Part III

Parallel and Scalable Architectures

Chapter 7
Multiprocessors and Multicomputers

Chapter 8
Multivector and SIMD Computers

Chapter 9
Scalable, Multithreaded, and Dataflow Architectures

Summary

Part III consists of three chapters dealing with parallel, vector, and scalable architectures for building high-performance computers. The multiprocessor system interconnects studied include crossbar switches, multistage networks, hierarchical buses, and multidimensional ring, mesh, and torus architectures. Three generations of multicomputer developments are reviewed. Then we consider message-passing mechanisms.

Vector supercomputers appear either as pipelined multiprocessors or as SIMD data-parallel computers. We study the architectures of the Cray Y-MP, C-90, Cray/MPP, NEC SX, Fujitsu VP-2000, VPP500, VAX 9000, Hitachi S-820, Stardent 3000, CM-2, MasPar MP-1, and CM-5 for concurrent scalar/vector processing.

Chapter 9 introduces scalable architectures for massively parallel processing applications. These include both von Neumann, fine-grain, multithreaded, and dataflow architectures. Various latency-hiding techniques are described, including the principles of multithreading. Case studies include the Intel Paragon, Stanford Dash, MIT Alewife, J-Machine and *T, Tera computer, KSR-I, Wisconsin Multicube, USC/OMP, ETL EM4, etc.

7

Multiprocessors and Multicomputers

In this chapter, we study system architectures of multiprocessors and multicomputers. Various cache coherence protocols, synchronization methods, crossbar switches, multiport memory, and multistage networks are described for building multiprocessor systems. Then we discuss multicomputers with distributed memories which are not globally shared. The Intel Paragon is used as a case study. Message-passing mechanisms required with multicomputers are also reviewed. Single-address-space multicomputers will be studied in Chapter 9.

7.1 MULTIPROCESSOR SYSTEM INTERCONNECTS

Parallel processing demands the use of efficient system interconnects for fast communication among multiple processors and shared memory, I/O, and peripheral devices. Hierarchical buses, crossbar switches, and multistage networks are often used for this purpose.

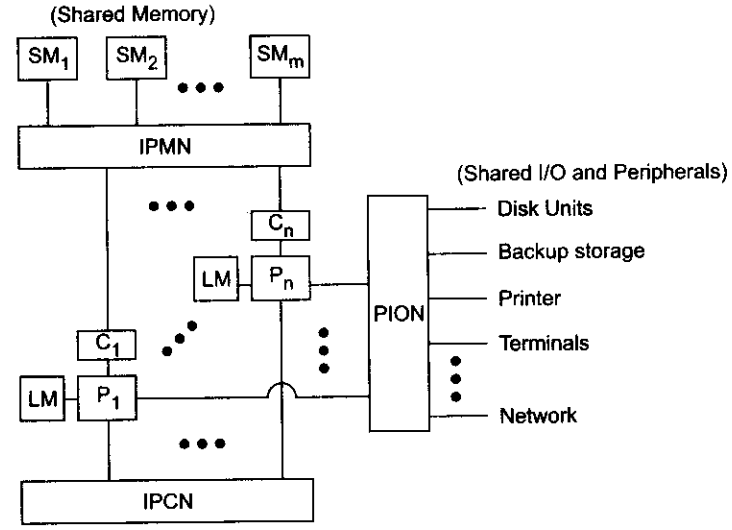
A generalized multiprocessor system is depicted in Fig. 7.1. This architecture combines features from the UMA, NUMA, and COMA models introduced in Section 1.4.1. Each processor P_i is attached to its own local memory and private cache. Multiple processors are connected to shared-memory modules through 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.

Network Characteristics Each of the above types of networks can be designed with many choices. The choices are based on the topology, timing protocol, switching method, and control strategy. Dynamic networks are used in multiprocessors in which the interconnections are under program control. Timing, switching, and control are three major operational characteristics of an interconnection network. 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.



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

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

Network control strategy is classified as *centralized* or *distributed*. With centralized 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.

7.1.1 Hierarchical Bus Systems

A *bus system* consists of a hierarchy of buses connecting various system and subsystem components in a computer. Each bus is formed with a number of signal, control, and power lines. Different buses are used to perform different interconnection functions.

In general, the hierarchy of bus systems are packaged at different levels as depicted in Fig. 7.2, including local buses on boards, backplane buses, and I/O buses.

Local Bus Buses implemented within processor chips or on *printed-circuit* boards are called *local buses*. On a processor board one may find a local bus which provides a common communication path among major components (chips) mounted on the board. A memory board uses a *memory bus* to connect the memory with

the interface logic. An I/O or network interface chip or board uses a *data bus*. Each of these local buses consists of signal and utility lines.

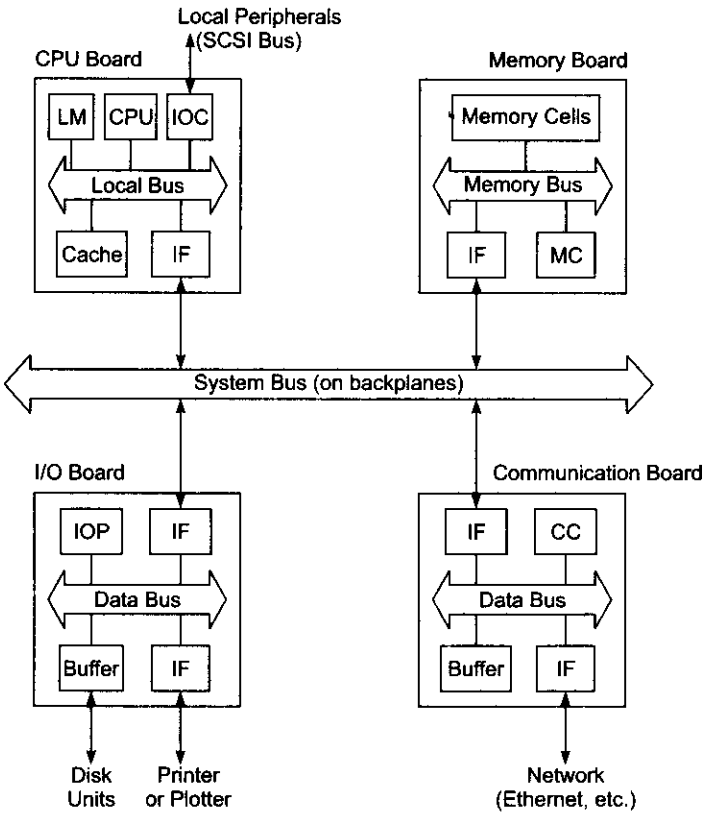


Fig. 7.2 Bus systems at board level, backplane level, and I/O level

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

Several backplane bus standards have been developed over time such as the VME bus (IEEE Standard 1014-1987), Multibus II (IEEE Standard 1296-1987), and Futurebus+ (IEEE Standard 896.1-1991) as introduced in Chapter 5. However, point to-point switched interconnects have emerged as more efficient alternatives, as discussed in Chapters 5 and 13.

I/O Bus Input/output devices are connected to a computer system through an *I/O bus* such as the SCSI (Small Computer Systems Interface) bus. This bus is made of coaxial cables with taps connecting disks,

printer, and other devices to a processor through an I/O controller (Fig. 7.2). Special interface logic is used to connect various board types to the backplane bus.

Complete specifications for a bus system include logical, electrical, and mechanical properties, various application profiles, and interface requirements. Our study will be confined to the logical and application aspects of system buses. Emphasis will be placed on the scalability and bus support for cache coherence and fast synchronization.

For example, the core of the Encore Multimax multiprocessor was the Nanobus, consisting of 20 slots, a 32-bit address, a 64-bit data path, and a 14-bit vector bus, and operating at a clock rate of 12.5 MHz with a total memory bandwidth of 100 Mbytes/s. The Sequent multiprocessor bus had a 64-bit data path, a 10-MHz clock rate, and a 32-bit address, for a channel bandwidth of 80 Mbytes/s. A write-back private cache was used to reduce the bus traffic by 50%.

Digital bus interconnects can be adopted in commercial systems ranging from workstations to minicomputers, mainframes, and multiprocessors. Hierarchical bus systems can be used to build medium-sized multiprocessors with less than 100 processors. However, the bus approach is limited by bandwidth scalability and the packaging technology employed.

Hierarchical Buses and Caches Wilson (1987) proposed a hierarchical cache/ bus architecture as shown in Fig. 7.3. This is a multilevel tree structure in which the leaf nodes are processors and their private caches (denoted P_j and C_{1j} in Fig. 7.3). These are divided into several clusters, each of which is connected through a cluster bus.

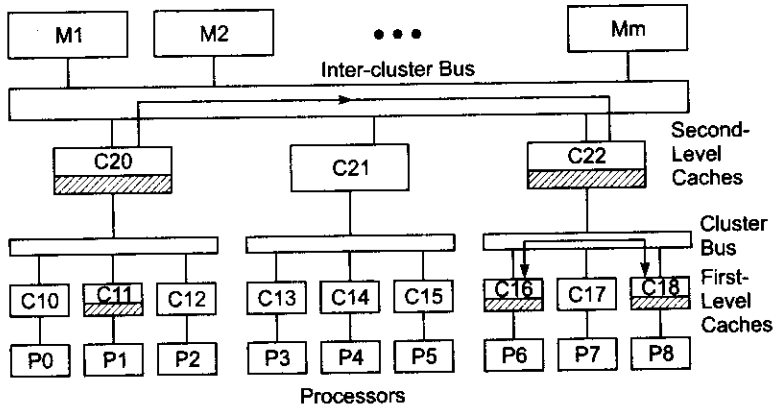


Fig. 7.3 A hierarchical cache/bus architecture for designing a scalable multiprocessor (Courtesy of Wilson; reprinted from *Proc. of Annual Int. Symp. on Computer Architecture, 1987*)

An intercluster bus is used to provide communications among the clusters. Second level caches (denoted as C_{2i}) are used between each cluster bus and the intercluster bus. Each second-level cache must have a capacity that is at least an order of magnitude larger than the sum of the capacities of all first-level caches connected beneath it.

Each single cluster operates as a single-bus system. Snoopy bus coherence protocols can be used to establish consistency among first-level caches belonging to the same cluster. Second-level caches are used to extend consistency from each local cluster to the upper level.

The upper-level caches form another level of shared memory between each cluster and the main memory modules connected to the intercluster bus. Most memory requests should be satisfied at the lower-level caches. Intercluster cache coherence is controlled among the second-level caches and the resulting effects are passed to the lower level.

Example 7.1 Encore Ultramax multiprocessor architecture

The Ultramax had a two-level hierarchical-bus architecture as depicted in Fig. 7.4. The Ultramax architecture was very similar to that characterized by Wilson, except that the global Nanobus was used only for intercluster communications.

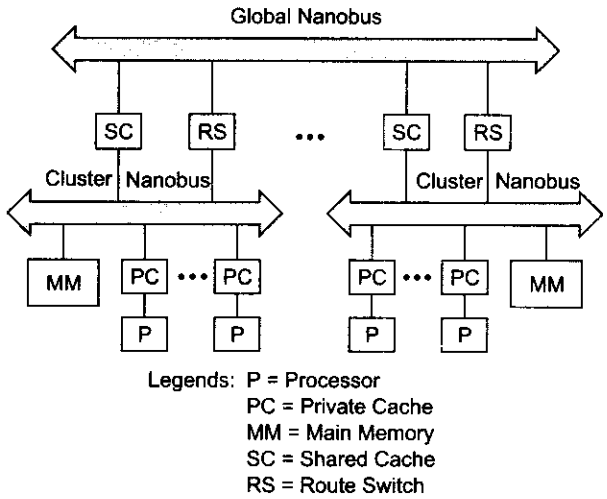


Fig. 7.4 The Ultramax multiprocessor architecture using hierarchical buses with multiple clusters (Courtesy of Encore Computer Corporation, 1987)

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 access 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. As exemplified in Fig. 7.5, multiple buses are used to build a very large system consisting of three multiprocessor clusters. The bus used in this example is Futurebus+, but the basic idea is more general. Bridges are used to interface the clusters. The main functions of a bridge include communication protocol conversion, interrupt handling in split transactions, and serving as cache and memory agents.

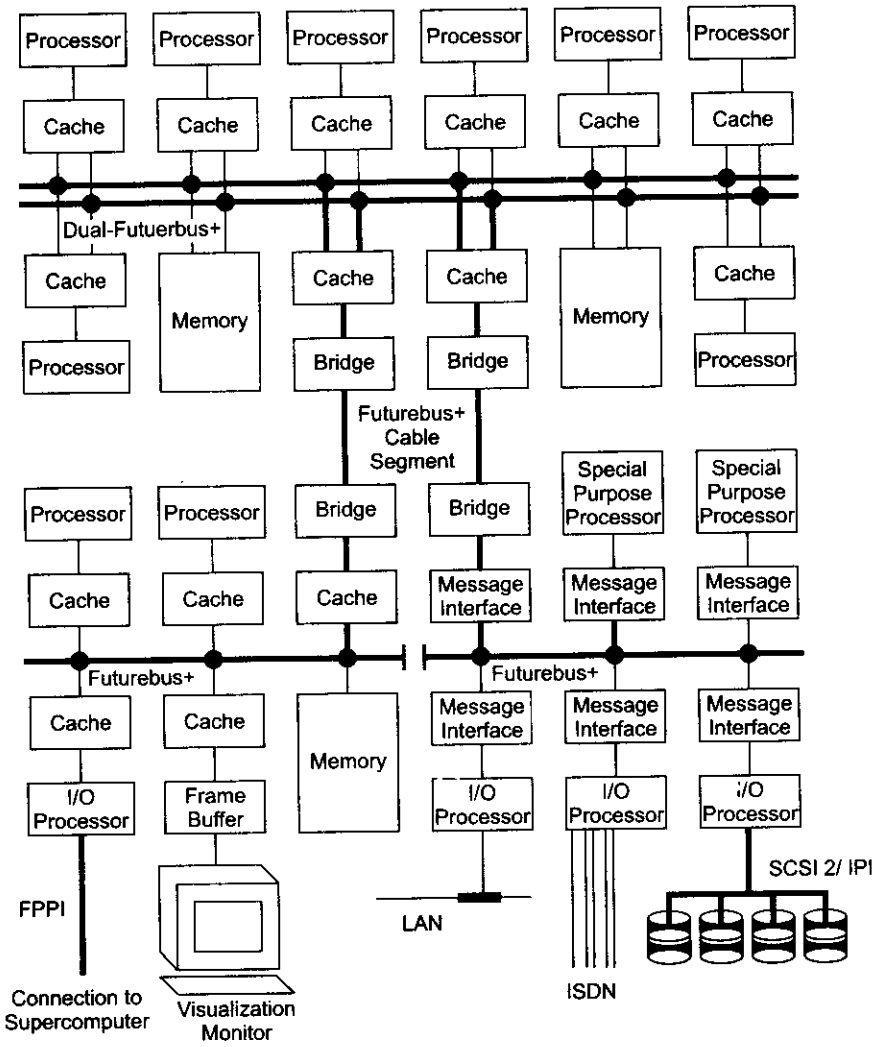


Fig. 7.5 A multiprocessor system using multiple Futurebus+ segments (Reprinted with permission from IEEE Standard 896.1-1991, copyright © 1991 by IEEE, Inc.)

7.1.2 Crossbar Switch and Multiport Memory

Switched networks provide dynamic interconnections between the inputs and outputs. Major classes of switched networks are specified below, based on the number of stages and blocking or nonblocking. We describe the crossbar networks and multiport memory structures first and then the multistage networks. Crossbar networks are mostly used in small or medium-size systems. The multistage networks can be extended to larger systems if the increased latency problem can be suitably addressed.

Network Stages Depending on the interstage connections used, a *single-stage network* is also called a *recirculating network* because data items may have to recirculate through the single stage many times before

reaching their destination. A single-stage network is cheaper to build, but multiple passes may be needed to establish certain connections. The crossbar switch and multiport memory organization are both single-stage networks.

A multistage network consists of more than one stage of switch boxes. Such a network should be able to connect from any input to any output. We will study unidirectional multistage networks in Section 7.1.3. The choice of interstage connection patterns determines the network connectivity. These patterns may be the same or different at different stages, depending the class of networks to be designed. 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 (Lawrie, 1975), Baseline (Wu and Feng, 1980), Banyan (Goke and Lipovski, 1973), and Delta networks (Patel, 1979). Some blocking networks are equivalent after graph transformations. In fact, most multistage networks are blocking in nature. In a blocking network, multiple passes through the network may be needed to achieve certain input-output connections.

A multistage network is called *nonblocking* if it can perform all possible connections between inputs and outputs by rearranging its connections. In such a network, a connection path can always be established between any input-output pair. The Benes networks (Benes, 1965) have such a capability. However, Benes networks require almost twice the number of stages to achieve the nonblocking connections. The Clos networks (Clos, 1953) can also perform all permutations in a single pass without blocking. Certain subclasses of blocking networks can also be made nonblocking if extra stages are added or connections are restricted. The blocking problem can be avoided by using combining networks to be described in the next section.

Crossbar Networks In a *crossbar network*, every input port is connected to a free output port through a crosspoint switch (circles in Fig. 2.26a) 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 returned to the requesting processor along the same crosspoint switch. In general, such a crossbar network requires the use of $n \times m$ crosspoint switches. A square crossbar ($n = m$) can implement any of the $n!$ permutations without blocking.

