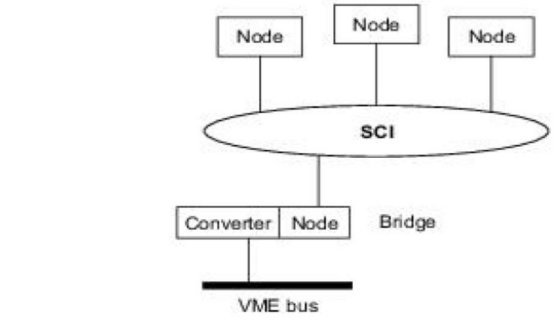
Prefetching offers another benefit in multiproccssors that use an ownership based cache coherence protocol. If a cache block line is to be modified, prefetching it directly with ownership can significantly reduce the write latencies and the ensuing network traffic for obtaining ownership.

Network traflic is reduced in read-modify-write instructions. since prefctching with ownership avoids first fetching a read-shared copy.

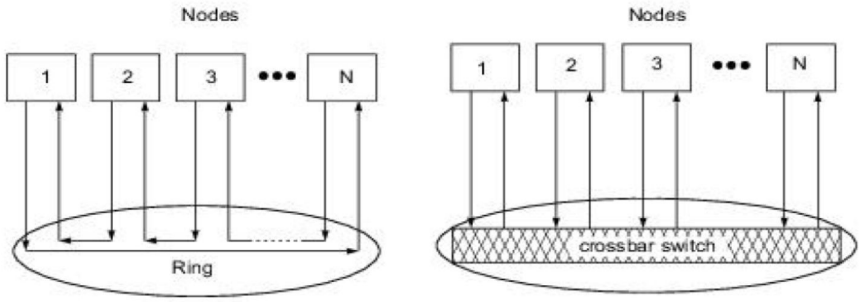
Scalable Coherence Interface

A scalable coherence interconnect. structure with low latency is needed to extend from conventional bused backplanes to a fully duplex, point-to-point interface specification.

SCI Interconnect Models SCI defines the interface between nodes and the external interconnect, using 16-bit links with a bandwidth of up to 1 Gbyte/s per link. As a result, backplane buses have been replaced by unidirectional point-to-point links. A typical SCI configuration is shown in Fig. 9.5a. Each SCI node can be a processor with attached memory and I/O devices. The SCI interconnect can assume a ring structure or a crossbar switch as depicted in Figs. 9.5b and 9.5c, respectively, among other configurations.



(a) Typical SCI configuration with bridge to other bus



(b) A ring for point-to-point transactions (b) A crossbar multiprocessor

Fig. 9.5 SCI interconnection configurations (Reprinted with permission from the IEEE Standard 1596-1992, copyright © 1992 by IEEE, Inc.)

The converter in Fig. 9.5a is used to bridge the SCI ring to the VME bus as shown. A mesh of rings can also be considered using some bridging modules. The bandwidth, arbitration, and addressing mechanisms of an SCI ring significantly outperform backplane buses. by eliminating the snoopy cache controllers, the SCI is also less expensive per node, but the main advantage lies in its low latency and scalability.

Sharing-List Structures

Sharing lists are used in SCI to build chained directories for cache coherence use. The length of the slrraiing lists is effectively unbounded. Sharing lists are dynamically created, pruned, and destroyed. Each coherently cached block is entered onto a list of processors sharing the block. Processors have the option of bypassing the coherence protocols for locally cached data. Cache blocks of 64 bytes are assumed. By distributing the directories among the sharing processors, SCI avoids sealing limitations imposed by using a central directory. Communications among sharing processors are supported by heavily shared memory controllers, as shown in Fig. 9.6.

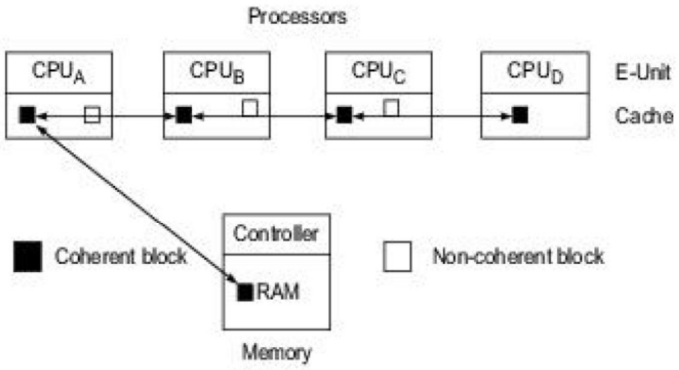


Fig. 9.6 SCI cache coherence protocol with distributed directories (Courtesy of D.V. James et al, *IEEE Computer*, 1990)

Sharing-List Creation

The states of the sharing list are defined by the state of the memory and the states of list entries.The shared memory is either in a home (uncached) or a cached (sharing-list) state. The sharing-list entries specify the cache properties, such as clean,dirty,valid, or stale. Thc head processor is always rcsponsible for list management. The memory is initially in the home state , and all cache copies are invalid. Sharing-list creation begins at the cache where an entry is changed from an invalid to a pending state. When a read-cache transaction is directed from a processor to the memory controller, the memory state is changed from uncachcd

to cached and thc requested data is returned. The requester's cachc entry statc is then changcd from a pending state to an only-clean state. Sharing-list creation is illustrated in Fig. 9.7a. Multiple requests can be simultaneously generated, but they are processed soquentially by thc memory controller.

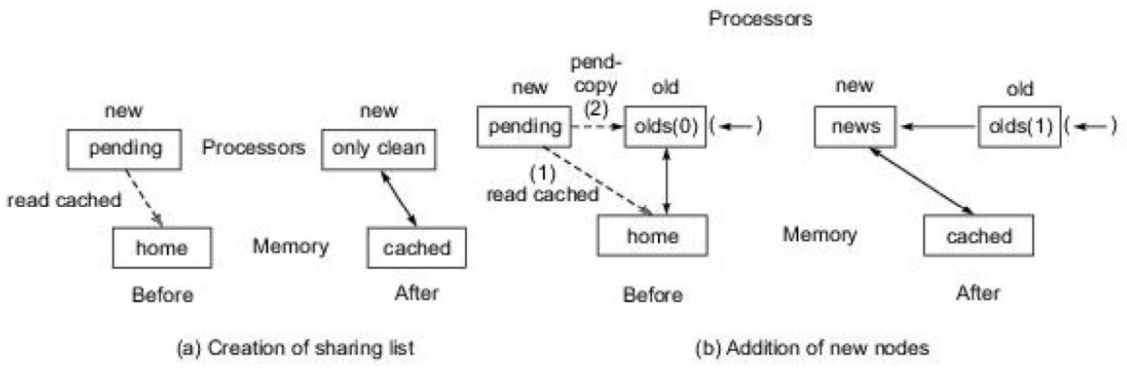


Fig. 9.7 Sharing-list creation and update examples (Courtesy of D.V. James et al, *IEEE Computer*, 1990)

Sharing-List Update

For subsequent memory access, the memory state is cached, and the cache head of the sharing list has possibly dirty data. As illustrated in Fig. 9.7b, a new requester (cache A) first directs its read-cache transaction to memory but receives a pointer to cache B instead of the requested data. A second cache-to-cache transaction, called prepend is directed from cache A to cache B. Cache B then sets its backward pointer to point to cache A and returns the requested data. The dashed lines correspond to transactions between a processor and memory or another processor. The solid lincs are sharing-list pointers After the transaction, the inserted cache A becomes the new head, and the old head, cache B, is in the middle as shown by the new sharing list on the right in Fig. 9.7b.

Implementation Issue

lt can support various multiprocessor topologies using Omega or crossbar network. Differential emitter coupled logic (ECL) signaling works well at SCI clock rates.

Relaxed Memory Consistency

Processor Consistency Goodman (I989) introduced the Processor Consistency (PC) model in which writes issued by each individual processor are always in program order.

The PC model relaxes from the sequencial consistency model by removing some restrictions on writes from different processors. Two conditions related to other proccssors are required for ensuring processor consistency:

1) Before a read is allowed to perform with respect to any other processor, all previous read accesses must be performed.
2) Before a write is allowed to perform with respect to any other processor, all previous read or write accesses must be performed.

These conditions allow reads following a write to by pass the write. To avoid deadlock the implementation should guarantee that a write that appears previously in program order will eventually be performed.

