Problem 8.17 Consider the use of a multivector multiprocessor system for computing the following linear combination of n vectors:

$$\mathbf{y} = \sum_{j=0}^{1023} a_j \times \mathbf{x}_j$$

where $\mathbf{y} = (y_0, y_1, \ldots, y_{1023})^T$ and $\mathbf{x}_j = (x_{0j}, x_{1j}, \ldots, x_{1023j})^T$ for $0 \le j \le 1023$ are column vectors; $\{a_j | 0 \le j \le 1023\}$ are scalar constants. You are asked to implement the above computations on a four-processor system with shared memory. Each processor is equipped with a vector-add pipeline and a vector-multiply pipeline. Assume four pipeline stages in each functional pipeline.

(a) Design a minimum-time parallel algorithm to perform concurrent vector operations on the given multiprocessor, ignoring all memory-access and I/O operations.

(b) Compare the performance of the multiprocessor algorithm with that of a sequential algorithm on a uniprocessor without the pipelined vector hardware.

Problem 8.18 The Burroughs Scientific Processor (BSP) was built as an SIMD computer consisting of 16 PEs accessing 17 shared memory modules. Prove that conflict-free memory access can be achieved on the BSP for vectors of an arbitrary length with a stride which is not a multiple of 17.

9
Scalable, Multithreaded, and Dataflow Architectures

This chapter discusses innovative computers built with scalable, multithreaded, or dataflow architectures. These architectures generated and validated many research ideas which led to the latter development of massively parallel processing (MPP) systems. Therefore, the material is presented with a strong research flavor benefiting mostly researchers, designers, and graduate students. More recent developments of these ideas are presented in Chapter 13.

Major research issues covered include latency-hiding techniques, principles of multithreading, multidimensional scalability, multithreaded architectures, fine-grain multicomputers, dataflow, and hybrid architectures. Example systems studied include the Stanford Dash, Wisconsin Multicube, USC/OMP, KSR-1, Tera, MIT Alewife and J-Machine, Caltech Mosaic C, ETL EM-4, and MIT/Motorola *T.

9.1 LATENCY-HIDING TECHNIQUES

Massively parallel and scalable systems may typically use distributed shared memory. The access of remote memory significantly increases memory latency. Furthermore, the processor speed has been increasing at a much faster rate than memory speeds. Thus any scalable multiprocessor or large-scale multicomputer must rely on the use of latency-reducing, -tolerating, or -hiding mechanisms. Four latency-hiding mechanisms are studied below for enhancing scalability and programmability.

Latency hiding can be accomplished through four complementary approaches: (i) using *prfetching techniques* which bring instructions or data close to the processor before they are actually needed; (ii) using *coherent caches* supported by hardware to reduce cache misses; (iii) using *relaxed memory consistency* models by allowing buffering and pipelining of memory references; and (iv) using *multiple-contexts* support to allow a processor to switch from one context to another when a long-latency operation is encountered.

The first three mechanisms are described in this section, supported by simulation results obtained by Stanford researchers. Multiple contexts will be treated with multithreaded processors and system architectures in Sections 9.2 and 9.4. However, the effect of multiple contexts is shown here in combination with other latency-hiding mechanisms.

9.1.1 Shared Virtual Memory

Single-address-space multiprocessors/multicomputers must use shared virtual memory. We present a model of such an architectural environment based on the Stanford Dash experience. Then we examine several shared-virtual-memory systems developed at Stanford, Yale, Carnegie-Mellon, and Princeton universities.

The Architecture Environment The Dash architecture was a large-scale, cache-coherent, NUMA multiprocessor system, as depicted in Fig. 9.1. It consisted of multiple multiprocessor clusters connected through a scalable, low-latency interconnection network. Physical memory was distributed among the processing nodes in various clusters. The distributed memory formed a global address space.

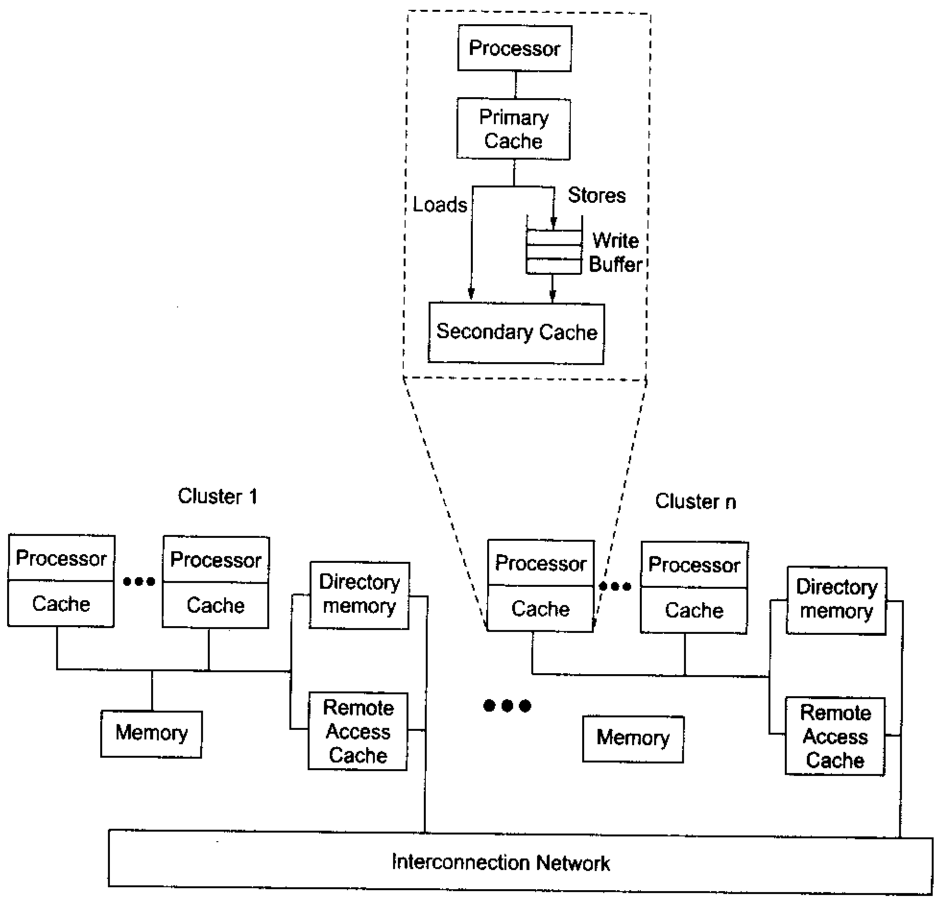


Fig. 9.1 A scalable coherent cache multiprocessor with distributed shared memory modeled after the Stanford Dash (Courtesy of Anoop Gupta et al, *Proc. 1991 Ann. Int. Symp. Computer Arch.*)

Cache coherence was maintained using an invalidating, distributed directory-based protocol (Section 7.2.3). For each memory block, the directory kept track of remote nodes cacheing it. When a write occurred, point-to-point messages were sent to invalidate remote copies of the block. Acknowledgment messages were used to inform the originating node when an invalidation was completed.

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

The SVM Concept Figure 9.2 shows the structure of a distributed shared memory. A global virtual address space is shared among processors residing at a large number of loosely coupled processing nodes. This *shared virtual memory* (SVM) concept was introduced in Section 4.4.1. Implementation and management issues of SVM are discussed below.

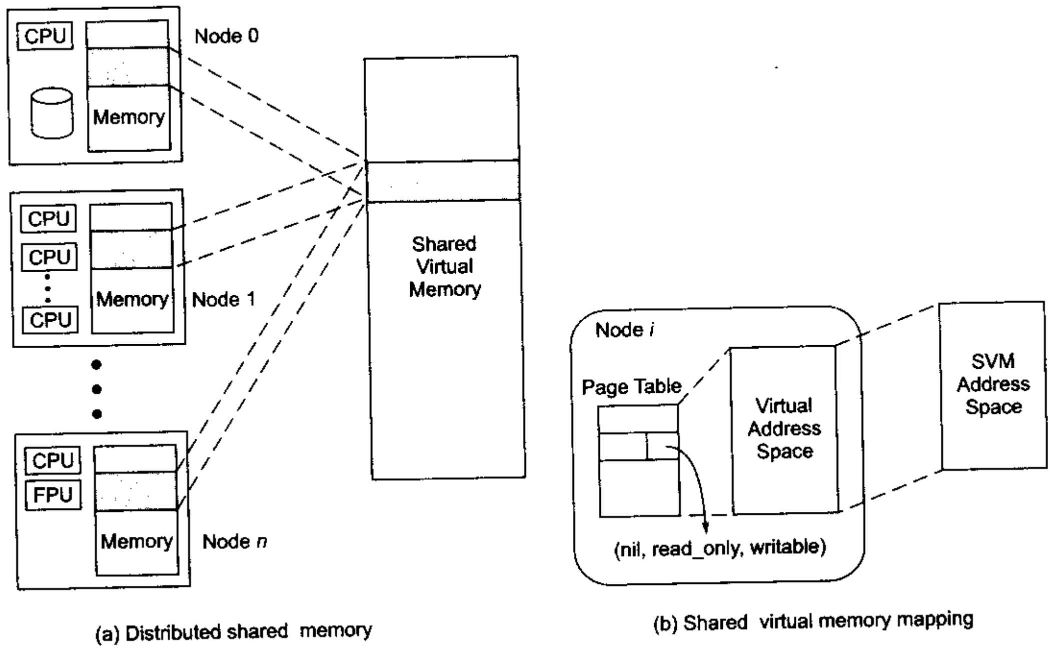


Fig. 9.2 The concept of distributed shared memory with a global virtual address space shared among all processors on loosely coupled processing nodes in a massively parallel architecture (Courtesy of Kai Li, 1992)

Shared virtual memory was first developed in a Ph.D. thesis by Li (1986) at Yale University. The idea is to implement coherent shared memory on a network of processors without physically shared memory. The coherent mapping of SVM on a message-passing multicomputer architecture is shown in Fig. 9.2b. The system uses virtual addresses instead of physical addresses for memory references.

Each virtual address space can be as large as a single node can provide and is shared by all nodes in the system. Li (1988) implemented the first SVM system, IVY, on a network of Apollo workstations. The SVM address space is organized in pages which can be accessed by any node in the system. A memory-mapping manager on each node views its local memory as a large cache of pages for its associated processor.

Page Swapping According to Kai Li (1992), pages that are marked read-only can have copies residing in the physical memories of other processors. A page currently being written may reside in only one local memory. When a processor writes a page that is also on other processors, it must update the page and then invalidate all copies on the other processors. Li described the page swapping as follows:

A memory reference causes a page fault when the page containing the memory location is not in a processor's local memory. When a page fault occurs, the memory manager retrieves the missing page from the memory of another processor. If there is a page frame available on the receiving node, the page is moved in. Otherwise, the SVM system uses page replacement policies to find an available page frame, swapping its contents to the sending node.

A hardware MMU can set the access rights (*nil, read-only, writable*) so that a memory access violating memory coherence will cause a page fault. The memory coherence problem is solved in IVY through distributed fault handlers and their servers. To client programs, this mechanism is completely transparent.

The large virtual address space allows programs to be larger in code and data space than the physical memory on a single node. This SVM approach offers the ease of shared-variable programming in a message-passing environment. In addition, it improves software portability and enhances system scalability through modular memory growth.

Example SVM Systems Nitzberg and Lo (1991) conducted a survey of SVM research systems. Excerpted from their survey, descriptions of four representative SVM systems are summarized in Table 9.1. Dash implemented SVM with a directory-based coherence protocol. Linda offered a shared associative object memory with access functions. Plus used a write-update coherence protocol and performed replication only by program request. Shiva extended the IVY system for the Intel iPSC/2 hypercube. In using SVM systems, there exists a tendency to use large block (page) sizes as units of coherence. This tends to increase false-sharing activity.

Table 9.1 *Representative SVM Research Systems (Excerpts from Nitzberg and Lo, IEEE Comput., August 1991)*

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

Scalability issues of SVM architectures include determining the sizes of data structures for maintaining memory coherence and how to take advantage of the fast data transmission among distributed memories in order to implement large SVM address spaces. Data structure compaction and page swapping can simplify the design of a large SVM address space without using disks as backing stores. A number of alternative choices are given in Li (1992).

9.1.2 Prefetching Techniques

Prefetching techniques are studied below. These involve both hardware and software approaches. Some benchmark results for prefetching on the Stanford Dash system are presented to illustrate the benefits.

Prefetching Techniques Prefetching uses knowledge about the expected misses in a program to move the corresponding data close to the processor before it is actually needed. Prefetching can be classified based on whether it is *binding* or *nonbinding*, and whether it is controlled by *hardware* or *software*.

With binding prefetching, the value of a later reference (e.g. a register load) is bound at the time when the prefetch completes. This places restrictions on when a binding prefetch can be issued, since the value will become stale if another processor modifies the same location during the interval between prefetch and reference. Binding prefetching may result in a significant loss in performance due to such limitations.

In contrast, nonbinding prefetching also brings the data close to the processor, but the data remains visible to the cache coherence protocol and is thus kept consistent until the processor actually reads the value.

Hardware-controlled prefetching includes schemes such as long cache lines and instruction lookahead. The effectiveness of long cache lines is limited by the reduced spatial locality in multiprocessor applications, while instruction lookahead is limited by branches and the finite lookahead buffer size.

With software-controlled prefetching, explicit prefetch instructions are issued. Software control allows the prefetching to be done selectively (thus reducing bandwidth requirements) and extends the possible interval between prefetch issue and actual reference, which is very important when latencies are large.

The disadvantages of software control include the extra instruction overhead required to generate the prefetches, as well as the need for sophisticated software intervention. In our study, we concentrate on *non-binding software controlled prefetching*.

Benefits of Prefetching The benefits of prefetching come from several sources. The most obvious benefit occurs when a prefetch is issued early enough in the code so that the line is already in the cache by the time it is referenced. However, prefetching can improve performance even when this is not possible (e.g. when the address of a data structure cannot be determined until immediately before it is referenced). If multiple prefetches are issued back to back to fetch the data structure, the latency of all but the first prefetched reference can be hidden due to the pipelining of the memory accesses.

Prefetching offers another benefit in multiprocessors that use an ownership-based cache coherence protocol. If a cache block line is to be modified, prefetching it directly with ownership can significantly reduce the write latencies and the ensuing network traffic for obtaining ownership. Network traffic is reduced in read-modify-write instructions, since prefetching with ownership avoids first fetching a read-shared copy.

Benchmark Results Stanford researchers (Gupta, Hennessy, Gharachorloo, Mowry, and Weber, 1991) reported some benchmark results for evaluating various latency-hiding mechanisms. Benchmark programs included a particle-based three-dimensional simulator used in aeronautics (MP3D), an LU-decomposition program (LU), and a digital logic simulation program (PTHOR). The effect of prefetching is illustrated in Fig. 9.3 for running the MP3D code on a simulated Dash multiprocessor (Fig. 9.1).

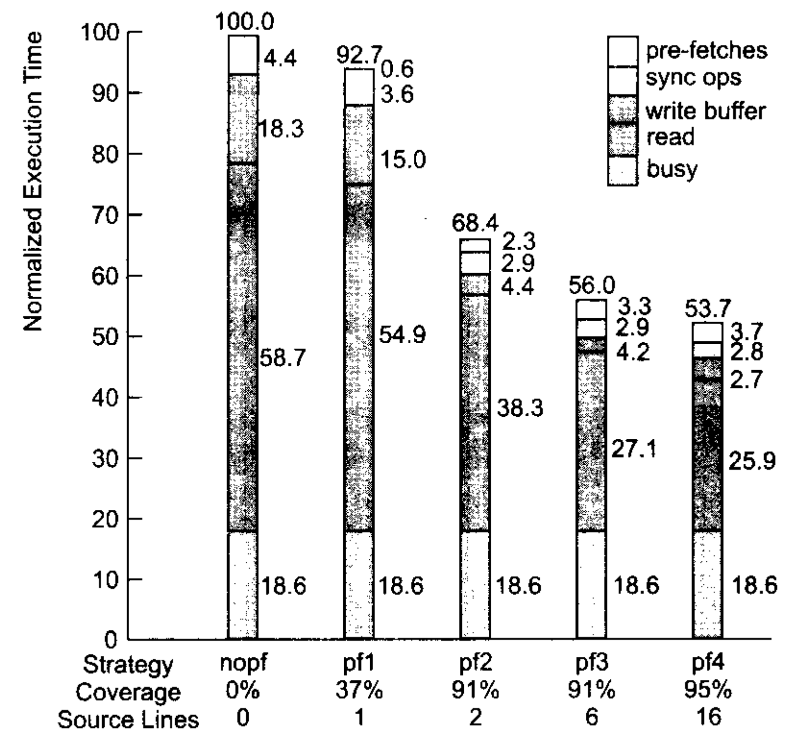


Fig. 9.3 Effect of various pre-fetching strategies for running the MP3D benchmark on a simulated Dash multiprocessor (Courtesy of Anoop Gupta et al, 1992)

The simulation runs involved 10,000 particles in a 64 × 8 × 8 space array with five time steps. Five prefetching strategies were tested (*nof, pf1, pf2, pf3,* and *pf4* in Fig. 9.3). These strategies range from no prefetching (*nopf*) to prefetching of the particle record in the same iteration or pipelined across increasing numbers of iterations (*pf1* through *pf4*). The bar diagrams in Fig. 9.3 show the execution times normalized with respect to the *nopf* strategy. Each bar shows a breakdown of the times required for prefetches, synchronization operations, using write buffers, reads, and busy in computing.

The end result was that prefetches were issued for up to 95% of the misses that occurred in the case without prefetching (referred to as the *coverage factor* in Fig. 9.3). Prefetching yielded significant time reduction in synchronization operations, using write buffers, and performing read operations. The best speedup achieved in Fig. 9.3 is 1.86, when the *pf4* prefetching strategy is compared with the *nopf* strategy. Still the prefetching benefits would be application-dependent. To introduce the pre-fetches in the MP3D code, only 16 lines of extra code were added to the source code.

9.1.3 Distributed Coherent Caches

While the coherence problem is easily solved for small bus-based multiprocessors through the use of snoopy cache coherence protocols, the problem is much more complicated for large-scale multiprocessors that use general interconnection networks. As a result, some large-scale multiprocessors did not provide caches (e.g. BBN Butterfly), others provided caches that must be kept coherent by software (e.g. IBM RP3), and still others provided full hardware support for coherent caches (e.g. Stanford Dash).

Dash Experience We evaluate the benefits when both private and shared read-write data are cacheable, as allowed by the Dash hardware coherent caches, versus the case where only private data are cacheable. Figure 9.4 presents a breakdown of the normalized execution times with and without cacheing of shared data for each of the applications. Private data are cached in both caches.