As introduced earlier, 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 memory requests independently and asynchronously. This poses the problem of multiple requests destined for the same memory module at the same time. In such cases, only one of the requests is serviced at a time. Let us characterize below the crosspoint switching operations.

Crosspoint Switch Design Out of n crosspoint switches in each column of an $n \times m$ crossbar mesh, only one can be connected at a time. To resolve the contention for each memory module, each crosspoint switch must be designed with extra hardware.

Furthermore, each crosspoint switch requires the use of a large number of connecting lines accommodating address, data path, and control signals. This means that each crosspoint has a complexity matching that of a bus of the same width.

For an $n \times n$ crossbar network, this implies that n^2 sets of crosspoint switches and a large number of lines are needed. What this amounts to is a crossbar network requiring extensive hardware when n is very large. So far only relatively small crossbar networks with $n \leq 16$ have been built into commercial machines.

On each row of the crossbar mesh, multiple crosspoint switches can be connected simultaneously. Simultaneous data transfers can take place in a crossbar between n pairs of processors and memories.

Figure 7.6 shows the schematic design of a row of crosspoint switches in a single crossbar network. Multiplexer modules are used to select one of n *read* or *write* requests for service. Each processor sends in an independent request, and the arbitration logic makes the selection based on certain fairness or priority rules.

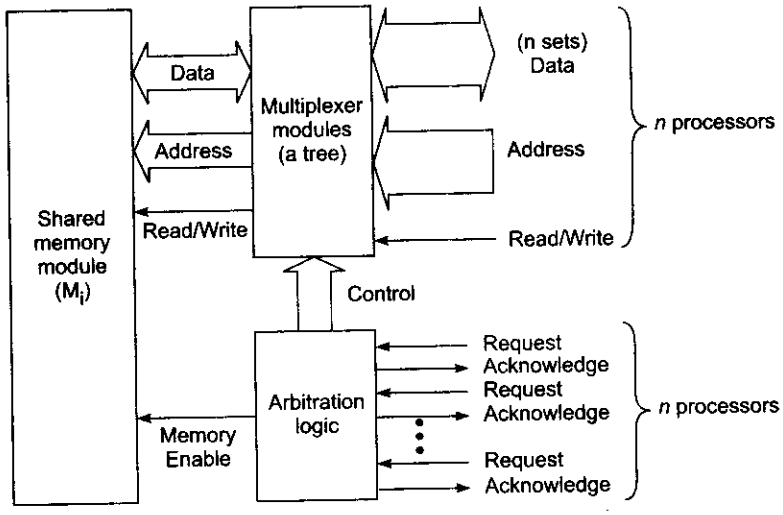


Fig. 7.6 Schematic design of a row of crosspoint switches in a crossbar network

For example, a 4-bit control signal will be generated for $n = 16$ processors. Note that n sets of data, address, and read/write lines are connected to the input of the multiplexer tree. 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 returned to the selected processor in the reverse direction using the data path established. In the case of *write*, the 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. Note that the data path established is bidirectional, in order to serve both *read* and *write* requests for different memory cycles.

Crossbar Limitations A single processor can send many requests to multiple memory modules. For an $n \times n$ crossbar network, at most n memory words can be delivered to at most n processors in each cycle.

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 multiprocessors with a few processors accessing a few memory modules. A single-stage crossbar network is not expandable once it is built.

Redundancy or parity-check lines can be built into each crosspoint switch to enhance the fault tolerance and reliability of the crossbar network.

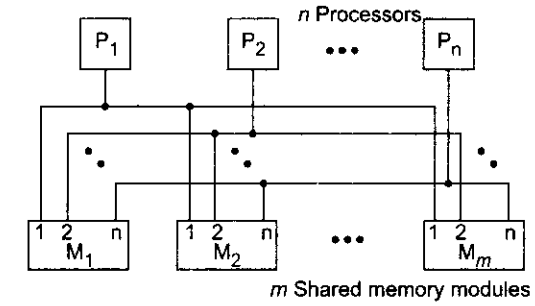
Multiport Memory Because building a crossbar network into a large system is cost prohibitive, some mainframe multiprocessors used a multiport memory organization. The idea is to move all crosspoint arbitration and switching functions associated with each memory module into the memory controller.

Thus the memory module becomes more expensive due to the added access ports and associated logic as demonstrated in Fig. 7.7a. 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.

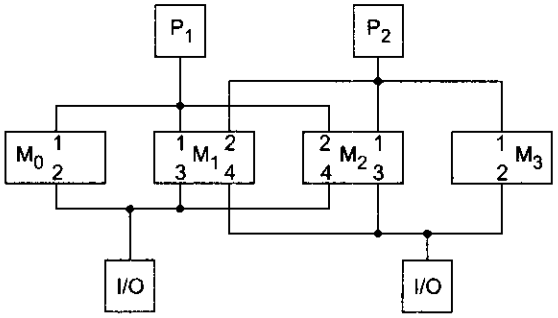
The multiport memory organization is a compromise solution between a low-cost, low-performance bus system and a high-cost, high-bandwidth crossbar system. The contention bus is time-shared by all processors and device modules attached. The multiport memory must resolve conflicts among processors.

This memory structure becomes expensive when m and n become large. A typical mainframe multiprocessor configuration may have $n = 4$ processors and $m = 16$ memory modules. A multiport memory multiprocessor is not scalable because once the ports are fixed, 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)

For example, the Univac 1100/94 multiprocessor consisted of four CPUs, four I/O processors, and two scientific vector processors connected to four shared-memory modules, each of which was 10-way ported. The access to these ports was prioritized under operating system control. In other multiprocessors, part of the memory module can be made private with ports accessible only to the owner processors.

7.1.3 Multistage and Combining Networks

Multistage networks are used to build larger multiprocessor systems. We describe two multistage networks, the Omega network and the Butterfly network, that have been built into commercial machines. We will study a special class of multistage networks, called combining networks, for resolving access conflicts automatically through the network. The combining network was built into the NYU's Ultracomputer.

Routing in Omega Network We have defined the Omega network in Chapter 2. In what follows, we describe the message-routing algorithm and broadcast capability of Omega network. This class of network was built into the Illinois Cedar multiprocessor (Kuck et al., 1987), into the IBM RP3 (Pfister et al., 1985), and into the NYU Ultracomputer (Gottlieb et al., 1983). An 8-input Omega network is shown in Fig. 7.8.

In general, an n-input Omega network has $\log_2 n$ stages. The stages are labeled from 0 to $\log_2 n - 1$ from the input end to the output end. Data routing is controlled by inspecting the destination code in binary. When the ith high-order bit of the destination code is a 0, a 2×2 switch at stage i connects the input to the upper output. Otherwise, the input is directed to the lower output.

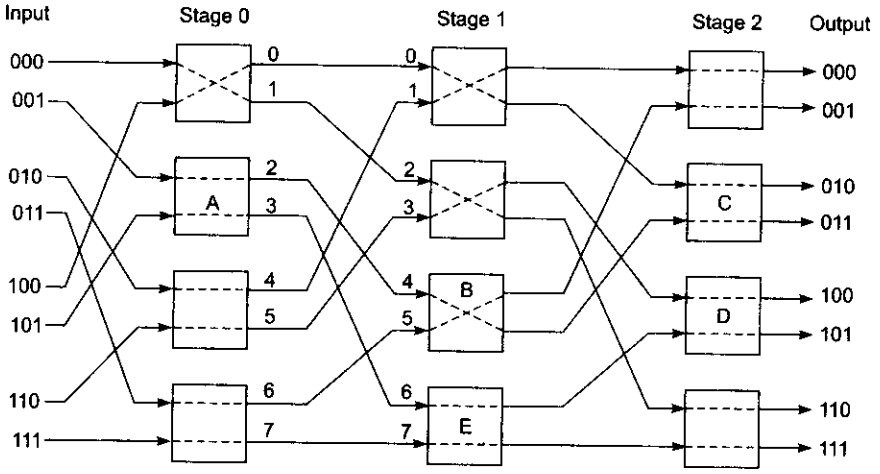
Two switch settings are shown in Figs. 7.8a and b with respect to permutations $\pi_1 = (0, 7, 6, 4, 2)\ (1, 3)\ (5)$ and $\pi_2 = (0, 6, 4, 7, 3)\ (1, 5)\ (2)$, respectively.

The switch settings in Fig. 7.8a are for the implementation of π_1, which maps $0 \rightarrow 7$, $7 \rightarrow 6$, $6 \rightarrow 4$, $4 \rightarrow 2$, $2 \rightarrow 0$, $1 \rightarrow 3$, $3 \rightarrow 1$, $5 \rightarrow 5$. 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 001 is connected to the upper output (labeled 2). The middle bit in 011 is a "one", thus input 4 to switch B is connected to the lower output with a "crossover" connection. The least significant bit in 011 is a "one", implying a flat connection in switch C. Similarly, the switches A, E, and D are set for routing a message from input 101 to output 101. There exists no conflict in all the switch settings needed to implement the permutation π_1 in Fig. 7.8a.

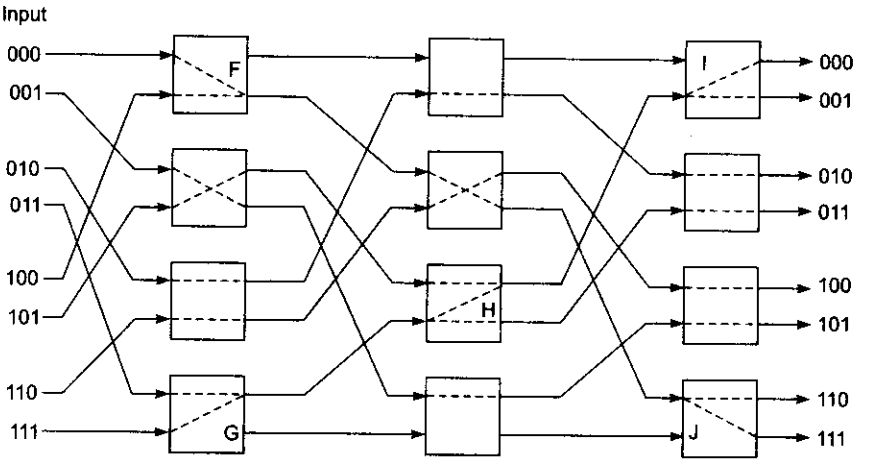
Now consider implementing the permutation π_2 in the 8-input Omega network (Fig. 7.8b). Conflicts in switch settings do exist in three switches identified as F, G, and H. The conflicts occurring at F are caused by the desired routings $000 \rightarrow 110$ and $100 \rightarrow 111$. Since both destination 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, we see 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, which is allowed if the hardware is built to have four legitimate states as shown in Fig. 2.24a. The above example indicates the fact that not all permutations can be implemented in one pass through the Omega network.

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.



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



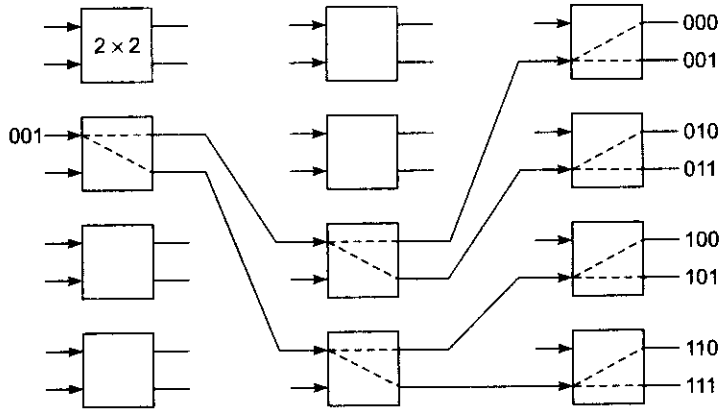
(b) Permutation π_2 = (0, 6, 4, 7, 3) (1, 5) (2) blocked at switches marked F, G, and H

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

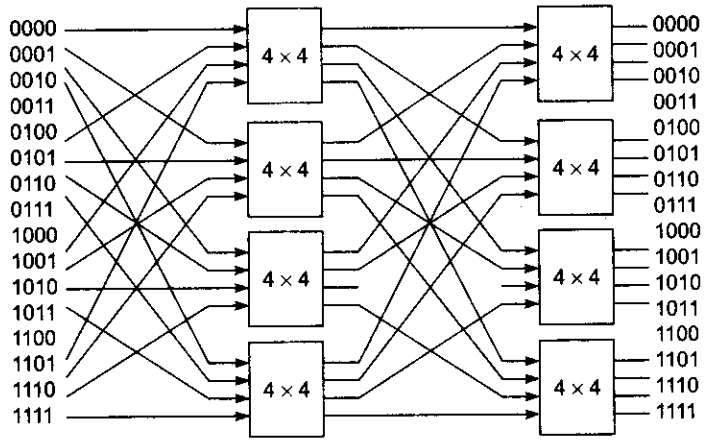
For $n = 8$, this implies that only $8^4/8! = 4096/40320 = 0.1016 = 10.16\%$ of all permutations are implementable in a single pass through an 8-input Omega network. All others will cause blocking and demand up to three passes to be realized. In general, a maximum of $\log_2 n$ passes are needed for an n-input Omega. Blocking is not a desired feature in any multistage network, since it lowers the effective bandwidth.

The Omega network can also be used to broadcast data from one source to many destinations, as exemplified 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 × 4 switch boxes are used as building blocks, as exemplified in Fig. 7.9b for a 16-input Omega network with $\log_4 16 = 2$ stages.



(a) Broadcast connections



(b) Using four-way shuffle interstage connections

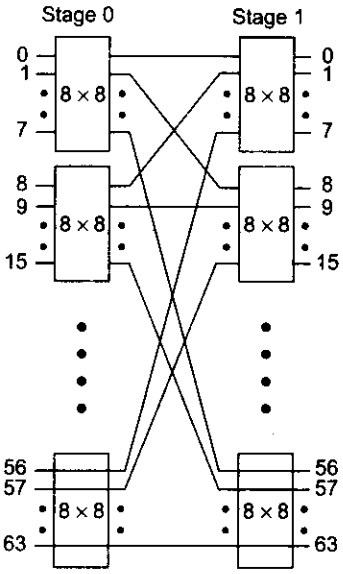
Fig. 7.9 Broadcast capability of an Omega network built with 4 × 4 switches

Note that a four-way shuffle corresponds to dividing the 16 inputs into four equal subsets and then shuffling them evenly among the four subsets. When $k \times k$ switch boxes are used, one can define a k-way shuffle function to build an even larger Omega network with $\log_k n$ stages.

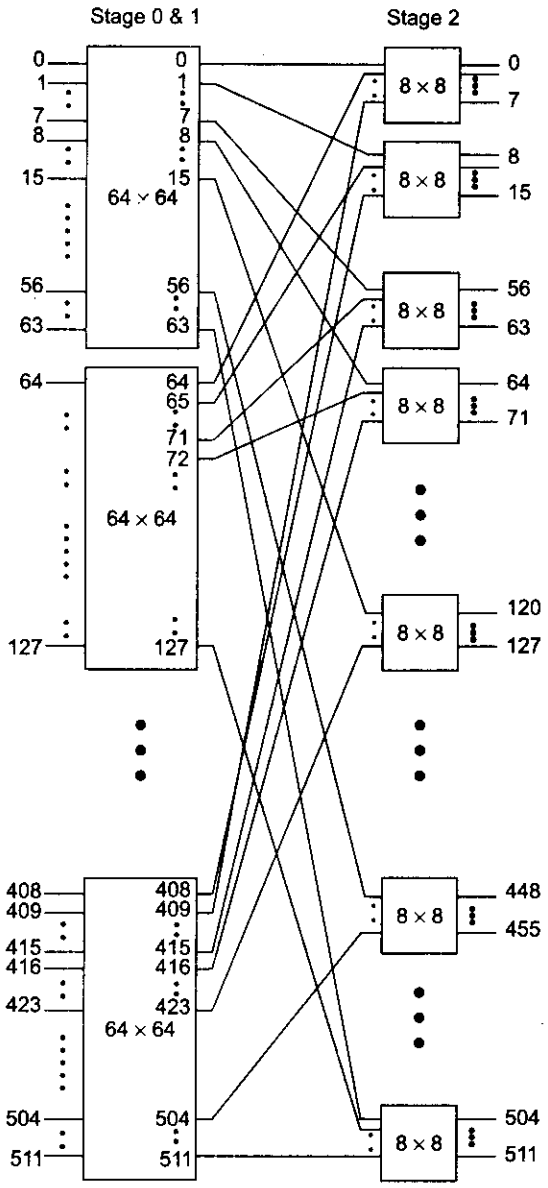
Routing in Butterfly Networks This class of networks is constructed with crossbar switches as building blocks. Figure 7.10 shows two Butterfly networks of different sizes. Figure 7.10a shows a 64-input Butterfly network built with two stages ($2 = \log_8 64$) of 8×8 crossbar switches. The eight-way shuffle 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 8×8 crossbar switches. Each of the 64×64 boxes in Fig. 7.10b is identical to the two-stage Butterfly network in Fig. 7.10a.

In total, sixteen 8×8 crossbar switches are used in Fig. 7.10a and $16 \times 8 + 8 \times 8 = 192$ are used in Fig. 7.10b. Larger Butterfly networks can be modularly constructed using more stages. Note that no broadcast

connections are allowed in a Butterfly network, making these networks a restricted subclass of Omega networks.