Release Consistency

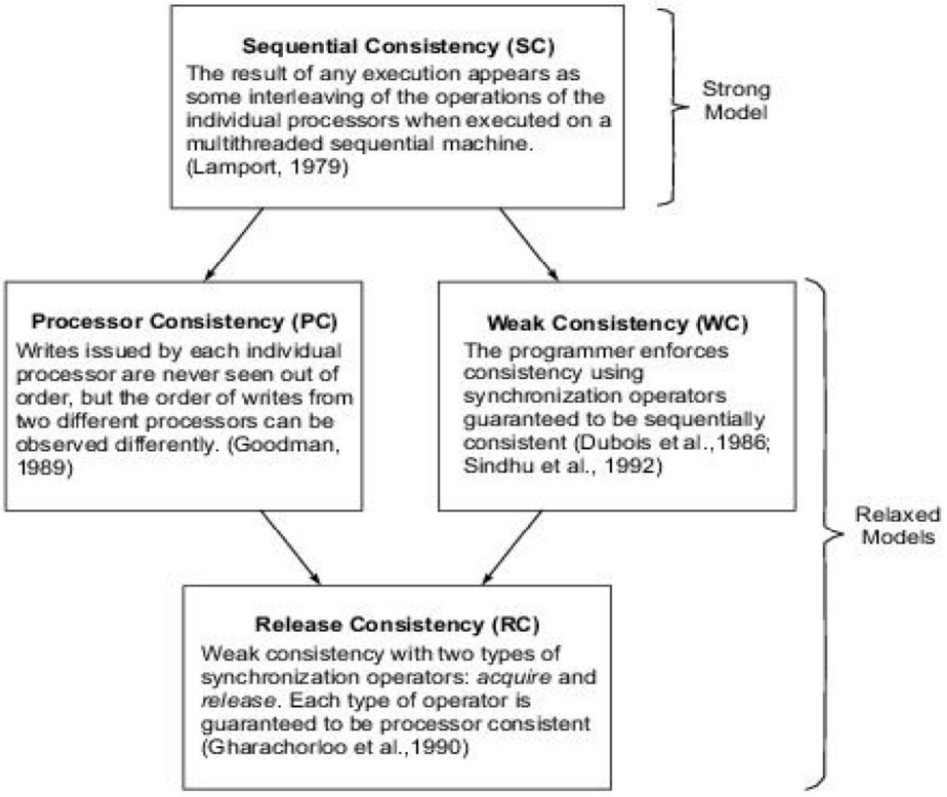
One of the most relaxed memory models is the Release Consistency (RC) model introduced by Gharochorloo et al (1990). Release consistency requires that synchronization accesses in the program be identified and classified as either acquires (e.g. locks) or releases (e.g. unlocks). An acquire is a read operation that gains permission lo access a set of data, while a release is a write operation that gives away such permission. This information is used to provide flexibility in buffering and pipelining of accesses between synchronization points.

The main advantage of the relaxed models is the potential for increased performance by hiding as much write latency as possible. The main disadvantage is increased hardware complexity and a more complex programming model.

Three conditions ensure release consistency:

1) Before an ordinary read or write access is allowed to perform with respect to any other processor, all previous acquire accesses must be performed.
2) Before a release access is allowed to perform with respect to any other processors all previous ordinary read and store accesses must bc performed.
3) special accesses are processor consistcnt with one anothcr.The ordering restrictions imposed by weak consistency are not present in rclease con sistency. lnstead, rclease consistency requires processor consistency and not sequential consistency.

Release consistency can be satisfied by (i) stalling the processor on an acquire access until it completes, and [ii] delaying the completion of release accesses until all previous memory accesses complete. Intuitive definitions of the four memory consistency models, the SC, WC, PC, and RC, are sununarized in Fig. 9.8.



Effect of Release Consistency

Figure 9.9 below presents the breakdown of execution times under SC and RC for the three applications. As can be seen from the results, RC removes all idle time due to write-miss latency.

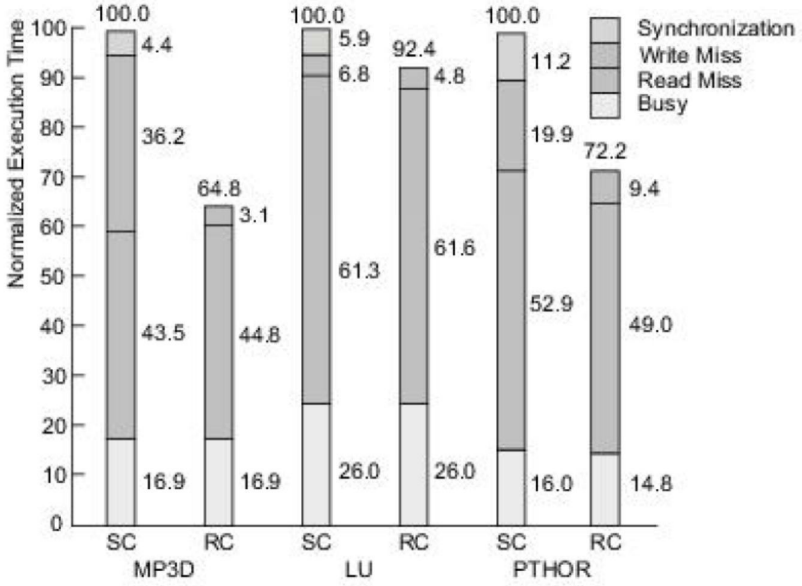


Fig. 9.9 Effect of relaxing the shared-memory model from sequential consistency (SC) to release consistency (RC) (Courtesy of Gupta et al, *Proc. Int. Symp. Comput. Archit.*, Toronto, Canada, May 1991)

Effect of Combining Mechanism

The effect of combining various latency-hiding mechanisms is illustrated by Fig. 9.10 based on the MP3D benchmark results obtained at Stanford University. The busy parts of the execution times in Fig. 9.10 are equal in all combinations. This is the CPU busy time for executing the MPED program. The idle part in the bar diagram corresponds to memory latency and includes all cache-miss penalties. All thc times arc normalized with respect to thc execution time required in a cache-Coherent system. The leftmost time bar (with 241 units) corresponds to the worst case of using a private cache ettclusivcly without shared reads or writes.

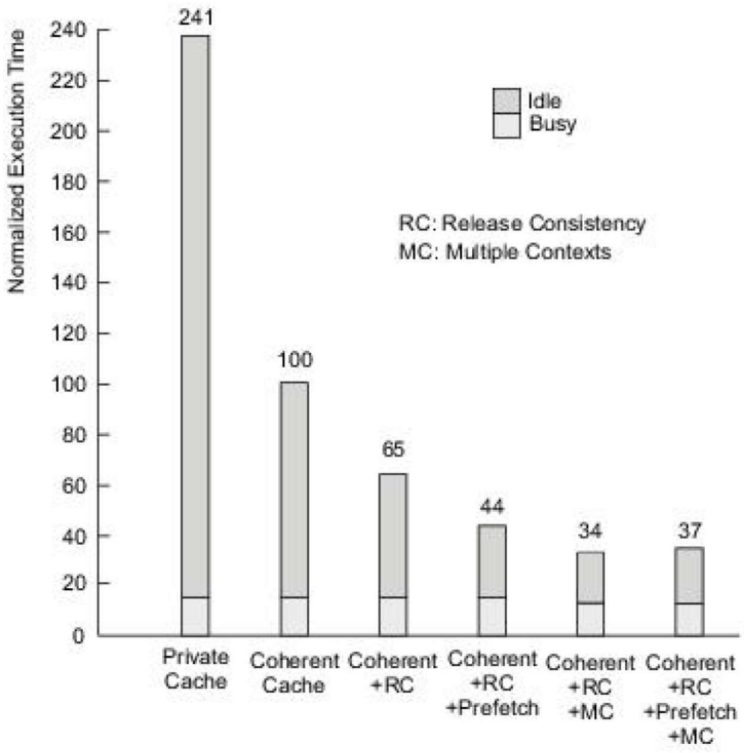
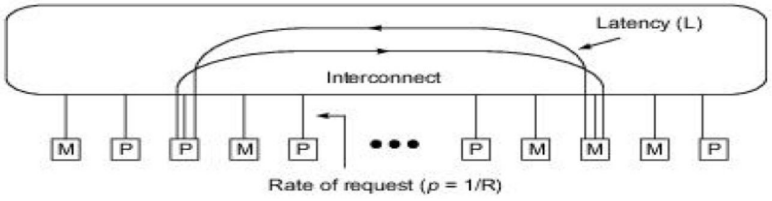


Fig. 9.10 Effect of combining various latency-hiding mechanisms from the MP3D benchmark on a simulated Dash multiprocessor (Courtesy of Gupta, 1992)

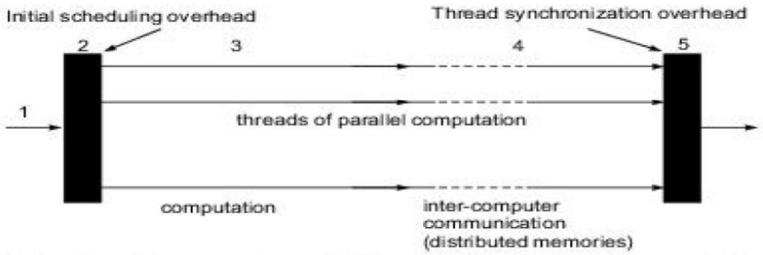
PRINCIPLES OF MULTITHREADING

Multithreading Issues and Solutions

Architecture Environment One possible multithreaded MPP system is modeled by a network of processor (P) and memory (M) nodes as depicted 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:



(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

(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.