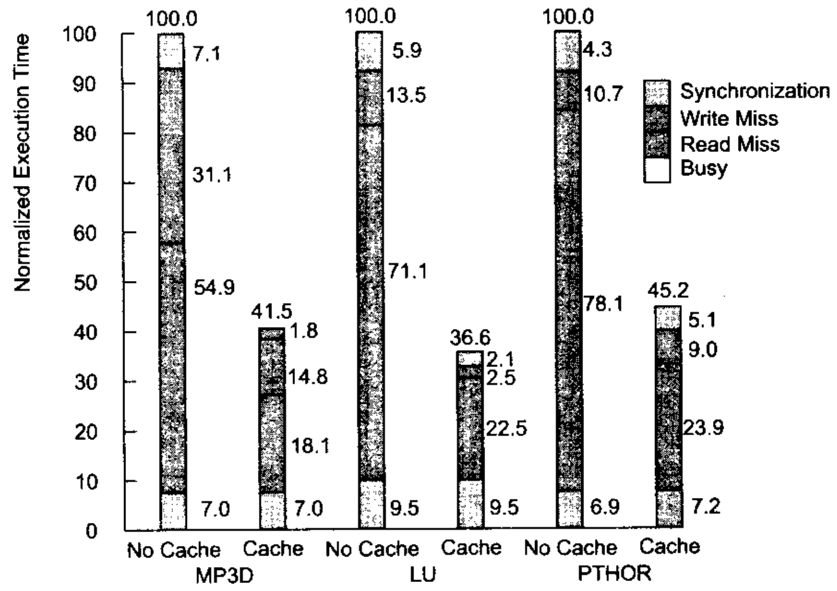


Fig. 9.4 Effect of cacheing shared data in simulated Dash benchmark experiments (Courtesy of Gupta et al, *Proc. Int. Symp. Comput. Archit.*, Toronto, Canada, May 1991)

The execution time of each application is normalized to the execution time of the case where shared data is not cached. The bottom section of each bar represents the busy time or useful cycles executed by the processor. The section above it represents the time that the processor is stalled waiting for reads. The section above that is the amount of time the processor is stalled waiting for writes to be completed. The top section, labeled "synchronization," accounts for the time processor is stalled due to locks and barriers.

Benefits of Cacheing As expected, the cacheing of shared read-write data provided substantial gains in performance, with benefits ranging from 2.2- to 2.7-fold improvement for the three Stanford benchmark programs. The largest benefit came from a reduction in the number of cycles wasted due to read misses. The cycles wasted due to write misses were also reduced, although the magnitude of the benefits varied across the three programs due to different write-hit ratios.

The cache-hit ratios achieved by MP3D, LU, and PTHOR were 80, 66, and 77%, respectively, for shared-read references, and 75, 97, and 47% for shared-write references. It is interesting to note that these hit ratios are substantially lower than the usual uniprocessor hit ratios.

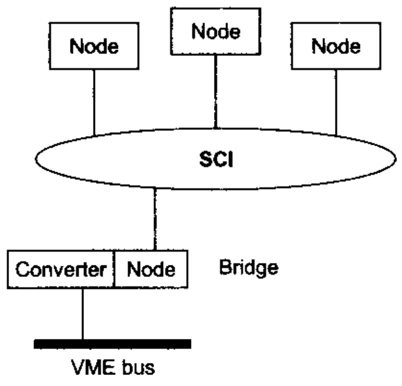
The low hit ratios arise from several factors: The data set size for engineering applications is large, parallelism decreases spatial locality in the application, and communication among processors results in invalidation misses. Still, hardware cache coherence is an effective technique for substantially increasing the performance with no assistance from the compiler or programmer.

9.1.4 Scalable Coherence Interface

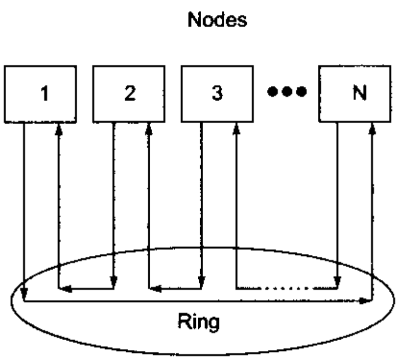
A scalable coherence interconnect structure with low latency is needed to extend from conventional bused backplanes to a fully duplex, point-to-point interface specification. The *scalable coherence interface* (SCI), which was introduced in Chapter 5, is specified in IEEE Standard 1596-1992. SCI supports unidirectional point-to-point connections, with two such links between each pair of nodes; packet-based communication is used, with routing.

Up to 64K processors, memory modules, or I/O nodes can effectively interface with a shared SCI interconnect. The cache coherence protocols used in SCI are directory-based. A sharing list is used to chain the distributed directories together for reference purposes.

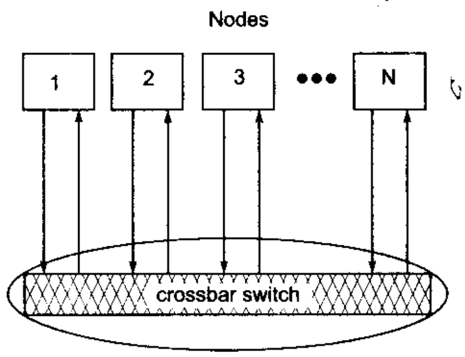
SCI Interconnect Models SCI defines the interface between nodes and the external interconnect, using 16-bit links with a bandwidth of up to 1 Gbyte/s per link. As a result, backplane buses have been replaced by unidirectional point-to-point links. A typical SCI configuration is shown in Fig. 9.5a. Each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



(a) Typical SCI configuration with bridge to other bus



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



(b) A crossbar multiprocessor

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

Each node has an input link and an output link which are connected from or to the SCI ring or crossbar. The bandwidth of SCI links depends on the physical standard chosen to implement the links and interfaces.

In such an environment, the concept of broadcast bus-based transactions is abandoned. Coherence protocols are based on point-to-point transactions initiated by a requester and completed by a responder. A ring interconnect provides the simplest feedback connections among the nodes.

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

Although SCI is scalable, the amount of memory used in the cache directories also scales up well. The performance of the SCI protocol does not scale, since when the sharing list is long, invalidations take proportionately longer time.

Sharing-List Structures Sharing lists are used in SCI to build chained directories for cache coherence use. The length of the sharing lists is effectively unbounded. Sharing lists are dynamically created, pruned, and destroyed. Each coherently cached block is entered onto a list of processors sharing the block.

Processors have the option of bypassing the coherence protocols for locally cached data. Cache blocks of 64 bytes are assumed. By distributing the directories among the sharing processors, SCI avoids scaling limitations imposed by using a central directory. Communications among sharing processors are supported by heavily shared memory controllers, as shown in Fig. 9.6.

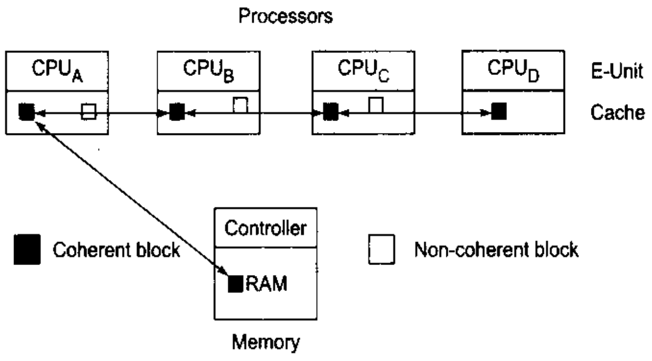


Fig. 9.6 SCI cache coherence protocol with distributed directories (Courtesy of D. V. James et al, *IEEE Computer*, 1990)

Other blocks may be locally cached and are not visible to the coherence protocols. For every block address, the memory and cache entries have additional tag bits which are used to identify the first processor (head) in the sharing list and to link the previous and following nodes.

Doubly linked lists are maintained between processors in the sharing list, with forward and backward pointers as shown by the double arrows in each link. Noncoherent copies may also be made coherent by page-level control. However, such higher-level software coherence protocols are beyond the scope of the SCI standard.

Sharing-List Creation The states of the sharing list are defined by the state of the memory and the states of list entries. Normally, the shared memory is either in a home (uncached) or a cached (sharing-list) state. The sharing-list entries specify the location of the entry in a multiple-entry sharing list, identify the only entry in the list, or specify the entry's cache properties, such as clean, dirty, valid, or stale.

The head processor is always responsible for list management. The stable and legal combinations of the memory and entry states can specify uncached data, clean or dirty data at various locations, and cached writable or stale data.

The memory is initially in the home state (uncached), and all cache copies are invalid. Sharing-list creation begins at the cache where an entry is changed from an invalid to a pending state. When a read-cache transaction is directed from a processor to the memory controller, the memory state is changed from uncached to cached and the requested data is returned.

The requester's cache entry state is then changed from a pending state to an only-clean state. Sharing-list creation is illustrated in Fig. 9.7a. Multiple requests can be simultaneously generated, but they are processed sequentially by the memory controller.

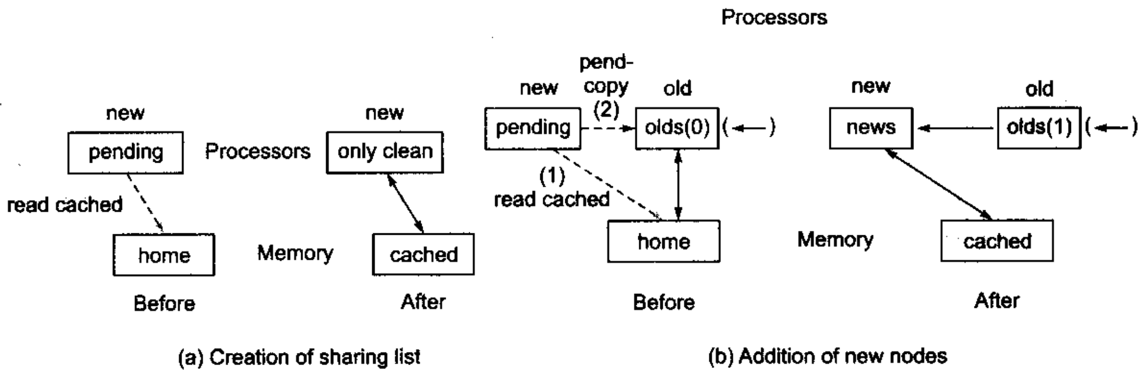


Fig. 9.7 **Sharing-list creation and update examples (Courtesy of D.V. James et al, *IEEE Computer*, 1990)**

Sharing-List Updates For subsequent memory access, the memory state is cached, and the cache head of the sharing list has possibly dirty data. As illustrated in Fig. 9.7b, a new requester (cache A) first directs its read-cache transaction to memory but receives a pointer to cache B instead of the requested data.

A second cache-to-cache transaction, called *prepend*, is directed from cache A to cache B. Cache B then sets its backward pointer to point to cache A and returns the requested data. The dashed lines correspond to transactions between a processor and memory or another processor. The solid lines are sharing-list pointers.

After the transaction, the inserted cache A becomes the new head, and the old head, cache B, is in the middle as shown by the new sharing list on the right in Fig. 9.7b.

Any sharing-list entry may delete itself from the list. Details of entry deletions are left as an exercise for the reader. Simultaneous deletions never generate deadlocks or starvation. However, the addition of new sharing-list entries must be performed in first-in-first-out order in order to avoid potential deadlocking dependences.

The head of the sharing list has the authority to purge other entries from the list to obtain an exclusive entry. Others may reenter as a new list head. Purges are performed sequentially. The chained-directory coherence protocols are fault-tolerant in that dirty data is never lost when transactions are discarded.

Implementation Issues SCI was developed to support multiprocessor systems with thousands of processors by providing a coherent distributed-cache image of distributed shared memory and bridges that interface with existing or future buses. It can support various multiprocessor topologies using Omega or crossbar networks.

Differential emitter coupled logic (ECL) signaling works well at SCI clock rates. The original SCI implementation uses a 16-bit data path at 2 ns per word. The interface is synchronously clocked. Several models of clock distribution are supported. With distributed shared-memory and distributed cache coherence protocols, the boundary between multiprocessors and multicomputers has become blurred in MIMD systems of this class.

9.1.5 Relaxed Memory Consistency

We have studied *weak consistency* (WC) (Sindhu et al, 1992) and *sequential consistency* (SC) in Section 5.4. Two additional memory models are introduced below for building scalable multiprocessors with distributed shared memory.

Processor Consistency Goodman (1989) introduced the *processor consistency* (PC) model in which writes issued by each individual processor are always in program order. However, the order of writes from two different processors can be out of program order. In other words, consistency in writes is observed in each processor, but the order of reads from each processor is not restricted as long as they do not involve other processors.

The PC model relaxes from the SC model by removing some restrictions on writes from different processors. This opens up more opportunities for write buffering and pipelining. Two conditions related to other processors are required for ensuring processor consistency:

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

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

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

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

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

(1) Before an ordinary *read* or *write* access is allowed to perform with respect to any other processor, all previous *acquire* accesses must be performed.

(2) Before a *release* access is allowed to perform with respect to any other processor, all previous ordinary *read* and *store* accesses must be performed.

(3) *Special accesses* are processor-consistent with one another. The ordering restrictions imposed by weak consistency are not present in release consistency. Instead, release consistency requires processor consistency and not sequential consistency.

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

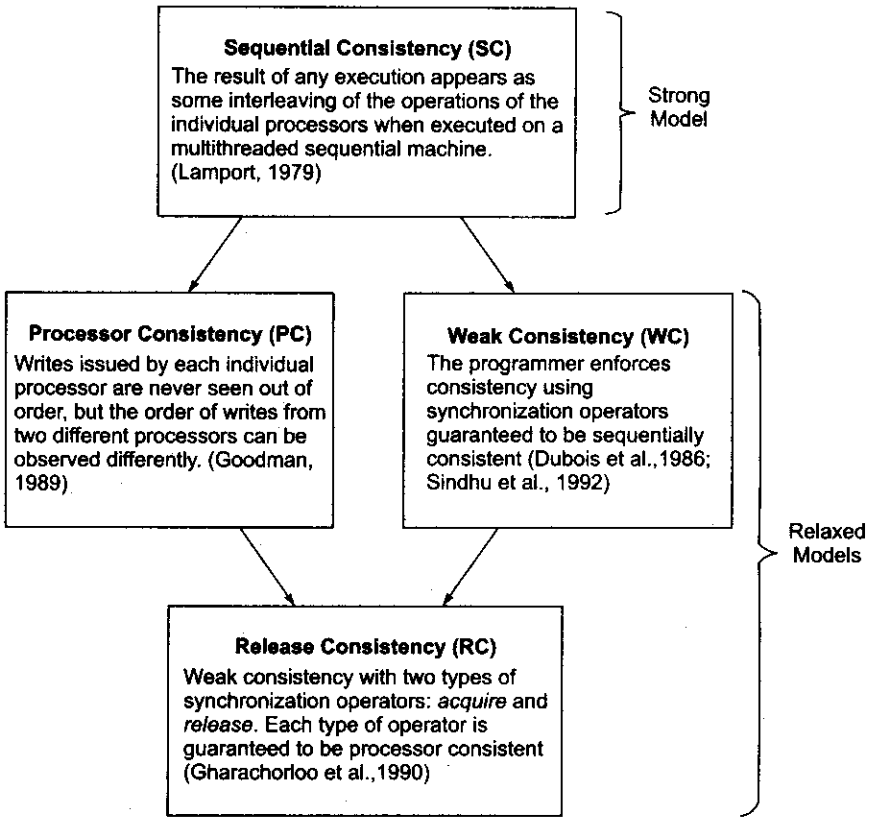

Sequential Consistency (SC)
The result of any execution appears as some interleaving of the operations of the individual processors when executed on a multithreaded sequential machine. (Lamport, 1979)
Strong Model
Processor Consistency (PC)
Writes issued by each individual processor are never seen out of order, but the order of writes from two different processors can be observed differently. (Goodman, 1989)
Weak Consistency (WC)
The programmer enforces consistency using synchronization operators guaranteed to be sequentially consistent (Dubois et al., 1986; Sindhu et al., 1992)
Relaxed Models
Release Consistency (RC)
Weak consistency with two types of synchronization operators: acquire and release. Each type of operator is guaranteed to be processor consistent (Gharachorloo et al., 1990)

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

The cost of implementing RC over that for SC arises from the extra hardware cost of providing a lockup-free cache and keeping track of multiple outstanding requests. Although this cost is not negligible, the same hardware features are also required to support prefetching and multiple contexts.

Effect of Release Consistency Figure 9.9 presents the breakdown of execution times under SC and RC for the three applications. The execution times are normalized to those shown in Fig. 9.3 with shared data cached. As can be seen from the results, RC removes all idle time due to write-miss latency.

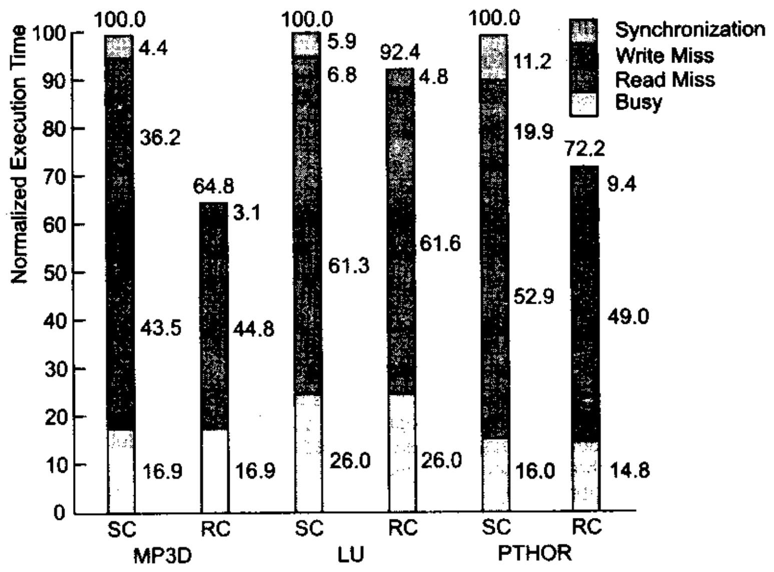


Fig. 9.9 Effect of relaxing the shared-memory model from sequential consistency (SC) to release consistency (RC) (Courtesy of Gupta et al, *Proc. Int. Symp. Comput. Archit.*, Toronto, Canada, May 1991)

The gains are large in MP3D and PTHOR since the write-miss time constitutes a large portion of the execution time under SC (35 and 20%, respectively), while the gain is small in LU due to the relatively small write-miss time under SC (7%).

Effect of Combining Mechanisms The effect of combining various latency-hiding mechanisms is illustrated by Fig. 9.10 based on the MP3D benchmark results obtained at Stanford University. The idea of using *multiple-context* processors will be described in Section 9.2. However, the effect of integrating MC with other latency-hiding mechanisms is presented below.

The busy parts of the execution times in Fig. 9.10 are equal in all combinations. This is the CPU busy time for executing the MP3D program. The idle part in the bar diagram corresponds to memory latency and includes all cache-miss penalties. All the times are normalized with respect to the execution time (100 units) required in a *cache-coherent* system. The leftmost time bar (with 241 units) corresponds to the worst case of using a private cache exclusively without shared reads or writes. Long overhead is experienced in this case due to excessive cache misses. The use of a cache-coherent system shows a 2.41-fold improvement over the private case. All the remaining cases are assumed to use hardware coherent caches.