(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 and the IBM RP3 multiprocessor, a combining mechanism has been added to the Omega network. The purpose was to combine multiple requests heading for the same destination at switch points where conflicts are taking place.

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

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

$$\begin{aligned} &\text{Fetch\&Add } (x, e) \\ &\quad \{temp \leftarrow x; \\ &\quad\ x \leftarrow temp + e; \\ &\quad \text{return } temp\} \end{aligned} \tag{7.1}$$

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, $e_1 + e_2 + \ldots + e_N$, is produced in any arbitrary serialization of the N requests.

This sum is added to the memory word x, resulting in a new value $x + e_1 + e_2 + \ldots + e_N$. The values returned to the N requests are all unique, depending on the serialization order followed. The net result is similar to a sequential execution of N Fetch&Adds but is performed in one indivisible operation. Two simultaneous requests are combined in a switch as illustrated in Fig. 7.11.

One of the following operations will be performed if processor P_1 executes $\text{Ans}_1 \leftarrow$ Fetch&Add(x, e_1) and P_2 executes $\text{Ans}_2 \leftarrow$ 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 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 Fetch&Add(x, e_1) and Fetch&Add(x, e_2), respectively, as illustrated in Fig. 7.11 in four steps.

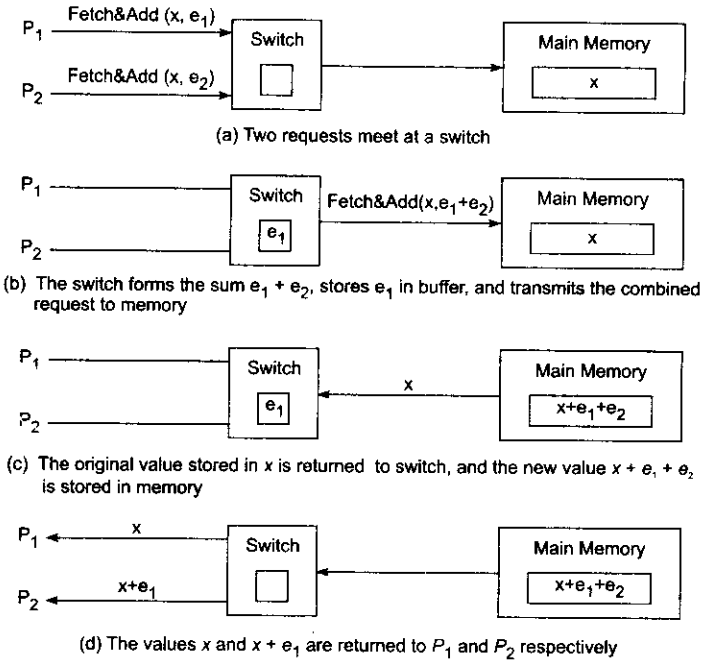


Fig. 7.11 Two Fetch&Add operations are combined to access a shared variable simultaneously via a combining network

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 Systems The IBM RP3 was designed to include 512 processors using a high-speed 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 of 13 Gbytes/s using a 50-MHz clock.

Multistage Omega networks were also built into the Cedar multiprocessor (Kuck et al., 1986) at the University of Illinois and in the Ultracomputer (Gottlieb et al., 1983) at New York University.

The BBN Butterfly processor (TC2000) used 8×8 crossbar switch modules to build a two-stage 64×64 Butterfly network for a 64-processor system, and a three-stage 512×512 Butterfly switch (see Fig. 7.10) for a 512-processor system in the TC2000 Series. The switch hardware was clocked at 38 MHz with a 1-byte data path. The maximum interprocessor bandwidth for a 64-processor TC2000 was designed at 2.4 Gbytes/s.

The Cray Y-MP multiprocessor used 64-, 128-, or 256-way interleaved memory banks, each of which could be accessed via four ports. Crossbar networks were used between the processors and memory banks in all Cray multiprocessors. The Alliant FX/2800 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.

7.2 CACHE COHERENCE AND SYNCHRONIZATION MECHANISMS

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

7.2.1 The Cache Coherence Problem

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

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

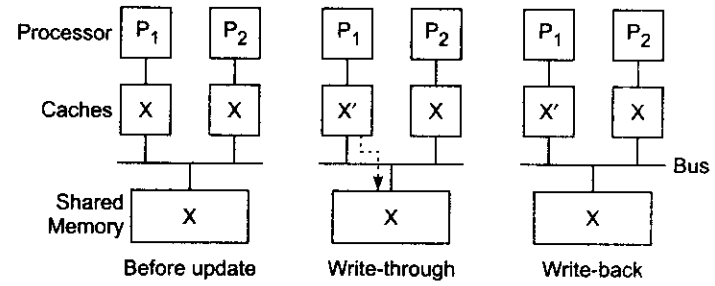
Inconsistency in Data Sharing The cache inconsistency problem occurs only when multiple private caches are used. In general, three sources of the problem are identified: *sharing of writable data, process migration*, and *I/O activity*. Figure 7.12 illustrates the problems caused by the first two sources.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

a shared data element which has been referenced by both processors. Before update, the three copies of X are consistent.

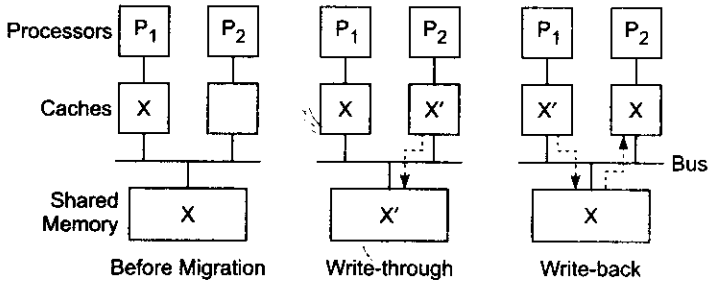
If processor P_1 writes new data X' into the cache, the same copy will be written immediately into the shared memory under a *write-through* policy. In this case, inconsistency occurs between the two copies (X' and X) in the two caches (Fig. 7.12a).

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

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



(a) Inconsistency in sharing of writable data



(b) Inconsistency after process migration

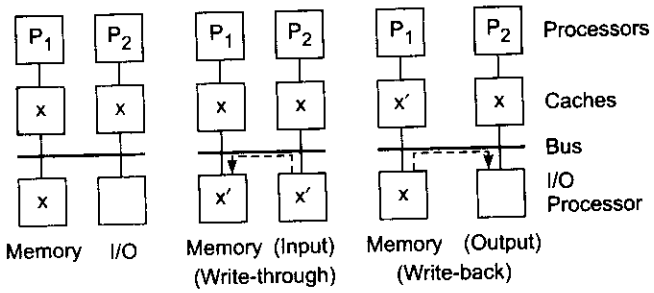
Fig. 7.12 Cache coherence problems in data sharing and in process migration (Adapted from Dubois, Scheurich, and Briggs 1988)

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

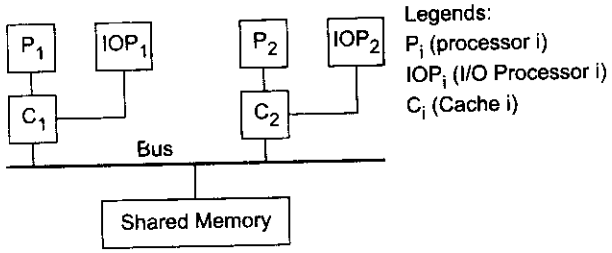
Inconsistency problems may occur during I/O operations that bypass the caches.

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

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



(a) I/O operations bypassing the cache



(b) A possible solution

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

Two Protocol Approaches Many of the early commercially available multiprocessors used bus-based memory systems. A bus is a convenient device for ensuring cache coherence because it allows all processors in the system to observe ongoing memory transactions. If a bus transaction threatens the consistent state of a locally cached object, the cache controller can take appropriate actions to invalidate the local copy. Protocols using this mechanism to ensure coherence are called *snoopy protocols* because each cache snoops on the transactions of other caches.

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

In general, a cache coherence protocol consists of the set of possible states in the local caches, the state in the shared memory, and the state transitions caused by the messages transported through the interconnection network to keep memory coherent. In what follows, we first describe the snoopy protocols and then the directory-based protocols. Other approaches to designing a scalable cache coherence interface will be studied in Chapter 9.

7.2.2 Snoopy Bus Protocols

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

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

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

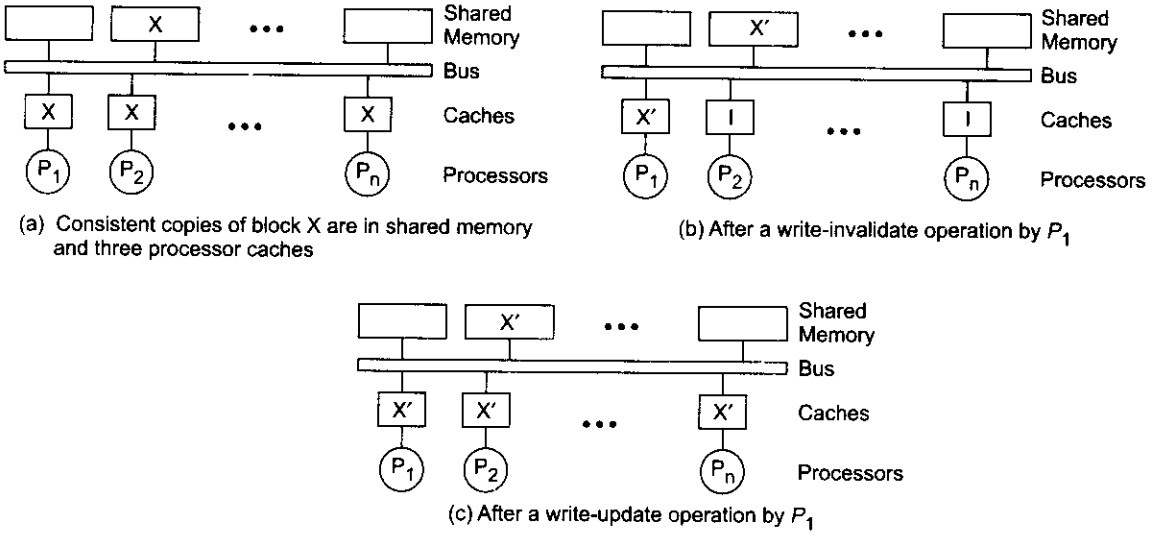


Fig. 7.14 Write-invalidate and write-update coherence protocols for write through caches (1: invalidate)

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

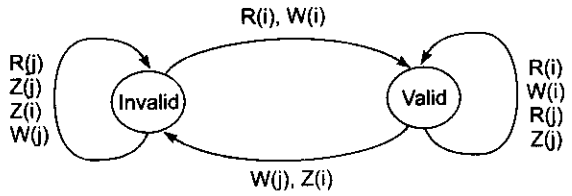
A remote processor is denoted j, where $j \neq i$. For each of the two cache states, six possible events may take place. Note that all cache copies of the same block use the same transition graph in making state changes.

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

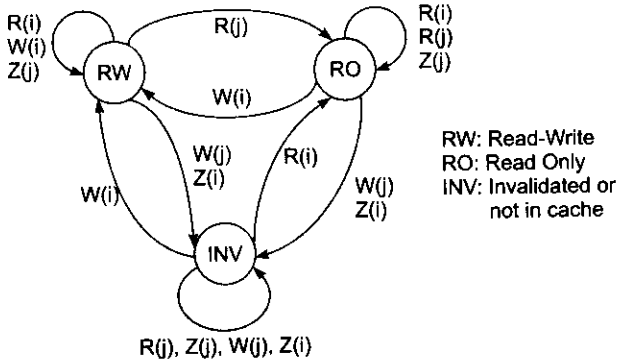
Wherever a remote processor *writes* ($W(j)$) into its cache copy, all other cache copies become invalidated. The cache block in cache i becomes valid whenever a successful read ($R(i)$) or write ($W(i)$) is carried out by a local processor i.

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

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



(a) Write-through cache



W(i) = Write to block by processor i.
R(i) = Read block by processor i.
Z(i) = Replace block in cache i.

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

(b) Write-back cache

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

When the memory owns a block, caches can contain only the RO copies of the block. In other words, multiple copies may exist in the RO state and every processor having a copy (called a *keeper* of the copy) can *read* ($R(i)$, $R(j)$) the copy safely.

The INV state is entered whenever a remote processor *writes* ($W(j)$) its local copy or the local processor replaces ($Z(i)$) its own block copy. The RW state corresponds to only one cache copy existing in the entire system owned by the local processor i. *Read* ($R(i)$) and *write* ($W(i)$) can be safely performed in the RW state. From either the RO state or the INV state, the cache block becomes uniquely owned when a local *write* ($W(i)$) takes place.

Other state transitions in Fig. 7.15b can be similarly figured out. Before a block is modified, ownership for exclusive access must first be obtained by a *read-only* bus transaction which is broadcast to all caches and memory. If a modified block copy exists in a remote cache, memory must first be updated, the copy invalidated, and ownership transferred to the requesting cache.

Write-once Protocol James Goodman (1983) proposed a cache coherence protocol for bus-based multiprocessors. This scheme combines the advantages of both write-through and write-back invalidations. In order to reduce bus traffic, the very first *write* of a cache block uses a write-through policy.

This will result in a consistent memory copy while all other cache copies are invalidated. After the first *write*, shared memory is updated using a write-back policy. This scheme can be described by the four-state transition graph shown in Fig. 7.16. The four cache states are defined below:

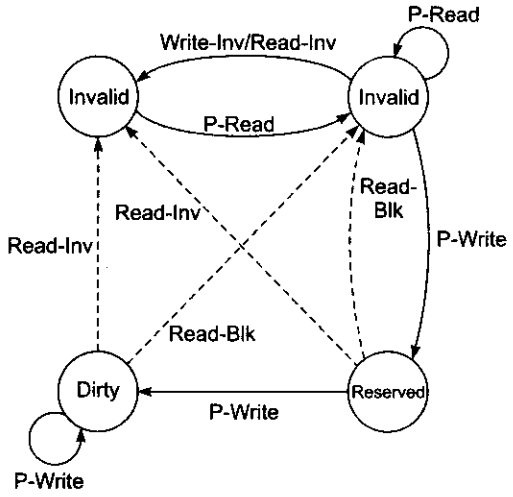


Fig. 7.16 Goodman's write-once cache coherence protocol using the write invalidate policy on write-back caches (Adapted from James Goodman 1983, reprinted from Stenstrom, *IEEE Computer*, June 1990)

- *Valid:* The cache block, which is consistent with the memory copy, has been *read* from shared memory and has not been modified.
- *Invalid:* The block is not found in the cache or is inconsistent with the memory copy.
- *Reserved:* Data has been *written* exactly *once* since being *read* from shared memory. The cache copy is consistent with the memory copy, which is the only other copy.

- *Dirty:* The cache block has been modified (*written*) more than once, and the cache copy is the only one in the system (thus inconsistent with all other copies).

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

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

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

Goodman's write-once protocol demands special bus lines to inhibit the main memory when the memory copy is invalid, and a *bus-read* operation is needed after a *read miss*. Most standard buses cannot support this inhibition operation.

The IEEE Futurebus+ proposed to include this special bus provision. Using a write-through policy after the first *write* and using a write-back policy in all additional *writes* eliminates unnecessary invalidations.

Snoopy cache protocols are popular in bus-based multiprocessors because of their simplicity of implementation. The write-invalidate policies were implemented on the Sequent Symmetry multiprocessor and on the Alliant FX multiprocessor.

Besides the DEC Firefly multiprocessor, the Xerox Palo Alto Research Center implemented another write-update protocol for its Dragon multiprocessor workstation. The Dragon protocol avoids updating memory until replacement, in order to improve the efficiency of intercache transfers.

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 among cache copies at the same level is maintained in the same way as described above. 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 P_1 issues a *write* request. The *write* request propagates up to the highest level and invalidates copies in C_{20}, C_{22}, C_{16}, and C_{18}, as shown by the arrows to all the shaded copies.

High-level caches such as C_{20} keep track of dirty blocks beneath them. A subsequent *read* request issued by P_7 will propagate up the hierarchy because no copies exist. When it reaches the top level, cache C_{20} issues a flush request down to cache C_{11} and the dirty copy is supplied to the private cache associated with processor P_7. Note that higher-level caches act as filters for consistency control. An invalidation command or a read request will not propagate down to clusters that do not contain a copy of the corresponding block. The cache C_{21} acts in this manner.

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. However, as the block size increases to a *data pollution* point, the miss ratio starts to increase. For larger caches, the data pollution point appears at a larger block size.

For a system requiring extensive process migration or synchronization, the write-invalidate protocol will perform better. However, a cache miss can result for an invalidation initiated by another processor prior to the cache access. Such *invalidation misses* may increase bus traffic and thus should be reduced.

Extensive simulation results have suggested that bus traffic in a multiprocessor may increase when the block size increases. Write-invalidate also facilitates the implementation of synchronization primitives. Typically, the average number of invalidated cache copies is rather small (one or two) in a small multiprocessor.

The write-update protocol requires a bus broadcast capability. This protocol also can avoid the ping-pong effect on data shared between multiple caches. Reducing the sharing of data will lessen bus traffic in a write-update multiprocessor. However, write–update cannot be used with long write bursts. Only through extensive program traces (trace-driven simulation) can one reveal the cache behavior, hit ratio, bus traffic, and effective memory-access time.

