IBM 3090 Vector Facility

A good example of a pipelined ALU organization for vector processing is the vector facility developed for the IBM 370 architecture and implemented on the high-end 3090 series [PADE88, TUCK87]. This facility is an optional add-on to the basic system but is highly integrated with it. It resembles vector facilities found on supercomputers, such as the Cray family.

The IBM facility makes use of a number of vector registers. Each register is actually a bank of scalar registers. To compute the vector sum $C = A + B$, the vectors A and B are loaded into two vector registers. The data from these registers are passed through the ALU as fast as possible, and the results are stored in a third vector register. The computation overlap, and the loading of the input data into the registers in a block, results in a significant speeding up over an ordinary ALU operation.

Organization The IBM vector architecture, and similar pipelined vector ALUs, provides increased performance over loops of scalar arithmetic instructions in three ways:

- The fixed and predetermined structure of vector data permits housekeeping instructions inside the loop to be replaced by faster internal (hardware or microcoded) machine operations.
- Data-access and arithmetic operations on several successive vector elements can proceed concurrently by overlapping such operations in a pipelined design or by performing multiple-element operations in parallel.
- The use of vector registers for intermediate results avoids additional storage reference.

Figure 18.18 shows the general organization of the vector facility. Although the vector facility is seen to be a physically separate add-on to the processor, its architecture

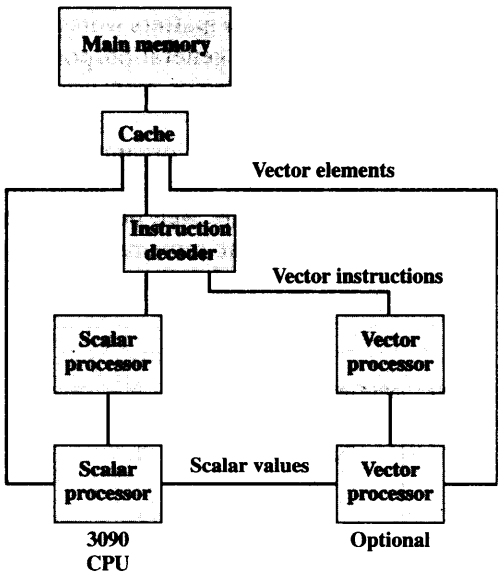


Figure 18.18 IBM 3090 with Vector Facility

Instructions 2 and 3 can be chained (pipelined) because they involve different memory locations and registers. Instruction 4 needs the results of instructions 2 and 3, but it can be chained with them as well. As soon as the first elements of vector registers 2 and 3 are available, the operation in instruction 4 can begin.

Another way to achieve vector processing is by the use of multiple ALUs in a single processor, under the control of a single control unit. In this case, the control unit routes data to ALUs so that they can function in parallel. It is also possible to use pipelining on each of the parallel ALUs. This is illustrated in Figure 18.16b. The example shows a case in which four ALUs operate in parallel.

As with pipelined organization, a parallel ALU organization is suitable for vector processing. The control unit routes vector elements to ALUs in a round-robin fashion until all elements are processed. This type of organization is more complex than a single-ALU CPI.

Finally, vector processing can be achieved by using multiple parallel processors. In this case, it is necessary to break the task up into multiple processes to be executed in parallel. This organization is effective only if the software and hardware for effective coordination of parallel processors is available.

We can expand our taxonomy of Section 18.1 to reflect these new structures, as shown in Figure 18.17. Computer organizations can be distinguished by the presence of one or more control units. Multiple control units imply multiple processors. Following our previous discussion, if the multiple processors can function cooperatively on a given task, they are termed *parallel processors.*

The reader should be aware of some unfortunate terminology likely to be encountered in the literature. The term *vector processor* is often equated with a pipelined ALU organization, although a parallel ALU organization is also designed for vector processing, and, as we have discussed, a parallel processor organization may also be designed for vector processing. *Array processing* is sometimes used to refer to a parallel ALU, although, again, any of the three organizations is optimized for the processing of arrays. To make matters worse, *array processor* usually refers to an auxiliary processor attached to a general-purpose processor and used to perform vector computation. An array processor may use either the pipelined or parallel ALU approach.

At present, the pipelined ALU organization dominates the marketplace. Pipelined systems are less complex than the other two approaches. Their control unit and operating system design are well developed to achieve efficient resource allocation and high performance. The remainder of this section is devoted to a more detailed examination of this approach, using a specific example.

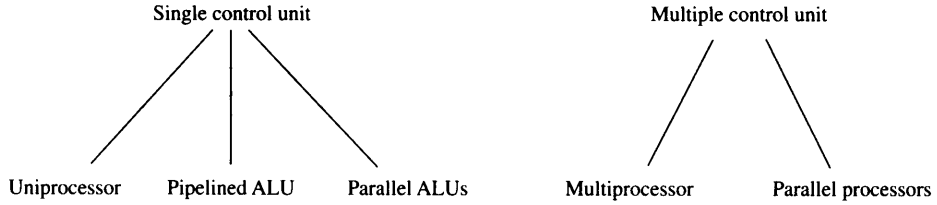


Figure 18.17 A Taxonomy of Computer Organizations

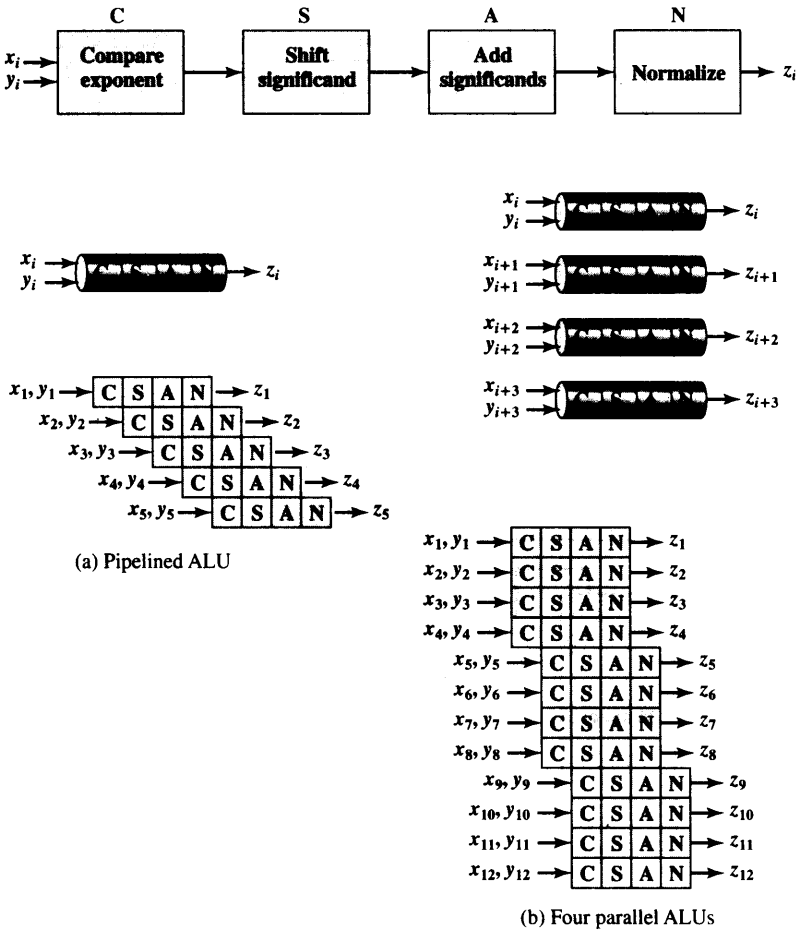
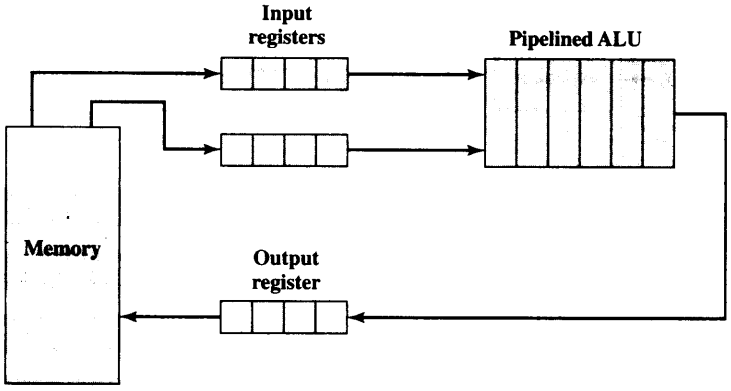


Figure 18.16 Pipelined Processing of Floating-Point Operations

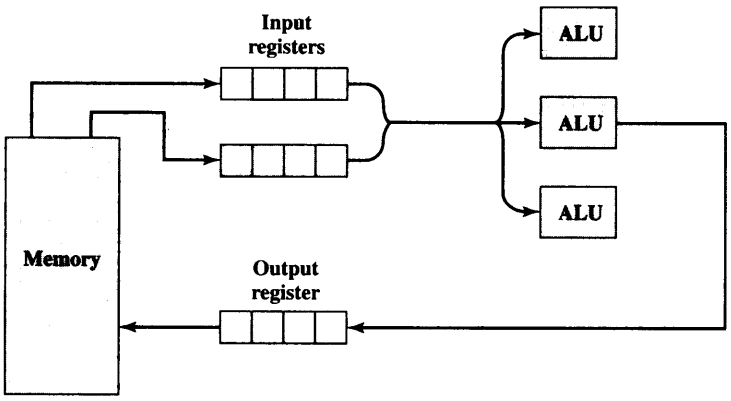
free. Essentially, chaining causes results issuing from one functional unit to be fed immediately into another functional unit and so on. If vector registers are used, intermediate results do not have to be stored into memory and can be used even before the vector operation that created them runs to completion.