Multithreaded Computations

The structure of the multithreaded parallel computations model is shown in Fig. 9.11. 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 a distributed memory (4), and finally by synchronization prior to beginning the next unit of parallel work (5).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 redueed by specialized hardware operating in parallel with computation.

Problems of Asynchronomy

Massively parallel processors operate asynchronously in a network environment. The asynchrony triggers two Fundamental latency problems: remote loads and synchronizing loads, as observed by Nikhil (1992). These two problems are explained by the following example:

Example 9.1 Latency problems for remote loads or synchronizing loads (Rishiyun Nikhil, 1992).

The remote load situation is illustrated in Fig. 9.12a. Variables *A* and B are located on nodes N2 and N3, respectively. They need to be brought to node N1 to compute the difference *A* – *B* in variable *C*. The basic computation demands the execution of two remote loads (rload) and then the subtraction.

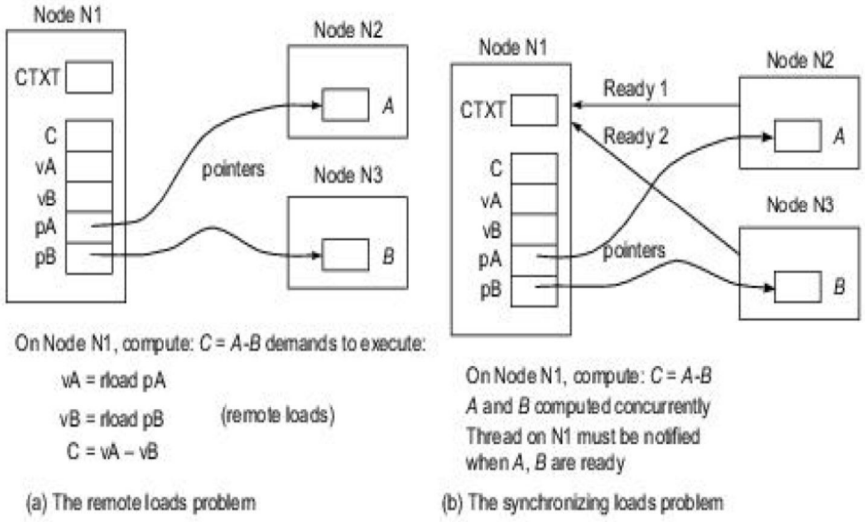


Fig. 9.12 Two common problems caused by asynchrony and communication latency in massively parallel processors (Courtesy of R.S. Nikhil, Digital Equipment Corporation, 1992)

Let pA and pB be the pointers to *A* and *B*, respectively. The two rloads can be issued from the same thread or from two different threads. The *context* of the computation on N1 is represented by the variable CTXT. It can be a stack pointer, a frame pointer, a current-object pointer, a process identifier, etc. In general, variable names like vA, vB, and C are interpreted relative to CTXT.

In Fig. 9.12b, the idling due to synchronizing loads is illustrated. In this case, *A* and *B* are computed by concurrent processes, and we are not sure exactly when they will be ready for node N1 to read. The ready signals (Ready1 and Ready2) may reach node N1 asynchronously. This is a typical situation in the producer-consumer problem. Busy-waiting may result.

Multithreading Solution

The solution to asynchrony problems is to multiplex among many threads. When one thread issues a remote-load request, the processor begins work on another thread, and so on as shown in Fig. 9.13a. Clearly, 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 as shown in Fig. 9.13a, we switch to thread E, which also issues a remote load. Thc 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. These thread identifiers are referred to as continuations on messages, A large continuation name space should be provided to name an adequate number of threads waiting for remote responses.

Distributed Cacheing

The concept of distributed cacheing is shown in Fig. 9.13-b. 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 bold copies) or exclusive (for writes, one cache holds the current value)

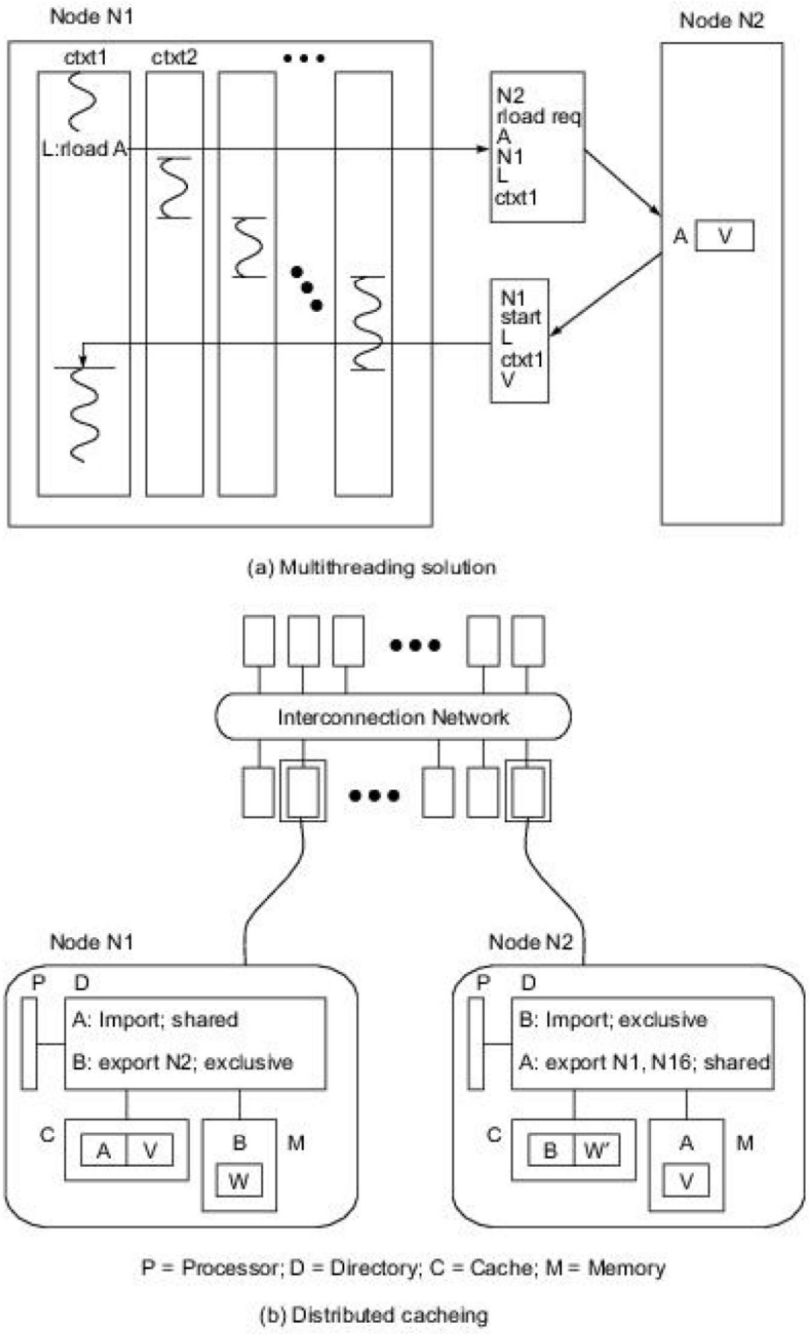
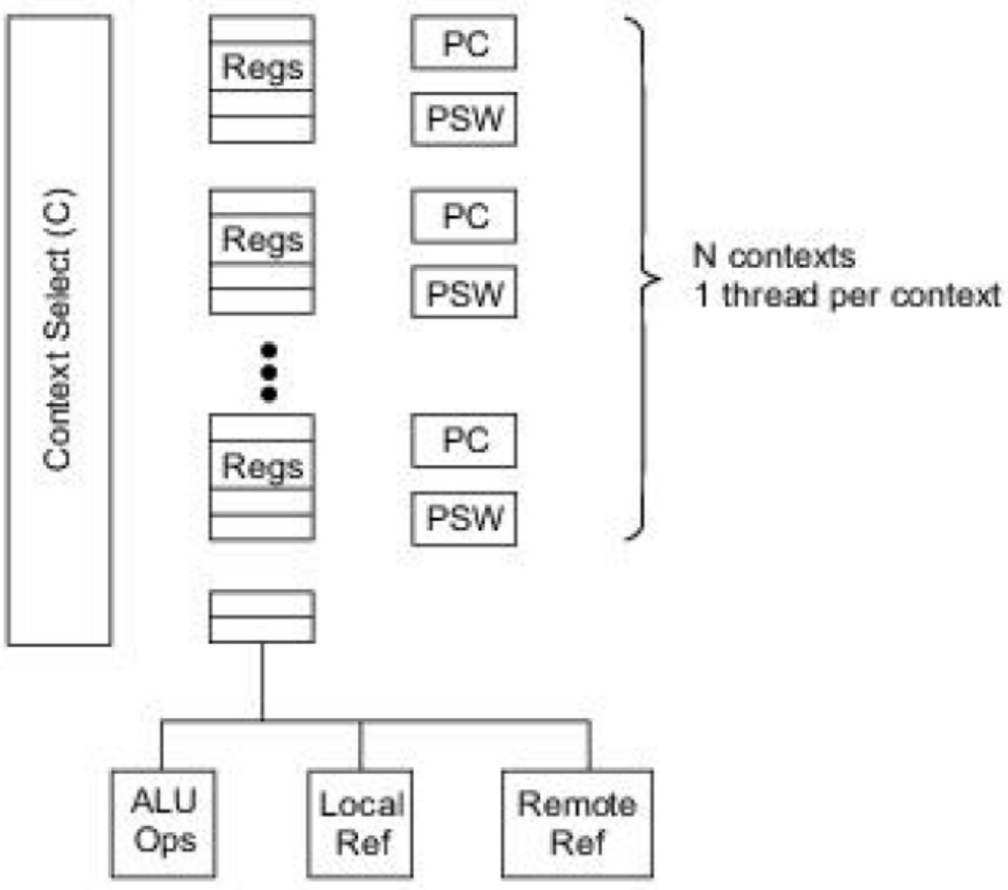