7.2.3 Directory-Based Protocols

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

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

Directory Structures In a multistage or packet switched network, cache coherence is supported by using cache directories to store information on where copies of cache blocks reside. Various directory-based protocols differ mainly in how the directory maintains information and what information it stores.

Tang (1976) proposed the first directory scheme, which used a *central directory* containing duplicates of all cache directories. This central directory, providing all the information needed to enforce consistency, is usually very large and must be associatively searched, like the individual cache directories. Contention and long search times are two drawbacks in using a central directory for a large multiprocessor.

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

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

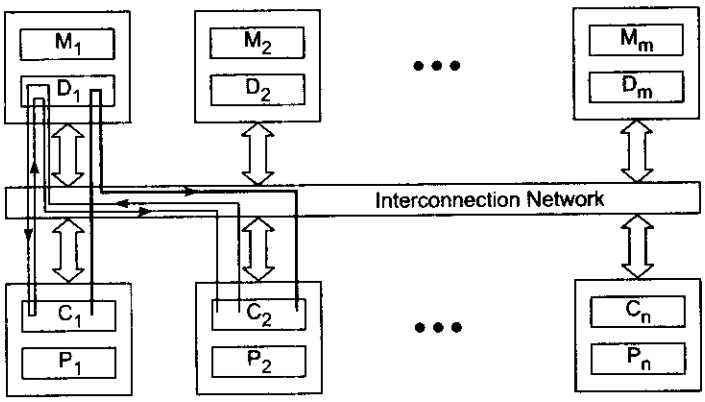


Fig. 7.17 Basic concept of a directory-based cache coherence scheme (Courtesy of Censier and Feautrier, *IEEE Trans. Computers*, Dec. 1978)

A cache-coherence protocol that does not use broadcasts must store the locations of all cached copies of each block of shared data. This list of cached locations, whether centralized or distributed, is called a *cache directory*. A directory entry for each block of data contains a number of *pointers* to specify the locations of copies of the block. Each directory entry also contains a dirty bit to specify whether a particular cache has permission to write the associated block of data.

Different types of directory protocols fall under three primary categories: *full map directories*, *limited directories*, and *chained directories*. Full-map directories store enough data associated with each block in global memory so that every cache in the system can simultaneously store a copy of any block of data. That is, each directory entry contains N pointers, where N is the number of processors in the system.

Limited directories differ from full-map directories in that they have a fixed number of pointers per entry, regardless of the system size. Chained directories emulate the full-map schemes by distributing the directory

among the caches. The following descriptions of the three classes of cache directories are based on the original classification by Chaiken, Fields, Kwihara, and Agarwal (1990):

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

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

Figure 7.18a illustrates three different states of a full-map directory. In the first state, location X is missing in all of the caches in the system. The second state results from three caches (C1, C2, and C3) requesting copies of location X. Three pointers (processor bits) are set in the entry to indicate the caches that have copies of the block of data. In the first two states, the dirty bit on the left side of the directory entry is set to clean (C), indicating that no processor has permission to write to the block of data. The third state results from cache C3 requesting write permission for the block. In the final state, the dirty bit is set to dirty (D), and there is a single pointer to the block of data in cache C3.

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

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

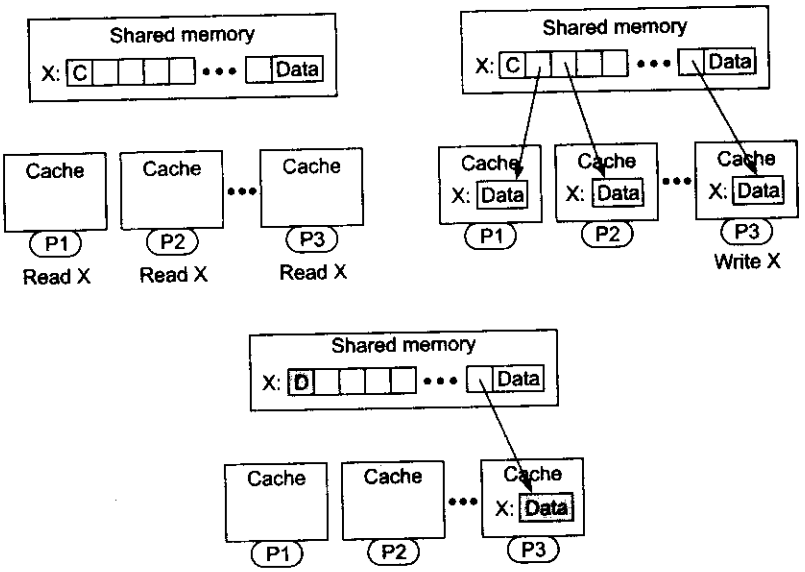
The memory module waits to receive the acknowledgments before allowing processor P3 to complete its write transaction. By waiting for acknowledgments, the protocol guarantees that the memory system ensures sequential consistency. The full-map protocol provides a useful upper bound for the performance of centralized directory-based cache coherence. However, it is not scalable due to excessive memory overhead.

Because the size of the directory entry associated with each block of memory is proportional to the number of processors, the memory consumed by the directory is proportional to the size of memory $O(N)$ multiplied by the size of the directory $O(N)$. Thus, the total memory overhead scales as the square of the number of processors $O(N^2)$.

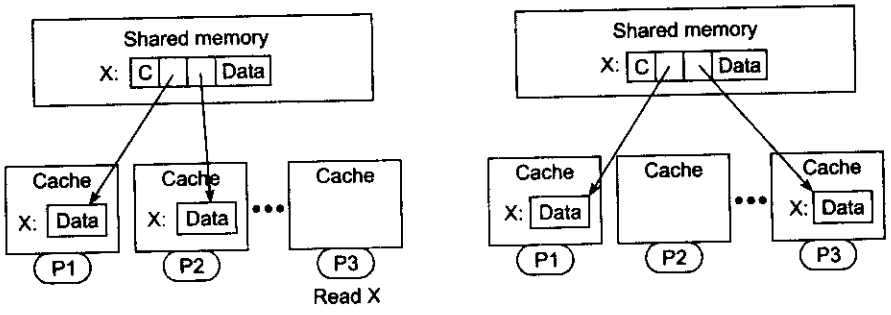
Limited Directories Limited directory protocols are designed to solve the directory size problem. Restricting the number of simultaneously cached copies of any particular block of data limits the growth of the directory to a constant factor.

A directory protocol can be classified as $Dir_i\ X$ using the notation from Agarwal et al (1988). The symbol i stands for the number of pointers, and X is NB for a scheme with no broadcast. A full-map scheme without

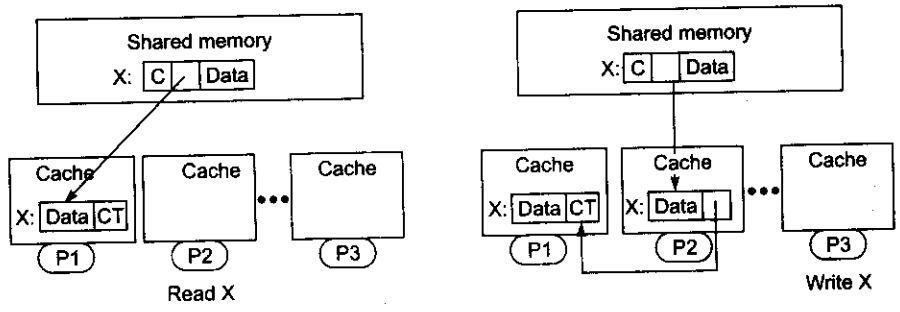
broadcast is represented as $Dir_N NB$. A limited directory protocol that uses $i < N$ pointers is denoted $Dir_i NB$. The limited directory protocol is similar to the full-map directory, except in the case when more than i caches request read copies of a particular block of data.



(a) Three states of a full-map directory



(b) Eviction in a limited directory



(c) The chained directory

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. In this case, we can view the two-pointer directory as a two-way set-associative cache of pointers to shared copies. When cache C3 requests a copy of location X, the memory module must invalidate the copy in either cache C1 or cache C2. This process of pointer replacement is called *eviction*. Since the directory acts as a set-associative cache, it must have a pointer replacement policy.

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

Directory pointers in a $Dir_i NB$ protocol encode binary processor identifiers, so each pointer requires $\log_2 N$ bits of memory, where N is the number of processors in the system. Given the same assumptions as for the full-map protocol, the memory overhead of limited directory schemes grows as $O(N \log_2 N)$.

These protocols are considered scalable with respect to memory overhead because the resource required to implement them grows approximately linearly with the number of processors in the system. $Dir_i B$ protocols allow more than i copies of each block of data to exist, but they resort to a broadcast mechanism when more than i cached copies of a block need to be invalidated. However, point-to-point interconnection networks do not provide an efficient systemwide broadcast capability. In such networks, it is difficult to determine the completion of a broadcast to ensure sequential consistency.

Chained Directories Chained directories realize the scalability of limited directories without restricting the number of shared copies of data blocks. This type of cache coherence scheme is called a *chained* scheme because it keeps track of shared copies of data by maintaining a chain of directory pointers.

The simpler of the two schemes implements a singly linked chain, which is best described by example (Fig. 7.18c). Suppose there are no shared copies of location X. If processor P1 reads location X, the memory sends a copy to cache C1, along with a *chain termination* (CT) pointer. The memory also keeps a pointer to cache C1. Subsequently, when processor P2 reads location X, the memory sends a copy to cache C2, along with the pointer to cache C1. The memory then keeps a pointer to cache C2.

By repeating the above step, all of the caches can cache a copy of the location X. If processor P3 writes to location X, it is necessary to send a data invalidation message down the chain. To ensure sequential consistency, the memory module denies processor P3 write permission until the processor with the chain termination pointer acknowledges the invalidation of the chain. Perhaps this scheme should be called a *gossip* protocol (as opposed to a snoopy protocol) because information is passed from individual to individual rather than being spread by covert observation.

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

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

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

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

The second scheme can be implemented by a less complex protocol than the first. In either case, sequential consistency is maintained by locking the memory location while invalidations are in progress. Another solution to the replacement problem is to use a doubly linked chain. This scheme maintains forward and backward chain pointers for each cached copy so that the protocol does not have to traverse the chain when

there is a cache replacement. The doubly linked directory optimizes the replacement condition at the cost of a larger average message block size (due to the transmission of extra directory pointers), twice the pointer memory in the caches, and a more complex coherence protocol.

Although the chained protocols are more complex than the limited directory protocols, they are still scalable in terms of the amount of memory used for the directories. The pointer sizes grow as the logarithm of the number of processors, and the number of pointers per cache or memory block is independent of the number of processors.

Cache Design Alternatives The relative merits of physical address caches and virtual address caches have to be judged based on the access time, the aliasing problem, the flushing problem, OS kernel overhead, special tagging at the process level, and cost/performance considerations. Beyond the use of private caches, three design alternatives are suggested below.

Each of the design alternatives has its own advantages and shortcomings. There exists insufficient evidence to determine whether any of the alternatives is always better or worse than the use of private caches. More research and trace data are needed to apply these cache architectures in designing high-performance multiprocessors.

Shared Caches 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 case. 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.

Shared caches can be built as second-level caches. Sometimes, one can make the second-level caches partially shared by different clusters of processors. Various cache architectures are possible if private and shared caches are both used in a memory hierarchy. The use of shared cache alone may be against the scalability of the entire system. Tradeoffs between using private caches, caches shared by multiprocessor clusters, and shared main memory are interesting topics for further research.

Noncacheable 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*. Special hardware tagging must be used to distinguish them. Cache systems with cacheable and noncacheable blocks demand more support from hardware and compilers.

Cache Flushing A third 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.

Flushing can be made very selective by the compiler in order to increase efficiency. Cache flushing at synchronization, I/O, and process migration may be carried out unconditionally or selectively. Cache flushing is more often used with virtual address caches.

7.2.4 Hardware Synchronization Mechanisms

Synchronization is a special form of communication in which control information is exchanged, instead of data, between communicating processes residing in the same or different processors. Synchronization

enforces correct sequencing of processors and ensures mutually exclusive access to shared writable data. 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 (operating system) level synchronization mechanisms such as *semaphores* or *monitors*. Only hardware synchronization mechanisms are studied below. Software approaches to synchronization will be treated in Chapter 10.

Atomic Operations Most multiprocessors 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. Besides atomic memory operations, some interprocessor interrupts can be used for synchronization purposes. For example, the synchronization primitives, Test&Set (*lock*) and Reset (*lock*), are defined below:

$$\begin{aligned} &\text{Test\&Set} \quad (lock) \\ &\qquad\qquad temp \leftarrow lock; \quad lock \leftarrow 1; \\ &\qquad\qquad \text{return } temp \\ &\text{Reset} \quad (lock) \\ &\qquad\qquad lock \leftarrow 0 \end{aligned} \tag{7.4}$$

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*. To avoid spinning, interprocessor interrupts can be used.

A lock tied to an interrupt is called a *suspend lock*. Using such a lock, a process does not relinquish the processor while it is waiting. 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 370 mainframes.

Concurrent processes residing in different processors can be synchronized using *barriers*. A barrier can be implemented 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.

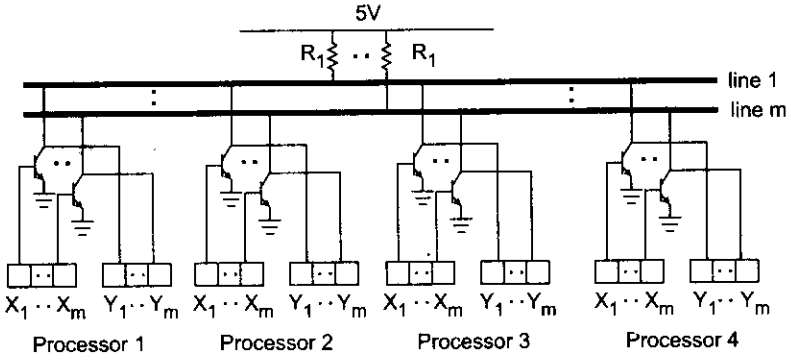
Wired Barrier Synchronization A wired-NOR logic is shown in Fig. 7.19 for implementing a barrier mechanism for fast synchronization. Each processor uses a dedicated control vector $X = (X_1, X_2, ..., X_m)$ and accesses a common monitor vector $Y = (Y_1, Y_2, ..., Y_m)$ in shared memory, where m corresponds to the barrier lines used.

The number of barrier lines needed for synchronization depends on the multiprogramming degree and the size of the multiprocessor system. Each control bit X_i is connected to the base (input) of a probing transistor. The monitor bit Y_i checks the collector voltage (output) of the transistor.

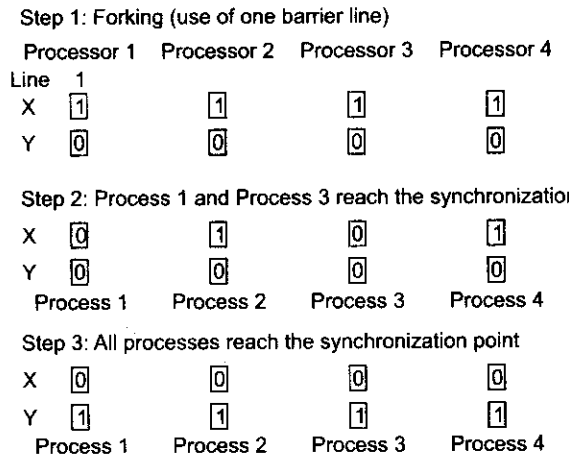
Each barrier line is wired-NOR to n transistors from n processors. Whenever bit X_i is raised to high (1), the corresponding transistor is closed, pulling down (0) the level of barrier line i. The wired-NOR connection implies that line i will be high (1) only if control bits X_i from all processors are low (0).

This demonstrates the ability to use the control bit X_i to signal the completion of a process on prccessor i. The bit X_i is set to 1 when a process is initiated and reset to 0 when the process finishes its execution.

When all processes finish their jobs, the X_i bits from 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. This timing is watched by all processors through snooping on the Y_i bits. Thus only one barrier line is needed to monitor the initiation and completion of a single synchronization involving many concurrent processes.



(a) Barrier lines and interface logic



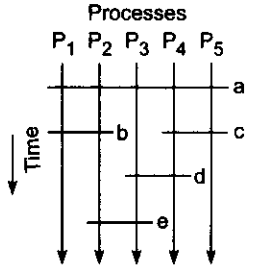
(b) Synchronization steps

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)

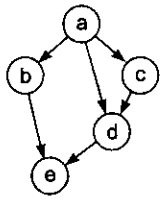
Multiple barrier lines can be used simultaneously to monitor several synchronization points. Figure 7.19 shows the synchronization of four processes residing on four processors using one barrier line. Note that other barrier lines can be used to synchronize other processes at the same time in a multiprogrammed multiprocessor environment.

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	11000	11001	10011	10110	10100
Y	00000	00000	00000	00000	00000
Step 1: Synchronization at barrier a					
X	01000	01001	00011	00110	00100
Y	10000	10000	10000	10000	10000
Step 2a: Synchronization at barrier b					
X	00000	00001	00011	00110	00100
Y	11000	11000	11000	11000	11000
Step 2b: Synchronization at barrier c					
X	00000	00001	00011	00010	00000
Y	11100	11100	11100	11100	11100
Step 3: Synchronization at barrier d					
X	00000	00001	00001	00000	00000
Y	11110	11110	11110	11110	11110
Step 4: Synchronization at barrier e					
X	00000	00000	00000	00000	00000
Y	11111	11111	11111	11111	11111

(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 2*a* and 2*b*. 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.