For example, when computing $C = (s \times A) + B$, where A, B, and C are vectors and s is a scalar, the Cray may execute three instructions at once. Elements fetched for a load immediately enter a pipelined multiplier, the products are sent to a pipelined adder, and the sums are placed in a vector register as soon as the adder completes them:

1. Vector load — A → Vector Register (VR1)
2. Vector load — B → VR2
3. Vector multiply — s × VR1 → VR3
4. Vector add — VR3 + VR2 → VR4
5. Vector store — VR4 → C



(a) Pipelined ALU



(b) Parallel ALUs

Figure 18.15 Approaches to Vector Computation

The pipeline operation can be further enhanced if the vector elements are available in registers rather than from main memory. This is in fact suggested by Figure 18.15a. The elements of each vector operand are loaded as a block into a vector register, which is simply a large bank of identical registers. The result is also placed in a vector register. Thus, most operations involve only the use of registers, and only load and store operations and the beginning and end of a vector operation require access to memory.

The mechanism illustrated in Figure 18.16 could be referred to as *pipelining within an operation.* That is, we have a single arithmetic operation (e.g., $C = A + B$) that is to be applied to vector operands, and pipelining allows multiple vector elements to be processed in parallel. This mechanism can be augmented with *pipelining across operations.* In this latter case, there is a sequence of arithmetic vector operations, and instruction pipelining is used to speed up processing. One approach to this, referred to as **chaining**, is found on the Cray supercomputers. The basic rule for chaining is this: A vector operation may start as soon as the first element of the operand vector(s) is available and the functional unit (e.g., add, subtract, multiply, divide) is

summations (across K) are done serially rather than in parallel. Even so, only $N2$ vector multiplications are required for this algorithm as compared with $N3$ scalar multiplications for the scalar algorithm.

Another approach, *parallel processing,* is illustrated in Figure 18.14c. This approach assumes that we have N independent processors that can function in parallel. To utilize processors effectively, we must somehow parcel out the computation to the various processors. Two primitives are used. The primitive FORK n causes an independent process to be started at location n. In the meantime, the original process continues execution at the instruction immediately following the FORK. Every execution of a FORK spawns a new process. The JOIN instruction is essentially the inverse of the FORK. The statement JOIN N causes N independent processes to be merged into one that continues execution at the instruction following the JOIN. The operating system must coordinate this merger, and so the execution does not continue until all N processes have reached the JOIN instruction.

The program in Figure 18.14c is written to mimic the behavior of the vector-processing program. In the parallel processing program, each column of C is computed by a separate process. Thus, the elements in a given row of C are computed in parallel.

The preceding discussion describes approaches to vector computation in logical or architectural terms. Let us turn now to a consideration of types of processor organization that can be used to implement these approaches. A wide variety of organizations have been and are being pursued. Three main categories stand out:

- Pipelined ALU
- Parallel ALUs
- Parallel processors

Figure 18.15 illustrates the first two of these approaches. We have already discussed pipelining in Chapter 12. Here the concept is extended to the operation of the ALU. Because floating-point operations are rather complex, there is opportunity for decomposing a floating-point operation into stages, so that different stages can operate on different sets of data concurrently. This is illustrated in Figure 18.16a. Floating-point addition is broken up into four stages (see Figure 9.22): compare, shift, add, and normalize. A vector of numbers is presented sequentially to the first stage. As the processing proceeds, four different sets of numbers will be operated on concurrently in the pipeline.

It should be clear that this organization is suitable for vector processing. To see this, consider the instruction pipelining described in Chapter 12. The processor goes through a repetitive cycle of fetching and processing instructions. In the absence of branches, the processor is continuously fetching instructions from sequential locations. Consequently, the pipeline is kept full and a savings in time is achieved. Similarly, a pipelined ALU will save time only if it is fed a stream of data from sequential locations. A single, isolated floating-point operation is not speeded up by a pipeline. The speedup is achieved when a vector of operands is presented to the ALU. The control unit cycles the data through the ALU until the entire vector is processed.

$C = A \times B$, where A, B, and C are $N \times N$ matrices. The formula for each element of C is

$$c_{i,j} = \sum_{k=1}^{N} a_{i,k} \times b_{k,j}$$

where A, B, and C have elements $a_{i,j}$, $b_{i,j}$, and $c_{i,j}$, respectively. Figure 18.14a shows a FORTRAN program for this computation that can be run on an ordinary scalar processor.

One approach to improving performance can be referred to as *vector processing*. This assumes that it is possible to operate on a one-dimensional vector of data. Figure 18.14b is a FORTRAN program with a new form of instruction that allows vector computation to be specified. The notation (J = 1, N) indicates that operations on all indices J in the given interval are to be carried out as a single operation. How this can be achieved is addressed shortly.

The program in Figure 18.14b indicates that all the elements of the ith row are to be computed in parallel. Each element in the row is a summation, and the

```
      DO 100 I = 1, N
      DO 100 J = 1, N
      C(I, J) = 0.0
      DO 100 K = 1, N
      C(I, J) = C(I, J) + A(I, K) + B(K, J)
100   CONTINUE
```

(a) Scalar processing

```
      DO 100 I = 1, N
      C(I, J) = 0.0 (J = 1, N)
      DO 100 K = 1, N
      C(I, J) = C(I, J) + A(I, K) + B(K, J) (J = 1, N)
100   CONTINUE
```

(b) Vector processing

```
      DO 50 J = 1, N - 1
      FORK 100
50    CONTINUE
      J = N
100   DO 200 I = 1, N
      C(I, J) = 0.0
      DO 200 K = 1, N
      C(I, J) = C(I, J) + A(I, K) + B(K, J)
200   CONTINUE
```

(c) Parallel processing

Figure 18.14 Matrix Multiplication (C = A × B)

rocket). This surface is approximated by a grid of points. A set of differential equations defines the physical behavior of the surface at each point. The equations are represented as an array of values and coefficients, and the solution involves repeated arithmetic operations on the arrays of data.

Supercomputers were developed to handle these types of problems. These machines are typically capable of hundreds of millions of floating-point operations per second and cost in the 10- to 15-million-dollar range. In contrast to mainframes, which are designed for multiprogramming and intensive I/O, the supercomputer is optimized for the type of numerical calculation just described.

The supercomputer has limited use and, because of its price tag, a limited market. Comparatively few of these machines are operational, mostly at research centers and some government agencies with scientific or engineering functions. As with other areas of computer technology, there is a constant demand to increase the performance of the supercomputer. Thus, the technology and performance of the supercomputer continues to evolve.

There is another type of system that has been designed to address the need for vector computation, referred to as the *array processor*. Although a supercomputer is optimized for vector computation, it is a general-purpose computer, capable of handling scalar processing and general data processing tasks. Array processors do not include scalar processing; they are configured as peripheral devices by both mainframe and minicomputer users to run the vectorized portions of programs.

Approaches to Vector Computation

The key to the design of a supercomputer or array processor is to recognize that the main task is to perform arithmetic operations on arrays or vectors of floating-point numbers. In a general-purpose computer, this will require iteration through each element of the array. For example, consider two vectors (one-dimensional arrays) of numbers, A and B. We would like to add these and place the result in C. In the example of Figure 18.13, this requires six separate additions. How could we speed up this computation? The answer is to introduce some form of parallelism.