Fig. 9.13 Two solutions for overcoming the asynchrony problems (Courtesy of R. S. Nikhil, Digital Equipment Corporation, 1992)

Multiple-Context: Proccessors

The Enhanced Processor Model

A conventional single-thread processor will wait during a remote reference, so we may say it is idle for a period of time L. A multithreaded processor, as modeled inFig. 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.



(a) Multithreaded model. (Courtesy of Rafael Saavedra, 1992)

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

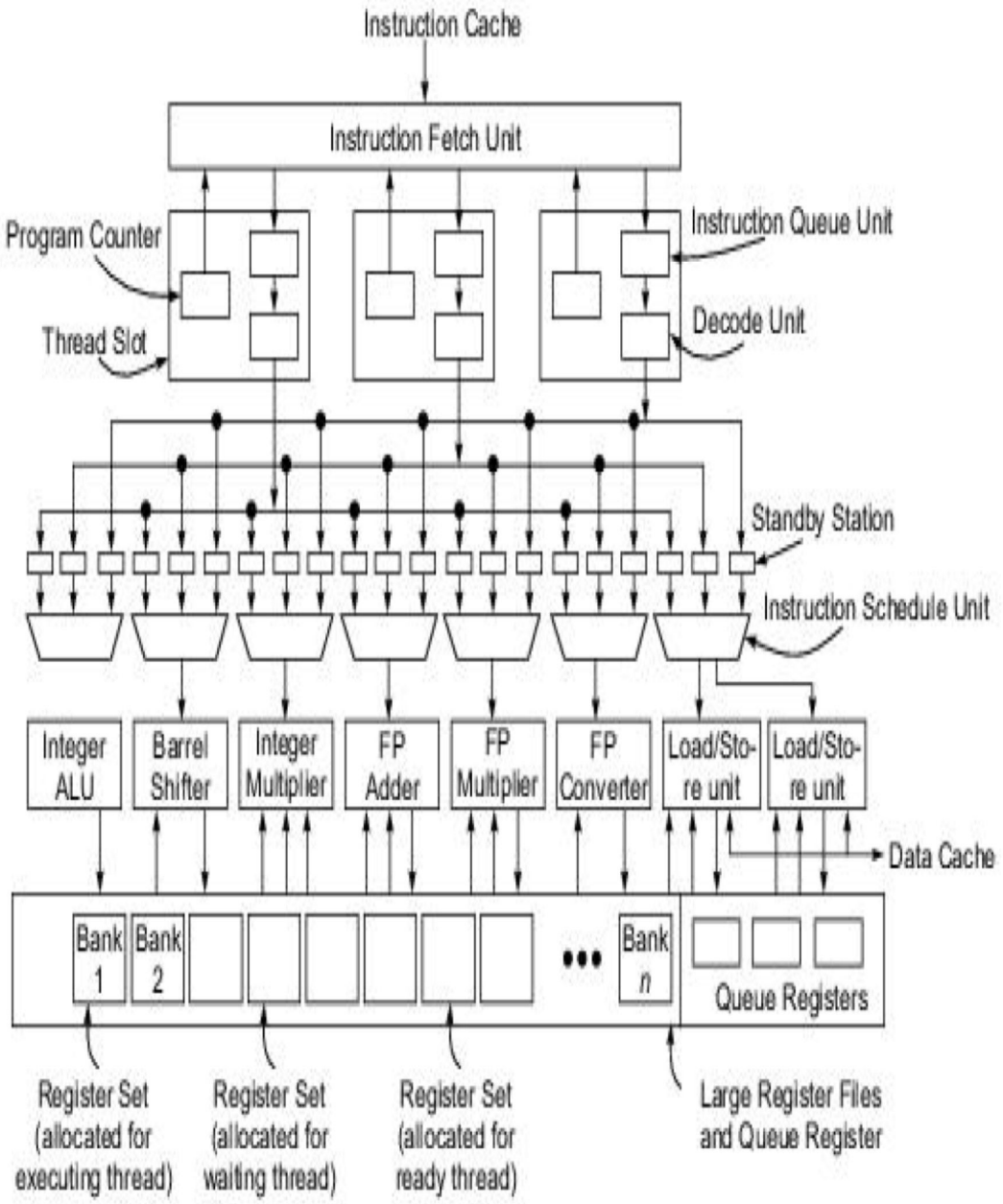
$$Efficiency = \frac{busy}{busy + switching + idle}$$

where busy, switching and idle represent the amount of time, measured over some large interval, that the processor is in the corresponding state. The state of a processor is determined by the disposition of the various contexts on the processor.During its lifetime, a context cycles through the following states: ready,running, leaveing and blocked. There can be at most one context running or leaving. A processor is busy if there is a context in the running state. it is switching while making the transition from one context to another, i.e. when a context is leaving. Otherwise,all contexts are blocked and we say the processor is idle.

Example 9.2 A multithreaded processor with three thread slots (Hiroaki Hirata et al., 1992).

As shown in Fig. 9.14b, the processor is provided with several instruction queue unit and decode unit pairs, called *thread slots*. Each thread slot, associated with a program counter, makes up a *logical processor,* while an instruction fetch unit and all functional units are physically shared among logical processors.

An instruction queue unit has a buffer which saves some instructions succeeding the instruction indicated by the program counter. The buffer size needs to be at least $B = N \times C$ words, where N is the number of thread slots and C is the number of cycles required to access the instruction cache.



(b) A three-thread processor example (Courtesy of H. Hirata et al, *Proc 19th Int. Symp. Comput. Archit.*, Australia, May 1992)

Fig. 9.14 Multiple-context processor model and an example design

Context-Switching Policies Different multithreaded architectures are distinguished by the context-switching policies adopted. Specified below are four switching policies:

(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.

The general model can be employed if it is postulated that there are two sources of latency (L_1 and L_2), each having a particular probability (p_1 and p_2) of occurring on every switch. If L_1 represents the latency on a cache miss, then p_1 corresponds to what is normally referred to as the miss ratio. L_2 is a zero-cycle memory latency with probability p_2.

(3) *Switch on every instruction*—This policy allows switching on every instruction, independent of whether it is a load or not. In other words, it interleaves the instructions from different threads on a cycle-by-cycle basis. Successive instructions become independent, which will benefit pipelined execution. However, the cache miss may increase due to breaking of locality. It has been verified by some trace-driven experiments at Stanford that cycle-by-cycle interleaving of contexts provides a performance advantage over switching on a cache miss in that the context interleaving could hide pipeline dependences and reduce the context switch cost.