Separate control vectors are used in local processors. The above dynamic barrier synchronization is possible only if the synchronization pattern is predicted at compile time and process preemption is not allowed. One can also use the barrier wires along with counting semaphores in memory to support multiprogrammed multiprocessors in which preemption is allowed.

7.3 THREE GENERATIONS OF MULTICOMPUTERS

Three early generations of multicomputers are reviewed in this section, which have contributed to the development of modern systems. Experiences from Intel, nCUBE, MIT, and Caltech are examined. In particular, we present the Intel Paragon system in some detail. The generic multicomputer model shown in Fig. 1.9 and various network topologies presented in Section 2.3 form the background needed for reading this section. Further discussion on related topics and current advances can be found in Chapter 13.

7.3.1 Design Choices in the Past

Before we examine these developments, let us identify the major design choices made so far in building multicomputers, as compared with the development of other types of parallel computers. As illustrated in Fig. 7.21, the choices made involve the selection of processors, memory structure, interconnection schemes, and control strategy.

Design Choices In selecting a processor technology, a multicomputer designer typically chooses low-cost so-called commodity processors as building blocks. In fact, the majority of parallel computers have been built with standard off-the-shelf processors. Even the custom-designed processors used in the AMT DAP, nCUBE, TMC/CM-2, and IBM RP3 computers were low-cost processors.

The next step was to choose distributed memory for multicomputers rather than using shared memory which would limit the scalability. Each processor has its own local memory to address. Scalability becomes more feasible without shared resources. With distributed memory, a new programming model and tools are needed for multicomputers.

Multicomputers have message-passing, point-to-point, direct networks as an interconnection scheme rather than the address-switching networks used in NUMA multiprocessors like the IBM RP3 and BBN Butterfly. A message-passing network routes messages between nodes. Any node can send a message to another. Send/receive semantics must be incorporated to guarantee consistent programming with or without uniform messaging speeds.

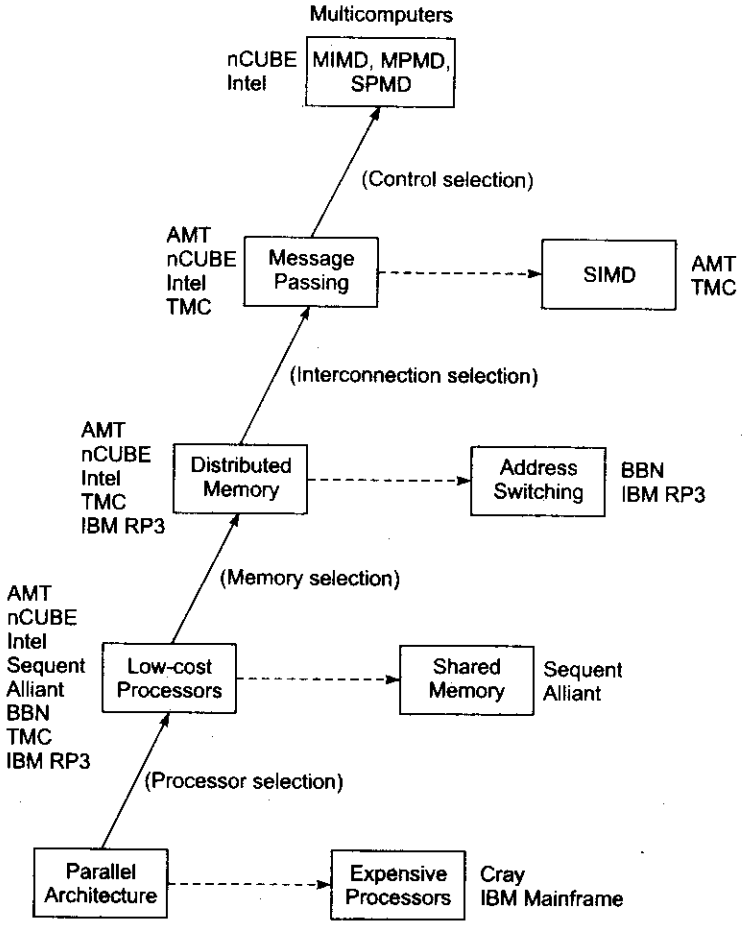


Fig. 7.21 Design choices made in the past for developing message-passing multicomputers compared to those made for other parallel computers (Courtesy of Intel Scientific Computers, 1988)

In selecting a control strategy, designers of multicomputers choose the asynchronous MIMD, MPMD, and SPMD operations, rather than the SIMD lockstep operations as in the CM-2 and DAP. Even though both support massive parallelism, the SIMD approach offers little or no opportunity to utilize existing multiprocessor code because radical changes must be made in the programming style.

On the other hand, multicomputers allow the use of existing software with minor changes from that developed for multiprocessors or for other types of parallel computers.

First Generation Caltech's Cosmic Cube (Seitz, 1983) was the first of the first generation multicomputers. The Intel iPSC/1, Ametek S/14, and nCUBE/10 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 ports. Seven I/O ports were used to form a seven-dimensional hypercube. The eighth port was used for an Ethernet connection from each node to the host.

Table 7.1 summarizes the important parameters used in designing the early three generations of multicomputers. The communication latency (for a 100-byte message) was rather long in the early 1980s. The 3-to-1 ratio between remote and local communication latencies was caused by the use of a *store-and-forward* routing scheme where the latency is proportional to the number of hops between two communicating nodes.

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	100K
Mflops scalar	6.4	512	40K
Mflops vector	640	10K	200K
Memory (Mbytes)	32	1K	32K
Communication latency (100-byte message)			
Neighbor (microseconds)	2000	5	0.5
Nonlocal (microseconds)	6000	5	0.5

(Modified from Athas and Seitz, "Multicomputers: Message-Passing Concurrent Computers", *IEEE Computer,* August 1988).

Vector hardware was added on a separate board attached to each processing node board. Or one could use the second board to hold extended local memory. The host used in the iPSC/1 was an Intel 310 microprocessor. All I/O must be done through the host.

7.3.2 Present and Future Development

The second and third generations of multicomputers are introduced below. The Intel Paragon is presented as a case study. More recent advances in high-performance computing are discussed in Chapter 13.

The Second Generation A major improvement of the second generation included the use of better processors, such as i386 in the iPSC/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 *wormhole routing*, reduced the communication latency significantly from 6000 μs to less than 5 μs. In fact, the latency for remote and local communications became almost the same, independent of the number of hops between any two nodes.

The architecture of a typical second-generation multicomputer is shown in Fig. 7.22. This corresponds to a 16-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.

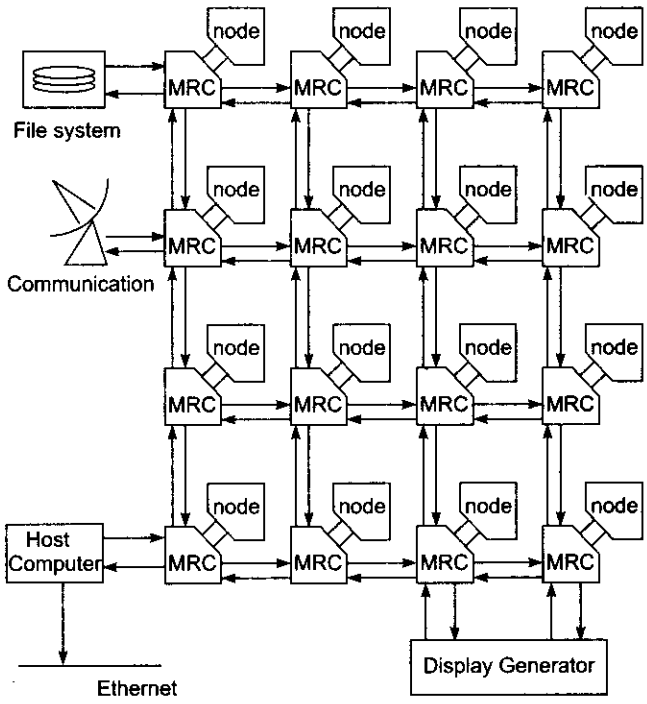


Fig. 7.22 The architecture of a second-generation multicomputer using a hardware-routed mesh interconnect (Courtesy of Charles Seitz; reprinted with permission from "Concurrent Architectures", *VLSI and Parallel Computation*, edited by Suaya and Birtwistle, Morgan Kaufmann Publishers, 1990)

Each node is implemented on a PC board plugged into the backplane at the proper MRC position. All I/O devices, graphics, and the host are connected to the periphery (boundary) of the mesh. The Intel Delta system had such a mesh architecture.

Another representative system was the nCUBE/2 which implemented a hypercube with up to 8192 nodes with a total of 512 Gbytes of distributed memory. Note that some parameters in Table 7.1 have been updated from the conservative estimates made by Atlas and Seitz in 1988. Typical figures representative of current systems can be found in Chapter 13.

The SuperNode 1000 was a Transputer-based multicomputer produced by Parsystem Ltd., England. Another second-generation system was Ametek's Series 2010, made with 25-MHz M68020 processors using a mesh-routed architecture with 225-Mbytes/s channels.

The Third Generation These designs laid the foundation for the current generation of multicomputers. Caltech had the Mosaic C project designed to use VLSI-implemented nodes, each containing a 14-MIPS processor, 20-Mbytes/s routing channels, and 16 Kbytes of RAM integrated on a single chip.

The full size of the Mosaic was targeted to have a total of 16,384 nodes organized in a three-dimensional mesh architecture. MIT built the J-machine which it planned to extend to a 65K-node multicomputer with VLSI nodes interconnected by a three-dimensional mesh network. We will study the J-machine experience in Section 9.3.2.

The J-machine planned to use message-driven processors to reduce the message handling overhead to less than 1 μs. Each processor chip would contain a 512-Kbit DRAM, a 32-bit processor, a floating-point unit, and a communication controller. The communication latency in systems was later reduced to a few ns using high-speed links and sophisticated communication protocols.

The significant reduction of overhead in communication and synchronization would permit the execution of much shorter tasks with grain sizes of 5 μs per processor in the J-machine, as opposed to executing tasks of 100 μs in the iPSc/1.This implies that concurrency may increase from 10^2 in the iPSc/1 to 10^5 in the J-machine.

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 multicomputers*.

Research is also underway to combine the private virtual address spaces distributed over the nodes into a globally shared virtual memory in MPP multicomputers. Instead of page-oriented message passing, 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 multiprocessors and multicomputers in a *heterogeneous processing* (HP) environment.

7.3.3 The Intel Paragon System

In the 1980s, hypercube multicomputers were made with homogeneous nodes because all I/O functions were given to the host. This limited the I/O bandwidth, and thus these computers could not be used in solving large-scale problems with efficiency or high throughput. The Intel Paragon was designed to overcome this difficulty. The usage model turned the multicomputer into an applications server with multiuser access in a network environment.

Ever since the introduction of the iPSC/2 CFS, parallel I/O has been possible with dedicated disk nodes in addition to the computing nodes. The iPSC/860 further pushed the idea of using heterogeneous node types. The Paragon system went further by making it a host-free multicomputer. We explain below the various node types used in the Paragon and present the hardware router design.

The architecture of the Intel Paragon system is shown in Fig. 7.23. This system was driven by applications which require solving general sparse matrix problems, performing parallel data manipulation, or making scientific predictions through simulation modeling.

These difficult problems demand heterogeneous node types for numeric, service, I/O, and network gateways, as demonstrated in the schematic diagram of the Paragon system. The mesh architecture of the Paragon was divided into three sections.

The middle section, called the compute partition, is a mesh of numeric nodes implemented with Intel i860XP microprocessors. This array had an aggregate of 8.8 Gbytes of distributed memory.

The system had a potential performance of 5 to 300 Gflops collectively. This mesh architecture eliminated the power-of-2 upgrade requirement of a hypercube architecture. 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 16×1 array of 16 disk nodes. The aggregate I/O bandwidth reached 48 Mbytes/s with a total of 27.4 Gbytes per disk I/O column.

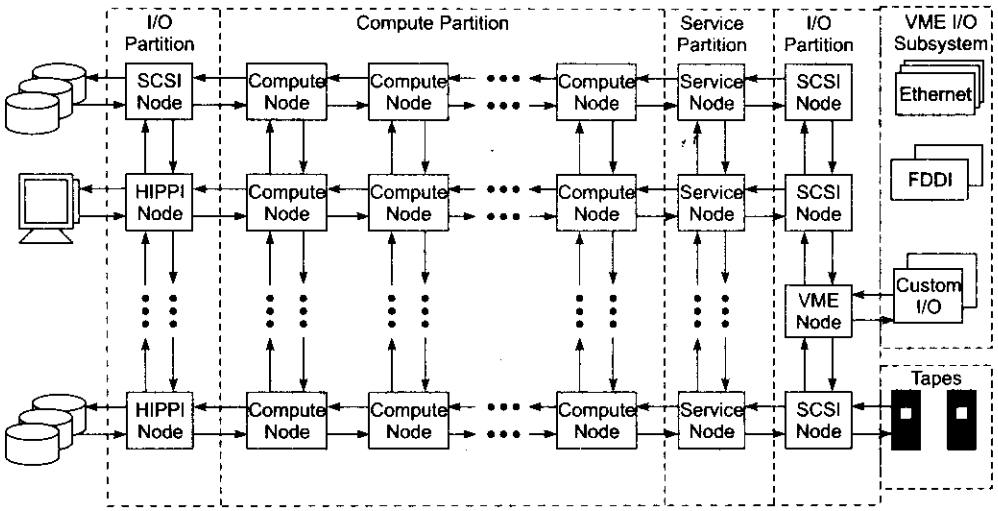


Fig. 7.23 The Intel Paragon system architecture (Courtesy of Intel Supercomputer Systems Division, 1991)

The processors 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. The service nodes were used for system diagnosis and handling of interrupts. The tape nodes were used for backup storage.

The Ethernet and HIPPI nodes were used for fast gateway connections with the outside world. Collectively, a 17,000-MIPS performance was claimed possible on the 570 numeric and disk I/O nodes involved in program execution. The system was designed to run iPSC/860-compatible software.

Node and Router Architecture The Paragon was designed as an experimental system. One unit was built and delivered to Caltech in May 1991 for research use by a consortium of 13 national laboratories and universities. The typical node architecture is shown in Fig. 7.24.

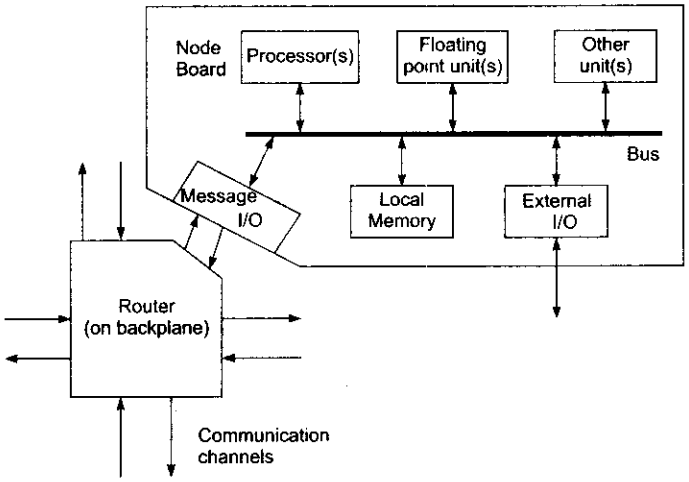


Fig. 7.24 Node architecture of the Paragon multicomputer

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 took 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.

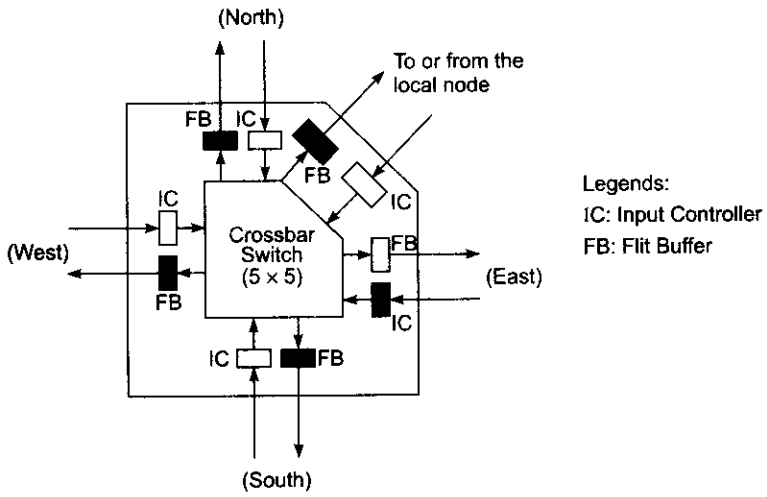


Fig. 7.25 The structure of a mesh-connected router with four pairs of I/O channels connected to neighboring routers

Each router had 10 I/O ports, 5 for input and 5 for output. Four pairs of I/O channels were used for mesh connection to the four neighbors at the north, south, east, and west nodes.

Flow control digits (flits) buffers were used at the end of input channels to hold the incoming flits. The concept of flits will be clarified in the next section. Besides four pairs of external channels, a fifth pair was used for internal connection between the router and the local node. A 5×5 crossbar switch was used to establish a connection between any input channel and any output channel.