The use of *release consistency* shows a 35% further improvement over the coherent system. The adding of prefetching reduces the time further to 44 units. The best case is the combination of using coherent caches, RC, and *multiple contexts* (MC). The rightmost time bar is obtained from applying all four mechanisms. The combined results show an overall speedup of 4 to 7 over the case of using private caches.

The above and other uncited benchmark results reported at Stanford suggest that a coherent cache and relaxed consistency uniformly improve performance. The improvements due to prefetching and multiple contexts are sizable but are much more application-dependent. Combinations of the various latency-hiding mechanisms generally attain a better performance than each one on its own.

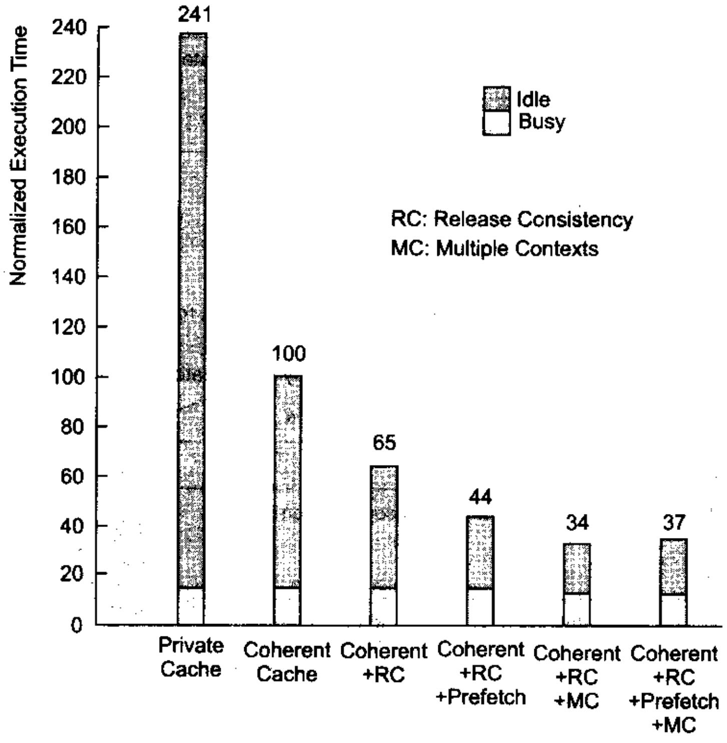


Fig. 9.10 Effect of combining various latency-hiding mechanisms from the MP3D benchmark on a simulated Dash multiprocessor (Courtesy of Gupta, 1992)

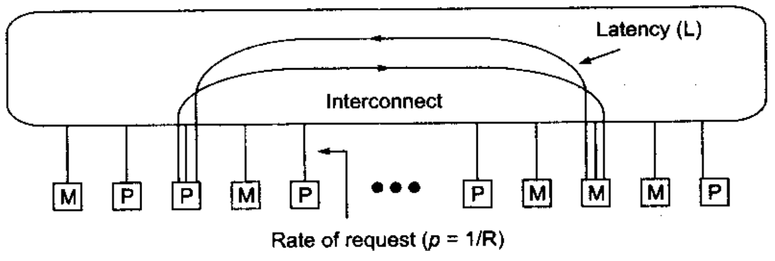
9.2 PRINCIPLES OF MULTITHREADING

This section considers multithreaded processors and multidimensional system architectures. Only control-flow approaches are described here. Fine-grain machines are studied in Section 9.3, von Neumann multithreading in Section 9.4, and dataflow multithreading in Section 9.5. Recent developments in multithreading support by processor hardware are discussed in Chapters 12 and 13.

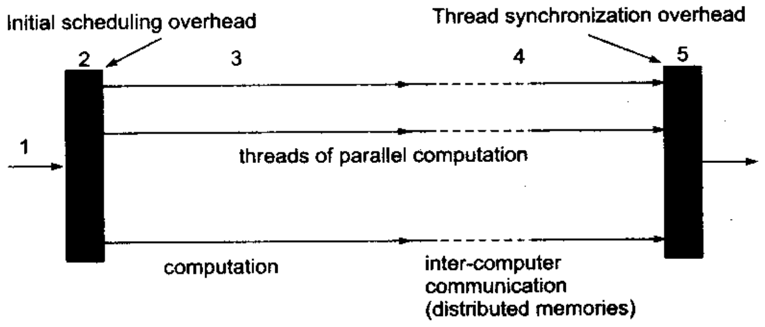
9.2.1 Multithreading Issues and Solutions

Multithreading demands that the processor be designed to handle multiple contexts simultaneously on a context-switching basis. We first specify the typical architecture environment using multiple-context processors. Next we present a multithreaded computation model. Then we look further into the latency and synchronization problems and discuss their solutions in this environment.

Architecture Environment One possible multithreaded MPP system is modeled by a network of processor (P) and memory (M) nodes as depicted in Fig. 9.11a. The distributed memories form a global address space. Four machine parameters are defined below to analyze the performance of this network:



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


Initial scheduling overhead
Thread synchronization overhead
1
2
3
4
5
threads of parallel computation
computation
inter-computer communication (distributed memories)

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

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

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

(2) *The number of threads* (N): This is the number of threads that can be interleaved in each processor. A *thread* is represented by a *context* consisting of a program counter, a register set, and the required context status words.

(3) *The context-switching overhead* (C): This refers to the cycles lost in performing context switching in a processor. This time depends on the switch mechanism and the amount of processor states devoted to maintaining active threads.

(4) *The interval between switches* (R): This refers to the cycles between switches triggered by remote reference. The inverse $p = 1/R$ is called the *rate of requests* for remote accesses. This reflects a combination of program behavior and memory system design.

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

Multithreaded Computations Bell (1992) has described the structure of the multithreaded parallel computations model shown in Fig. 9.11b. The computation starts with a sequential thread (1), followed by supervisory scheduling (2) where the processors begin threads of computation (3), by intercomputer messages that update variables among the nodes when the computer has a distributed memory (4), and finally by synchronization prior to beginning the next unit of parallel work (5).

The communication overhead period (4) inherent in distributed memory structures is usually distributed throughout the computation and is possibly completely overlapped. Message-passing overhead (send and receive calls) in multicomputers can be reduced by specialized hardware operating in parallel with computation.

Communication bandwidth limits granularity, since a certain amount of data has to be transferred with other nodes in order to complete a computational grain. Message-passing calls (4) and synchronization (5) are nonproductive. Fast mechanisms to reduce or to hide these delays are therefore needed. Multithreading is not capable of speedup in the execution of single threads, while weak ordering or relaxed consistency models are capable of doing this.

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

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

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

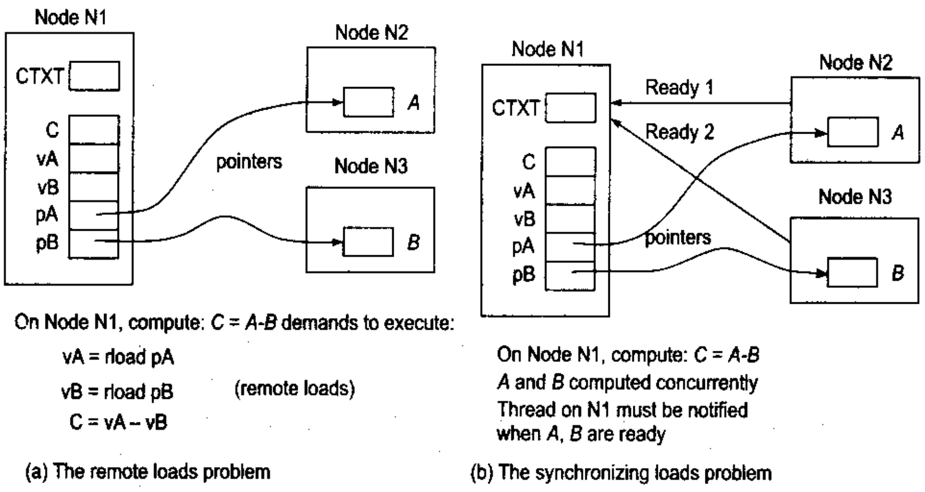

Node N1
CTXT
C
vA
vB
pA
pB
Node N2
A
pointers
Node N3
B
On Node N1, compute: C = A-B demands to execute:
vA = rload pA
vB = rload pB
(remote loads)
C = vA – vB
(a) The remote loads problem
Node N1
CTXT
C
vA
vB
pA
pB
Ready 1
Ready 2
Node N2
A
pointers
Node N3
B
On Node N1, compute: C = A-B
A and B computed concurrently
Thread on N1 must be notified when A, B are ready
(b) The synchronizing loads problem

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

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

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

The key issue involved in remote loads is how to avoid idling in node N1 during the load operations. The latency caused by remote loads is an architectural property. The latency caused by synchronizing loads also depends on scheduling and the time it takes to compute A and B, which may be much longer than the transit latency. The synchronization latency is often unpredictable, while the remote-load latencies are often predictable.

Multithreading Solutions This solution to asynchrony problems is to multiplex among many threads: When one thread issues a remote-load request, the processor begins work on another thread, and so on (Fig. 9.13a). Clearly, the cost of thread switching should be much smaller than that of the latency of the remote load, or else the processor might as well wait for the remote load's response.

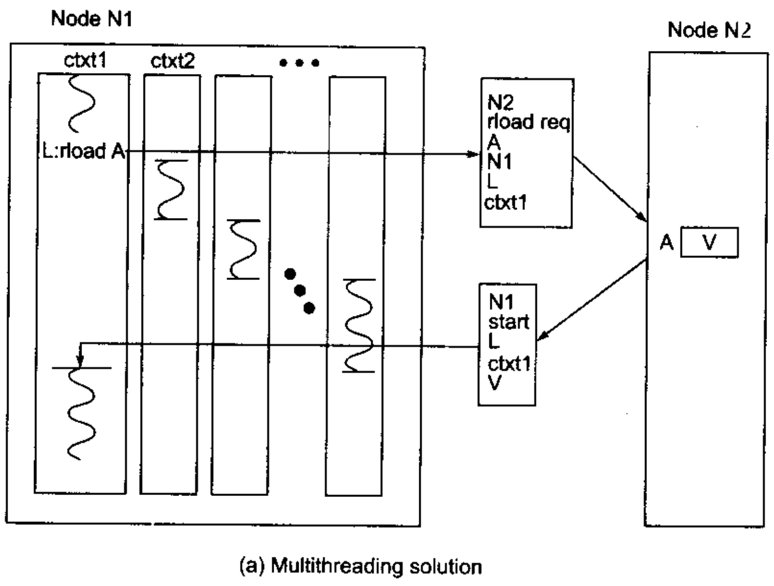
As the internode latency increases, more threads are needed to hide it effectively. Another concern is to make sure that messages carry continuations. Suppose, after issuing a remote load from thread T_1 (Fig. 9.13a), we switch to thread T_2, which also issues a remote load. The responses may not return in the same order. This may be caused by requests traveling different distances, through varying degrees of congestion, to destination nodes whose loads differ greatly, etc.

One way to cope with the problem is to associate each remote load and response with an identifier for the appropriate thread, so that it can be reenabled on the arrival of a response. These thread identifiers are referred to as *continuations* on messages. A large *continuation name space* should be provided to name an adequate number of threads waiting for remote responses.

The size of the hardware-supported continuation in a name space varies greatly in different system designs: from 1 in the Dash, 4 in the Alewife, 64 in the HEP, and 1024 in the Tera (Section 9.4) to the local memory address space in the Monsoon, Hybrid Dataflow/von Neumann, MDP (Section 9.3), and *T (Section 9.5). Of course, if the hardware-supported name space is small, one can always virtualize it by multiplexing in software, but this has an associated overhead.

Distributed Cacheing The concept of distributed cacheing is shown in Fig. 9.13b. Every memory location has an owner node. For example, N1 owns B and N2 owns A. The directories are used to contain import-export lists and state whether the data is *shared* (for reads, many caches may hold copies) or *exclusive* (for writes, one cache holds the current value).

The directories multiplex among a small number of contexts to cover the cache loading effects. The MIT Alewife, KSR-l, and Stanford Dash have implemented directory-based coherence protocols. It should be noted that distributed cacheing offers a solution for the remote-loads problem, but not for the synchronizing-loads problem. Multithreading offers a solution for remote loads and possibly for synchronizing loads. However, the two approaches can be combined to solve both types of remote-access problems.



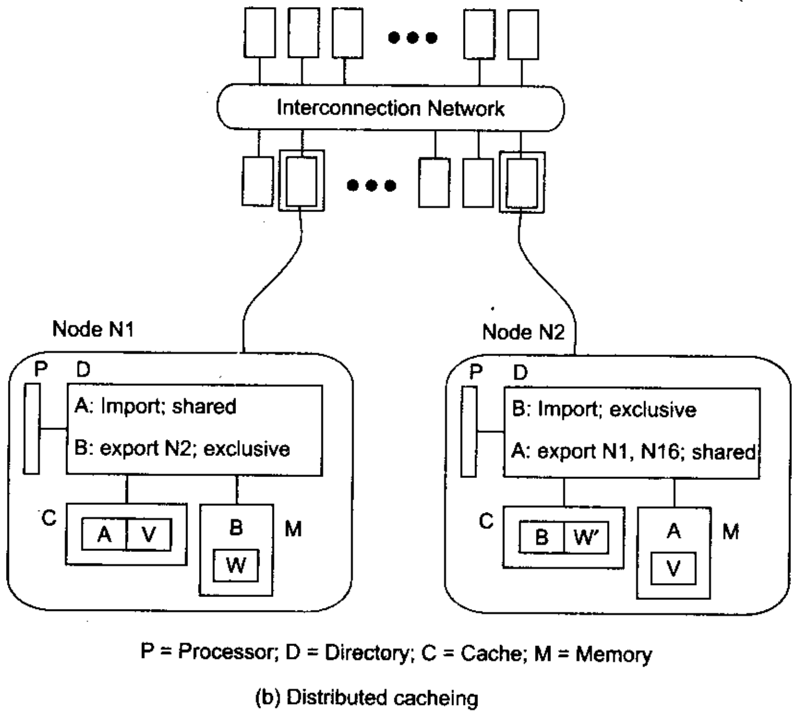


Fig. 9.13 Two solutions for overcoming the asynchrony problems (Courtesy of R. S. Nikhil, Digital Equipment Corporation, 1992)

9.2.2 Multiple-Context Processors

Multithreaded systems are constructed with *multiple-context* (or *multithreaded)* processors. In this section, we study an abstract model based on the work of Saavedra et al (1990). We then present an example of this type of processor. We discuss the processor efficiency issue as a function of memory latency (L), the number of contexts (N), and context-switching overhead (C).

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

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

$$Efficiency = \frac{busy}{busy + switching + idle} \tag{9.1}$$

where *busy, switching,* and *idle* represent the amount of time, measured over some large interval, that the processor is in the corresponding state. The basic idea behind a multithreaded machine is to interleave the execution of several contexts in order to dramatically reduce the value of *idle,* but without overly increasing the magnitude of *switching.*

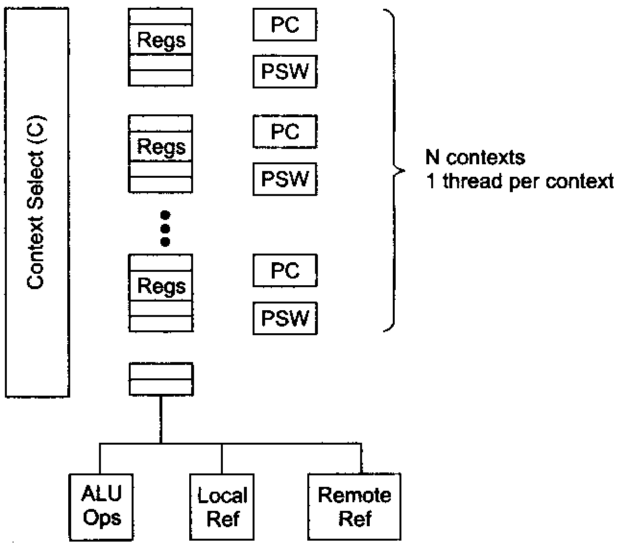
The state of a processor is determined by the disposition of the various contexts on the processor. During its lifetime, a context cycles through the following states: *ready, running, leaving,* and *blocked.* There can be at most one context running or leaving. A processor is *busy* if there is a context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

A running context keeps the processor busy until it issues an operation that requires a context switch. The context then spends C cycles in the *leaving* state, then goes into the *blocked* state for L cycles, and finally reenters the *ready* state. Eventually the processor will choose it and the cycle will start again.

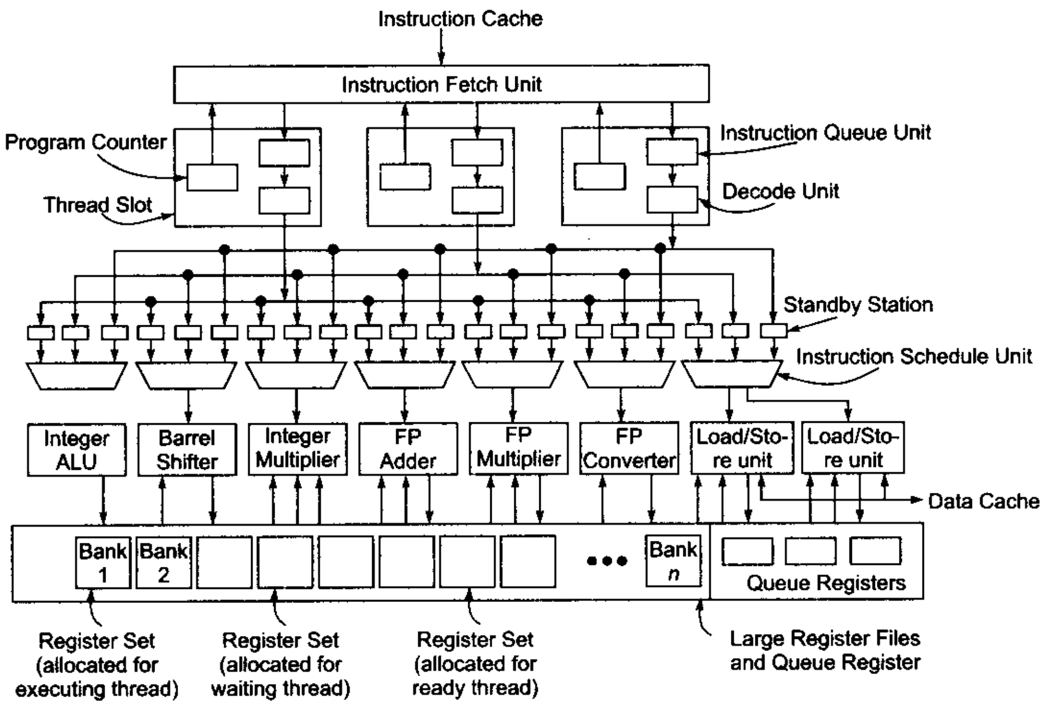
The abstract model shown in Fig. 9.14a assumes one thread per context, and each context is represented by its own program counter (PC), register set, and process status word (PSW). An example multithreaded processor in which three thread slots ($N = 3$) are provided is shown in Fig. 9.14b.