Several approaches have been taken to achieving parallelism in vector computation. We illustrate this with an example. Consider the vector multiplication

$$\begin{bmatrix} 1.5 \\ 7.1 \\ 6.9 \\ 100.5 \\ 0 \\ 59.7 \end{bmatrix} + \begin{bmatrix} 2.0 \\ 39.7 \\ 1000.003 \\ 11 \\ 21.1 \\ 19.7 \end{bmatrix} = \begin{bmatrix} 3.5 \\ 46.8 \\ 1006.093 \\ 111.5 \\ 21.1 \\ 79.4 \end{bmatrix}$$

$$A + B = C$$

Figure 18.13 Example of Vector Addition

any local cache has that line and, if so, cause it to be purged. If the actual memory location is at the node receiving the broadcast notification, then that node's directory needs to maintain an entry indicating that that line of memory is invalid and remains so until a write back occurs. If another processor (local or remote) requests the invalid line, then the local directory must force a write back to update memory before providing the data.

NUMA Pros and Cons

The main advantage of a CC-NUMA system is that it can deliver effective performance at higher levels of parallelism than SMP, without requiring major software changes. With multiple NUMA nodes, the bus traffic on any individual node is limited to a demand that the bus can handle. However, if many of the memory accesses are to remote nodes, performance begins to break down. There is reason to believe that this performance breakdown can be avoided. First, the use of L1 and L2 caches is designed to minimize all memory accesses, including remote ones. If much of the software has good temporal locality, then remote memory accesses should not be excessive. Second, if the software has good spatial locality, and if virtual memory is in use, then the data needed for an application will reside on a limited number of frequently used pages that can be initially loaded into the memory local to the running application. The Sequent designers report that such spatial locality does appear in representative applications [LOVE96]. Finally, the virtual memory scheme can be enhanced by including in the operating system a page migration mechanism that will move a virtual memory page to a node that is frequently using it; the Silicon Graphics designers report success with this approach [WHIT97].

There are disadvantages to the CC-NUMA approach as well. Two in particular are discussed in detail in [PFIS98]. First, a CC-NUMA does not transparently look like an SMP; software changes will be required to move an operating system and applications from an SMP to a CC-NUMA system. These include page allocation, already mentioned, process allocation, and load balancing by the operating system. A second concern is that of availability. This is a rather complex issue and depends on the exact implementation of the CC-NUMA system; the interested reader is referred to [PFIS98].

18.7 VECTOR COMPUTATION

Although the performance of mainframe general-purpose computers continues to improve relentlessly, there continue to be applications that are beyond the reach of the contemporary mainframe. There is a need for computers to solve mathematical problems of physical processes, such as occur in disciplines including aerodynamics, seismology, meteorology, and atomic, nuclear, and plasma physics.

Typically, these problems are characterized by the need for high precision and a program that repetitively performs floating-point arithmetic operations on large arrays of numbers. Most of these problems fall into the category known as *continuous-field simulation.* In essence, a physical situation can be described by a surface or region in three dimensions (e.g., the flow of air adjacent to the surface of a

multiple processors, each with its own L1 and L2 caches, plus main memory. The node is the basic building block of the overall CC-NUMA organization. For example, each Silicon Graphics Origin node includes two MIPS R10000 processors; each Sequent NUMA-Q node includes four Pentium II processors. The nodes are interconnected by means of some communications facility, which could be a switching mechanism, a ring, or some other networking facility.

Each node in the CC-NUMA system includes some main memory. From the point of view of the processors, however, there is only a single addressable memory, with each location having a unique system wide address. When a processor initiates a memory access, if the requested memory location is not in that processor's cache, then the L2 cache initiates a fetch operation. If the desired line is in the local portion of the main memory, the line is fetched across the local bus. If the desired line is in a remote portion of the main memory, then an automatic request is sent out to fetch that line across the interconnection network, deliver it to the local bus, and then deliver it to the requesting cache on that bus. All of this activity is automatic and transparent to the processor and its cache.

In this configuration, cache coherence is a central concern. Although implementations differ as to details, in general terms we can say that each node must maintain some sort of directory that gives it an indication of the location of various portions of memory and also cache status information. To see how this scheme works, we give an example taken from [PFIS98]. Suppose that processor 3 on node 2 (P2-3) requests a memory location 798, which is in the memory of node 1. The following sequence occurs:

1. P2-3 issues a read request on the snoopy bus of node 2 for location 798.
2. The directory on node 2 sees the request and recognizes that the location is in node 1.
3. Node 2's directory sends a request to node 1, which is picked up by node 1's directory.
4. Node 1's directory, acting as a surrogate of P2-3, requests the contents of 798, as if it were a processor.
5. Node 1's main memory responds by putting the requested data on the bus.
6. Node 1's directory picks up the data from the bus.
7. The value is transferred back to node 2's directory.
8. Node 2's directory places the data back on node 2's bus, acting as a surrogate for the memory that originally held it.
9. The value is picked up and placed in P2-3's cache and delivered to P2-3.

The preceding sequence explains how data are read from a remote memory using hardware mechanisms that make the transaction transparent to the processor. On top of these mechanisms, some form of cache coherence protocol is needed. Various systems differ on exactly how this is done. We make only a few general remarks here. First, as part of the preceding sequence, node 1's directory keeps a record that some remote cache has a copy of the line containing location 798. Then, there needs to be a cooperative protocol to take care of modifications. For example, if a modification is done in a cache, this fact can be broadcast to other nodes. Each node's directory that receives such a broadcast can then determine if

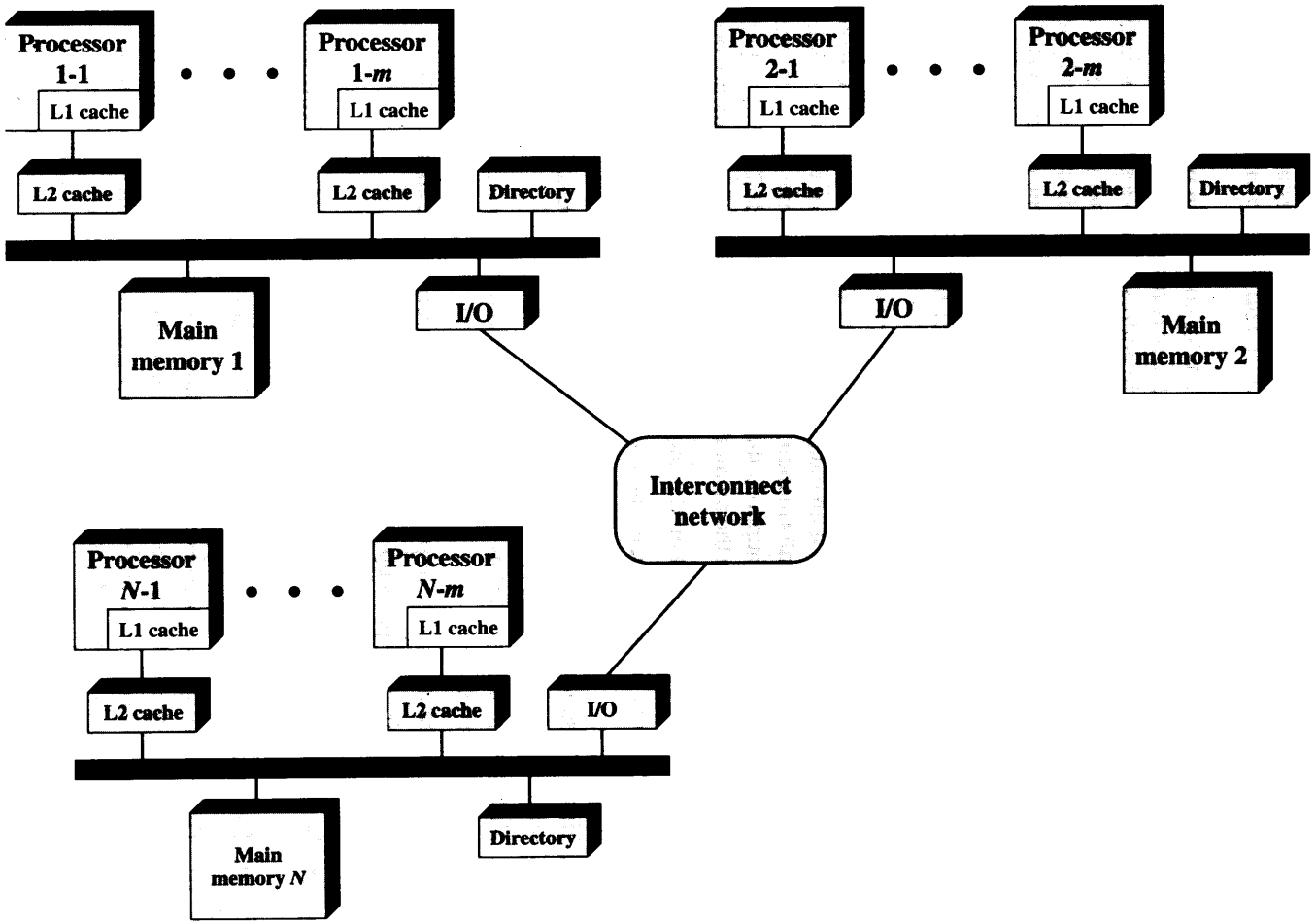


Figure 18.12 CC-NUMA Organization

- **Uniform memory access (UMA):** All processors have access to all parts of main memory using loads and stores. The memory access time of a processor to all regions of memory is the same. The access times experienced by different processors are the same. The SMP organization discussed in Sections 18.2 and 18.3 is UMA.
- **Nonuniform memory access (NUMA):** All processors have access to all parts of main memory using loads and stores. The memory access time of a processor differs depending on which region of main memory is accessed. The last statement is true for all processors; however, for different processors, which memory regions are slower and which are faster differ.
- **Cache-coherent NUMA (CC-NUMA):** A NUMA system in which cache coherence is maintained among the caches of the various processors.