The functions of the hardware router included pipelined message routing at the flit level and resolving buffer or channel deadlock situations to achieve deadlock-free routing. In the next section, we will explain various routing mechanisms and deadlock avoidance schemes.

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. Through time-sharing, one can also implement *virtual channels* to multiplex the use of physical channels as described in the next section.

7.4 MESSAGE-PASSING MECHANISMS

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

Both deterministic and adaptive routing algorithms are presented for achieving deadlock-free message routing. We first study deterministic dimension-order routing schemes such as E-cube routing for hypercubes and X-Y routing for two-dimensional meshes. Then we discuss adaptive routing using virtual channels or virtual subnets. Besides one-to-one unicast routing, we will consider one-to-many multicast and one-to-all broadcast operations using virtual subnets and greedy routing algorithms.

7.4.1 Message-Routing Schemes

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

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

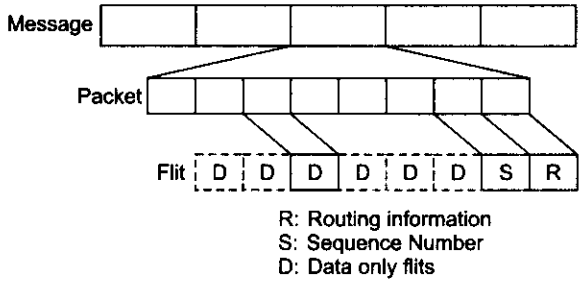


Fig. 7.26 The format of message, packets, and flits (control flow digits) used as information units of communication in a message-passing network

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

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

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

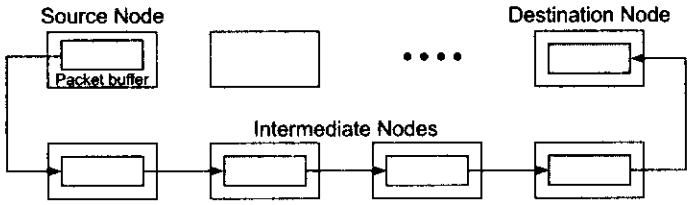
The packet length is determined by the routing scheme and network implementation. Typical packet lengths range from 64 to 512 bits. The sequence number may occupy one to two flits depending on the message length. Other factors affecting the choice of packet and flit sizes include channel bandwidth, router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

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.

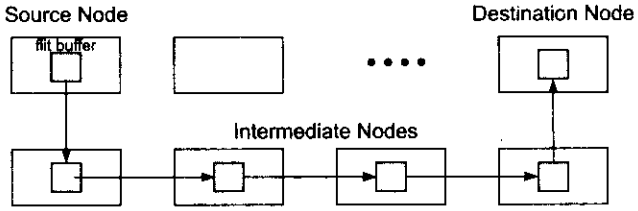
When a packet reaches an intermediate node, it is first stored in the buffer. Then it is forwarded to the next node if the desired output channel and a packet buffer in the receiving node are both available.

The latency in store-and-forward networks is directly proportional to the distance (the number of hops) between the source and the destination. This routing scheme was implemented in the first generation of multicomputers.

Wormhole Routing By subdividing the packet into smaller flits, latter generations of multicomputers implement the *wormhole routing* scheme, as illustrated in 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.



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



(b) Wormhole routing using flit buffers in successive routers

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

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 cars (data flits).

Only the header flit knows where the train (packet) is going. All the data flits (box cars) must follow the header flit. Different packets can be interleaved during transmission. However, the flits from different packets cannot be mixed up. Otherwise they may be towed to the wrong destinations.

We prove below that wormhole routing has a latency almost independent of the distance between the source and the destination.

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

When the receiving router (D) is ready (Fig. 7.28a) 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. 7.28d), the cycle repeats itself for the transmission of the next flit $i + 1$ until the entire packet is transmitted.

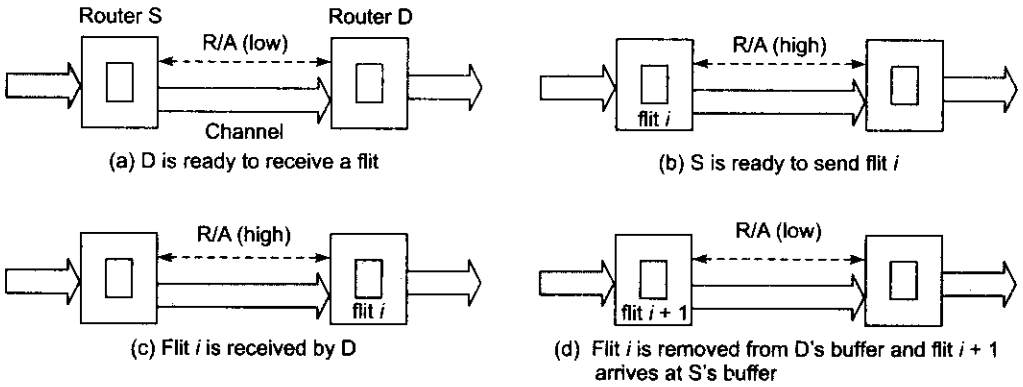


Fig. 7.28 Handshaking protocol between two wormhole routers (Courtesy of Lionel Ni, 1991)

Asynchronous pipelining can be very efficient, and the clock used can be faster than that used in a synchronous pipeline. However, the pipeline can be stalled if flit buffers or successive channels along the path are not available during certain cycles. Should that happen, the packet can be buffered, blocked, dragged, or detoured. We will discuss these flow control methods in Section 7.4.3.

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).

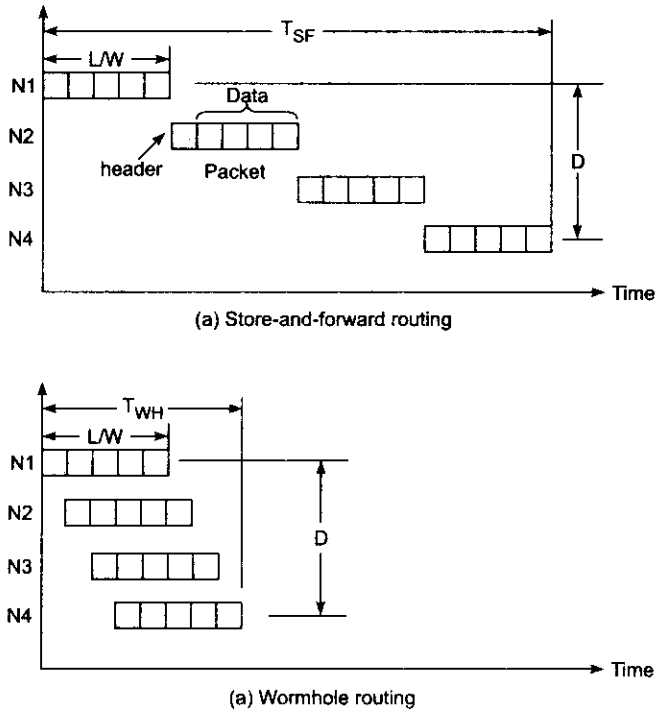


Fig. 7.29 Time comparison between the two routing techniques

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}.

According to the estimate given in Table 7.1, a typical first generation value of T_{SF} is between 2000 and 6000 μs, while a typical value of T_{WH} is 5 μs or less. Current systems employ much faster processors, data links and routers. Both the latency figures above would therefore be smaller, but wormhole routing would still have much lower latency than packet store-and-forward routing.

7.4.2 Deadlock and Virtual Channels

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

We introduce below the concept and explain its applications in avoiding deadlocks in this section and in facilitating network partitioning for multicasting in Section 7.4.4.

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.

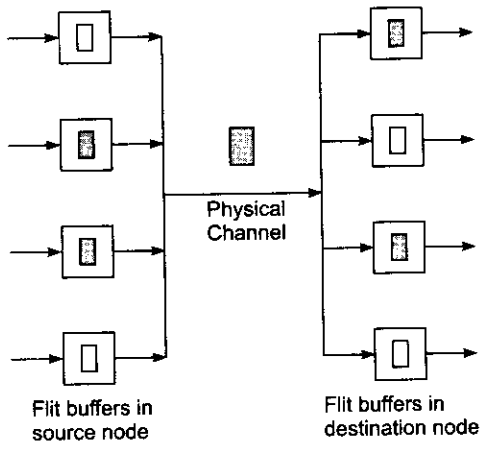
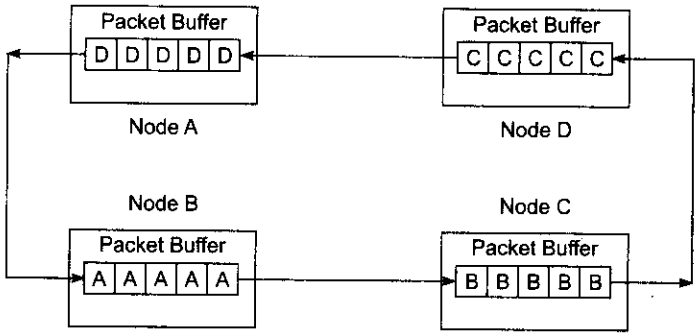


Fig. 7.30 Four virtual channels sharing a physical channel with time multiplexing on a flit-by-flit basis

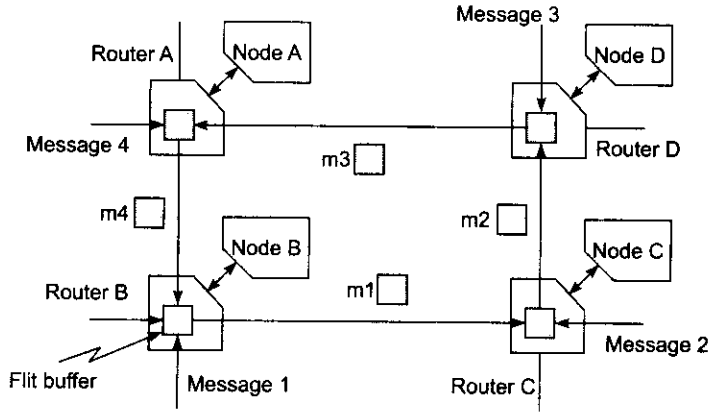
Comparing the setup in Fig. 7.30 with that in Fig. 7.28, the difference lies in the added buffers at both ends. The sharing of a physical channel by a set of virtual channels is conducted by time-multiplexing on a flit-by-flit basis.

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

As illustrated in Fig. 7.31, two types of deadlock situations are caused by a circular wait at buffers 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 buffers in four nodes. Unless one packet is discarded or misrouted, the deadlock cannot be broken. In Fig. 7.31b, a *channel deadlock* results from four messages being simultaneously transmitted along four channels in a mesh-connected network using wormhole routing.



(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

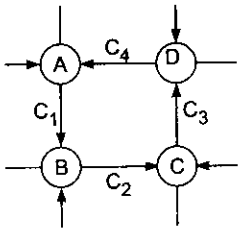
Fig. 7.31 Deadlock situations caused by a circular wait at buffers or at communication channels

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. Circular waits are further illustrated in Fig. 7.32 using a *channel-dependence graph.*

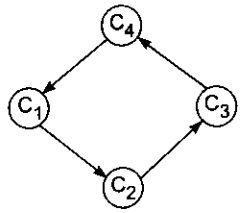
The channels involved are represented by nodes, and directed arrows are used to show the dependence relations among them. A deadlock avoidance scheme is presented using virtual channels.

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

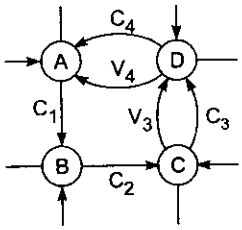
The cycle in Fig. 7.32b is being converted to a spiral, thus avoiding a deadlock. Channel multiplexing can be done at the flit level or at the packet level if the packet length is sufficiently short. Virtual channels can be implemented with either *unidirectional channels* or *bidirectional channels.*



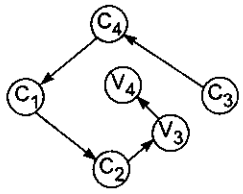
(a) Channel deadlock



(b) Channel-dependence graph containing a cycle



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



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

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

The use of virtual channels may reduce the effective channel bandwidth available to each request. There exists a tradeoff between network throughput and communication latency in determining the degree of using virtual channels. High-speed multiplexing is required for implementing a large number of virtual channels.

7.4.3 Flow Control Strategies

In this section, we examine various strategies developed to control smooth network traffic flow without causing congestion or deadlock situations. 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, we describe below deterministic and adaptive routing algorithms developed for one-to-one i.e. unicast communication.

Packet Collision Resolution In order to move a flit between adjacent nodes in a pipeline of channels, three elements must be 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 be done with the packet being denied the channel? These decisions lead to the four methods illustrated in Fig. 7.33 for coping with the packet collision problem.

Figure 7.33 illustrates four methods for resolving the conflict between two packets competing for the use of the same outgoing channel at an intermediate node. Packet 1 is being allocated the channel, and packet 2 being denied. A *buffering* method has been proposed with the *virtual cut-through routing* scheme devised by Kermani and Kleinrock (1979).

Packet 2 is temporarily stored in a packet buffer. When the channel becomes available later, it will be transmitted then. This buffering approach has the advantage of not wasting the resources already allocated. However, it requires the use of a large buffer to hold the entire packet.

Furthermore, the packet buffers along the communication path should not form a cycle as shown in Fig. 7.31a. The packet buffer however may cause significant storage delay. The virtual cut-through method offers a compromise by combining the store-and-forward and wormhole routing schemes. When collisions do not occur, the scheme should perform as well as wormhole routing. In the worst case, it will behave like a store-and-forward network.

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; however, it is not being abandoned. 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.

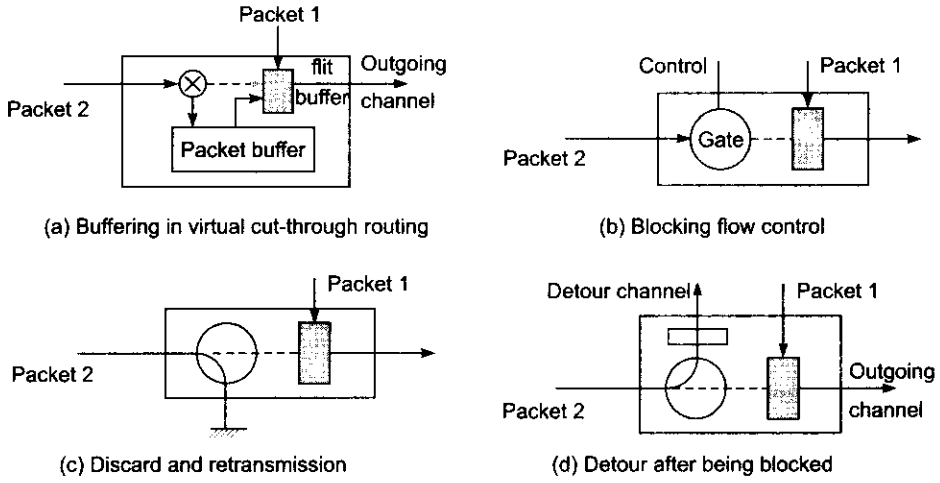


Fig. 7.33 Flow control methods for resolving a collision between two packets requesting the same outgoing channel (packet 1 being allocated the channel and packet 2 being denied)

The discard policy may result in a severe waste of resources, and it demands packet retransmission and acknowledgment. Otherwise, a packet may be lost after discarding. This policy is rarely used now because of its unstable packet delivery rate. The BBN Butterfly network had used this discard policy.

Detour routing offers more flexibility in packet routing. However, the detour may waste more channel resources than necessary to reach the destination. Furthermore, a re-routed packet may enter a cycle of *livelock*, which wastes network resources. Both the Connection Machine and the Denelcor HEP had used this detour policy.

In practice, some multicomputer networks use hybrid policies which may combine the advantages of some of the above flow control policies.

Dimension-Order Routing Packet routing can be conducted deterministically or adaptively. In *deterministic routing*, the communication path is completely determined by the source and destination addresses. In other words, the routing path is uniquely predetermined in advance, independent of network condition.

Adaptive routing may depend on network conditions, and alternate paths are possible. In both types of routing, deadlock-free algorithms are desired. Two such deterministic routing algorithms are given below, based on a concept called *dimension order routing*.

Dimension-order routing requires the selection of successive channels to follow a specific order based on the dimensions of a multidimensional network. In the case of a two-dimensional mesh network, the scheme is called *X-Y routing* because a routing path along the X-dimension is decided first before choosing a path along the Y-dimension. For hypercube (or n-cube) networks, the scheme is called *E-cube routing* as originally proposed by Sullivan and Bashkow (1977). These two routing algorithms are described below by presenting examples.

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 $r_4 = 1$.

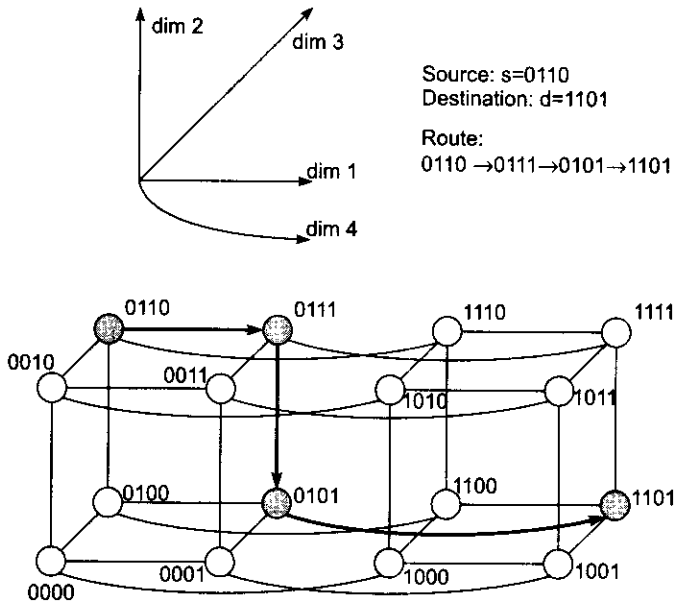


Fig. 7.34 E-cube routing on a hypercube computer with 16 nodes

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.