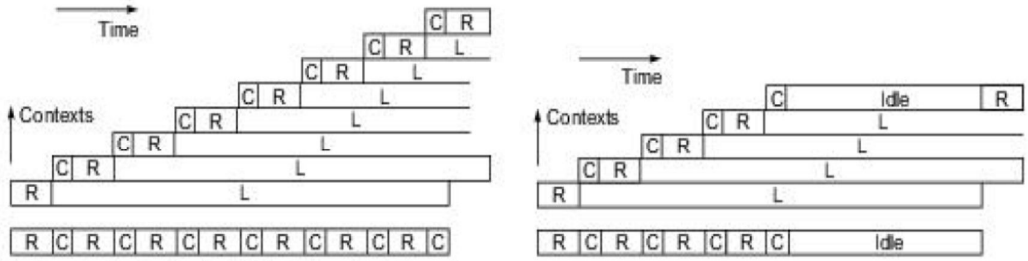
(4) *Switch on block of instruction*—Blocks of instructions from different threads are interleaved. This will improve the cache-hit ratio due to locality. It will also benefit single-context performance.

Processor Efficiencies

A single-thread processor executes a context until a remote reference is issued (R Cycles) and then is idle until the reference completes (L cycles). R and L correspond to the amount of time during a cycle that the processor is busy and idle, respectively. Thus the efficiency of a single-threaded machine is given by

$$E_1 = \frac{R}{R+L} = \frac{1}{1+L/R}$$

With multiple contexts, memory latency can be hidden by switching to a new context, but we assume that the switch takes C cycles of overhead. Assuming the run length between switches is constant with a sufficient number of contexts, there is always a context ready to execute when a switch occurs, so the processor is never idle. The processor efficiency is analyzed below under two different conditions as illustrated in Fig. 9.15.



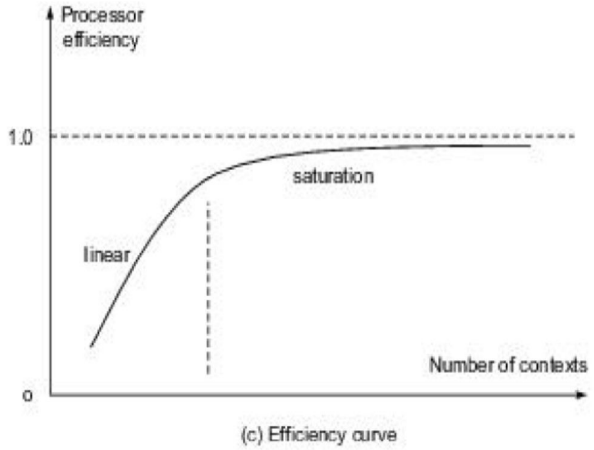


Fig. 9.15 Context switching and processor efficiency as a function of the number of contexts (Courtesy of Rafael Saavedra, 1992)

(1) *Saturation region*—In this saturated region, the processor operates with maximum utilization. The cycle of the renewal process in this case is $R + C$, and the efficiency is simply

$$E_{sat} = \frac{R}{R+C} = \frac{1}{1+C/R} \tag{9.3}$$

(2) *Linear region*—When the number of contexts is below the saturation point, there may be no ready contexts after a context switch, so the processor will experience idle cycles. The time required to switch to a ready context, execute it until a remote reference is issued, and process the reference is equal to $R + C + L$. Assuming N is below the saturation point, during this time all the other contexts have a turn in the processor. Thus, the efficiency is given by

$$E_{lin} = \frac{NR}{R+C+L} \tag{9.5}$$

Figures 9.15a and 9.15b show snapshots of context switching in the saturation and linear regions, respectively. The processor efficiency is plotted as a function of the number of contexts in Fig. 9.15c.

In Fig. 9.16, the processor efficiency is plotted as a function of the memory latency L with an average run length $R = 16$ cycles. The $C = 0$ curve corresponds to zero switching overhead. With $C = 16$ cycles, about 50% efficiency can be achieved. These results are based on a Markov model of multithreaded architecture

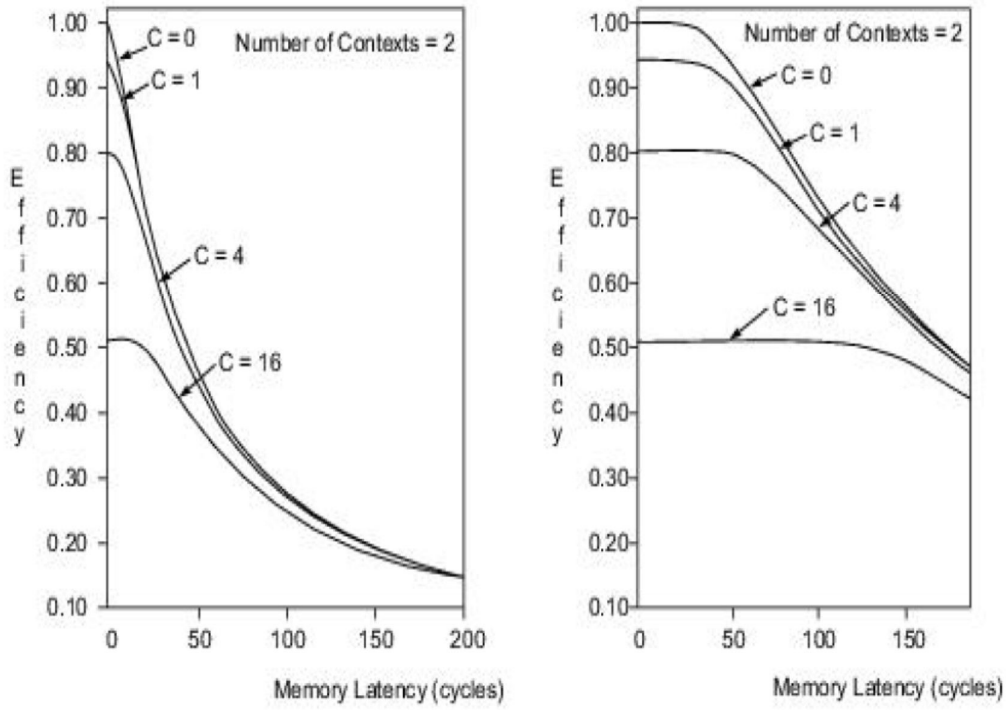


Fig. 9.16 Processor efficiency of a multithreaded architecture (Courtesy of R. Saavedra, D. E. Culler, and T. von Eicken, 1992)

Multidimensional Architectures

The architecture of massively parallel processors has evolved from one-dimensional *rings* to two-dimensional and three-dimensional meshes or tori as illustrated in Fig. 9.17. The Maryland Zmob experimented on a *slotted token ring* for building a multiprocessor. Both the CDC Cyberplus and KSR-1 used hierarchical (two-level) ring architectures. The ring is the simplest architecture to implement from the viewpoint of backplane packaging.

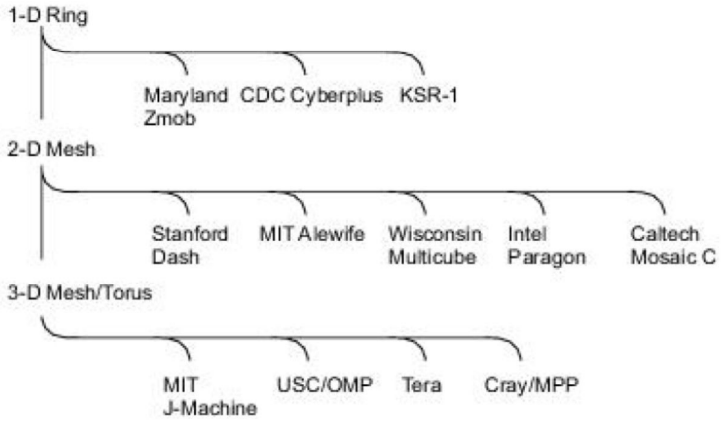


Fig. 9.17 The evolution from one-dimensional ring to two-dimensional mesh and then to three-dimensional mesh/torus architecture for building massively parallel processors.

FINE-GRAIN MULTICOMPUTERS

Fine-Grain Parallelism

Latency Analysis The computing granularity and communication latency of leading early examples of multiprocessors, data-parallel computers, and medium-and fine-grain multicomputers are summarized in Table 9.3. These table entries summarize what we have learned in Chapters 7 and 8. Four attributes are identified to characterize these machines. Only typical values for a typical program mix are shown. The intention is to show the order of magnitude in these entries.