A NUMA system without cache coherence is more or less equivalent to a cluster. The commercial products that have received much attention recently are CC-NUMA systems, which are quite distinct from both SMPs and clusters. Usually, but unfortunately not always, such systems are in fact referred to in the commercial literature as CC-NUMA systems. This section is concerned only with CC-NUMA systems.

Motivation

With an SMP system, there is a practical limit to the number of processors that can be used. An effective cache scheme reduces the bus traffic between any one processor and main memory. As the number of processors increases, this bus traffic also increases. Also, the bus is used to exchange cache-coherence signals, further adding to the burden. At some point, the bus becomes a performance bottleneck. Performance degradation seems to limit the number of processors in an SMP configuration to somewhere between 16 and 64 processors. For example, Silicon Graphics' Power Challenge SMP is limited to 64 R10000 processors in a single system; beyond this number performance degrades substantially.

The processor limit in an SMP is one of the driving motivations behind the development of cluster systems. However, with a cluster, each node has its own private main memory; applications do not see a large global memory. In effect, coherency is maintained in software rather than hardware. This memory granularity affects performance and, to achieve maximum performance, software must be tailored to this environment. One approach to achieving large-scale multiprocessing while retaining the flavor of SMP is NUMA. For example, the Silicon Graphics Origin NUMA system is designed to support up to 1024 MIPS R10000 processors [WHIT97] and the Sequent NUMA-Q system is designed to support up to 252 Pentium II processors [LOVE96].

The objective with NUMA is to maintain a transparent system wide memory while permitting multiple multiprocessor nodes, each with its own bus or other internal interconnect system.

Organization

Figure 18.12 depicts a typical CC-NUMA organization. There are multiple independent nodes, each of which is, in effect, an SMP organization. Thus, each node contains

- **Single memory space:** Distributed shared memory enables programs to share variables.
- **Single job-management system:** Under a cluster job scheduler, a user can submit a job without specifying the host computer to execute the job.
- **Single user interface:** A common graphic interface supports all users, regardless of the workstation from which they enter the cluster.
- **Single I/O space:** Any node can remotely access any I/O peripheral or disk device without knowledge of its physical location.
- **Single process space:** A uniform process-identification scheme is used. A process on any node can create or communicate with any other process on a remote node.
- **Checkpointing:** This function periodically saves the process state and intermediate computing results, to allow rollback recovery after a failure.
- **Process migration:** This function enables load balancing.

The last four items on the preceding list enhance the availability of the cluster. The remaining items are concerned with providing a single system image.

Returning to Figure 18.11, a cluster will also include software tools for enabling the efficient execution of programs that are capable of parallel execution.

Clusters Compared to SMP

Both clusters and symmetric multiprocessors provide a configuration with multiple processors to support high-demand applications. Both solutions are commercially available, although SMP schemes have been around far longer.

The main strength of the SMP approach is that an SMP is easier to manage and configure than a cluster. The SMP is much closer to the original single-processor model for which nearly all applications are written. The principal change required in going from a uniprocessor to an SMP is to the scheduler function. Another benefit of the SMP is that it usually takes up less physical space and draws less power than a comparable cluster. A final important benefit is that the SMP products are well established and stable.

Over the long run, however, the advantages of the cluster approach are likely to result in clusters dominating the high-performance server market. Clusters are far superior to SMPs in terms of incremental and absolute scalability. Clusters are also superior in terms of availability, because all components of the system can readily be made highly redundant.

18.6 NONUNIFORM MEMORY ACCESS

In terms of commercial products, the two common approaches to providing a multiple-processor system to support applications are SMPs and clusters. For some years, another approach, known as nonuniform memory access (NUMA), has been the subject of research and commercial NUMA products are now available.

Before proceeding, we should define some terms often found in the NUMA literature.

Figure 18.11 Cluster Computer Architecture [BUYY99a]

be incrementally scalable. When a new computer is added to the cluster, the load-balancing facility should automatically include this computer in scheduling applications. Middleware mechanisms need to recognize that services can appear on different members of the cluster and may migrate from one member to another.

Parallelizing Computation In some cases, effective use of a cluster requires executing software from a single application in parallel. [KAPP00] lists three general approaches to the problem:

- **Parallelizing compiler:** A parallelizing compiler determines, at compile time, which parts of an application can be executed in parallel. These are then split off to be assigned to different computers in the cluster. Performance depends on the nature of the problem and how well the compiler is designed.
- **Parallelized application:** In this approach, the programmer writes the application from the outset to run on a cluster, and uses message passing to move data, as required, between cluster nodes. This places a high burden on the programmer but may be the best approach for exploiting clusters for some applications.
- **Parametric computing:** This approach can be used if the essence of the application is an algorithm or program that must be executed a large number of times, each time with a different set of starting conditions or parameters. A good example is a simulation model, which will run a large number of different scenarios and then develop statistical summaries of the results. For this approach to be effective, parametric processing tools are needed to organize, run, and manage the jobs in an orderly manner.

Cluster Computer Architecture

Figure 18.11 shows a typical cluster architecture. The individual computers are connected by some high-speed LAN or switch hardware. Each computer is capable of operating independently. In addition, a middleware layer of software is installed in each computer to enable cluster operation. The cluster middleware provides a unified system image to the user, known as a **single-system image**. The middleware is also responsible for providing high availability, by means of load balancing and responding to failures in individual components. [HWAN99] lists the following as desirable cluster middleware services and functions:

- **Single entry point:** A user logs onto the cluster rather than to an individual computer.
- **Single file hierarchy:** The user sees a single hierarchy of file directories under the same root directory.
- **Single control point:** There is a default workstation used for cluster management and control.
- **Single virtual networking:** Any node can access any other point in the cluster, even though the actual cluster configuration may consist of multiple interconnected networks. There is a single virtual network operation.

In one approach to clustering, each computer is a **separate server** with its own disks and there are no disks shared between systems (Figure 18.10a). This arrangement provides high performance as well as high availability. In this case, some type of management or scheduling software is needed to assign incoming client requests to servers so that the load is balanced and high utilization is achieved. It is desirable to have a failover capability, which means that if a computer fails while executing an application, another computer in the cluster can pick up and complete the application. For this to happen, data must constantly be copied among systems so that each system has access to the current data of the other systems. The overhead of this data exchange ensures high availability at the cost of a performance penalty.

To reduce the communications overhead, most clusters now consist of servers connected to common disks (Figure 18.10b). In variation on this approach, called **shared nothing**, the common disks are partitioned into volumes, and each volume is owned by a single computer. If that computer fails, the cluster must be reconfigured so that some other computer has ownership of the volumes of the failed computer.

It is also possible to have multiple computers share the same disks at the same time (called the **shared disk** approach), so that each computer has access to all of the volumes on all of the disks. This approach requires the use of some type of locking facility to ensure that data can only be accessed by one computer at a time.

Operating System Design Issues

Full exploitation of a cluster hardware configuration requires some enhancements to a single-system operating system.

Failure Management How failures are managed by a cluster depends on the clustering method used (Table 18.2). In general, two approaches can be taken to dealing with failures: highly available clusters and fault-tolerant clusters. A highly available cluster offers a high probability that all resources will be in service. If a failure does occur, such as a system goes down or a disk volume is lost, then the queries in progress are lost. Any lost query, if retried, will be serviced by a different computer in the cluster. However, the cluster operating system makes no guarantee about the state of partially executed transactions. This would need to be handled at the application level.