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

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



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



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

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

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

An instruction fetch unit fetches at most B instructions for one thread every C cycles from the instruction cache and attempts to fill the buffers in the instruction queue unit. This fetching operation is done in an interleaved fashion for multiple threads. So, on the average, the buffer in one instruction queue unit is filled once in B cycles.

When one of the threads encounters a branch instruction, however, that thread can preempt the prefetching operation. The instruction cache and fetch unit might become a bottleneck for a processor with many thread slots. In such cases, a bigger and/or faster cache and another fetch unit would be needed.

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

(1) *Switch on cache miss*—This policy corresponds to the case where a context is preempted when it causes a cache miss. In this case, R is taken to be the average interval between misses (in cycles), and L the time required to satisfy the miss. Here, the processor switches contexts only when it is certain that the current one will be delayed for a significant number of cycles.

(2) *Switch on every load*—This policy allows switching on every load, independent of whether it will cause a miss or not. In this case, R represents the average interval between loads. A general multithreading model assumes that a context is blocked for L cycles after every switch; but in the case of a switch-on-load processor, this happens only if the load causes a cache miss.

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

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

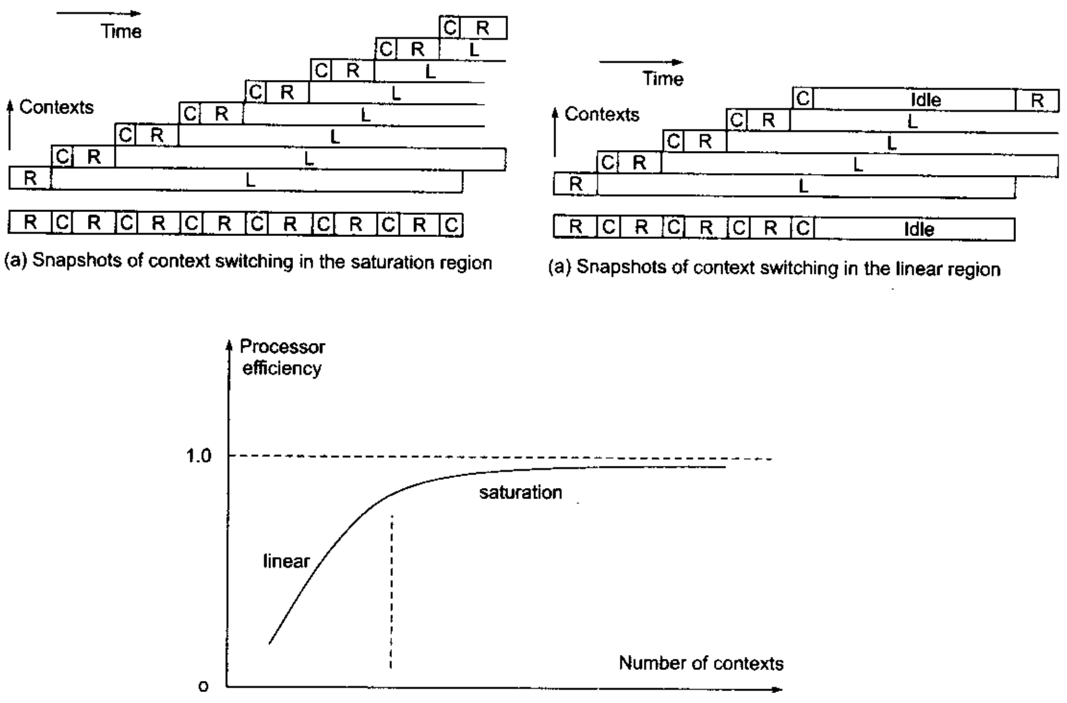
(4) *Switch on block of instruction*—Blocks of instructions from different threads are interleaved. This will improve the cache-hit ratio due to locality. It will also benefit single-context performance.

Processor Efficiencies A single-thread processor executes a context until a remote reference is issued (R cycles) and then is idle until the reference completes (L cycles). There is no context switch and obviously no switch overhead. We can model this behavior as an alternating renewal process having a cycle of $R + L$. In terms of Eq. 9.1, R and L correspond to the amount of time during a cycle that the processor is *busy* and *idle*, respectively. Thus the efficiency of a single-threaded machine is given by

$$E_1 = \frac{R}{R+L} = \frac{1}{1+L/R} \tag{9.2}$$

This shows clearly the performance degradation of such a processor in a parallel system with a large memory latency.

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



(a) Snapshots of context switching in the saturation region

(a) Snapshots of context switching in the linear region

(c) Efficiency curve

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

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

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

Observe that the efficiency in saturation is independent of the latency and also does not change with a further increase in the number of contexts.

Saturation is achieved when the time the processor spends servicing the other threads exceeds the time required to process a request, i.e., when $(N-1)(R+C) > L$. This gives the saturation point, under constant run length, as

$$N_d = \frac{L}{R+C} + 1 \tag{9.4}$$

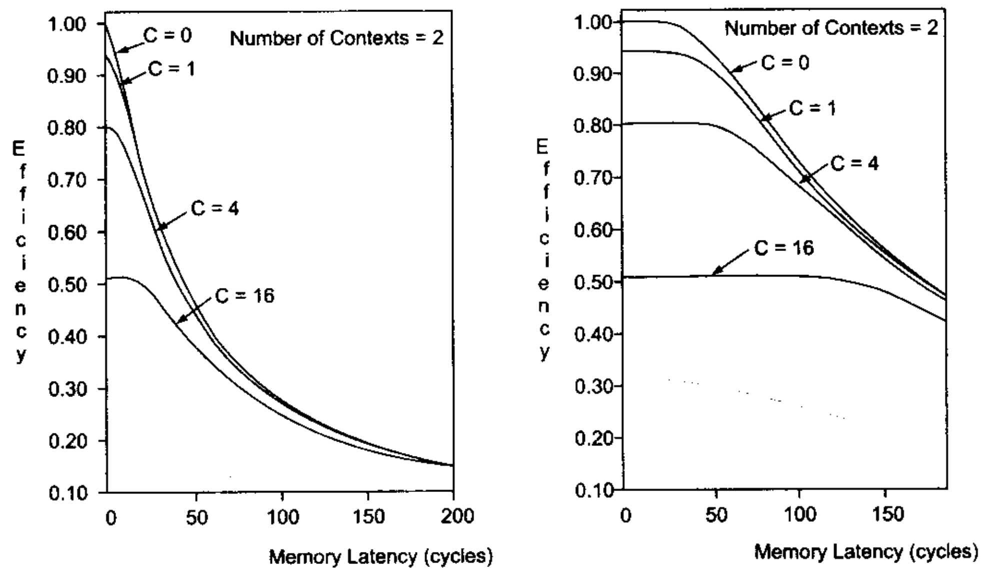
(2) *Linear region*—When the number of contexts is below the saturation point, there may be no ready contexts after a context switch, so the processor will experience idle cycles. The time required to switch to a ready context, execute it until a remote reference is issued, and process the reference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

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

Observe that the efficiency increases linearly with the number of contexts until the saturation point is reached and beyond that remains constant. The equation for E_{sat} gives the fundamental limit on the efficiency of a multithreaded processor and underlines the importance of the ratio C/R. Unless the context switch is extremely cheap, the remote reference rate must be kept low.

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

In Fig. 9.16, the processor efficiency is plotted as a function of the memory latency L with an average run length $R = 16$ cycles. The $C = 0$ curve corresponds to zero switching overhead. With $C = 16$ cycles, about 50% efficiency can be achieved. These results are based on a Markov model of multithreaded architecture by Saavedra (1992). It should be noted that multithreading increases both processor efficiency and network traffic. Tradeoffs do exist between these two opposing goals, and this has been discussed in a paper by Agarwal (1992).



(a) Two contexts per processor (b) Six contexts per processor

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

9.2.3 Multidimensional Architectures

In order to enhance the scalability of multiprocessor systems, many research groups have explored economical and multidimensional architectures that support fast communication, coherence extension, distributed shared memory, and modular packaging.

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

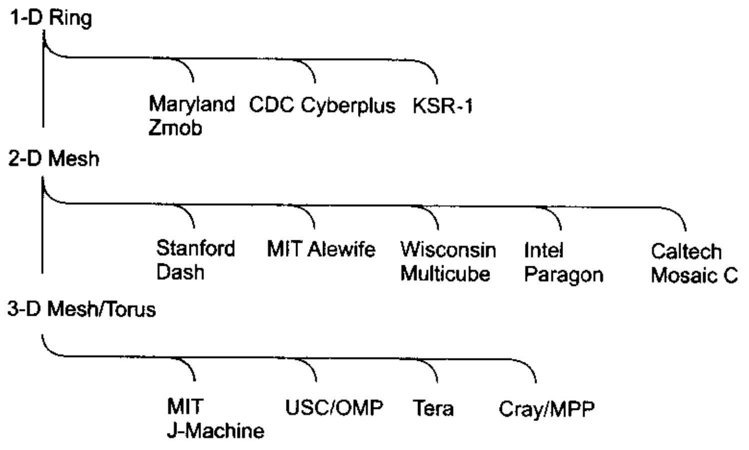
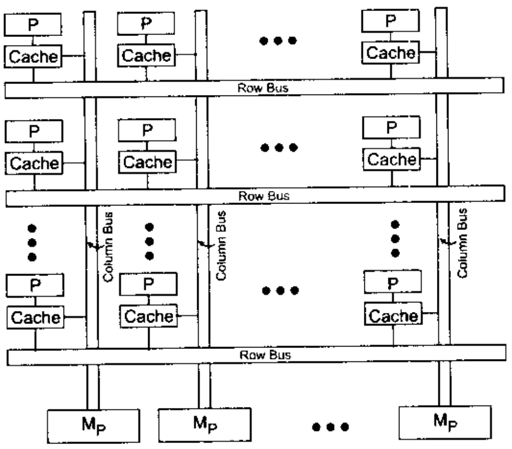


Fig. 9.17 The evolution from one-dimensional ring to two-dimensional mesh and then to three-dimensional mesh/torus architecture for building massively parallel processors.

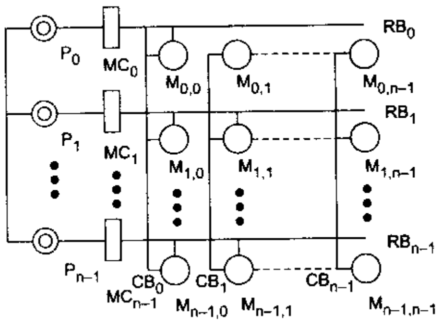
Two-dimensional meshes were adopted in the Stanford Dash, the MIT Alewife, the Wisconsin Multicube, the Intel Paragon, and the Caltech Mosaic C. A three-dimensional mesh/torus was implemented in the MIT J-Machine, the Tera computer, and in the Cray/MPP architecture, called T3D. The USC *orthogonal multiprocessor* (OMP) could be extended to higher dimensions. However, it becomes more difficult to build higher-dimensional architectures with conventional two-dimensional circuit boards.

Instead of using hierarchical buses or switched network architectures in one dimension, multiprocessor architectures can be extended to a higher *dimensionality* or *multiplicity* along each dimension. The concepts are described below for two- and three-dimensional meshes proposed for the Multicube and OMP architectures, respectively.

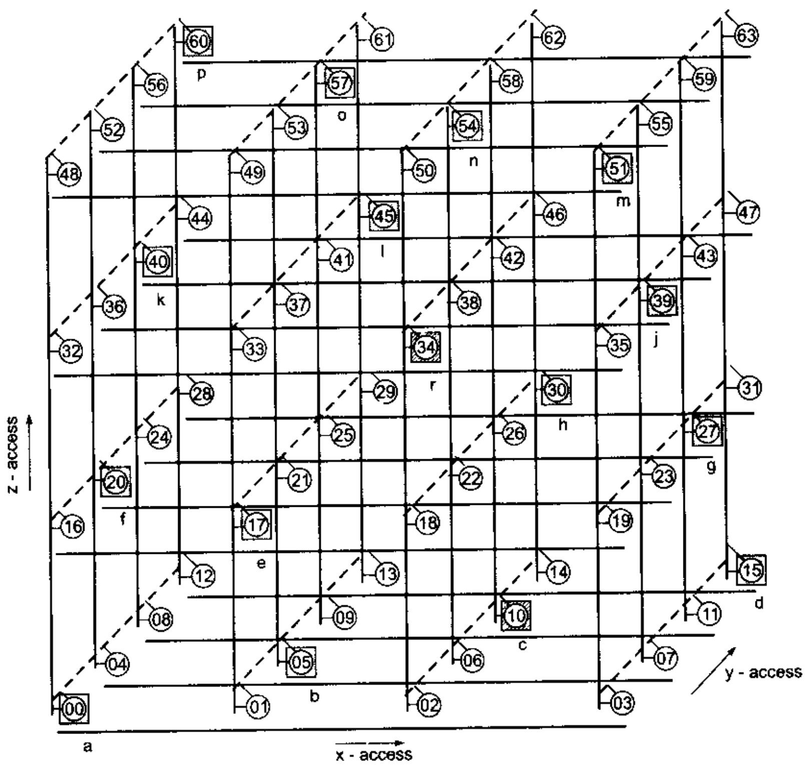
The Wisconsin Multicube This architecture was proposed by Goodman and Woest (1988) at the University of Wisconsin. It employed a snooping cache system over a grid of buses, as shown in Fig. 9.18a. Each processor was connected to a multilevel cache.



(a) The Wisconsin Multicube



(b) The two-dimensional OMP(2,*n*). (Pi: Processors; Mij: memory modules; RB_i: row buses; CBj: column buses)



(c) The 3-D OMP (3,4) architecture. (Processors are labeled a, b, ..., p; memory modules are labeled 00.01..., 63)

Fig. 9.18 The Multicube and orthogonal multiprocessor architectures (Courtesy of Goodman and Woest, 1988, and of Hwang et al, 1989)

The first-level cache, called the *processor cache*, was a high-performance SRAM cache designed with the traditional goal of minimizing memory latency. A second-level cache, referred to as the *snooping cache*, was a very large cache designed to minimize bus traffic.

Each snooping cache monitored two buses, a row bus and a column bus, in order to maintain data consistency among the snooping caches. Consistency between the two cache levels was maintatined by using a write-through strategy to ensure that the processor cache is always a strict subset of the snooping cache. The main memory was divided up among the column buses. All processors tied to the same column shared the same home memory. The row buses were used for intercolumn communication and cache coherence control.

The proposed architecture was an example of a new class of interconnection topologies, the *multicube*, consisting of $N = n^k$ processors, where each processor was connected to k buses and each bus was connected to n processors. The hypercube is a special case where $n = 2$. The Wisconsin Multicube was a two-dimensional multicube ($k = 2$), where n scaled to about 32, resulting in a proposed system of over 1000 processors.

The Orthogonal Multiprocessor In the proposed OMP architecture (Fig. 9.18b), n processors simultaneously access n rows or n columns of interleaved memory modules. The *$n \times n$ memory mesh* is interleaved in both dimensions. In other words, each row is n-way interleaved and so is each column of memory modules. There are $2n$ logical buses spanning in two orthogonal directions.

The synchronized row access or column access must be performed exclusively. In fact, the row bus R_i and the column bus C_i can be the same physical bus because only one of the two will be used at a time. The memory controller (MC) in Fig. 9.18b synchronizes the row access and column access of the shared memory.

The OMP architecture supports special-purpose computations in which data sets can be regularly arranged as matrices. Simulated performance results obtained at USC verified the effectiveness of using an OMP in matrix algebraic computations or in image processing operations.

In Fig. 9.18b, each of the memory modules M_{ij} is shared by two processors P_i and P_j. In other words, the physical address space of processor P_i covers only the ith row or the ith column of the memory mesh. The OMP is well suited for SPMD operations, in which n processors are synchronized at the memory-access level when data sets are vectorized in matrix format.

Multidimensional Extensions The above OMP architecture can be generalized to higher dimensions. A generalized orthogonal multiprocessor is denoted as an OMP(n, k), where n is the *dimension* and k is the *multiplicity*. There are $p = k^{n-1}$ processors and $m = k^n$ memory modules in the system, where $p \gg n$ and $p \gg k$.

The system uses p memory buses, each spanning into n dimensions. But only one dimension is used in a given memory cycle. There are k memory modules attached to each spanning bus.

Each module is connected to n out of p buses through an n-way switch. It should be noted that the dimension n corresponds to the number of accessible ports that each memory module has. This implies that each module is shared by n out of $p = k^{n-1}$ processors. For example, the architecture of an OMP(3,4) is shown in Fig. 9.18c, where the circles represent memory modules, the squares processor modules, and the circles inside squares computer modules.

The 16 processors orthogonally access 64 memory modules via 16 buses, each spanning into three directions, called the *x-access*, *y-access*, and *z-access*, respectively. Various sizes of OMP architecture for different values of n and k are given in Table 9.2. A five-dimensional OMP with multiplicity $k = 16$ has 64K processors.

Table 9.2 *Orthogonal Multiprocessor of Dimension n and Multiplicity k*

$OMP(n, k)$	$p = k^{n-1}$	$m = k^n$
OMP(2, 8)	8	64
OMP(2, 16)	16	256
OMP(3, 8)	64	512
OMP(3, 16)	256	4096
OMP(4, 8)	512	4096
OMP(4, 16)	4096	65,536
OMP(5,16)	65,536	1,048,576

Note: p = number of processors; m = number of memory modules.

9.3 FINE-GRAIN MULTICOMPUTERS

Traditionally, shared-memory multiprocessors like the Cray Y-MP were used to perform coarse-grain computations in which each processor executed programs having tasks of a few seconds or longer. Message-passing multicomputers are used to execute medium-grain programs with approximately 10-ms task size as in the iPSC/1. In order to build MPP systems, we may have to explore a higher degree of parallelism by making the task grain size even smaller.

Fine-grain parallelism was utilized in SIMD or data-parallel computers like the CM-2 or on the message-driven J-Machine and Mosaic C to be described below. We first characterize fine-grain parallelism and discuss the network architectures proposed for such systems. Special attention is paid to the efficient hardware or software mechanisms developed for achieving fine-grain MIMD computation.

9.3.1 Fine-Grain Parallelism

We compare below the grain sizes, communication latencies, and concurrency in four classes of parallel computers. This comparison leads to the rationales for developing fine-grain multicomputers. In Chapter 13 we shall review recent developments.

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

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

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

Fine-Grain Parallelism The *grain size* T_g is measured by the execution time of a typical program, including both computing time and communication time involved. Supercomputers handle large grain. Both the CM-2 and the J-Machine were designed as fine-grain machines. The iPSC/1 was a relatively medium-grain machine compared with the rest.