The *communication latency* T_c measures the data or message transfer time on a system interconnect. This corresponds to the shared-memory access time on the Cray Y-MP, the time required to send a 32-bit value across the hypercube network in the CM-2, and the network latency on the iPSC/1 or J-Machine. The *synchronization overhead* T_s is the processing time required on a processor, or on a PE, or on a processing node of a multicomputer for the purpose of synchronization.

The sum $T_c + T_s$ gives the total time required for IPC. The shared-memory Cray Y-MP had a short T_c but a long T_s. The SIMD machine CM-2 had a short T_s but a long T_c. The long latency of the iPSC/1 made it unattractive based on fast advancing standards. The MIT J-Machine was designed to make a major improvement in both of these communication delays.

Table 9.3 *Fine-Grain, Medium-Grain, and Coarse-Grain Machine Characteristics of Some Example Systems*

Characteristics	*Machine*			
	Cray Y-MP	*Connection Machine CM-2*	*Intel iPSC/1*	*MIT J-Machine*
Communication latency, T_c	40 ns via shared memory	600 μs per 32-bit *send* operation	5 ms	2 μs
Synchronization overhead, T_s	20 μs	125 ns per bit-slice operation in lock step	500 μs	1 μs
Grain size, T_g	20 s	4 μs per 32-bit result per PE instruction	10 ms	5 μs
Concurrency (DOP)	2–16	8K–64K	8–128	1K–64K
Remark	Coarse-grain supercomputer	Fine-grain data parallelism	Medium-grain multicomputer	Fine-grain multicomputer

9.3.2 The MIT J-Machine

The architecture and building block of the MIT J-Machine, its instruction set, and system design considerations are described below based on the paper by Dally et al (1992). The building block was the *message-driven processor* (MDP), a 36-bit microprocessor custom-designed for a fine-grain multicomputer.

The J-Machine Architecture The *k*-ary *n-cube* networks were applied in the MIT J-Machine. The initial prototype J-Machine used a 1024-node network ($8 \times 8 \times 16$), which was a reduced 16-ary 3-cube with 8 nodes along the *x*- and *y*-dimensions and 16 nodes along the *z*-dimension. A 4096-node J-Machine would use a full 16-ary 3-cube with $16 \times 16 \times 16$ nodes. The J-Machine designers called their network a three-dimensional mesh.

The MDP Design The MDP chip included a processor, a 4096-word by 36-bit memory, and a built-in rc with network ports as shown in Fig. 9.19. An on-chip memory controller with error checking and correc (ECC) capability permitted local memory to be expanded to 1 million words by adding external DR chips. The processor was message-driven in the sense that it executed functions in response to messages the dispatch mechanism. No receive instruction was needed.

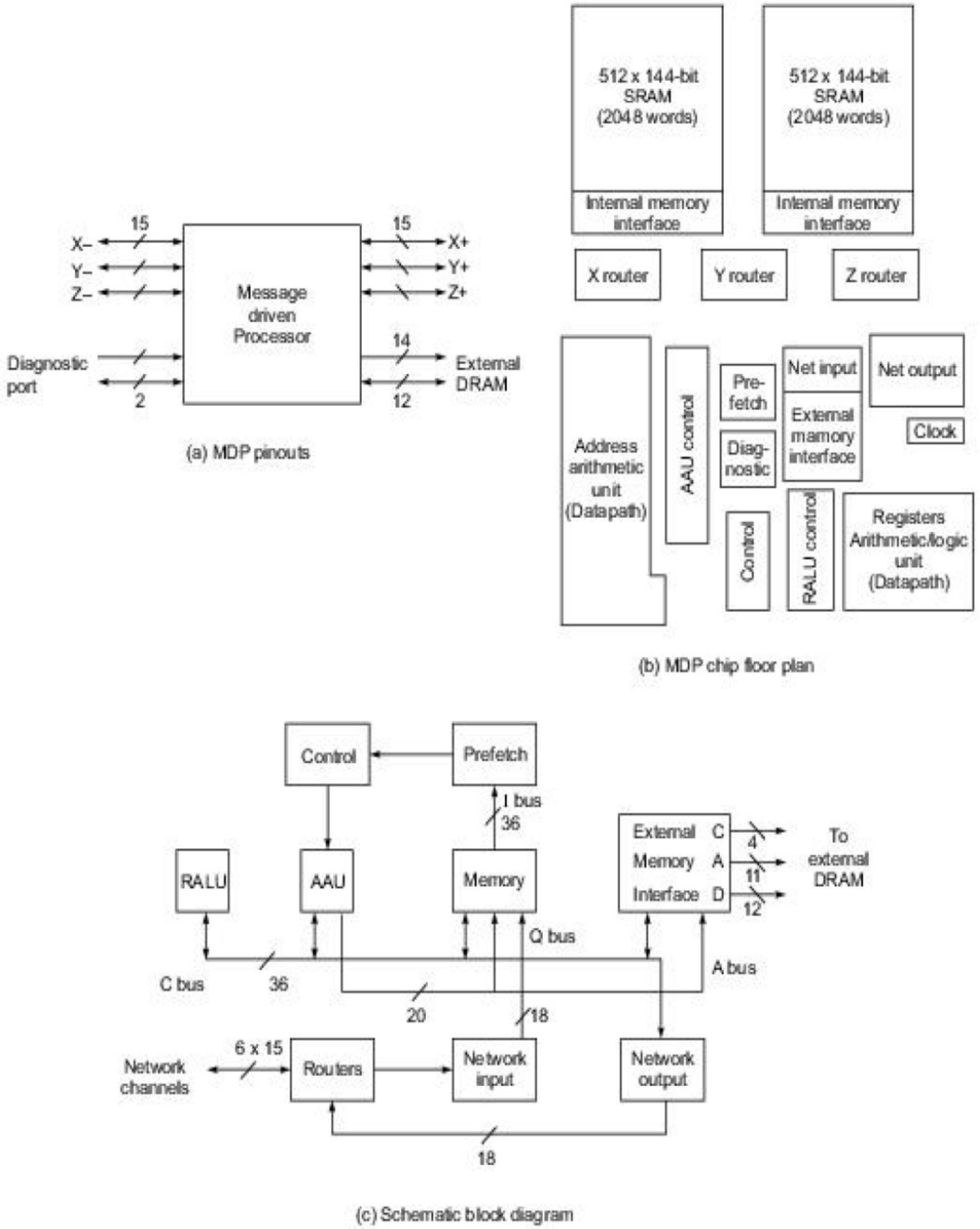


Fig. 9.19 The message-driven processor (MDP) architecture (Courtesy of W. Dally et al; reprinted with permission from *IEEE Micro*, April 1992)

Instruction-Set Architecture The MDP extended a conventional microprocessor instruction-set architecture with instructions to support parallel processing. The instruction set contained fixed-format, three-address instructions. Two 17-bit instructions fit into each 36-bit word with 2 bits reserved for type checking.

Communication Support The MDP provided hardware support for end-to-end message delivery including formatting, injection, delivery, buffer allocation, buffering, and task scheduling. An MDP transmitted a message using a series of SEND instructions, each of which injected one or two words into the network at either priority 0 or 1.

Consider the following MDP assembly code for sending a four-word message using three variants of the SEND instruction.

```
SEND     R0,0       ;   send net address (priority 0)
SEND2    R1,R2,0    ;   header and receiver (priority 0)
SEND2E   R3,[3,A3],0 ;  selector and continuation end message (priority 0)
```

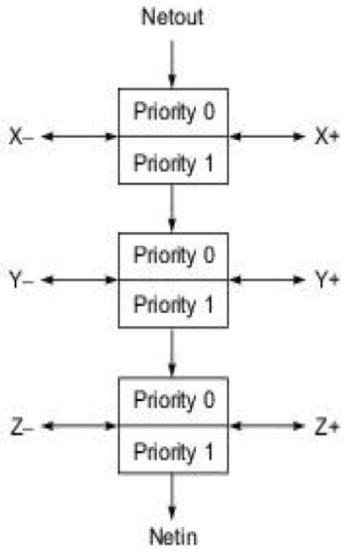
The first SEND instruction reads the absolute address of the destination node in $< X, Y, Z >$ format from R0 and forwards it to the network hardware. The SEND2 instruction reads the first two words of the message out of registers R1 and R2 and enqueues them for transmission. The final instruction enqueues two additional words of data, one from R3 and one from memory. The use of the SEND2E instruction marks the end of the message and causes it to be transmitted into the network.