X-Y Routing on a 2D Mesh The same idea is applicable to mesh-connected networks. X-Y routing is illustrated by the example in Fig. 7.35. From any source node $s = (x_1y_1)$ to any destination node $d = (x_2y_2)$, route from s along the X-axis first until it reaches the column Y_2, where d is located. Then route to d along the Y-axis.

There are four possible X-Y routing patterns corresponding to the east-north, east-south, west-north, and west-south paths chosen.

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

Four (source, destination) pairs are shown in Fig. 7.35 to illustrate the four possible routing patterns on a two-dimensional mesh.

An east-north route is needed from 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.

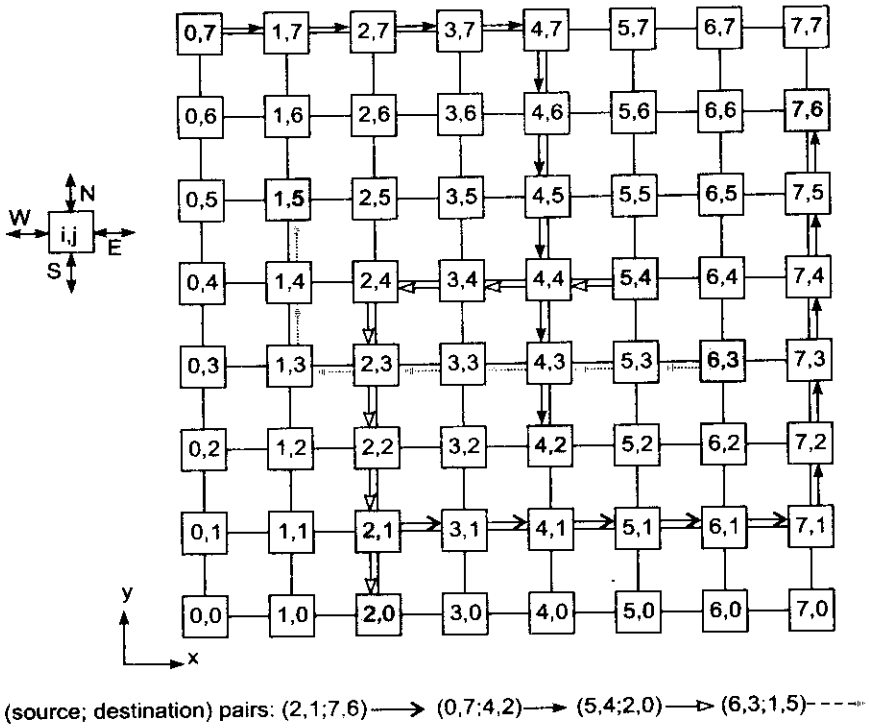
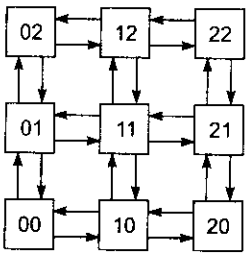


Fig. 7.35 X-Y routing on a 2D mesh computer with 8 × 8 = 64 nodes

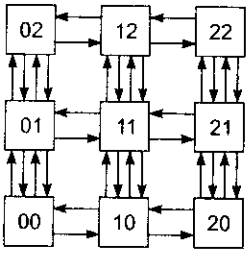
It is left as an exercise for the reader to prove that both E-cube and X-Y schemes result in deadlock-free routing. Both can be applied in either store-and-forward or wormhole-routed networks, resulting in a minimal route with the shortest distance between source and destination.

However, the same dimension order routing scheme cannot produce minimal routes for torus networks. Nonminimal routing algorithms, producing deadlock-free routes, allow packets to traverse through longer paths, sometimes to reduce network traffic or for other reasons.

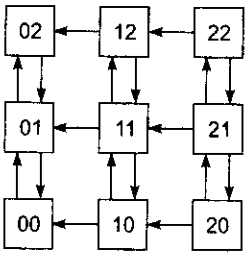
Adaptive Routing The main purpose of using adaptive routing is to achieve efficiency and avoid deadlock. The concept of virtual channels makes adaptive routing more economical and feasible to implement. We have shown in Fig. 7.32 how to apply virtual channels for this purpose. The idea can be further extended by having virtual channels in all connections along the same dimension of a mesh-connected network (Fig. 7.36).



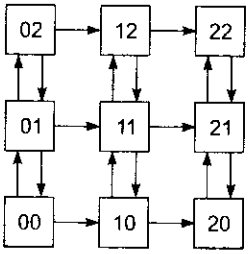
(a) Original mesh without virtual channel



(b) Two pairs of virtual channels in Y-dimension



(c) For a westbound message



(d) For an eastbound message

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)

Example 7.6 Adaptive X-Y routing using virtual channels

This example uses two pairs of virtual channels in the Y-dimension of a mesh using X-Y routing.

For westbound traffic, the *virtual network* 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.

The two virtual networks are used at different times; thus deadlock can be adaptively avoided. This concept will be further elaborated for achieving deadlockfree multicast routing in the next section.

7.4.4 Multicast Routing Algorithms

Various communication patterns are specified below. Routing efficiency is defined. The concept of virtual networks and network partitioning are applied to realize the complex communication patterns with efficiency.

Communication Patterns Four types of communication patterns may appear in multicomputer networks. What we have implemented in previous sections is the one-to-one unicast pattern with one source and one destination.

A *multicast* pattern 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.

In what follows, we consider the requirements for implementing multicast, broadcast, and conference communication patterns. Of course, all patterns can be implemented with multiple unicasts sequentially, or even simultaneously if resource conflicts can be avoided. Special routing schemes must be used to implement these multi-destination patterns.

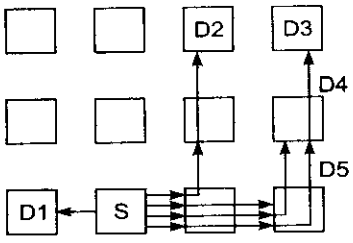
Routing Efficiency Two commonly used efficiency parameters are *channel bandwidth* and *communication latency*. The channel bandwidth at any time instant (or during any time period) indicates the effective data transmission rate achieved to deliver the messages. The latency is indicated by the packet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

An optimally routed network should achieve both maximum bandwidth and minimum latency for the communication patterns involved. However, these two parameters are not totally independent. Achieving maximum bandwidth may not necessarily achieve minimum latency at the same time, and vice versa.

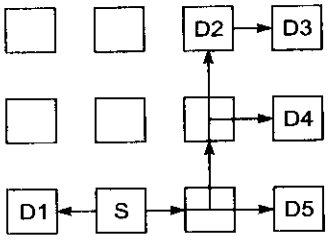
Depending on the switching technology used, latency is the more important issue in a store-and-forward network, while in general the bandwidth affects efficiency more in a wormhole-routed network.

Example 7.7 Multicast and broadcast on a mesh-connected computer

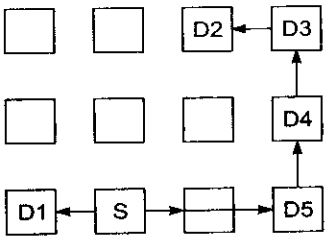
Multicast routing is implemented on a 3 × 3 mesh in Fig. 7.37. The source node is identified as S, which transmits a packet to five destinations labeled D_i for $i = 1, 2, ..., 5$.



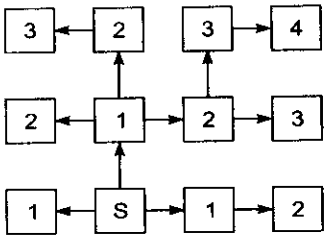
(a) Five unicasts with traffic = 13 and distance = 4



(b) A multicast pattern with traffic = 7 and distance = 4



(c) Another multicast pattern with traffic = 6 and distance = 5



(d) Broadcast to all nodes via a tree (numbers in nodes correspond to levels of the tree)

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

This five-destination multicast can be implemented by five unicasts, as shown in Fig. 7.37a. The X-Y routing traffic requires the use of 1 + 3 + 4 + 3 + 2 = 13 channels, and the latency is 4 for the longest path leading to D3.

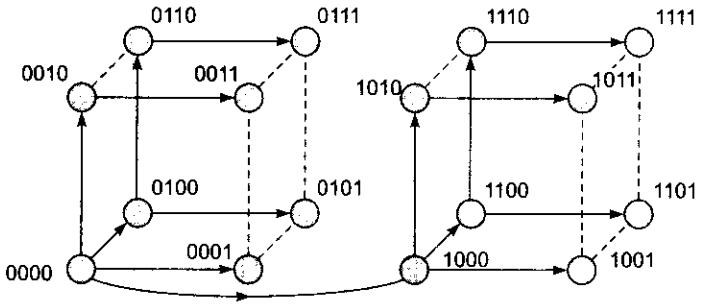
A multicast can be implemented by replicating the packet at an intermediate node, and multiple copies of the packet reach their destinations with significantly reduced channel traffic.

Two multicast routes are given in Figs. 7.37b and 7.37c, resulting in traffic of 7 and 6, respectively. On a wormhole-routed network, the multicast route in Fig. 7.37c is better. For a store-and-forward network, the route in Fig. 7.37b is better and has a shorter latency.

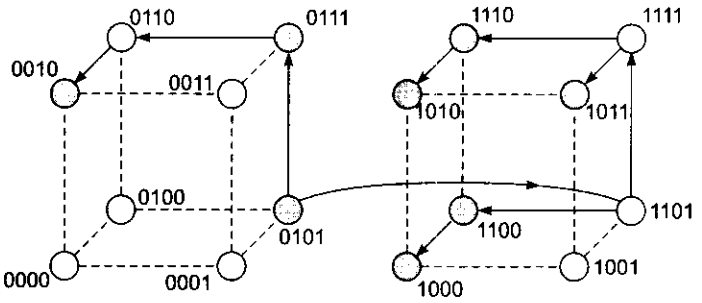
A four-level spanning tree is used from node S to broadcast a packet to all the mesh nodes in Fig. 7.37d. Nodes reached at level i of the tree have latency i. This broadcast tree should result in minimum latency as well as in minimum traffic.

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 $1110 \rightarrow 1010$.

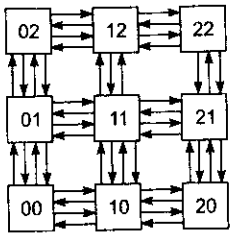
Extending the multicast tree, one should compare the reachability via all dimensions before selecting certain dimensions to obtain a minimum cover set for the remaining nodes. In case of a tie between two dimensions, selecting any one of them is sufficient. Therefore, the tree may not be uniquely generated.

It has been proved that this greedy multicast algorithm requires the least number of traffic channels compared with multiple unicasts or a broadcast tree. To implement multicast operations on wormhole-routed networks, the router in each node should be able to replicate the data in the flit buffer.

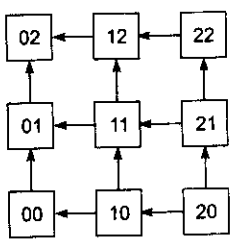
In order to synchronize the growth of a multicast tree or a broadcast tree, all outgoing channels at the same level of the tree must be ready before transmission can be pushed one level down. Otherwise, additional buffering is needed at intermediate nodes.

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

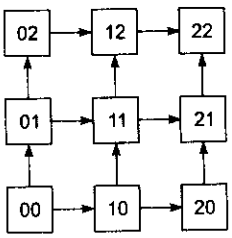
These virtual channels can be used to generate four possible virtual networks. For west-north traffic, the virtual network in Fig. 7.39b should be used.



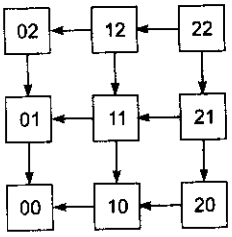
(a) A dual-channel 3 × 3 mesh



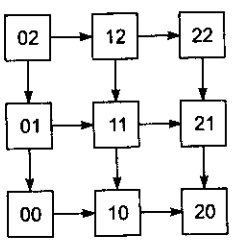
(b) West-north subnet



(c) East-north subnet



(d) West-south subnet



(e) East-south subnet

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

Similarly, one can construct three other virtual nets for other traffic orientations. Note that no cycle is possible on any of the virtual networks. Thus deadlock can be completely avoided when X-Y routing is implemented on these networks.

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 (e) or (c) and (d) can be used simultaneously.

Other combinations, such as (b) and (c), or (b) and (d), or (c) and (e), or (d) and (e), cannot coexist at the same time due to a shortage of channels.

Obviously, adding channels to the network will increase the adaptivity in making routing decisions. However, the increased cost can be appreciable and thus prevent the use of redundancy.

Network Partitioning The concept of virtual networks leads to the partitioning of a given physical network into logical subnetworks for multicast communications. The idea is illustrated in Fig. 7.40.

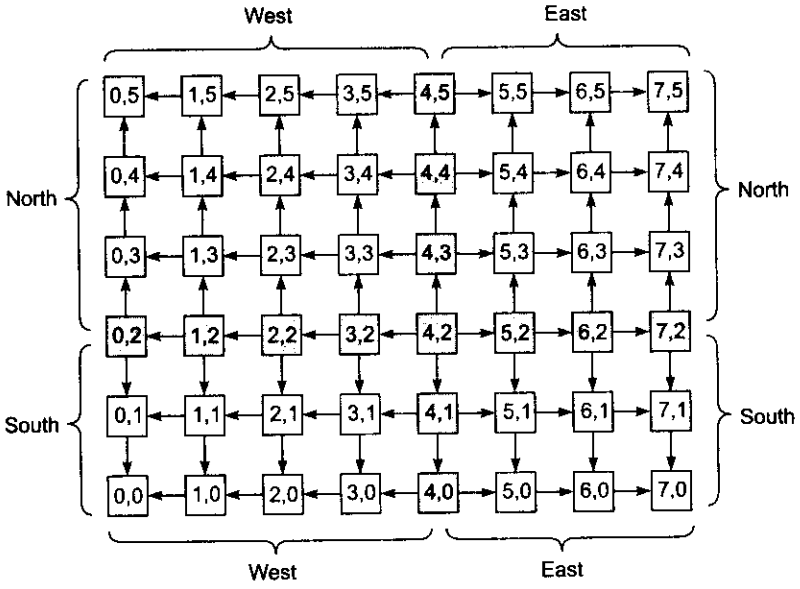


Fig. 7.40 Partitioning of a 6 × 8 mesh into four subnets for a multicast from source node (4,2). Shaded nodes are along the boundary of adjacent subnets (Courtesy of Lin, McKinly, and Ni, 1991)

Suppose source node (4, 2) wants to transmit to a subset of nodes in the 6×8 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.

Similarly, one can partition a binary n-cube into 2^{n-1} subcubes to provide deadlock-free adaptive routing. Each subcube has $n + 1$ levels with 2^n virtual channels per level for the bidirectional network. The number

of required virtual channels increases rapidly with n. It has been shown that for low-dimensional cubes (n = 2 to 4), this method is best for general-purpose routing.

Summary

In a multiprocessor system, interconnects between sub-systems such as processors, memories and network controllers play a crucial role in determining system performance. The earliest multiprocessor systems were bus-based, with shared main memory. The bus is a simple interconnect, but it has limitations in scalability. Hierarchical bus systems can address the problem to a limited extent, but as systems grow larger, more sophisticated and scalable system interconnects are needed.

A network may be of blocking or non-blocking type. We studied the crossbar network and the basic design of a row of crosspoint switches, with its arbitration and multiplexer modules. While it has better aggregate bandwidth than the bus, the crossbar network also has limitations of scalability. Multi-port memory can be used to enhance the aggregate bandwidth of a memory module.

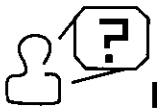
We studied Omega and Butterfly multistage networks. Larger Omega networks can be built using 2x2 and 4x4 basic switches, while the Butterfly network is built from modules of crossbar switches. When network traffic is non-uniform, so-called 'hot-spots' may develop which may degrade network performance. The concept of combining networks was developed in an attempt to address this performance limitation.

We studied the related issues of maintaining cache coherence and synchronization. Write operations on shared cache data, process migration and I/O operations can cause loss of cache coherence. If all the caches are on a common bus, then the snoopy bus protocol can be used to maintain cache coherence. Directory-based cache coherence protocols—using full map, limited or chained directories—can be used on more general types of system interconnects. Details of the schemes vary between write-back and write-through types of cache.

Hardware synchronization mechanisms between processors make use of atomic operations typified by Test&Set. However, at a still lower level of hardware, in theory wired barrier synchronization can also be used, of which we saw examples.