Large grain implies lower concurrency or a lower DOP (degree of parallelism). Fine grain leads to a much higher DOP and also to higher communication overhead. SIMD machines used hardwired synchronization and massive parallelism to overcome the problems of long network latency and slow processor speed. *Fine-grain multicomputers*, like the J-Machine *and* Caltech Mosaic, were designed to lower both the grain size and the communication overhead compared to those of traditional multicomputers.

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

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

9.3.2 The MIT J-Machine

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

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

Network addressing limited the size of the J-Machine to a maximum configuration of 65,536 nodes, corresponding to a three-dimensional mesh with $32 \times 32 \times 64$ nodes. The architecture of the three-dimensional mesh or a general *k*-ary *n-cube* was shown in Fig. 2.20 for the case of $k = 4$. All hidden parts (nodes and links) are not shown for purposes of clarity. Clearly, every node has a constant node degree of 6, and there are three rings crossing each node along the three dimensions. The end-around connections can be folded (Fig. 2.21b) to balance the wire length on all channels.

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

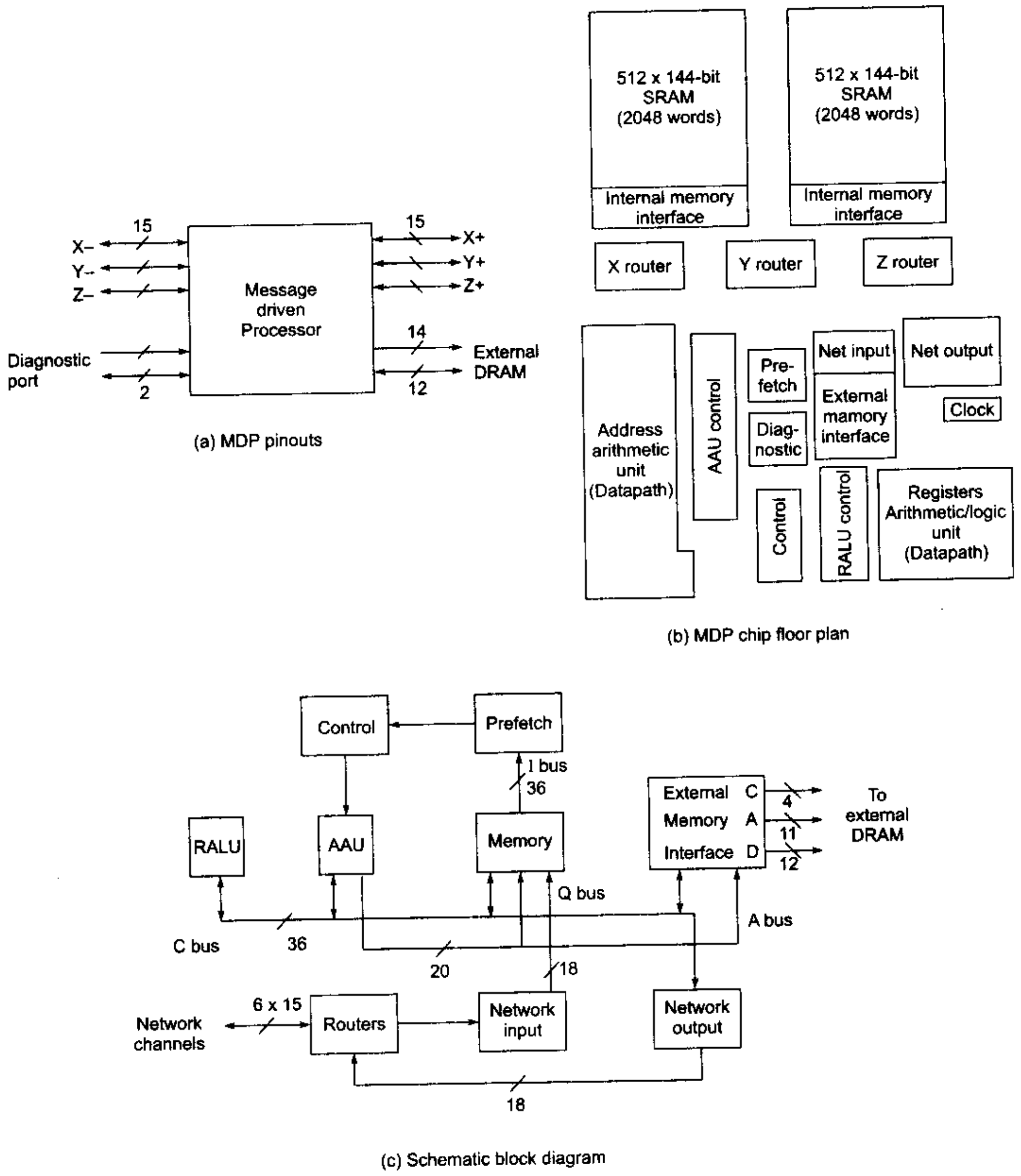


Fig. 9.19 **The message-driven processor (MDP) architecture (Courtesy of W. Dally et al; reprinted with permission from *IEEE Micro*, April 1992)**

The MDP created a task to handle each arriving message. Messages carrying these tasks drove each computation. MDP was a general-purpose multicomputer processing node that provided the communication, synchronization, and global naming mechanisms required to efficiently support fine-grain, concurrent programming models. The grain size was as small as 8-word objects or 20-instruction tasks. As we have seen, fine-grain programs typically execute from 10 to 100 instructions between communication and synchronization actions.

MDP chips provided inexpensive processing nodes with plentiful VLSI commodity parts to construct the Jellybean Machine (J-Machine) multicomputer. As shown in Fig. 9.19a, the MDP appeared as a component with a memory port, six two-way network ports, and a diagnostic port.

The memory port provided a direct interface to up to 1M words of ECC DRAM, consisting of 11 multiplexed address lines, a 12-bit data bus, and 3 control signals. Prototype J-Machines used three 1M × 4 static-column DRAMs to form a four-chip processing node with 262,144 words of memory. The DRAMs cycled three times to access a 36-bit data word and a fourth time to check or update the ECC check bits.

The network ports connected MDPs together in a three-dimensional mesh network. Each of the six ports corresponded to one of the six cardinal directions (+x, −x, +y, −y, +z, −z) and consisted of nine data and six control lines. Each port connected directly to the opposite port on an adjacent MDP.

The diagnostic port could issue supervisory commands and read and write MDP memory from a console processor (host). Using this port, a host could read or write at any location in the MDP's address space, as well as reset, interrupt, halt, or single-step the processor. The MDP chip floor plan is shown Fig. 9.19b.

Figure 9.19c shows the components built inside the MDP chip. The chip included a conventional microprocessor with prefetch, control, register file and ALU (RALU), and memory blocks. The network communication subsystem comprised the routers and network input and output interfaces. The *address arithmetic unit* (AAU) provided addressing functions. The MDP also included a DRAM interface, control clock, and diagnostic interface.

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

Separate register sets were provided to support rapid switching among three execution levels: background, priority 0 (P0), and priority 1 (P1). The MDP executed at the background level while no message created a task, and initiated execution upon message arrival at P0 or P1 level depending on the message priority.

P1 level had higher priority than P0 level. The register set at each priority level included four GPRs, four address registers, four ID registers, and one instruction pointer (IP). The ID registers were not used in the background register set.

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

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

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

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

The J-Machine was a three-dimensional mesh with two-way channels, dimension-order routing, and blocking flow control (Fig. 9.20). The faces of the network cube were open for use as I/O ports to the machine. Each channel could sustain a data rate of 288 Mbps (million bits per second). All three dimensions could operate simultaneously for an aggregate data rate of 864 Mbps per node.

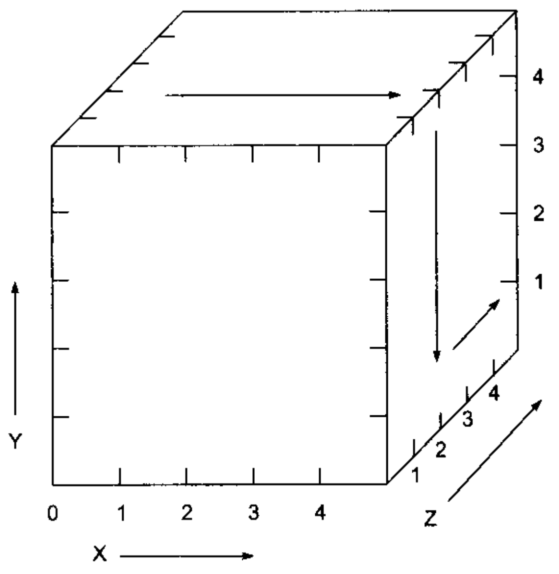


Fig. 9.20 E-cube routing from node (1, 5, 2) to node (5, 1, 4) on a 6-ary 3-cube

Message Format and Routing The J-Machine used deterministic dimension-order E-cube routing. As shown in Fig. 9.20, all messages routed first in the x-dimension, then in the y-dimension, and then in the z-dimension. Since messages routed in dimension order and messages running in opposite directions along the same dimension cannot block, resource cycles were thus avoided, making the network provably deadlock-free.

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

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

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

The MDP supported a broad range of parallel programming models, including shared memory, data-parallel, dataflow, actor, and explicit message passing, by providing a low-overhead primitive mechanism for communication, synchronization, and naming.

Its communication mechanisms permitted a user-level task on one node to send a message to any other node in a 4096-node machine in less than 2 μs. This process did not consume any processing resources on intermediate nodes, and it automatically allocated buffer memory on the receiving node. On message arrival, the receiving node created and dispatched a task in less than 1 μs.

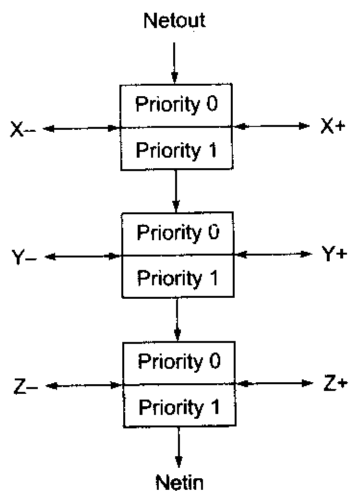
Presence tags provided synchronization on all storage locations. Three separate register sets allowed fast context switching. A translation mechanism maintained bindings between arbitrary names and values and supported a global virtual address space. These mechanisms were selected to be general and amenable to efficient hardware implementation. The J-Machine used wormhole routing and blocking flow control. A combining-tree approach was used for synchronization.

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

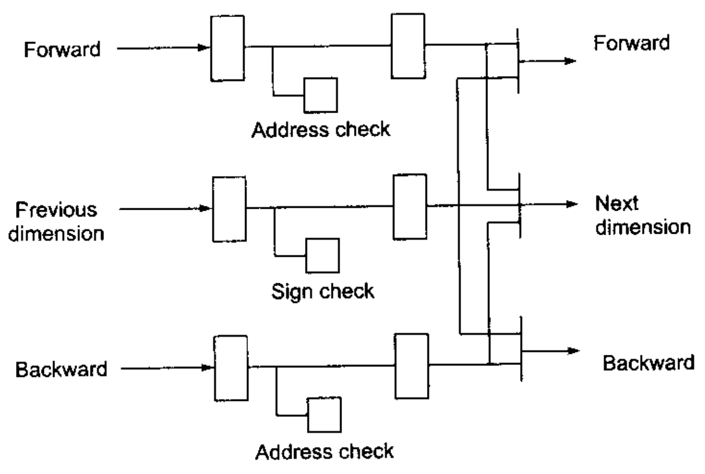
Each router contained two separate virtual networks with different priorities that shared the same physical channels. The priority-1 network could preempt the wires even if the priority-0 network was congested or jammed. The priority levels supported multi-threaded operations.

Each of the 18 router paths contained buffers, comparators, and output arbitration (Fig. 9.21). On each data path, a comparator compared the lead flit, which contained the destination address in that dimension, to the node coordinate. If the head flit did not match, the message continued in the current direction. Otherwise the message was routed to the next dimension.

A message entering the dimension competed with messages continuing in the dimension at a two-to-one switch. Once a message was granted this switch, all other input was locked out for the duration of the message. Once the head flit of the message had set up the route, subsequent flits followed directly behind it.



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



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

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

Two priorities of messages shared the physical wires but used completely separate buffers and routing logic. This allowed priority-1 messages to proceed through blockages at priority 0. Without this ability, the system would not be able to redistribute data that caused hot spots in the network.

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

In response to an arriving message, the processor may set presence tags for task synchronization. For example, access to the value produced by the combining tree may be synchronized by initially tagging as empty the location that will hold this value. An attempt to read this location before the combining tree has written it will raise an exception and suspend the reading task until the root of the tree writes the value.

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

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

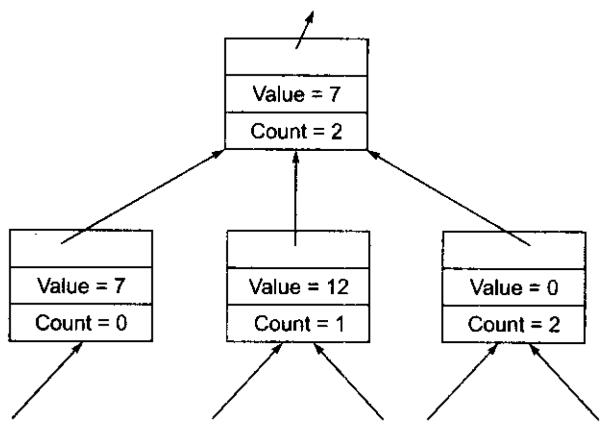


Fig. 9.22 A combining tree for internode communication or synchronization (Courtesy of W. Dally et al, 1992)

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

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

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

Research Issues The J-Machine was an exploratory research project. Rather than being specialized for a single model of computation, the MDP incorporated primitive mechanisms for efficient communication, synchronization, and naming. The machine was used as a platform for software experiments in fine-grain parallel programming.

Reducing the grain size of a program increases both the potential speedup due to parallel execution and the potential overhead associated with parallelism. Special hardware mechanisms for reducing the overhead due to communication, process switching, synchronization, and multi-threading were therefore central to the design of the MDP. Software issues such as load balancing, scheduling, and locality also remained open questions.

The MIT research group led by Dally implemented two languages on the J-Machine: the actor language Concurrent Smalltalk and the dataflow language Id. The machine's mechanism also supported dataflow and object-oriented programming models using a global name space. The use of a few simple mechanisms provided orders of magnitude lower communication and synchronization overhead than was possible with multicomputers built from then available off-the-shelf microprocessors.

9.3.3 The Caltech Mosaic C

The Caltech Mosaic C was an experimental fine-grain multicomputer that employed single-chip nodes and advanced packaging technology to demonstrate the performance/cost advantages of fine-grain multicomputer architecture. We describe below the architecture of the Mosaic C and review its application potentials, based on a report by Seitz (1992), the project leader at Caltech.

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

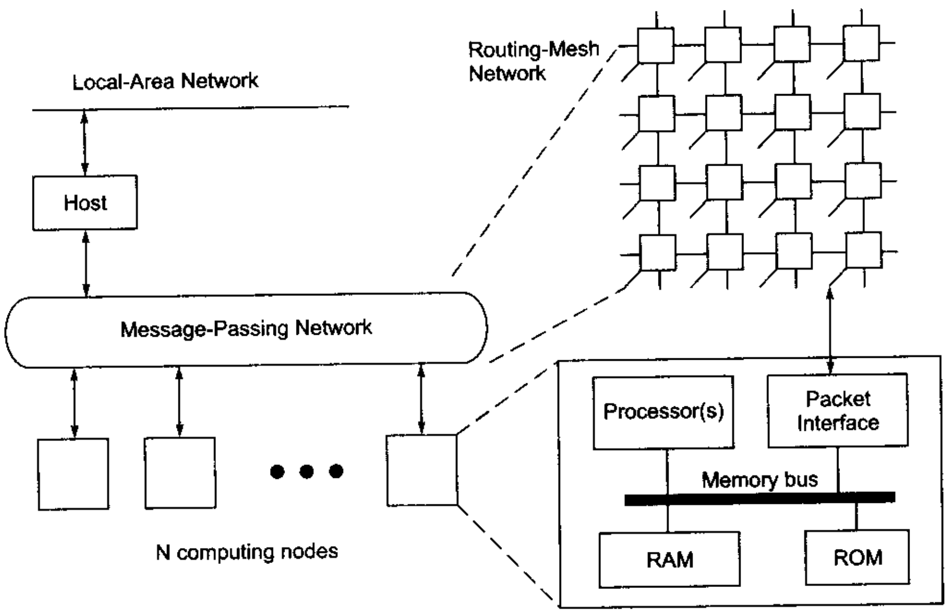


Fig. 9.23 The Caltech Mosaic architecture (Courtesy of C. Seitz, 1992)

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

The compilation-based programming system allowed fine-grain reactive-process message-passing programs to be expressed in C+–, an extension of C++, and the run-time system performed automatic distributed management of system resources.

Mosaic C Node The Mosaic C multicomputer node was a single 9.25 mm × 10.00 mm chip fabricated in a 1.2-μm-feature-size, two-level-metal CMOS process. At 5-V operation, the synchronous parts of the chip operated with large margins at a 30-MHz clock rate, and the chip dissipated $\approx$ 0.5 W.

The processor also included two program counters and two sets of general-purpose registers to allow zero-time context switching between user programs and message handling. Thus, when the packet interface received a complete packet, received the header of a packet, completed the sending of a packet, exhausted the allocated space for receiving packets, or any of several other events that could be selected, it could interrupt the processor by switching it instantly to the message-handling context.

Instead of several hundred instructions for handling a packet, the Mosaic typically required only about 10 instructions. The number of clock cycles for the message-handling routines could be reduced to insignificance by placing them in hardware, but the Caltech group chose the more flexible software mechanism so that they could experiment with different message-handling strategies.

Mosaic C 8 × 8 Mesh Boards The choice of a two-dimensional mesh for the Mosaic was based on a 1989 engineering analysis; originally, a three-dimensional mesh network was planned. But the mutual fit of the two-dimensional mesh network and the circuit board medium provided high packaging density and allowed the high-speed signals between the routers to be conveyed on shorter wires.

Sixty-four Mosaic chips were packaged by tape-automated bonding (TAB) in an 8 × 8 array on a circuit board. These boards allowed the construction of arbitrarily large, two-dimensional arrays of nodes using stacking connectors. This style of packaging was meant to demonstrate some of the density, scaling, and testing advantages of mesh-connected systems. Host-interface boards were also used to connect the Mosaic arrays and workstations.

Applications and Future Trends Charles Seitz determined that the most profitable niche and scaling track for the multicomputer, a highly scalable and economical MIMD architecture, was the fine-grain multicomputer. The Mosaic C demonstrated many of the advantages of this architecture, but the major part of the Mosaic experiment was to explore the programmability and application span of this class of machine.