Example 9.3 A typical message in the MIT J-Machine (W. Dally et al, 1992)

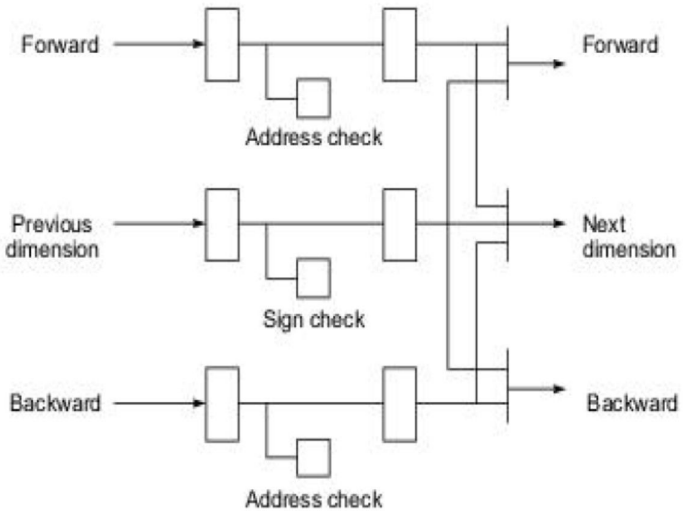
The following message consists of nine flits. The first three flits of the message contain the x-, y-, and z-addresses. Each node along the path compares the address in the head flit of the message. If the two indices match, the node routes the rest to the next dimension. The final flit in the message is marked as the tail.

Flit	Contents		Remarks
1	5:+		x-address
2	1:–		y-address
3	4:+		z-address
4	Msg: 00		Method to call
5	00440		
6	INT: 00		Argument to method
7	0023		
8	INT: 0 0		Reply address
9	< 1 : 5 : 2 >	T	

The Router Design The routers formed the switches in a J-Machine network and delivered messages to their destinations. As shown in Fig. 9.21a, the MDP contained three independent routers, one for each bidirectional dimension of the network.



(a) Dual-priority levels per dimension in the router



(a) Each priority with forward, reverse, and previous data paths to the next dimension.

Fig. 9.21 Priority control and dimension-order router design in the MDP chip (Courtesy of W. Dally et al; reprinted with permission from *IEEE Micro*, April 1992)

Synchronization The MDP synchronized using message dispatch and presence tags on all states. Because each message arrival dispatched a process, messages could signal events on remote nodes. For example, in the following combining-tree example, each COMBINE message signals its own arrival and initiates the COMBINE routine.

Example 9.4 Using a combining tree for synchronization of events (W. Dally et al, 1992)

A combining tree is shown in Fig. 9.22. This tree sums results produced by a distributed computation. Each node sums the input values as they arrive and then passes a result message to its parent.

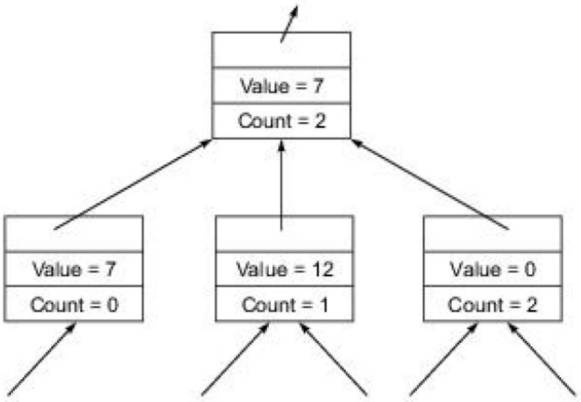


Fig. 9.22 A combining tree for internode communication or synchronization (Courtesy of W. Dally et al, 1992)

A pair of SEND instructions was used to send the COMBINE message to a node. Upon message arrival, the MDP buffered the message and created a task to execute the following COMBINE routine written in MDP assembly code:

```
COMBINE:  MOVE    [1, A3], COMB              ;  get node pointer from message
          MOVE    [2, A3], R1                ;  get value from message
          ADD     R1, COMB.VALUE, R1
          MOVE    R1, COMB.VALUE             ;  store result
          MOVE    COMB.COUNT, R2             ;  get Count
          ADD     R2, -1, R2
          MOVE    R2, COMB.COUNT             ;  store decremented Count
          BNZ     R2, DONE
          MOVE    HEADER, R0                 ;  get message header
          SEND2   COMB.PARENT_NODE, R0       ;  send message to parent
          SEND2E  COMB.PARENT, R1            ;  with value
DONE:     SUSPEND
```

If the node was idle, execution of this routine began three cycles after message arrival. The routine loaded the combining-node pointer and value from the message, performed the required add and decrement, and, if Count reached zero, sent a message to its parent.

9.3.3 The Caltech Mosaic C

From Cosmic Cube to Mosaic C The evolution from the Cosmic Cube to the Mosaic is an example of one type of *scaling track* in which advances in technology are employed to reimplement nodes of a similar logical complexity but which are faster and smaller, have lower power, and are less expensive. The progress in microelectronics over the preceding decade was such that Mosaic nodes were = 60 times faster, used = 20 times less power, were = 100 times smaller, and were (in constant dollars) = 25 times less expensive to manufacture than Cosmic Cube nodes.

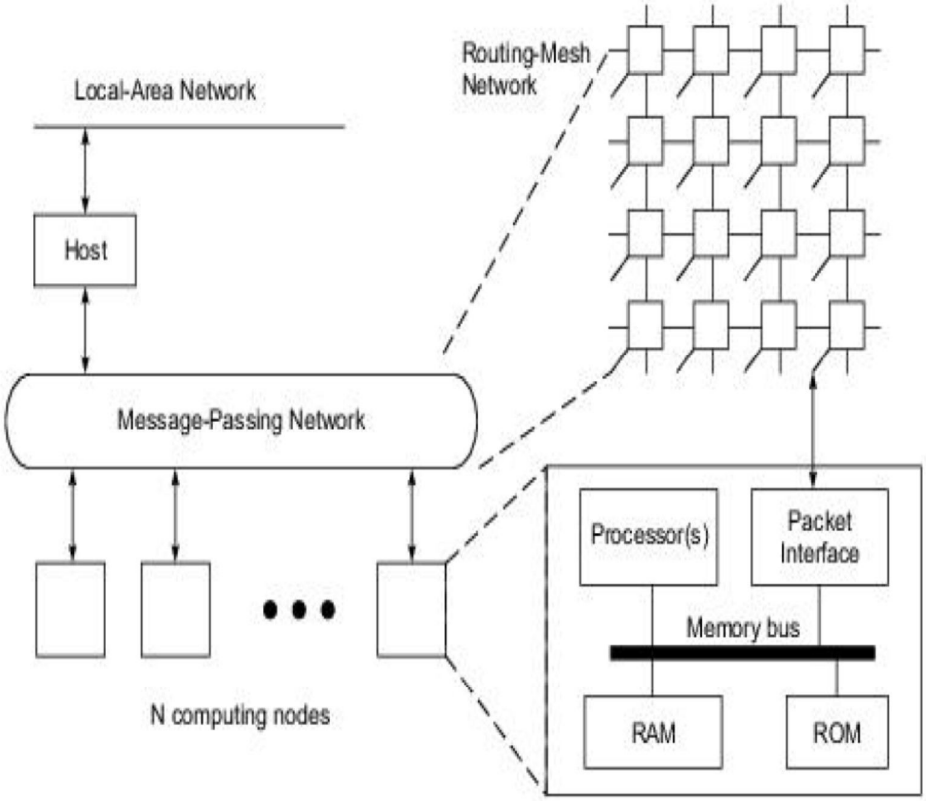


Fig. 9.23 The Caltech Mosaic architecture (Courtesy of C. Seitz, 1992)

Each Mosaic node included 64 Mbytes of memory and an 11-MIPS processor, a packet interface, and a router. The nodes were tied together with a 60-Mbytes/s, two-dimensional routing-mesh network (Fig. 9.23).