A fault-tolerant cluster ensures that all resources are always available. This is achieved by the use of redundant shared disks and mechanisms for backing out uncommitted transactions and committing completed transactions.

The function of switching applications and data resources over from a failed system to an alternative system in the cluster is referred to as **failover**. A related function is the restoration of applications and data resources to the original system once it has been fixed; this is referred to as **failback**. Failback can be automated, but this is desirable only if the problem is truly fixed and unlikely to recur. If not, automatic failback can cause subsequently failed resources to bounce back and forth between computers, resulting in performance and recovery problems.

Load Balancing A cluster requires an effective capability for balancing the load among available computers. This includes the requirement that the cluster

Table 18.2 Clustering Methods: Benefits and Limitations

Clustering Method	Description	Benefits	Limitations
Passive Standby	A secondary server takes over in case of primary server failure.	Easy to implement	High cost because the secondary server is unavailable for other processing tasks.
Active Secondary:	The secondary server is also used for processing tasks.	Reduced cost because secondary servers can beused for processing.	Increased complexity.
Separate Servers	Separate servers have their own disks. Data is continuously copied from primary to secondary server.	High availability.	High network and server overhead due to copying operations.
Servers Connected to Disks	Servers are cabled to the same disks, but each server owns its disks. If one server fails, its disks are taken over by the other server.	Reduced network and server overhead due to elimination of copying operations.	Usually requires disk mirroring or RAID technology to compensate for risk of disk failure.
Servers Share Disks	Multiple servers simultaneously share access to disks.	Low network and server overhead. Reduced risk of downtime caused by disk failure.	Requires lock manager software. Usually used with disk mirroring or RAID technology.

use of RAID or some similar redundant disk technology is common in clusters so that the high availability achieved by the presence of multiple computers is not compromised by a shared disk that is a single point of failure.

A clearer picture of the range of cluster options can be gained by looking at functional alternatives. Table 18.2 provides a useful classification along functional lines, which we now discuss.

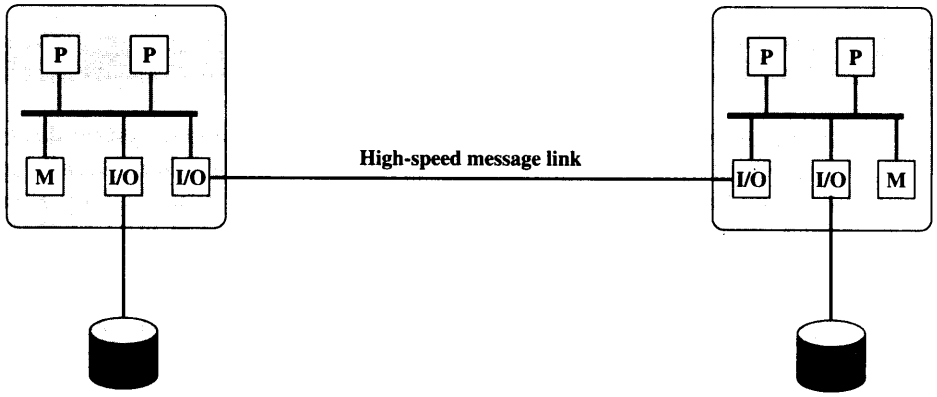
A common, older method, known as **passive standby**, is simply to have one computer handle all of the processing load while the other computer remains inactive, standing by to take over in the event of a failure of the primary. To coordinate the machines, the active, or primary, system periodically sends a "heartbeat" message to the standby machine. Should these messages stop arriving, the standby assumes that the primary server has failed and puts itself into operation. This approach increases availability but does not improve performance. Further, if the only information that is exchanged between the two systems is a heartbeat message, and if the two systems do not share common disks, then the standby provides a functional backup but has no access to the databases managed by the primary.

The passive standby is generally not referred to as a cluster. The term *cluster* is reserved for multiple interconnected computers that are all actively doing processing while maintaining the image of a single system to the outside world. The term **active secondary** is often used in referring to this configuration. Three classifications of clustering can be identified: separate servers, shared nothing, and shared memory.

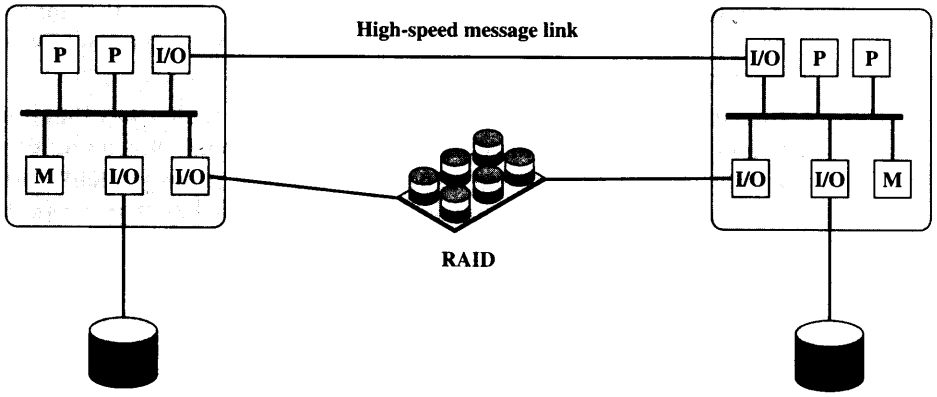
Cluster Configurations

In the literature, clusters are classified in a number of different ways. Perhaps the simplest classification is based on whether the computers in a cluster share access to the same disks. Figure 18.10a shows a two-node cluster in which the only interconnection is by means of a high-speed link that can be used for message exchange to coordinate cluster activity. The link can be a LAN that is shared with other computers that are not part of the cluster or the link can be a dedicated interconnection facility. In the latter case, one or more of the computers in the cluster will have a link to a LAN or WAN so that there is a connection between the server cluster and remote client systems. Note that in the figure, each computer is depicted as being a multiprocessor. This is not necessary but does enhance both performance and availability.

In the simple classification depicted in Figure 18.10, the other alternative is a shared-disk cluster. In this case, there generally is still a message link between nodes. In addition, there is a disk subsystem that is directly linked to multiple computers within the cluster. In this figure, the common disk subsystem is a RAID system. The



(a) Standby server with no shared disk



(b) Shared Disk

Figure 18.10 Cluster Configurations

compiler, which places operations that may be executed in parallel in the same word. In a simple VLIW machine (Figure 18.8g), if it is not possible to completely fill the word with instructions to be issued in parallel, no-ops are used.

- **Interleaved multithreading VLIW:** This approach should provide similar efficiencies to those provided by interleaved multithreading on a superscalar architecture.
- **Blocked multithreaded VLIW:** This approach should provide similar efficiencies to those provided by blocked multithreading on a superscalar architecture.

The final two approaches illustrated in Figure 18.8 enable the parallel, simultaneous execution of multiple threads:

- **Simultaneous multithreading:** Figure 18.8i shows a system capable of issuing 8 instructions at a time. If one thread has a high degree of instruction-level parallelism, it may on some cycles be able fill all of the horizontal slots. On other cycles, instructions from two or more threads may be issued. If sufficient threads are active, it should usually be possible to issue the maximum number of instructions on each cycle, providing a high level of efficiency.
- **Chip multiprocessor:** Figure 18.8k shows a chip containing four processors, each of which has a two-issue superscalar processor. Each processor is assigned a thread, from which it can issue up to two instructions per cycle.

Comparing Figures 18.8j and 18.8k, we see that a chip multiprocessor with the same instruction issue capability as an SMT cannot achieve the same degree of instruction-level parallelism. This is because the chip multiprocessor is not able to hide latencies by issuing instructions from other threads. On the other hand, the chip multiprocessor should outperform a superscalar processor with the same instruction issue capability, because the horizontal losses will be greater for the superscalar processor. In addition, it is possible to use multithreading within each of the processors on a chip multiprocessor, and this is done on some contemporary machines.

Example Systems

Pentium 4 More recent models of the Pentium 4 use a multithreading technique that the Intel literature refers to as *hyperthreading* [MARR02]. In essence, the Pentium 4 approach is to use SMT with support for two threads. Thus, the single multithreaded processor is logically two processors.

IBM Power5 The IBM Power5 chip, which is used in high-end PowerPC products, combines chip multiprocessing with SMT [KALL04]. The chip has two separate processors, each of which is a multithreaded processor capable of supporting two threads concurrently using SMT. Interestingly, the designers simulated various alternatives and found that having two two-way SMT processors on a single chip provided superior performance to a single four-way SMT processor. The simulations showed that additional multithreading beyond the support for two threads might decrease performance because of cache thrashing, as data from one thread displaces data needed by another thread.