The Mosaic may be taken as the origin of two scaling tracks: (1) Single-chip nodes are a technologically attractive point in the design space of multicomputers. Constant-node-size scaling results in single-chip nodes of increasing memory size, processing capability, and communication bandwidth in larger systems than centralized shared-memory multiprocessors. (2) It was also forecasts that constant-node-complexity scaling would allow a Mosaic 8 × 8 board to be implemented as a single chip, with about 20 times the performance per node, within 10 years. In this context, see also the discussion in Chapter 13.

A 16K-node machine was constructed at Caltech to explore the programmability and application span of the Mosaic C architecture for large-scale computing problems. For the loosely coupled computations in which it excels, a multicomputer can be more economically implemented as a network of high-performance workstations connected by a high-bandwidth local-area network. In fact, the Mosaic components and programming tools were used by a USC Information Science Institute project (led by Danny Cohen, 1992) to implement a 400-Mbits/s ATOMIC local-area network for this purpose.

9.4 SCALABLE AND MULTITHREADED ARCHITECTURES

Three pioneering and landmark scalable multiprocessor systems are discussed in this section. The Stanford Dash combined several latency-hiding mechanisms. The Kendall Square Research KSR-1 offered the first commercial attempt to produce a multiprocessor with cache-only memory. The Tera computer evolved from the HEP/Horizon series developed by Burton Smith. Only the main architectural features are described below. All three systems were extensions of the traditional von Neumann model. By far, the Tera system represented the most aggressive attempt to build a multi-threaded multiprocessor.

9.4.1 The Stanford Dash Multiprocessor

This was an experimental multiprocessor system developed by John Hennessy and coworkers at Stanford University beginning in 1988. The name Dash is an abbreviation for *Directory Architecture for Shared Memory*. The fundamental premise behind Dash was that it is possible to build a scalable high-performance machine with a single address space, coherent caches, and distributed memories. The directory-based coherence gave Dash the ease of use of shared-memory architectures, while maintaining the scalability of message-passing machines.

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

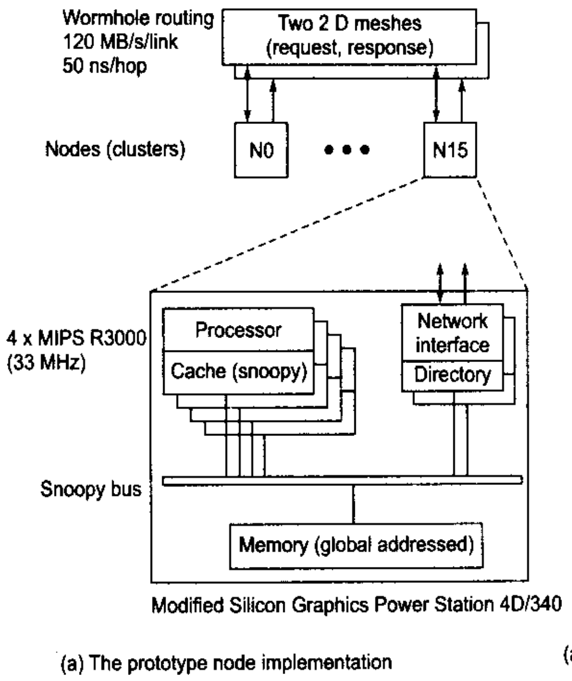
The interconnection network among the 16 multiprocessor clusters was a pair of wormhole-routed mesh networks. The channel width was 16 bits with a 50-ns fall-through time and a 35-ns cycle time. One mesh network was used to *request* remote memory, and the other was a *reply* mesh as depicted in Fig.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

The Dash designers claimed scalability for the Dash approach. Although the prototype was limited to at most 16 clusters (a 4×4 mesh), due to the limited physical memory addressability (256 Mbytes) of the 4D/340 system, the system was scalable to support hundreds to thousands of processors.

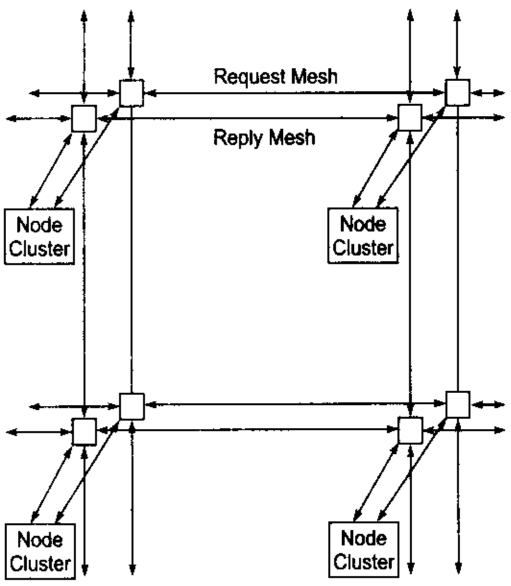
To use the 4D/340 in the Dash, the Stanford team made minor modifications to the existing system boards and designed a pair of new boards to support the directory memory and intercluster interface. The main modification to the existing boards was to add a bus retry signal, to be used when a request required service from a remote cluster.

The central bus arbiter was modified to accept a mask from the directory. The mask held off a processor's retry until the remote request was serviced. This effectively created a split-transaction bus protocol for requests requiring remote service.

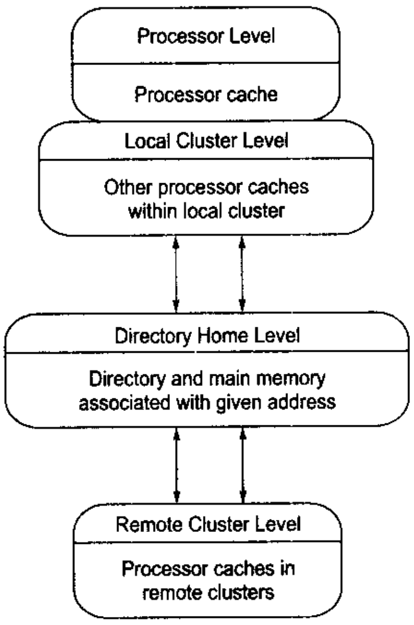
The new directory controller boards contained the directory memory, the intercluster coherence state machines and buffers, and a local section of the global interconnection network. The directory logic was split between the two logic boards along the lines of the logic used for outbound and inbound portions of intercluster transactions.



(a) The prototype node implementation



(a) Block diagram of 2 × 2 mesh interconnect


Processor Level
Processor cache
Local Cluster Level
Other processor caches within local cluster
Directory Home Level
Directory and main memory associated with given address
Remote Cluster Level
Processor caches in remote clusters

(c) Logic memory hierarchy

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

The mesh networks supported a scalable local and global memory bandwidth. The single-address space with coherent caches permitted incremental porting or tuning of applications, and exploited temporal and spatial locality. Other factors contributing to improved performance included mechanisms for reducing and tolerating latency, and well-designed I/O capabilities.

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

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

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

The first level was the processor cache which was designed to match the processor speed and support snooping from the bus. It took only one clock to access the processor cache. A request that could not be serviced by the processor cache was sent to the *local cluster.* The prototype allowed 30 processor clocks to access the local cluster. This level included the other processors' caches within the requesting processor's cluster.

Otherwise, the request was sent to the *home cluster* level. The home level consisted of the cluster that contained the directory and physical memory for a given memory address. It took 100 processor clocks to access the directory at the home level. For many accesses (for instance, most private data references), the local and home cluster were the same, and the hierarchy collapsed to three levels. In general, however, a request would travel through the interconnection network to the home cluster.

The home cluster could usually satisfy the request immediately, but if the directory entry was in a dirty state, or in a shared state when the requesting processor requested exclusive access, the fourth level had to be accessed. The *remote cluster* level for a memory block consisted of the clusters marked by the directory as holding a copy of the block. It took 135 processor clocks to access processor caches in remote clusters in the prototype design.

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

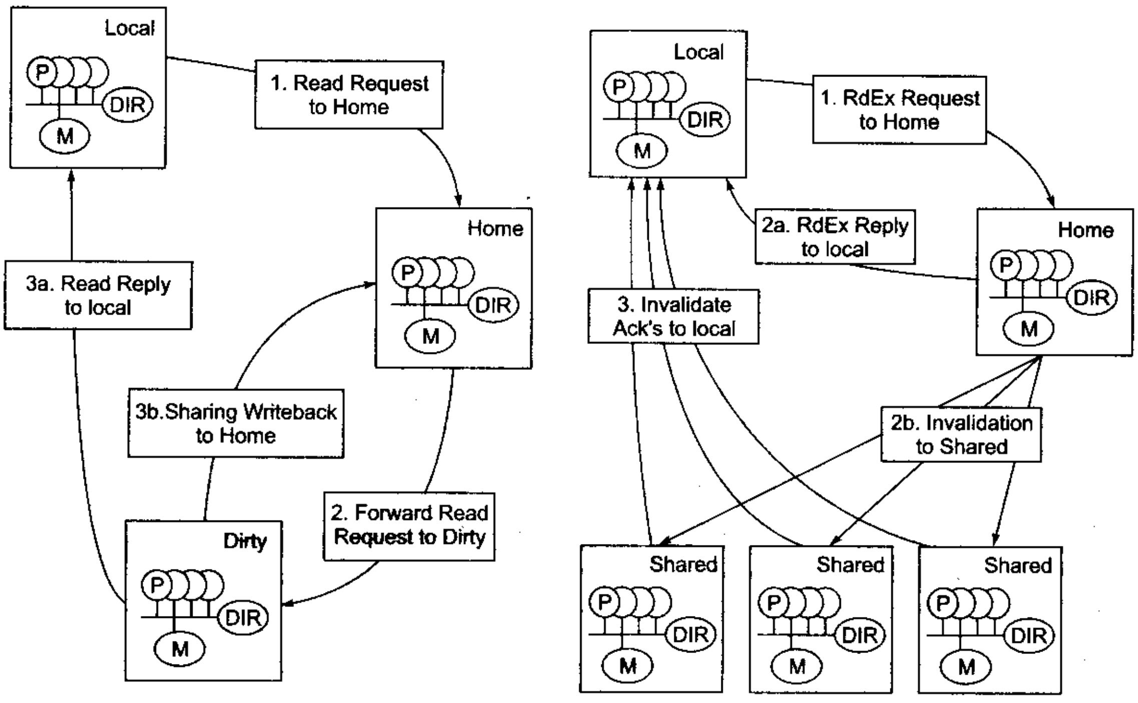
Using the state and presence bits, the memory could tell which caches needed to be invalidated when a location was written. Likewise, the directory indicated whether the memory copy of the block was up-to-date or which cache held the most recent copy.

By using the directory memory, a node writing a location could send point-to-point invalidation or update messages to the processors actually cacheing that block. This is in contrast to the invalidating broadcast required by the snoopy protocol. The scalability of the Dash depended on this ability to avoid broadcasts.

Another important attribute of a directory-based protocol is that it does not depend on any specific interconnection network topology. As a result, the designer can readily use any of the low-latency scalable networks, such as meshes or hypercubes, that were originally developed for message-passing machines.

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

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



(a) Read of dirty remote cache block (a) Write to shared remote cache block

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

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

Figure 9.25b shows the corresponding sequence for a write operation that requires remote service. The invalidation-based protocol requires the processor (actually the write buffer) to acquire exclusive ownership of the cache block before completing the store. Thus, if a write is made to a block that the processor does not have cached, or only has cached in a shared state, the processor issues a read-exclusive request on the local bus.

In this case, no other cache holds the block entry dirty in the local cluster, so a RdEx Request (message 1) is sent to the home cluster. As before, a remote-access cache entry is allocated in the local cluster. At the home cluster, the pseudo-CPU issues the read-exclusive request to the bus. The directory indicates that the line is in the shared state. This results in the directory controller sending a RdEx Reply (message 2a) to the local cluster and invalidation requests (Inv-Req, message 2b) to the sharing cluster.

The home cluster owns the block, so it can immediately update the directory to the dirty state, indicating that the local cluster now holds an exclusive copy of the memory line. The RdEx Reply message is received in the local cluster by the reply controller, which can then satisfy the read-exclusive request.

To ensure consistency at release points, however the remote-access cache entry is deallocated only when it receives the number of invalidate acknowledgments (Inv-Ack, message 3) equal to an invalidation count sent in the original reply message.

The Dash prototype with 64 nodes was rather small in size. If each processor had a five-issue superscalar operation with a 100-MHz clock, an extended machine with 2K nodes would have the potential to become a system with 1 tera operations per second, with higher performance at higher clock rates.

This demands an integrated implementation with lower overhead in the scalable directory structure. A three-dimensional torus network was considered with 16-bit data paths, a 20-ns fall-through delay, and a 4-ns cycle time. The access time ratio among the four levels of memory hierarchy was to be approximately 1:5:16:80:120, where 1 corresponds to one processor clock. The larger version of DASH was not implemented; however, the concept of distributed directory-based cache coherence was validated.

9.4.2 The Kendall Square Research KSR-1

This was the first commercial attempt to build a scalable multiprocessor with *cache-only memory architecture* (COMA). The Kendall Square Research KSR-1 was a size- and generation-scalable shared-memory multiprocessor computer. It was formed as a hierarchy of "ring multis" as depicted in Fig. 9.26.

The KSR-1 Architecture Scalability in the KSR-1 was achieved by connecting 32 processors to form a ring multi (search engine 0 in Fig. 9.26) operating at 1 Gbyte/s (128 million accesses per second). Interconnection bandwidth within a ring scales linearly, since every ring slot has roughly the capacity of a typical crosspoint switch found in a supercomputer that interconnects eight to sixteen 100-Mbytes/s HIPPI channels.

The KSR-1 used a two-level hierarchy to interconnect 34 Ring:0s by a top-level Ring:1 (1088 processors) and was therefore massive. The ring design supported an arbitrary number of levels, permitting ultras to be built (Fig. 9.27).

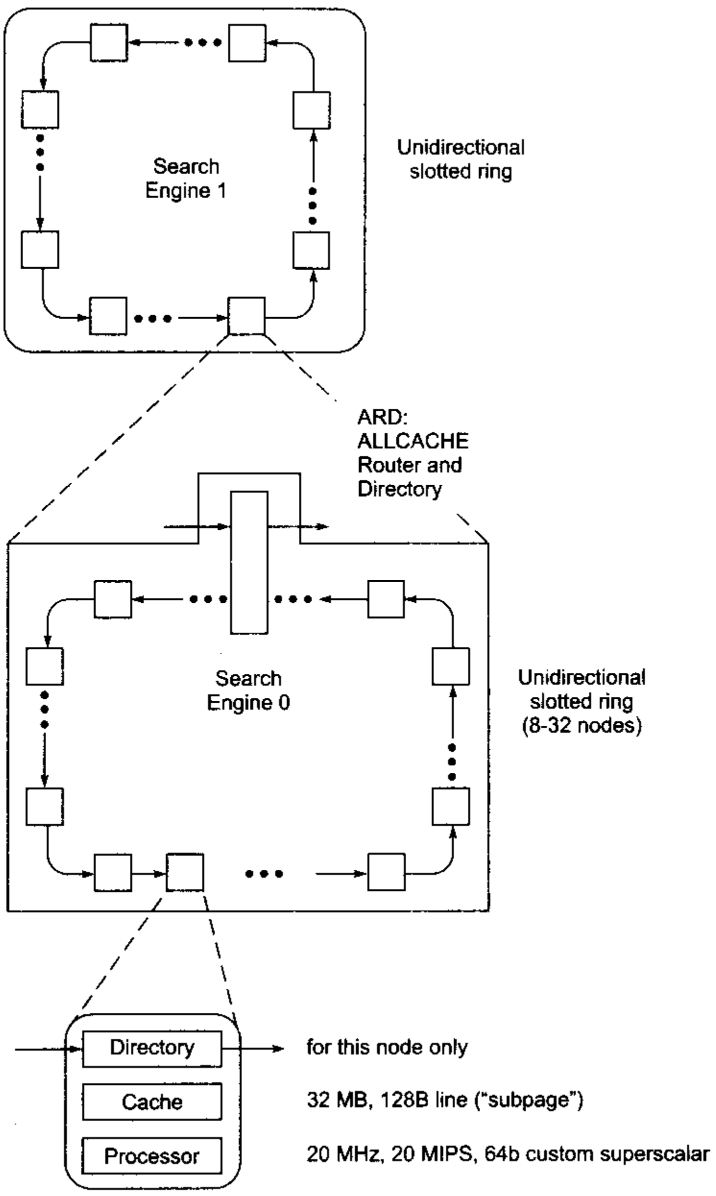

Search Engine 1
Unidirectional slotted ring
ARD: ALLCACHE Router and Directory
Search Engine 0
Unidirectional slotted ring (8-32 nodes)
Directory
Cache
Processor
for this node only
32 MB, 128B line ("subpage")
20 MHz, 20 MIPS, 64b custom superscalar

Fig. 9.26 The KSR-1 architecture with a slotted ring for communication (Courtesy of Kendall Square Research Corporation, 1991)

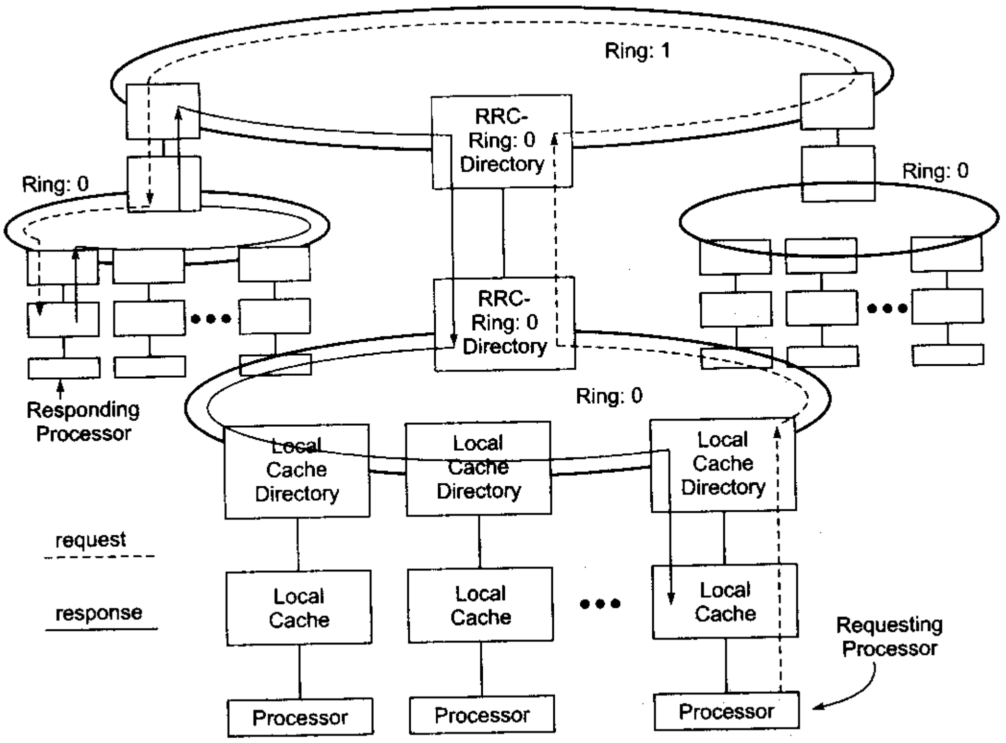


Fig. 9.27 Remote cache (memory) access through two levels of communication rings in the KSR-1 (Courtesy of Kendall Square Research Corporation, 1991)

Each node comprised a primary cache, acting as a 32-Mbyte primary memory, and a 64-bit superscalar processor with roughly the same performance as an IBM RS/6000 operating at the same clock rate. The superscalar processors containing 64 floating-point and 32 fixed-point registers of 64 bits were designed for both scalar and vector operations.