9.4 SCALABLE AND MULTITHREADED ARCHITECTURES

9.4.1 The Stanford Dash Multiprocessor

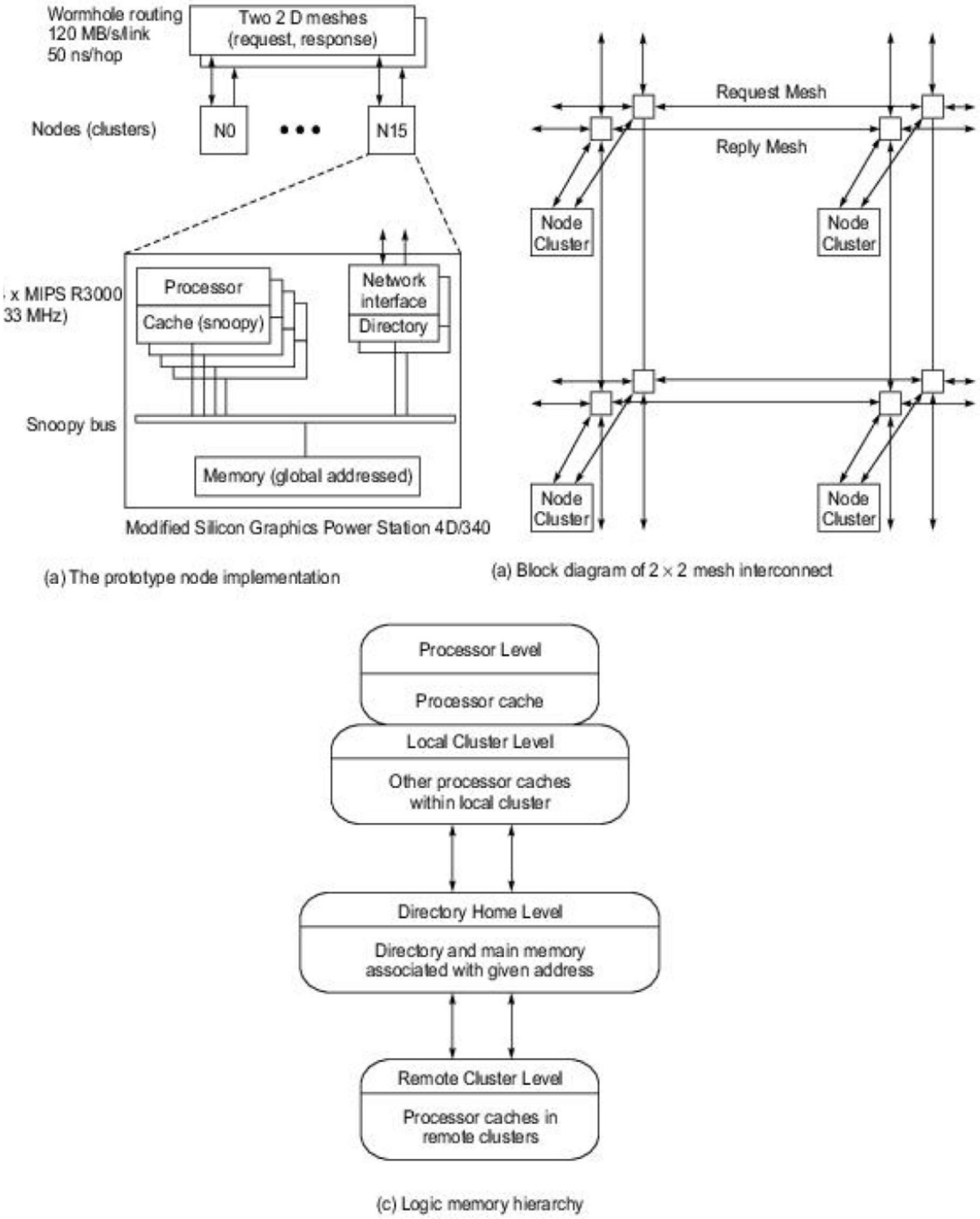


Fig. 9.24 The Stanford Dash prototype system (Courtesy of D. Lenoski et al, *Proc. 19th Int. Symp. Comput. Archit.*, Australia, May 1992)

The Prototype Architecture A high-level organization of the Dash architecture was illustrated in Fig. 9.1 when we studied the various latency-hiding techniques. The Dash prototype is illustrated in Fig. 9.24. It incorporated up to 64 MIPS R3000/R3010 microprocessors with 16 clusters of 4 PEs each. The cluster hardware was modified from Silicon Graphics 4D/340 nodes with new directory and reply controller boards as depicted in Fig. 9.24a.

The interconnection network among the 16 multiprocessor clusters was a pair of wormhole-routed mesh networks. The channel width was 16 bits with a 50-ns fall-through time and a 35-ns cycle time. One mesh network was used to *request* remote memory, and the other was a *reply* mesh as depicted in Fig. 9.24b, where the small squares at mesh intersections are the 5×5 mesh routers.

Dash Memory Hierarchy Dash implemented an invalidation-based cache coherence protocol. A memory location could be in one of three states:

- *Uncached*—not cached by any cluster;
- *Shared*—in an unmodified state in the caches of one or more clusters; or
- *Dirty*—modified in a single cache of some cluster.

The directory kept the summary information for each memory block, specifying its state and the clusters cacheing it. The Dash memory system could be logically broken into four levels of hierarchy, as illustrated in Fig. 9.25c.

The Directory Protocol The directory memory relieved the processor caches of snooping on memory requests by keeping track of which caches held each memory block. In the home node, there was a directory entry per block frame. Each entry contained one *presence bit* per processor cache. In addition, a *state bit* indicated whether the block was uncached, shared in multiple caches, or held exclusively by one cache (i.e. whether the block was dirty).

Using the state and presence bits, the memory could tell which caches needed to be invalidated when a location was written. Likewise, the directory indicated whether the memory copy of the block was up-to-date or which cache held the most recent copy.

Example 9.5 Cache coherence protocol using distributed directories in the Dash multiprocessor (Daniel Lenoski and John Hennessy et al, 1992.)

Figure 9.25a illustrates the flow of a read request to remote memory with the directory in a dirty remote state. The read request is forwarded to the owning dirty cluster. The owning cluster sends out two messages in response to the read. A message containing the data is sent directly to the requesting cluster, and a sharing writeback request is sent to the home cluster. The sharing writeback request writes the cache block back to memory and also updates the directory.

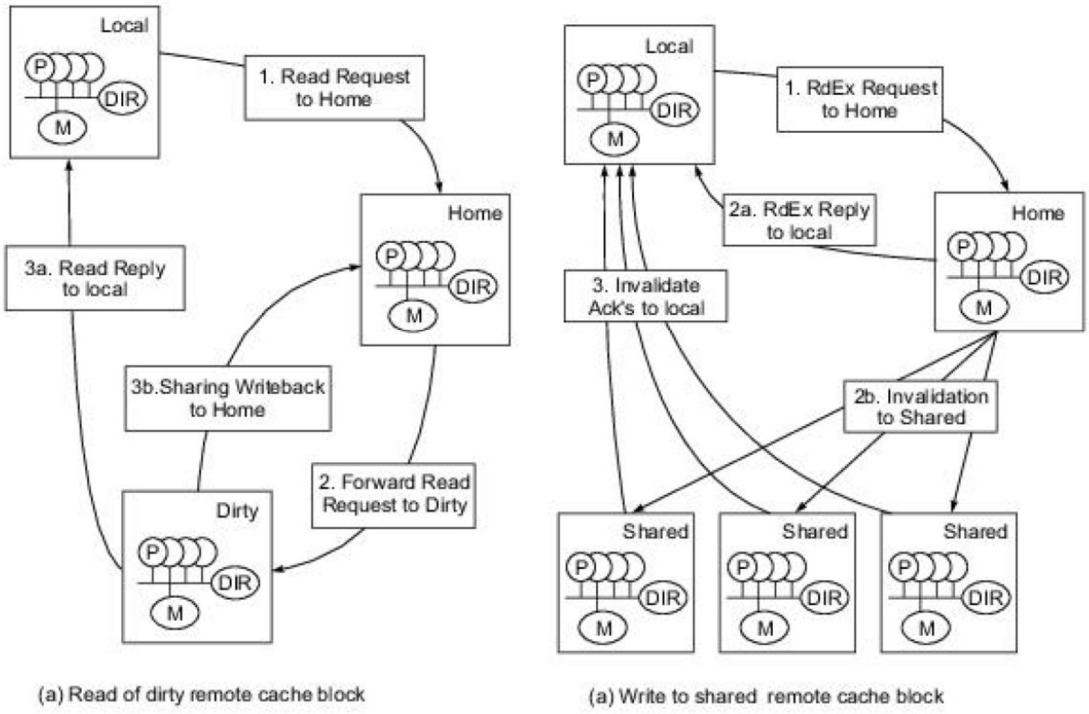


Fig. 9.25 Two examples of a directory-based cache coherence protocol in the Dash (Courtesy of Lenoski and Hennessy, 1992)

This protocol reduces latency by permitting the dirty cluster to respond directly to the requesting cluster. In addition, this forwarding strategy allows the directory controller to simultaneously process many requests (i.e. to be multithreaded) without the added complexity of maintaining the state of outstanding requests. Serialization is reduced to the time of a single intercluster bus transaction. The only resource held while intercluster messages are being sent is a single entry in the originating cluster's remote-access cache.