Figure 18.9 shows the Power5's instruction flow diagram. Only a few of the elements in the processor need to be replicated, with separate elements dedicated to

The first three illustrations in Figure 18.8 show different approaches with a scalar (i.e., single-issue) processor:

- **Single-threaded scalar:** This is the simple pipeline found in traditional RISC and CISC machines, with no multithreading.
- **Interleaved multithreaded scalar:** This is the easiest multithreading approach to implement. By switching from one thread to another at each clock cycle, the pipeline stages can be kept fully occupied, or close to fully occupied. The hardware must be capable of switching from one thread context to another between cycles.
- **Blocked multithreaded scalar:** In this case, a single thread is executed until a latency event occurs that would stop the pipeline, at which time the processor switches to another thread.

Figure 18.8c shows a situation in which the time to perform a thread switch is one cycle, whereas Figure 18.8b shows that thread switching occurs in zero cycles. In the case of interleaved multithreading, it is assumed that there are no control or data dependencies between threads, which simplifies the pipeline design and therefore should allow a thread switch with no delay. However, depending on the specific design and implementation, block multithreading may require a clock cycle to perform a thread switch, as illustrated in Figure 18.8. This is true if a fetched instruction triggers the thread switch and must be discarded from the pipeline [UNGE03].

Although interleaved multithreading appears to offer better processor utilization than blocked multithreading, it does so at the sacrifice of single-thread performance. The multiple threads compete for cache resources, which raises the probability of a cache miss for a given thread.

More opportunities for parallel execution are available if the processor can issue multiple instructions per cycle. Figures 18.8d through 18.8i illustrate a number of variations among processors that have hardware for issuing four instructions per cycle. In all these cases, only instructions from a single thread are issued in a single cycle. The following alternatives are illustrated:

- **Superscalar:** This is the basic superscalar approach with no multithreading. Until relatively recently, this was the most powerful approach to providing parallelism within a processor. Note that during some cycles, not all of the available issue slots are used. During these cycles, less than the maximum number of instructions is issued; this is referred to as *horizontal loss*. During other instruction cycles, no issue slots are used; these are cycles when no instructions can be issued; this is referred to as *vertical loss*.
- **Interleaved multithreading superscalar:** During each cycle, as many instructions as possible are issued from a single thread. With this technique, potential delays due to thread switches are eliminated, as previously discussed. However, the number of instructions issued in any given cycle is still limited by dependencies that exist within any given thread.
- **Blocked multithreaded superscalar:** Again, instructions from only one thread may be issued during any cycle, and blocked multithreading is used.
- **Very long instruction word (VLIW):** A VLIW architecture, such as IA-64, places multiple instructions in a single word. Typically, a VLIW is constructed by the

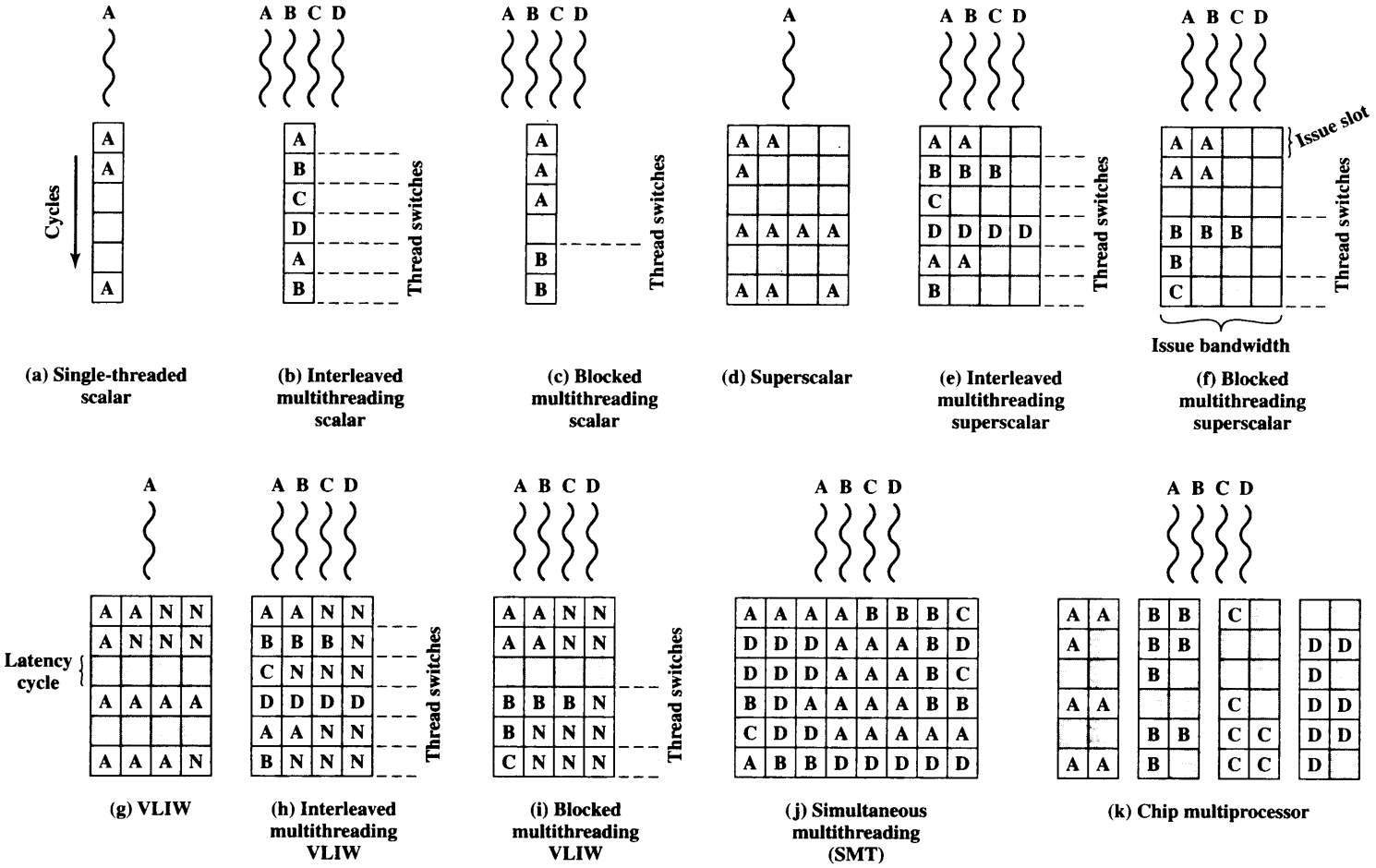


Figure 18.8 Approaches to Executing Multiple Threads

amount and type of additional hardware used to support concurrent thread execution. In general, instruction fetching takes place on a thread basis. The processor treats each thread separately and my use a number of techniques for optimizing single-thread execution, including branch prediction, register renaming, and superscalar techniques. What is achieved is thread-level parallelism, which may provide for greatly improved performance when married to instruction-level parallelism.

Broadly speaking, there are four principal approaches to multithreading:

- **Interleaved multithreading:** This is also known as **fine-grained multithreading**. The processor deals with two or more thread contexts at a time, switching from one thread to another at each clock cycle. If a thread is blocked because of data dependencies or memory latencies, that thread is skipped and a ready thread is executed.
- **Blocked multithreading:** This is also known as **coarse-grained multithreading**. The instructions of a thread are executed successively until an event occurs that may cause delay, such as a cache miss. This event induces a switch to another thread. This approach is effective on an in-order processor that would stall the pipeline for a delay event such as a cache miss.
- **Simultaneous multithreading (SMT):** Instructions are simultaneously issued from multiple threads to the execution units of a superscalar processor. This combines the wide superscalar instruction issue capability with the use of multiple thread contexts.
- **Chip multiprocessing:** In this case, the entire processor is replicated on a single chip and each processor handles separate threads. The advantage of this approach is that the available logic area on a chip is used effectively without depending on ever-increasing complexity in pipeline design.

For the first two approaches, instructions from different threads are not executed simultaneously. Instead, the processor is able to rapidly switch from one thread to another, using a different set of registers and other context information. This results in a better utilization of the processor's execution resources and avoids a large penalty due to cache misses and other latency events. The SMT approach involves true simultaneous execution of instructions from different threads, using replicated execution resources. Chip multiprocessing also enables simultaneous execution of instructions from different threads.

Figure 18.8, based on one in [UNGE02], illustrates some of the possible pipeline architectures that involve multithreading and contrasts these with approaches that do no use multithreading. Each horizontal row represents the potential issue slot or slots for a single execution cycle; that is, the width of each row corresponds to the maximum number of instructions that can be issued in a single clock cycle.[3] The vertical dimension represents the time sequence of clock cycles. An empty (shaded) slot represents an unused execution slot in one pipeline. A no-op is indicated by N.