For example, 16 elements could be prefetched at one time. A processor also had a 0.5-Mbyte subcache supplying 20 million accesses per second to the processor (a computational efficiency of 0.5). A processor operated at 20 MHz and was fabricated in 1.2-μm CMOS.

The processor, without caches, contained 3.9 million transistors on 6 types of 12 custom chips. Three-quarters of each processor consisted of the search engine responsible for migrating data to and from other nodes, for maintaining memory coherence throughout the system using distributed directories, and for ring control.

The ALLCACHE Memory The KSR-1 eliminated the memory hierarchy found in conventional computers and the corresponding physical memory addressing overhead. Instead, it offered a single-level memory, called *ALLCACHE* by KSR designers. This ALLCACHE design represented the confluence of cache and shared virtual memory concepts that exploit locality required by scalable distributed computing. Each local cache had a capacity of 32 Mbytes (2^{25} bytes). The global virtual address space had 2^{40} bytes.

Bell (1992) considered the KSR machine the most likely blueprint for future scalable MPP systems. This was a revolutionary architecture and thus was more controversial when it was first introduced in 1991. The architecture provided size (including I/O) and generation scalability in that every node was identical, and it offered an efficient environment for both arbitrary workloads and sequential to parallel processing through a large hardware-supported address space with an unlimited number of processors.

Programming Model The KSR machine provided a strict sequentially consistent programming model and dynamic management of memory through hardware migration and replication of data throughout the distributed processor memory nodes using its ALLCACHE mechanism.

With sequential consistency, every processor returns the latest value of a written value, and results of an execution on multiple processors appear as some interleaving of operations of individual nodes when executed on a multithreaded machine. With ALLCACHE, an address became a name, and this name automatically migrated throughout the system and was associated with a processor in a cache-like fashion as needed.

Copies of a given cell were made by the hardware and sent to other nodes to reduce access time. A processor could prefetch data into a local cache and post-store data for other cells. The hardware was designed to exploit spatial and temporal locality.

For example, in the SPMD programming model, copies of the program moved dynamically and were cached in each of the operating nodes' primary and processor caches. Data such as elements of a matrix moved to the nodes as required simply by accessing the data, and the processor had instructions to prefetch data to the processor's registers. When a processor wrote to an address, all cells were updated and thus memory coherence was maintained. Data movement occurred in subpages of 128 bytes of the 16K pages.

Example 9.6 Multi-ring searching with requesting and responding processors on different Ring: Os (Courtesy of Kendall Square Research Corporation, 1991).

Internode communication for remote memory access was achieved through a searching process. When the requester and responder were in the same Ring:0, the searching was restricted to a single connected Ring:0. Local cache directories showed what addresses could be found in the connected local cache. Each Ring:0 was a unidirectional slotted ring for pipelined searching until the destination was reached.

Figure 9.27 illustrates the situation when the requester and responder resided in different Ring:0s. The top level, Ring:1, consisted entirely of *ring routing cells* (RRCs), each containing a directory for the Ring:0 to which it was connected. Each RRC directory on Ring:1 was essentially a duplicate of the RRC directory on the corresponding Ring:0.

When a packet reached an RRC on Ring:1, it was moved to the next RRC on the ring if the RRC directory indicated that the requested data was not on the corresponding ring. Otherwise, the packet was routed down to the RRC on Ring:0. The packet-passing speed of a Ring:0 was 8 million packets per second. Ring:1 could be configured to handle 8, 16, 32, or 64 million packets per second.

Environment and Performance Every known form of parallelism was supported via the KSR's Mach-based operating system. Multiple users could run multiple sessions comprising multiple applications or multiple processes (each with independent address space), each of which might consist of multiple threads of control running and simultaneously sharing a common address space. Message passing was supported by pointer passing in the shared memory to avoid data copying and enhance performance.

The KSR also provided a commercial programming environment for transaction processing that accessed relational databases in parallel with unlimited scalability as an alternative to multicomputers formed from multiprocessor mainframes. A 1K-node system provided almost two orders of magnitude more processing power, primary memory, I/O bandwidth, and mass storage capacity than a multiprocessor mainframe available at that time.

For example, unlike other contemporary candidates, a 1088-node system could be configured with 15.3 terabytes of disk memory, providing 500 times the capacity of its main memory. The 32- and 320-node systems were designed to deliver over 1000 and 10,000 transactions per second, respectively, giving them over 100 times the throughput of a multiprocessor mainframe available at the time.

With rapid advances in VLSI and interconnect technologies, the mid-1990s saw a major shakeout in the supercomputer business. Kendall Square Research, the developers of KSR-1 and its sequel KSR-2 systems, were forced to exit from hardware business during that period. As in the case of other innovative and pioneering attempts at the development of parallel computer architectures, knowledge gained from the KSR development was also useful in the design and development of MPP computer systems of subsequent generations. Our next case study on MPP system will also bring out clearly this important point.

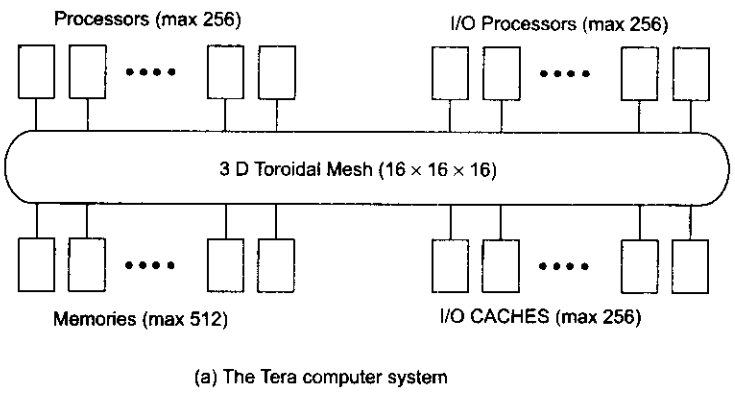
9.4.3 The Tera Multiprocessor System

Multithreaded von Neumann architecture can be traced back to the CDC 6600 manufactured in the mid-1960s. Multiple functional units in the 6600 CPU could execute different operations simultaneously using a score-boarding control. The very first multithreaded multiprocessor was the Denelcor HEP designed by Burton Smith in 1978. The HEP was built with 16 processors driven by a 10-MHz clock, and each processor could execute 128 *threads* (called *processes* in HEP terminology) simultaneously.

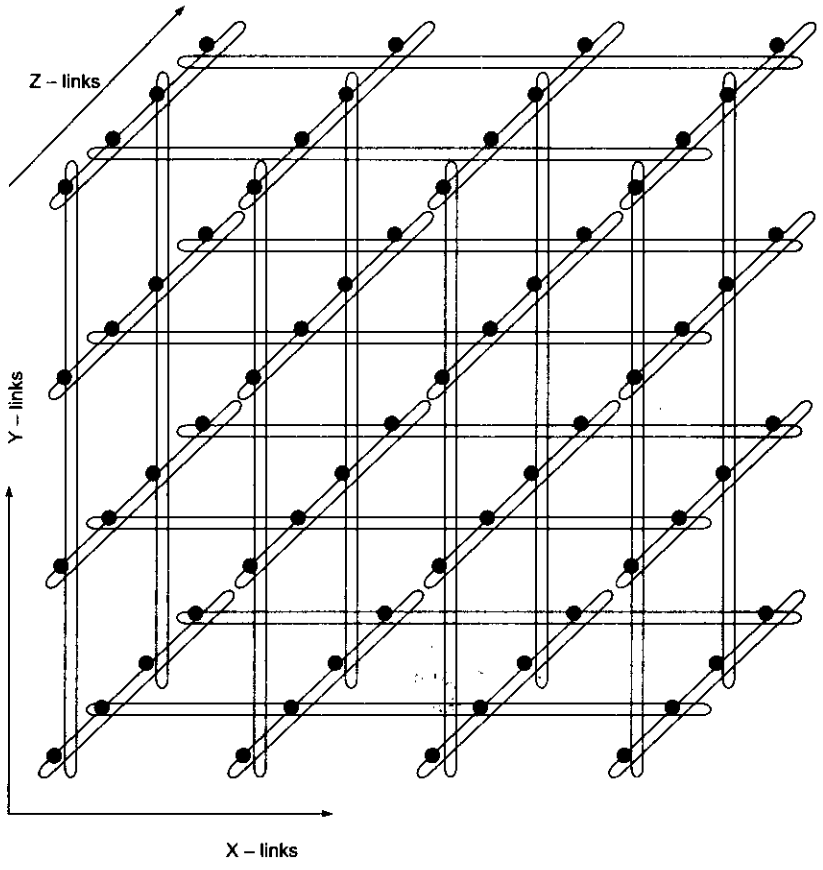
The HEP failed to survive due to inadequate software and compiler support. The Tera was very much a HEP descendant but was implemented with VLSI circuits and packaging technology. A 400-MHz clock was proposed for use in the Tera system, again with a maximum of 128 threads (*i-streams* in Tera terminology) per processor.

In this section, we describe the Tera architecture, its processors and thread state, and the tagged memory/registers. The unique features of the Tera included not only the high degree of multithreading but also the explicit-dependence lookahead and the high degree of pipelining in its processor-network-memory operations. These advanced features were mutually supportive. The first Tera Multithreaded Architecture (MTA) system was delivered in 1998.

The Tera Design Goals The Tera architecture was designed with several major goals in mind. First, it needed to be suitable for very high-speed implementations, i.e. have a short clock period and be scalable to many processors. A maximum configuration of the first implementation of the architecture (Fig. 9.28a) was 256 processors, 512 memory units, 256 I/O cache units, 256 I/O processors, 4096 interconnection network nodes, and a clock period of less than 3 ns.



(a) The Tera computer system



(b) A sparse 4 × 4 × 4 torus with X-links and Y-links missing on alternate Z-layers, respectively

Fig. 9.28 **The Tera multiprocessor and its three-dimensional sparse torus architecture shown with a 4 × 4 × 4 configuration (Courtesy of Tera Computer Company, 1992)**

Second, it was important that the architecture be applicable to a wide spectrum of problems. Programs that do not vectorize well, perhaps because of a preponderance of scalar operations or too frequent conditional branches, will execute efficiently as long as there is sufficient parallelism to keep the processors busy. Virtually any parallelism applicable in the total computational workload can be turned into speed, from operation-level parallelism within program basic blocks to multiuser time and space sharing.

A third goal was ease of compiler implementation. Although the instruction set did have a few unusual features, they did not pose unduly difficult problems for the code generator. There were no register or memory addressing constraints and only three addressing modes. Condition code setting was consistent and orthogonal.

Because the architecture permitted free exchange of spatial and temporal locality for parallelism, a highly optimizing compiler could improve locality and trade the parallelism thereby saved for more speed. On the other hand, if there was sufficient parallelism, the compiler could exploit it efficiently.

The Sparse Three-Dimensional Torus The interconnection network was a three-dimensional sparsely populated torus (Fig. 9.28b) of pipelined packet-switching nodes, each of which was linked to some of its neighbors. Each link could transport a packet containing source and destination addresses, an operation, and 64 data bits in both directions simultaneously on every clock tick. Some of the nodes were also linked to *resources*, i.e. processors, data memory units, I/O processors, and I/O cache units.

Instead of locating the processors on one side of the network and the memories on the other (a "dance hall" configuration), the resources were distributed more-or-less uniformly throughout the network. This permitted data to be placed in memory units near the appropriate processor when possible, and otherwise generally maximized the distance between possibly interfering resources.

The interconnection network of one 256-processor Tera system contained 4096 nodes arranged in a $16 \times 16 \times 16$ toroidal mesh; i.e. the mesh "wrapped around" in all three dimensions. Of the 4096 nodes, 1280 were attached to the resources comprising 256 cache units and 256 I/O processors. The 2816 remaining nodes did not have resources attached but still provided message bandwidth.

To increase node performance, some of the links were omitted. If the three directions are named x, y, and z, then x-links and y-links were omitted on alternate z-layers (Fig. 9.28b). This reduces the node degree from 6 to 4, or from 7 to 5, counting the resource link. In spite of its missing links, the bandwidth of the network was very large.

Any plane bisecting the network crossed at least 256 links, giving the network a data bisection bandwidth of one 64-bit data word per processor per tick in each direction. This bandwidth was needed to support shared-memory addressing in the event that all 256 processors addressed memory on the other side of some bisecting plane simultaneously.

As the Tera architecture scaled to larger numbers of processors p, the number of network nodes grew as $p^{3/2}$ rather than as the $p \log p$ associated with the more commonly used multistage networks. To see this, we first assume that memory latency is fully masked by parallelism only when the number of messages being routed by the network is at least $p \times l$, where l is the (round-trip) latency. Since messages occupy volume, the network must have a volume proportional to $p \times l$; since the speed of light is finite, the volume is also proportional to l^3 and therefore l is proportional to $p^{1/2}$ rather than $\log p$.

Pipelined Support Each processor in a Tera computer could execute multiple instruction streams (threads) simultaneously. In the initial implementation, as few as 1 or as many as 128 program counters could be active at once. On every tick of the clock, the processor logic selected a ready-to-execute thread and allowed it to issue its next instruction. Since instruction interpretation was completely pipelined by the processor and by the network and memories as well (Fig. 9.29), a new instruction from a different thread could be issued during each tick without interfering with its predecessors.

When an instruction finished, the thread to which it belonged became ready to execute the next instruction. As long as there were enough threads in the processor so that the average instruction latency was filled with instructions from other threads, the processor was fully utilized. Thus, it was only necessary to have enough threads to hide the expected latency (perhaps 70 ticks on average); once latency was hidden, the processor would run at peak performance and additional threads would not speed the result.

If a thread were not alicwed to issue its next instruetion until the previous instruction completed, then approximately 70 different threads would be required on each processor to hide the expected latency. The lookahead described later allowed threads to issue multiple instructions in parallel, thereby reducing the number of threads needed to achieve peak performance.

As seen in Fig. 9.29, three operations could be executed simultaneously per instruction per processor. The *M-pipeline* was for memory-access operations, the *A-pipeline* for arithmetic operations, and the *C-pipeline* for control or arithmetic operations. The instructions were 64 bits wide. If more than one operation in an instruction specified the same register or setting of condition codes, the priority was $M > A > C$.

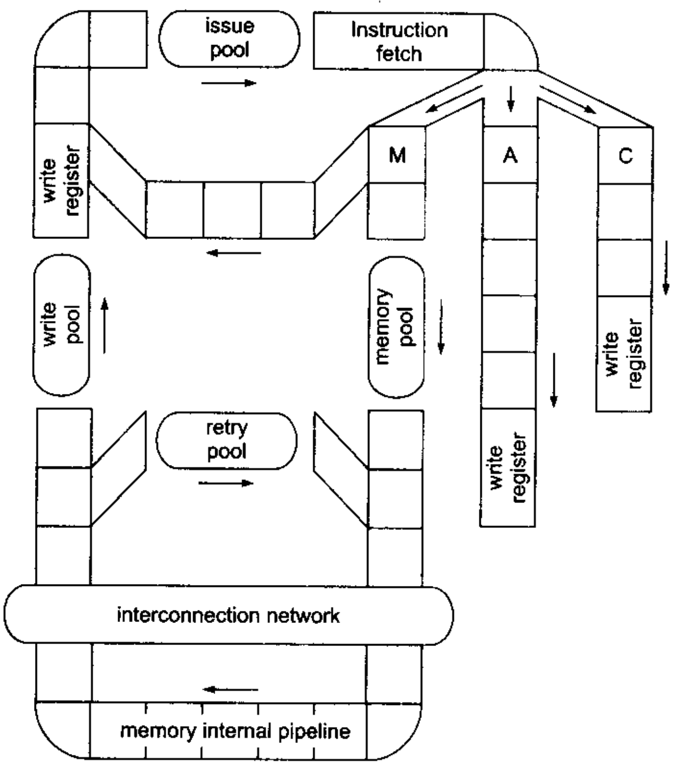
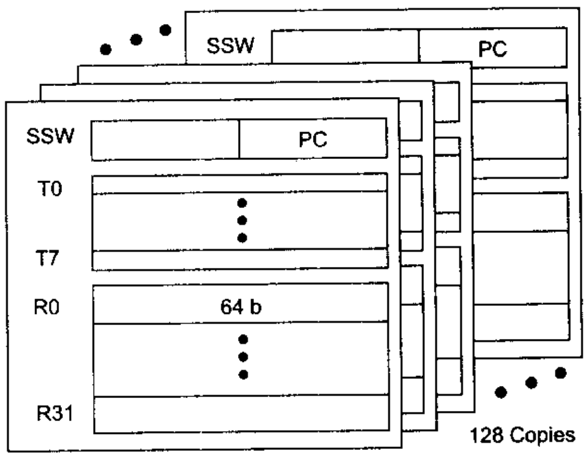


Fig. 9.29 Pipelined processor-network-memory structure (Courtesy of Tera Computer Company, 1992)

It was estimated that a peak speed of 1G operations per second could be achieved per processor if driven by a 333-MHz clock. However, a particular thread would not exceed about 100M operations per second because of interleaved execution. The processor pipeline was rather deep, about 70 ticks, as compared with 8 ticks in the earlier HEP pipeline.

Thread State and Management Figure 9.30 shows that each thread had the following state associated with it:

- One 64-bit stream status word (SSW);
- Thirty-two 64-bit general-purpose registers (R0-R31);
- Eight 64-bit target registers (T0–T7).



Stream Status Word (SSW)
• 32 bit PC (Program Counter)
• Modes (e.g. rounding, lookahead disable)
• Trap disable mask (e.g. data alignment, overflow)
• Condition codes (last four emitted)
No synchronization bits on R0-R31
Target Registers (T0-T7) look like SSWs

Fig. 9.30 **The thread management scheme used in the Tera computer (Courtesy of Tera Computer Company, 1992)**

Context switching was so rapid that the processor had no time to swap the processor-resident thread state. Instead, it had 128 of everything, i.e. 128 SSWs, 4096 general purpose registers, and 1024 target registers. It is appropriate to compare these registers in both quantity and function to vector registers or words of caches in other architectures. In all three cases, the objective is to improve locality and avoid reloading data.

Program addresses were 32 bits in length. Each thread's current program counter (PC) was located in the lower half of its SSW. The upper half described various modes (e.g. floating-point rounding, lookahead disable), the trap disable mask (e.g. data alignment, floating overflow), and the four most recently generated condition codes.