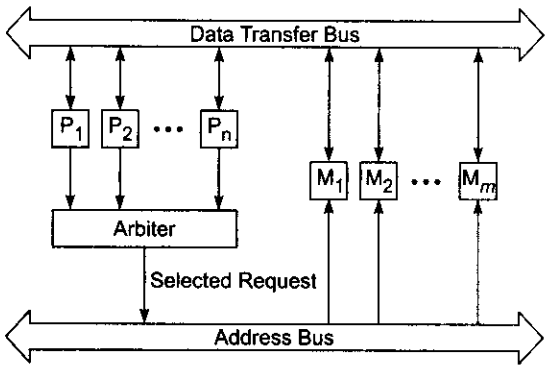
Three early generations of multicomputer systems were studied, providing a picture of how multicomputer architecture has evolved over time. Broadly, the trend has been from expensive to low cost processors, from shared to distributed memory, and (with higher speed processors) to higher speed interconnects. We studied the Intel Paragon system as a specific example, laying the basis to review more recent advances in Chapter 13.

Message-passing communication uses networks of point-to-point links, the basic aim of routing protocols being to achieve low network latency and high bandwidth. We studied the typical formats of messages, packets, and flits (flow control digits); routing schemes were studied from the points of view of latency analysis and the avoidance of deadlocks. We examined the important concepts of virtual channels, wormhole routing, flow control, collision resolution, dimension order routing, and multicast communication.



Exercises

Problem 7.1 Consider a multiprocessor with n processors and m shared-memory modules, all connected to the same backplane bus with a central arbiter as depicted below:



Assume $m > n$ and all memory modules are equally accessible to each processor. In other words, each processor generates a request for any module with probability $1/m$. The address bus and the DTB can be used at the same time to serve different requests. Both buses take one cycle to pass the address of a request or to transfer one word of 4 bytes between memory and processor. At each bus cycle (τ), the arbiter randomly selects one of the requests from the processors.

Once a memory module is identified at the end of the address cycle (one bus cycle), it takes a memory cycle (which equals c bus cycles) to retrieve the addressed word from the memory module, and another bus cycle to transfer the word to the requesting processor via the data transfer bus.

Until a memory cycle is completed, the arbiter will not issue another request to the same module. All rejected requests are ignored and resubmitted in subsequent bus cycles until being selected.

(a) Calculate the memory bandwidth defined as the average number of memory words transferred per second over the DTB if $n = 8$, $m = 16$, $\tau = 10$ ns, and $c \cdot \tau = 8\tau = 80$ ns.

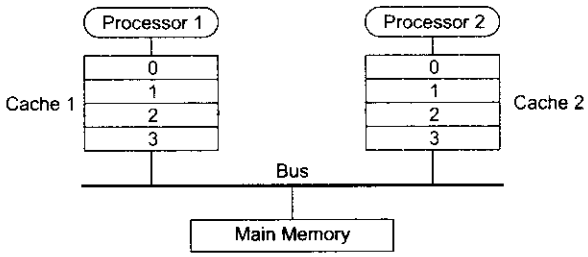
(b) Calculate the memory utilization defined as the average number of requests accepted by all memory modules per memory cycle using the same set of parameters used in part (a).

Problem 7.2 Use two-input AND and OR gates (no wired-OR) to construct an $n \times n$ crossbar switch network between n processors and n memory modules. Let the width of each crosspoint be w bits (or a word) in each direction.

(a) Prepare a schematic design of a typical crosspoint switch using c_{ij} as the enable signal for the switch in the ith row and jth column. Estimate the total number of AND and OR gates needed as a function of n and w.

(b) Assume that processor P_i has higher priority over processor P_j if $i < j$ when they are competing for access to the same memory module. Let $k = \log_2 n$ be the address width. Design an arbiter which generates all the crosspoint enable signals c_{ij}, again using only two-input AND and OR gates and some inverters if needed. The memory address decoder is assumed available from each processor and thus is not included in the arbiter design. Indicate the complexity of the arbiter design as a function of n and k.

Problem 7.3 Consider a dual-processor (P1 and P2) system using write-back private caches and a shared memory, all connected to a common contention bus. Each cache has four block frames labeled below as 0, 1, 2, 3.



The shared memory is divided into eight cache blocks as 0, 1, ..., 7. To maintain cache coherence, the system uses a three-state (RO, RW, and invalid) snoopy protocol based on the write-invalidate policy described in Fig. 7.12b.

Assume the same clock drives the processors and the memory bus. Within each cycle, any processor can submit a request to access the bus. In case of simultaneous bus requests from both processors, the request from P1 is granted and P2 must wait one or more cycles to access the bus.

In all cases, the bus allows only one transaction per cycle. Once a bus access is granted, the transaction must be completed before the next request is granted. When there is no bus contention, memory-access events from each processor may require one to two cycles to complete, as specified below separately:

- Read-hit in cache requires one cycle and no bus request at all.
- Read-miss in cache requires two cycles without contention: one for block fetch and one for CPU read from cache.
- Write-hit requires one cycle for CPU write and bus invalidation simultaneously.
- Write-miss requires two cycles: one for block fetch and bus invalidation, and one for CPU write.
- Replacement of a dirty block requires one cycle to update memory via the bus.

(a) In the case of bus contention, one additional cycle is needed for bus arbitration in all the above cases except a read-hit.

(i) Show how to map the eight cache blocks to four cache block frames using a direct-mapping cache organization.

(ii) Show how to map the eight cache block frames using a two-way set-associative cache organization.

(b) Consider the following two asynchronous sequences of memory-access events, where boldface numbers are for write and the remaining are for read.

Processor #1 : 0,**0**,0,**1**,1,4,3,3,5,**5**,5

Processor #2 : 2,**2**,0,0,**7**,5,5,5,**7**,7,0

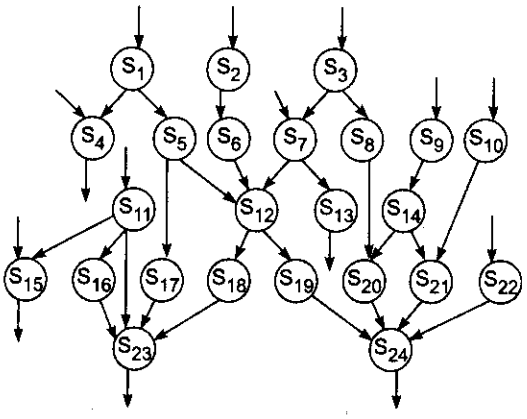
(i) Trace the execution of these two sequences on the two processors by executing the successive blocks. Both caches are initially flushed (empty). Assume a direct-mapping organization in both caches. Indicate the state (RO or RW) of each valid cache block and mark cache miss and bus utilization (busy or idle) in the block trace for each cycle. Assume that the very first memory-access events from both processors take place in cycle 1 simultaneously. Calculate the hit ratio of cache 1 and cache 2, respectively.

(ii) Assume a two-way set-associative cache organization and a LRU cache block replacement policy.

Problem 7.4 Consider the execution of 24 code segments, S_1 through S_{24}, following a given precedence graph on a multiprocessor with four processors and six memory modules as shown below. Assume all segments have the same gain size and execute with equal time. When two or more processors try to access the same memory module at the same time, the request of the lowest numbered processor is granted and the rest of the requests are deferred to later segment time steps.

A processor waiting from an earlier memory-access rejection has seniority priority over new requests to access the same memory module. No processor should wait for more than three steps

to access any given memory module. Each code segment takes a fixed unit time to access a memory and to execute. Assume that the four processors are synchronized in each segment execution instruction cycle.



In some cases, a single segment may require access to several memory modules simultaneously. Ignore the contention problem in the interconnection network. The four processors operate in MIMD mode, and different instructions can be executed by different processors during the same cycle.

What is the average memory bandwidth in words per unit time? Try to achieve the minimum execution time by maximizing the degree of parallelism at all steps.

Note that at each step some of the memory modules may be idle. The highest possible memory bandwidth is six words per step. Some segments may require a wait of no more than three steps before granting of the memory access requested. But such a waiting period should be minimized.

	Processor			
Instr.	P_1	P_2	P_3	P_4
S_1	M_1		M_5	M_1
S_2	M_1	M_2	M_2	M_2
S_3			M_3	M_3
S_4	M_5	M_3	M_2	M_4
S_5	M_1	M_6	M_2	
S_6		M_2	M_1	M_3
S_7			M_6	M_5
S_8		M_2	M_3	
S_9	M_3	M_4	M_4	
S_{10}	M_1	M_3	M_4	M_4
S_{11}	M_2	M_4	M_5	M_1
S_{12}		M_2	M_6	M_5
S_{13}		M_1		M_6
S_{14}		M_4	M_5	M_4
S_{15}	M_3	M_3	M_3	
S_{16}	M_2	M_2	M_2	M_4
S_{17}	M_1			
S_{18}		M_2	M_5	
S_{19}	M_2	M_2	M_2	M_1
S_{20}		M_3	M_3	M_4
S_{21}	M_2			M_4
S_{22}	M_3	M_1		M_6
S_{23}	M_1	M_2	M_5	M_3
S_{24}			M_3	M_4

Problem 7.5 This problem is based on Fig. 7.11 which combines multiple Fetch&Add requests to the same shared variable in a common memory.

(a) Show the necessary combining network components needed to combine four Fetch&Add (x, e_i) for $i = 1, 2, 3, 4$.

(b) Show the successive snapshots and variations in switch and memory contents, as in Fig. 7.11, for combining the four requests.

Problem 7.6 You have learned about a two-way shuffle (perfect shuffle) in Fig. 2.14 and a four-way shuffle in Fig. 7.9. Generalize the mappings to an m-way shuffle over n objects, where $m \times k = n$ for some integer $k \geq 2$, for the construction of the class of Delta networks introduced by Patel (1980).

(a) Show how to perform a four-way shuffle over 12 objects.

(b) Use a minimum number of 4×3 switch modules and a four-way shuffle mapping as an interstage connection pattern to build a 64-input, 27-output Delta network in three stages.

(c) In general, an n-stage $a^n \times b^n$ Delta network is implemented with $a \times b$ switch modules as shown in Fig. 2.23. Calculate the total number of switch modules needed and specify the

interstage connection pattern from b^n inputs to a^n outputs.

(d) Figure out a simple routing scheme to control the switch settings from stage to stage in an $a^n \times b^n$ Delta network with n stages.

(e) What is the relationship between Omega networks and Delta networks?

Problem 7.7 Prove the following properties associated with multistage Omega networks using different-sized building blocks:

(a) Prove that the number of legitimate states (connections) in a $k \times k$ switch module equals k^k.

(b) Determine the percentage of permutations that can be realized in one pass through a 64-input Omega network built with 2×2 switch modules.

(c) Repeat part (b) for a 64-input Omega network built with 8×8 switch modules.

(d) Repeat part (b) for a 512-input Omega network built with 8×8 switch modules.

Problem 7.8 Consider the interleaved execution of k programs in a multiprogrammed multiprocessor using m wired-NOR synchronization lines on n processors as described in Fig. 7.19a.

In general, the number m_i of barrier lines needed for a program i is estimated as $m_i = b_i [q_i/P_i] + 1$, where b_i = the number of barriers demanded in program i, q_i = the number of processes created in program i, and P_i = the number of processors allocated to program i.

Thus $m = m_1 + m_2 + \ldots + m_k$. For simplicity, assume $b_i = b$ and $q_i = q$ for $i = 1, 2, \ldots, k$, and $P_i = \min(n/k, q)$ processors are allocated to each program i.

Prove that m can be approximated by $b \cdot q \cdot k^2/n + k$, or that the degree of multiprogramming is $k \leq \left(-n + \sqrt{n^2 + 4bqmn}\right)/(2bq)$ in such a multiprocessor system. Note that bq represents the number of required synchronization points, which depends on the parallelism profiles in user programs. For fixed values of bq and n, the maximally allowed multiprogramming degree k increases with respect to $\sqrt{m}$.

Problem 7.9 Wilson (1987) proposed a hierarchical cache/bus architecture (Fig. 7.3) and outlined how multilevel cache coherence can be enforced by extending the write-invalidate protocol. Can you figure out a write-broadcast protocol for achieving multilevel cache coherence on the same hardware platform? Comment on the relative merits of the two protocols. Feel free to modify the hardware in Fig. 7.3 if needed to implement the write-broadcast protocol on the hierarchical bus/cache architecture.

Problem 7.10 Answer the following questions on design choices of multicomputers made in the past;

(a) Why were low-cost processors chosen over expensive processors as processing nodes?

(b) Why was distributed memory chosen over shared memory?

(c) Why was message passing chosen over address switching?

(d) Why was MIMD, MPMD, or SPMD control chosen over SIMD data parallelism?

Problem 7.11 Explain the following terms associated with multicomputer networks and message-passing mechanisms:

(a) Message, packets, and flits.

(b) Store-and-forward routing at packet level.

(c) Wormhole routing at flit level.

(d) Virtual channels versus physical channels.

(e) Buffer deadlock versus channel deadlock.

(f) Buffering flow control using virtual cut-through routing.

(g) Blocking flow control in wormhole routing.

(h) Discard and retransmission flow control.

(i) Detour flow control after being blocked.

(j) Virtual networks and subnetworks.

Problem 7.12

(a) Draw a 16-input Omega network using 2×2 switches as building blocks.

(b) Show the switch settings for routing a message from node 1011 to node 0101 and from node 0111 too node 1001 simultaneously. Does blocking exist in this case?

(c) Determine how many permutations can be implemented in one pass through this Omega network. What is the percentage of one-pass permutations among all permutations?

(d) What is the maximum number of passes needed to implement any permutation through the network?

Problem 7.13 Explain the following terms as applied to communication patterns in a message-passing network:

(a) Unicast versus multicast

(b) Broadcast versus conference

(c) Channel bandwidth

(d) Communication latency

(e) Network partitioning for multicasting communications

Problem 7.14 Determine the optimal routing paths in the following mesh and hypercube multicomputers.

(a) Consider a 64-node hypercube network. Based on the E-cube routing algorithm, show how to route a message from node (101101) to node (011010). All intermediate nodes must be identified on the routing path.

(b) Determine two optimal routes for multicast on an 8×8 mesh, subject to the following constraints separately. The source node is (3, 5), and there are 10 destination nodes (1, 1), (1,2), (1,6), (2, 1), (4, 1), (5, 5), (5, 7), (6, 1), (7, 1), (7, 5). (i) The first multicast route should be implemented with a minimum number of channels. (ii) The second multicast route should result in minimum distances from the source to each of the 10 destinations.

(c) Based on the greedy algorithm (Fig. 7.38), determine a suboptimal multicast route, with minimum distances from the source to all destinations using as few traffic channels as possible, on a 16-node hypercube network. The source node is (1010), and there are 9 destination nodes (0000), (0001), (0011), (0100), (0101), (0111), (1111), (1101), and (1001).

Problem 7.15 Prove the following statements with reasoning or analysis or counter-examples:

(a) Prove that E-cube routing is deadlock-free on a wormhole-routed hypercube with a pair of opposite unidirectional channels between adjacent nodes.

(b) Prove that X-Y routing is deadlock-free on a 2D mesh.

(c) Prove that E-cube routing on the 3D mesh (k-ary n-cube) used in the J-Machine is deadlock-free with wormhole routing and blocking flow control.

Problem 7.16 Study the Turn model for adaptive routing proposed by Glass and Ni (1992) in the 1992 *Annual International Symposium on Computer Architecture*. Answer the following questions:

(a) Why is the Turn model deadlock-free from having cycles?

(b) How can the Turn model be applied on an n-dimensional mesh to prevent deadlock?

(c) How can the Turn model be applied on a k-ary n-cube to prevent deadlock?

Problem 7.17 The following assignments are related to the greedy algorithm for multicast routing on a wormhole-routed hypercube network.

(a) Formulate the successive steps of the greedy algorithm (Example 7.8) as a minimum cover problem, similar to that practiced in Karnaugh maps.

(b) Prove that the greedy algorithm always yields the minimum network traffic and minimum distance from the source to any of the destinations.

Problem 7.18 Consider the implementation of Goodman's write-once cache coherence protocol in a bus-connected multiprocessor system. Specify the use of additional bus lines to inhibit the main memory when the memory copy is invalid. Also specify all other hardware mechanisms and software support needed for an economical and fast implementation of the Goodman protocol.

Explain why this protocol will reduce bus traffic and how unnecessary invalidations can be eliminated. Consult if necessary the two related papers published by Goodman in 1983 and 1990.

Problem 7.19 Study the paper by Archibald and Baer (1986) which evaluated various cache coherence protocols using a multiprocessor simulation model. Explain the Dragon protocol implemented in the Dragon multiprocessor workstation at the Xerox Palo Alto Research Center. Compare the relative merits of the Goodman protocol, the Firefly protocol, and the Dragon protocol in the context of implementation requirements and expected performance.

Problem 7.20 The Cedar multiprocessor at Illinois was built with a clustered Omega network as shown below. Four 8×4 crossbar switches were used in the first stage and four 4×8 crossbar switches were used in the second stage. There were 32 processors and 32 memory modules, divided into four clusters with eight of each per cluster.

(a) Figure out a fixed priority scheme to avoid conflicts in using the crossbar switches for nonblocking connections. For simplicity, consider only the forward connections from the processors to the memory modules.

(b) Suppose both stages use 8×8 crossbar switches. Design a two-stage Cedar network to provide switched connections between 64 processors and 64 memory modules, again in a clustered manner similar to the above Cedar network design.

(c) Further expand the Cedar network to three stages using 8×8 crossbar switches as building blocks to connect 512 processors and 512 memory modules. Show the schematic interconnections in all three stages from the input end to the output end.