[3]Issue slots are the position from which instructions can be issued in a given clock cycle. Recall from Chapter 14 that instruction issue is the process of initiating instruction execution in the processor's functional units. This occurs when an instruction moves from the decode stage of the pipeline to the first execute stage of the pipeline.

—**Resource ownership:** A process includes a virtual address space to hold the process image; the process image is the collection of program, data, stack, and attributes that define the process. From time to time, a process may be allocated control or ownership of resources, such as main memory, I/O channels, I/O devices, and files.

—**Scheduling/execution:** The execution of a process follows an execution path (trace) through one or more programs. This execution may be interleaved with that of other processes. Thus, a process has an execution state (Running, Ready, etc.) and a dispatching priority and is the entity that is scheduled and dispatched by the operating system.

- **Process switch:** An operation that switches the processor from one process to another, by saving all the process control data, registers, and other information for the first and replacing them with the process information for the second.[2]
- **Thread:** A dispatchable unit of work within a process. It includes a processor context (which includes the program counter and stack pointer) and its own data area for a stack (to enable subroutine branching). A thread executes sequentially and is interruptible so that the processor can turn to another thread.
- **Thread switch:** The act of switching processor control from one thread to another within the same process. Typically, this type of switch is much less costly than a process switch.

Thus, a thread is concerned with scheduling and execution, whereas a process is concerned with both scheduling/execution and resource ownership. The multiple threads within a process share the same resources. This is why a thread switch is much less time consuming than a process switch. Traditional operating systems, such as earlier versions of Unix, did not support threads. Most modern operating systems, such as Linux, other versions of Unix, and Windows, do support threads. A distinction is made between user-level threads, which are visible to the application program, and kernel-level threads, which are visible only to the operating system. Both of these may be referred to as explicit threads, defined in software.

All of the commercial processors and most of the experimental processors so far have used explicit multithreading. These systems concurrently execute instructions from different explicit threads, either by interleaving instructions from different threads on shared pipelines or by parallel execution on parallel pipelines. Implicit multithreading refers to the concurrent execution of multiple threads extracted from a single sequential program. These implicit threads may be defined either statically by the compiler or dynamically by the hardware. In the remainder of this section we consider explicit multithreading.

Approaches to Explicit Multithreading

At minimum, a multithreaded processor must provide a separate program counter for each thread of execution to be executed concurrently. The designs differ in the

[2]The term *context switch* is often found in OS literature and textbooks. Unfortunately, although most of the literature uses this term to mean what is here called a process switch, other sources use it to mean a thread switch. To avoid ambiguity, the term is not used in this book.

memory. The L1 write-through policy forces any modification to an L1 line out to the L2 cache and therefore makes it visible to other L2 caches. The use of the L1 write-through policy requires that the L1 content must be a subset of the L2 content. This in turn suggests that the associativity of the L2 cache should be equal to or greater than that of the L1 associativity. The L1 write-through policy is used in the IBM S/390 SMP.

If the L1 cache has a write-back policy, the relationship between the two caches is more complex. There are several approaches to maintaining coherence. For example, the approach used on the Pentium II is described in detail in [SHAN05].

18.4 MULTITHREADING AND CHIP MULTIPROCESSORS

The most important measure of performance for a processor is the rate at which it executes instructions. This can be expressed as

$$\text{MIPS rate} = f \times IPC$$

where f is the processor clock frequency, in MHz, and IPC (instructions per cycle) is the average number of instructions executed per cycle. Accordingly, designers have pursued the goal of increased performance on two fronts: increasing clock frequency and increasing the number of instructions executed or, more properly, the number of instructions that complete during a processor cycle. As we have seen in earlier chapters, designers have increased IPC by using an instruction pipeline and then by using multiple parallel instruction pipelines in a superscalar architecture. With pipelined and multiple-pipeline designs, the principal problem is to maximize the utilization of each pipeline stage. To improve throughput, designers have created ever more complex mechanisms, such as executing some instructions in a different order from the way they occur in the instruction stream and beginning execution of instructions that may never be needed. But as was discussed in Section 2.2, this approach may be reaching a limit due to complexity and power consumption concerns.

An alternative approach, which allows for a high degree of instruction-level parallelism without increasing circuit complexity or power consumption, is called multithreading. In essence, the instruction stream is divided into several smaller streams, known as threads, such that the threads can be executed in parallel.

The variety of specific multithreading designs, realized in both commercial systems and experimental systems, is vast. In this section, we give a brief survey of the major concepts.

Implicit and Explicit Multithreading

The concept of thread used in discussing multithreaded processors may or may not be the same as the concept of software threads in a multiprogrammed operating system. It will be useful to briefly define terms:

- **Process:** An instance of a program running on a computer. A process embodies two key characteristics:

Write Miss When a write miss occurs in the local cache, the processor initiates a memory read to read the line of main memory containing the missing address. For this purpose, the processor issues a signal on the bus that means *read-with-intent-to-modify* (RWITM). When the line is loaded, it is immediately marked modified. With respect to other caches, two possible scenarios precede the loading of the line of data.

First, some other cache may have a modified copy of this line (state = modify). In this case, the alerted processor signals the initiating processor that another processor has a modified copy of the line. The initiating processor surrenders the bus and waits. The other processor gains access to the bus, writes the modified cache line back to main memory, and transitions the state of the cache line to invalid (because the initiating processor is going to modify this line). Subsequently, the initiating processor will again issue a signal to the bus of RWITM and then read the line from main memory, modify the line in the cache, and mark the line in the modified state.

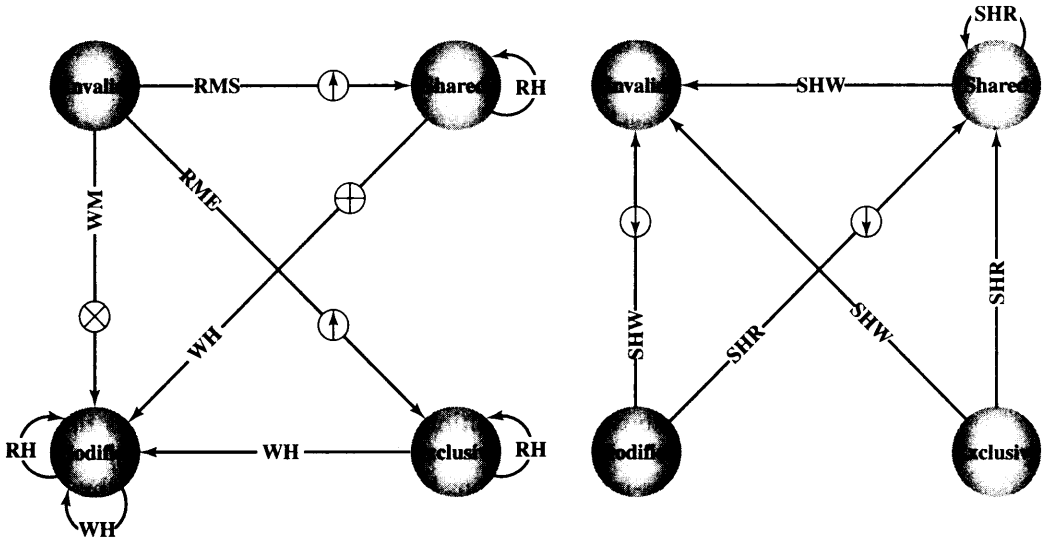
The second scenario is that no other cache has a modified copy of the requested line. In this case, no signal is returned, and the initiating processor proceeds to read in the line and modify it. Meanwhile, if one or more caches have a clean copy of the line in the shared state, each cache invalidates its copy of the line, and if one cache has a clean copy of the line in the exclusive state, it invalidates its copy of the line.

Write Hit When a write hit occurs on a line currently in the local cache, the effect depends on the current state of that line in the local cache:

- **Shared:** Before performing the update, the processor must gain exclusive ownership of the line. The processor signals its intent on the bus. Each processor that has a shared copy of the line in its cache transitions the sector from shared to invalid. The initiating processor then performs the update and transitions its copy of the line from shared to modified.
- **Exclusive:** The processor already has exclusive control of this line, and so it simply performs the update and transitions its copy of the line from exclusive to modified.
- **Modified:** The processor already has exclusive control of this line and has the line marked as modified, and so it simply performs the update.