Most operations had a _TEST variant which emitted a condition code; and branch operations could examine any subset of the last four condition codes emitted and branch appropriately. Also associated with each thread were thirty-two 64-bit general-purpose registers. Register R0 was special in that it read as 0 and output to it was discarded. Otherwise, all general-purpose registers were identical.

The target registers were used as branch targets. The format of the target registers was identical to that of the SSW, though most control transfer operations used only the low 32 bits to determine a new PC. Separating the determination of the branch target address from the decision to branch allowed the hardware to prefetch instructions at the branch targets, thus avoiding delay when the branch decision was made. Using target registers also made branch operations smaller, resulting in tighter loops. There were also skip operations which obviated the need to set targets for short forward branches.

One target register (T0) pointed to the trap handler which was nominally an unprivileged program. When a trap occurred, the effect was as if a coroutine call to a T0 had been executed. This made trap handling extremely lightweight and independent of the operating system. Trap handlers could be changed by the user to achieve specific trap capabilities and priorities without loss of efficiency.

Explicit-Dependence Lookahead If there were enough threads executing on each processor to hide the pipeline latency (about 70 ticks), then the machine would run at peak performance. However, if each thread could execute some of its instructions in parallel (e.g. two successive loads), then fewer threads and parallel activities would be required to achieve peak performance.

The obvious solution was to introduce instruction lookahead; the difficulty was that the traditional register reservation approach requires far too much scoreboard bandwidth in this kind of architecture. Either multithreading or horizontal instruction alone would preclude scoreboarding.

The Tera architecture used a new technique called *explicit-dependence lookahead.* Each instruction contained a 3-bit lookahead field that explicitly specified how many instructions from this thread would be issued before encountering an instruction that depended on the current one. Since seven was the maximum possible lookahead value, at most 8 instructions and 24 operations could be concurrently executing from each thread.

A thread was ready to issue a new instruction when all instructions with lookahead values referring to the new instruction had completed. Thus, if each thread maintained a lookahead of seven, then nine threads were needed to hide 72 ticks of latency.

Lookahead across one or more branch operations was handled by specifying the minimum of all distances involved. The variant branch operations `JUMP_OFTEN` and `JUMP_SELDOM`, for high-and low-probability branches, respectively, facilitated optimization by providing a barrier to lookahead along the less likely path. There were also `SKIP_OFTEN` and `SKIP_SELDOM` operations. The overall approach was conceptually similar to exposed-pipeline lookahead except that the quanta were instructions instead of ticks.

Advantages and Drawbacks The Tera used multiple contexts to hide latency. The machine performed a context switch every clock cycle. Both pipeline latency and memory latency were hidden in the HEP/Tera approach. The major focus was on latency tolerance rather than latency reduction.

With 128 contexts per processor, a large number (2K) of registers must be shared finely between threads. The thread creation must be very cheap (a few clock cycles). Tagged memory and registers with full/empty bits were used for synchronization. As long as there was plenty of parallelism in user programs to hide latency and plenty of compiler support, the performance was potentially very high.

However, these Tera advantages were embedded in a number of potential drawbacks. The performance must be bad for limited parallelism, such as guaranteed low single-context performance. A large number of contexts (threads) demanded lots of registers and other hardware resources which in turn implied higher cost and complexity. Finally, the limited focus on latency reduction and cacheing entailed lots of slack parallelism to hide latency as well as lots of memory bandwidth; both required a higher cost for building the machine.

In the year 1996, the independent company Cray Research, Inc. founded by Seymour Cray merged with the high-performance graphics workstation producer Silicon Graphics, Inc. (SGI); Cray Research then became a business division of SGI. In the year 2000, Tera Computer Company, originators and developers of the Tera MTA massively parallel system which we have studied in this section, took over Cray Research. The merged company was named Cray, Inc., and it is in active operation today (see www.cray.com). Cray has continued with the development of the MTA architecture, as we shall review in Chapter 13.

9.5 DATAFLOW AND HYBRID ARCHITECTURES

Multithreaded architectures can in theory be designed with a pure dataflow approach or with a hybrid approach combining von Neumann and data-driven mechanisms. In this final section, we briefly review the historical development of dataflow computers. Then we consider the design of the ETL/EM-4 in Japan and the prototype design of the MIT/Motorola *T project.

9.5.1 The Evolution of Dataflow Computers

As introduced in Section 2.3, dataflow computers have the potential for exploiting all the parallelism available in a program. Since execution is driven only by the availability of operands at the inputs to the functional units, there is no need for a program counter in this architecture, and its parallelism is limited only by the actual data dependences in the application program. While the dataflow concept offers the potential of high performance, the performance of an actual dataflow implementation can be restricted by a limited number of functional units, limited memory bandwidth, and the need to associatively match pending operations with available functional units.

Arvind and Iannucci (1987) identified *memory latency* and *synchronization overhead* as two fundamental issues in multiprocessing. Scalable multiprocessors must address the loss in processor efficiency in these cases. Using various latency-hiding mechanisms and multiple contexts per processor can make the conventional von Neumann architecture relatively expensive to implement, and only certain types of parallelism can be exploited efficiently.

HEP/Tera computers offered an evolutionary step beyond the von Neumann architectures. Dataflow architectures represent a radical alternative to von Neumann architectures because they use dataflow graphs as their machine languages. Dataflow graphs, as opposed to conventional machine languages, specify only a partial order for the execution of instructions and thus provide opportunities for parallel and pipelined execution at the level of individual instructions.

Dataflow Graphs We have seen a dataflow graph in Fig. 2.13. Dataflow graphs can be used as a machine language in dataflow computers. Another example of a dataflow graph (Fig. 9.31a) is given below.

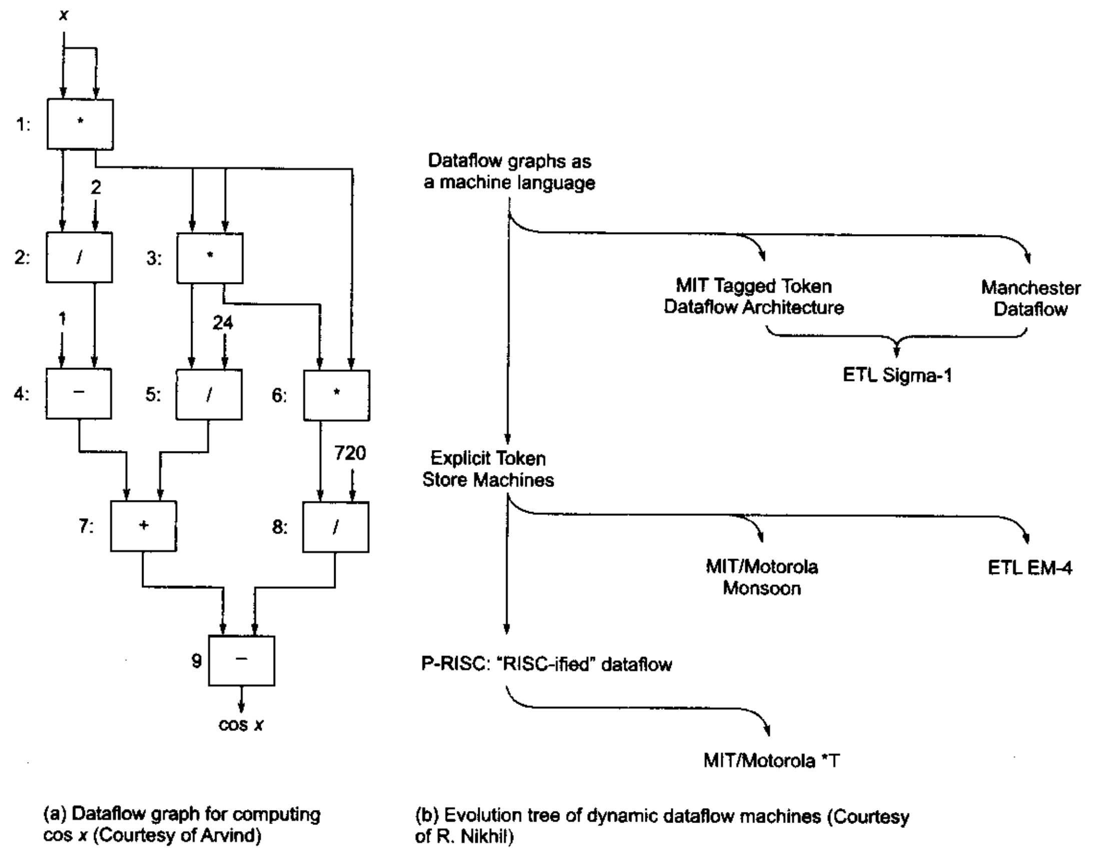

x
1: *
2
2: /
3: *
1
24
4: −
5: /
6: *
720
7: +
8: /
9 −
cos x
Dataflow graphs as a machine language
MIT Tagged Token Dataflow Architecture
Manchester Dataflow
ETL Sigma-1
Explicit Token Store Machines
MIT/Motorola Monsoon
ETL EM-4
P-RISC: "RISC-ified" dataflow
MIT/Motorola *T
(a) Dataflow graph for computing cos x (Courtesy of Arvind)
(b) Evolution tree of dynamic dataflow machines (Courtesy of R. Nikhil)

Fig. 9.31 An example dataflow graph and dataflow machine projects

Example 9.7 The dataflow graph for the calculation of cos*x* (Arvind, 1991).

This dataflow graph shows how to obtain an approximation of cos$x$ by the following power series computation:

$$\cos x \simeq 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} = 1 - \frac{x^2}{2} + \frac{x^4}{24} - \frac{x^6}{720} \tag{9.6}$$

The corresponding dataflow graph consists of nine operators (actors or nodes). The edges in the graph interconnect the operator nodes. The successive powers of x are obtained by repeated multiplications. The constants (divisors) are fed into the nodes directly. All intermediate results are forwarded among the nodes.

Static versus Dynamic Dataflow *Static dataflow computers* simply disallow more than one token to reside on any one arc, which is enforced by the firing rule: A node is enabled as soon as tokens are present on all input arcs and there is no token on any of its output arcs. Jack Dennis proposed the very first static dataflow computer in 1974.

The static firing rule is difficult to implement in hardware. Special feedback *acknowledge signals* are needed to secure the correct token passing between producing nodes and consuming nodes. Also, the static rule makes it very inefficient to process arrays of data. The number of acknowledge signals can grow too fast to be supported by hardware.

However, static dataflow inspired the development of *dynamic dataflow computers,* which were researched vigorously at MIT and in Japan. In a dynamic architecture, each data token is tagged with a context descriptor, called a *tagged token.* The firing rule of tagged-token dataflow is changed to: A node is enabled as soon as tokens with identical tags are present at each of its input arcs.

With tagged tokens, tag matching becomes necessary. Special hardware mechanisms are needed to achieve this. In the rest of this section, we discuss only dynamic dataflow computers. Arvind of MIT pioneered the development of tagged-token architecture for dynamic dataflow computers.

Although data dependence does exist in dataflow graphs, it does not force unnecessary sequentialization, and dataflow computers schedule instructions according to the availability of the operands. Conceptually, "token"-carrying values flow along the edges of the graph. Values or tokens may be memory locations.

Each instruction waits for tokens on all inputs, consumes input tokens, computes output values based on input values, and produces tokens on outputs. No further restriction on instruction ordering is imposed. No side effects are produced with the execution of instructions in a dataflow computer. Both dataflow graphs and machines implement only functional languages.

Pure Dataflow Machines Figure 9.31b shows the evolution of dataflow computers. The MIT *tagged-token dataflow architecture* (TTDA) (Arvind et al, 1983), the Manchester Dataflow Computer (Gurd and Watson, 1982), and the ETL Sigma-1 (Hiraki and Shimada, 1987) were all pure dataflow computers. The TTDA was simulated but never built. The Manchester machine was actually built and became operational in mid-1982. It operated asynchronously using a separate clock for each processing element with a performance comparable to that of the VAX/780.

The ETL Sigma-1 was developed at the Electrotechnical Laboratory, Tsukuba, Japan. It consisted of 128 PEs fully synchronous with a 10-MHz clock. It implemented the I-structure memory proposed by Arvind. The full configuration became operational in 1987 and achieved a 170-Mflops performance. The major problem in using the Sigma-1 was lack of high-level language for users.

Explicit Token Store Machines These were successors to the pure dataflow machines. The basic idea is to eliminate associative token matching. The waiting token memory is directly addressed, with the use of full/empty bits. This idea was used in the MIT/Motorola Monsoon (Papadopoulos and Culler, 1988) and in the ETL EM-4 system (Sakai et al, 1989).

Multithreading was supported in Monsoon using multiple register sets. Thread-based programming was conceptually introduced in Monsoon. The maximum configuration built consisted of eight processors and eight I-structure memory modules using an 8×8 crossbar network. It became operational in 1991.

EM-4 was an extension of the Sigma-1. It was designed for 1024 nodes, but only an 80-node prototype became operational in 1990. The prototype achieved 815 MIPS in an 80×80 matrix multiplication benchmark. We will study the details of EM-4 in Section 9.5.2.

Hybrid and Unified Architectures These are architectures combining positive features from the von Neumann and dataflow architectures. The best research examples include the MIT P-RISC (Nikhil and Arvind, 1988), the IBM Empire (Iannucci et al., 1991), and the MIT/Motorola *T (Nikhil, Papadopoulos, Arvind, and Greiner, 1991).

P-RISC was a "RISC-ified" dataflow architecture. It allowed tighter encodings of the dataflow graphs and produced longer threads for better performance. This was achieved by splitting "complex" dataflow instructions into separate "simple" component instructions that could be composed by the compiler. It used traditional instruction sequencing. It performed all intraprocessor communication via memory and implemented "joins" explicitly using memory locations.

P-RISC replaced some of the dataflow synchronization with conventional program counter-based synchronization. IBM Empire was a von Neumann/dataflow hybrid architecture under development at IBM based on the thesis of Iannucci (1988). The *T was a latter effort at MIT joining both the dataflow and von Neumann ideas, to be discussed in Section 9.5.3.

9.5.2 ETL/EM-4 in Japan

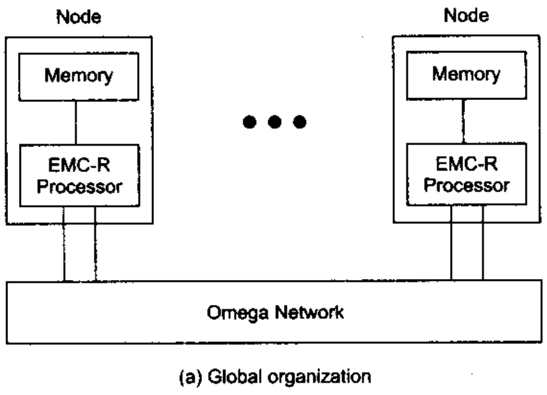
EM-4 had the overall system organization as shown in Fig. 9.32a. Each EMC-R node was a single-chip processor without floating-point hardware but including a switch of the network. Each node played the role of I-structure memory and had 1.31 Mbytes of static RAM. An Omega network was used to provide interconnections among the nodes.

The Node Architecture The internal design of the processor chip and of the node memory are shown in Fig. 9.32b. The processor chip communicated with the network through a 3×3 crossbar *switch unit.* The processor and its memory were interfaced with *a memory control unit.* The memory was used to hold programs (template segments) as well as tokens (operand segments, heaps, or frames) waiting to be fetched.

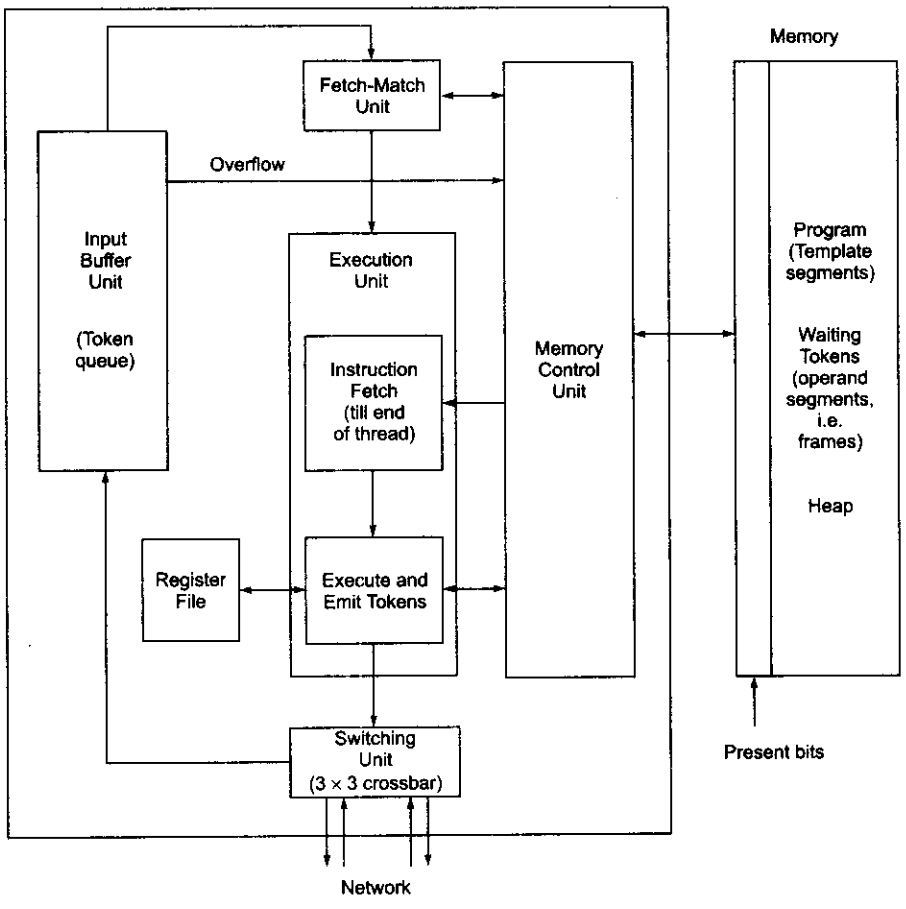
The processor consisted of six component units. The *input buffer* was used as a token store with a capacity of 32 words. The *fetch-match unit* fetched tokens from the memory and performed tag-matching operations among the tokens fetched in. Instructions were directly fetched from the memory through the memory controller.

The heart of the processor was the *execution unit,* which fetched instructic ns until the end of a thread. Instructions with matching tokens were executed. Instructions could emit tokens or write to registers. Instructions were fetched continually using traditional sequencing (PC + 1 or branch) until a "stop" flag was raised to indicate the end of a thread. Then another pair of tokens was accepted. Each instruction in a thread specified the two sources for the next instruction in the thread.

Static versus Dynamic Dataflow *Static dataflo*
reside on any one are whi



(a) Global organization



(b) The EMC-R processor design

Fig. 9.32 **The ETL EM-4 dataflow architecture (Courtesy of Sakai, Yamaguchi et al, Electrotechnical Laboratory, Tsukuba, Japan, 1991)**