L1–L2 Cache Consistency We have so far described cache coherency protocols in terms of the cooperate activity among caches connected to the same bus or other SMP interconnection facility. Typically, these caches are L2 caches, and each processor also has an L1 cache that does not connect directly to the bus and that therefore cannot engage in a snoopy protocol. Thus, some scheme is needed to maintain data integrity across both levels of cache and across all caches in the SMP configuration.

The strategy is to extend the MESI protocol (or any cache coherence protocol) to the L1 caches. Thus, each line in the L1 cache includes bits to indicate the state. In essence, the objective is the following: for any line that is present in both an L2 cache and its corresponding L1 cache, the L1 line state should track the state of the L2 line. A simple means of doing this is to adopt the write-through policy in the L1 cache; in this case the write through is to the L2 cache and not to the



(a) Line in cache at initiating processor (b) Line in snooping cache

RH	Read hit		Dirty line copyback
RMS	Read miss, shared		
RME	Read miss, exclusive		Invalidate transaction
WH	Write hit		
WM	Write miss		Read-with-intent-to-modify
SHR	Snoop hit on read		
SHW	Snoop hit on write or read-with-intent-to-modify		Cache line fill

Figure 18.7 MESI State Transition Diagram

- If one other cache has a modified copy of the line, then that cache blocks the memory read and provides the line to the requesting cache over the shared bus. The responding cache then changes its line from modified to shared.[1]
- If no other cache has a copy of the line (clean or modified), then no signals are returned. The initiating processor reads the line and transitions the line in its cache from invalid to exclusive.

Read Hit When a read hit occurs on a line currently in the local cache, the processor simply reads the required item. There is no state change: The state remains modified, shared, or exclusive.

[1]In some implementations, the cache with the modified line signals the initiating processor to retry. Meanwhile, the processor with the modified copy seizes the bus, writes the modified line back to main memory, and transitions the line in its cache from modified to shared. Subsequently, the requesting processor tries again and finds that one or more processors have a clean copy of the line in the shared state, as described in the preceding point.

- **Modified:** The line in the cache has been modified (different from main memory) and is available only in this cache.
- **Exclusive:** The line in the cache is the same as that in main memory and is not present in any other cache.
- **Shared:** The line in the cache is the same as that in main memory and may be present in another cache.
- **Invalid:** The line in the cache does not contain valid data.

Table 18.1 summarizes the meaning of the four states. Figure 18.7 displays a state diagram for the MESI protocol. Keep in mind that each line of the cache has its own state bits and therefore its own realization of the state diagram. Figure 18.7a shows the transitions that occur due to actions initiated by the processor attached to this cache. Figure 18.7b shows the transitions that occur due to events that are snooped on the common bus. This presentation of separate state diagrams for processor-initiated and bus-initiated actions helps to clarify the logic of the MESI protocol. At any time a cache line is in a single state. If the next event is from the attached processor, then the transition is dictated by Figure 18.7a and if the next event is from the bus, the transition is dictated by Figure 18.7b. Let us look at these transitions in more detail.

Read Miss When a read miss occurs in the local cache, the processor initiates a memory read to read the line of main memory containing the missing address. The processor inserts a signal on the bus that alerts all other processor/cache units to snoop the transaction. There are a number of possible outcomes:

- If one other cache has a clean (unmodified since read from memory) copy of the line in the exclusive state, it returns a signal indicating that it shares this line. The responding processor then transitions the state of its copy from exclusive to shared, and the initiating processor reads the line from main memory and transitions the line in its cache from invalid to shared.
- If one or more caches have a clean copy of the line in the shared state, each of them signals that it shares the line. The initiating processor reads the line and transitions the line in its cache from invalid to shared.

Table 18.1 MESI Cache Line States

	M Modified	E Exclusive	S Shared	I Invalid
This cache line valid?	Yes	Yes	Yes	No
The memory copy is ...	out of date	valid	valid	—
Copies exist in other caches?	No	No	Maybe	Maybe
A write to this line ...	does not go to bus	does not go to bus	goes to bus and updates cache	goes directly to bus

processor to do a write back to main memory. The line may now be shared for reading by the original processor and the requesting processor.

Directory schemes suffer from the drawbacks of a central bottleneck and the overhead of communication between the various cache controllers and the central controller. However, they are effective in large-scale systems that involve multiple buses or some other complex interconnection scheme.

Snoopy Protocols Snoopy protocols distribute the responsibility for maintaining cache coherence among all of the cache controllers in a multiprocessor. A cache must recognize when a line that it holds is shared with other caches. When an update action is performed on a shared cache line, it must be announced to all other caches by a broadcast mechanism. Each cache controller is able to "snoop" on the network to observe these broadcasted notifications, and react accordingly.

Snoopy protocols are ideally suited to a bus-based multiprocessor, because the shared bus provides a simple means for broadcasting and snooping. However, because one of the objectives of the use of local caches is to avoid bus accesses, care must be taken that the increased bus traffic required for broadcasting and snooping does not cancel out the gains from the use of local caches.

Two basic approaches to the snoopy protocol have been explored: write invalidate and write update (or write broadcast). With a write-invalidate protocol, there can be multiple readers but only one writer at a time. Initially, a line may be shared among several caches for reading purposes. When one of the caches wants to perform a write to the line, it first issues a notice that invalidates that line in the other caches, making the line exclusive to the writing cache. Once the line is exclusive, the owning processor can make cheap local writes until some other processor requires the same line.

With a write-update protocol, there can be multiple writers as well as multiple readers. When a processor wishes to update a shared line, the word to be updated is distributed to all others, and caches containing that line can update it.

Neither of these two approaches is superior to the other under all circumstances. Performance depends on the number of local caches and the pattern of memory reads and writes. Some systems implement adaptive protocols that employ both write-invalidate and write-update mechanisms.

The write-invalidate approach is the most widely used in commercial multiprocessor systems, such as the Pentium 4 and PowerPC. It marks the state of every cache line (using two extra bits in the cache tag) as modified, exclusive, shared, or invalid. For this reason, the write-invalidate protocol is called MESI. In the remainder of this section, we will look at its use among local caches across a multiprocessor. For simplicity in the presentation, we do not examine the mechanisms involved in coordinating among both level 1 and level 2 locally as well as at the same time coordinating across the distributed multiprocessor. This would not add any new principles but would greatly complicate the discussion.

The MESI Protocol

To provide cache consistency on an SMP, the data cache often supports a protocol known as MESI. For MESI, the data cache includes two status bits per tag, so that each line can be in one of four states:

compile-time software approaches generally must make conservative decisions, leading to inefficient cache utilization.

Compiler-based coherence mechanisms perform an analysis on the code to determine which data items may become unsafe for caching, and they mark those items accordingly. The operating system or hardware then prevents noncacheable items from being cached.

The simplest approach is to prevent any shared data variables from being cached. This is too conservative, because a shared data structure may be exclusively used during some periods and may be effectively read-only during other periods. It is only during periods when at least one process may update the variable and at least one other process may access the variable that cache coherence is an issue.

More efficient approaches analyze the code to determine safe periods for shared variables. The compiler then inserts instructions into the generated code to enforce cache coherence during the critical periods. A number of techniques have been developed for performing the analysis and for enforcing the results; see [LILJ93] and [STEN90] for surveys.

Hardware Solutions

Hardware-based solutions are generally referred to as cache coherence protocols. These solutions provide dynamic recognition at run time of potential inconsistency conditions. Because the problem is only dealt with when it actually arises, there is more effective use of caches, leading to improved performance over a software approach. In addition, these approaches are transparent to the programmer and the compiler, reducing the software development burden.

Hardware schemes differ in a number of particulars, including where the state information about data lines is held, how that information is organized, where coherence is enforced, and the enforcement mechanisms. In general, hardware schemes can be divided into two categories: directory protocols and snoopy protocols.

Directory Protocols Directory protocols collect and maintain information about where copies of lines reside. Typically, there is a centralized controller that is part of the main memory controller, and a directory that is stored in main memory. The directory contains global state information about the contents of the various local caches. When an individual cache controller makes a request, the centralized controller checks and issues necessary commands for data transfer between memory and caches or between caches. It is also responsible for keeping the state information up to date; therefore, every local action that can affect the global state of a line must be reported to the central controller.

Typically, the controller maintains information about which processors have a copy of which lines. Before a processor can write to a local copy of a line, it must request exclusive access to the line from the controller. Before granting this exclusive access, the controller sends a message to all processors with a cached copy of this line, forcing each processor to invalidate its copy. After receiving acknowledgments back from each such processor, the controller grants exclusive access to the requesting processor. When another processor tries to read a line that is exclusively granted to another processor, it will send a miss notification to the controller. The controller then issues a command to the processor holding that line that requires the