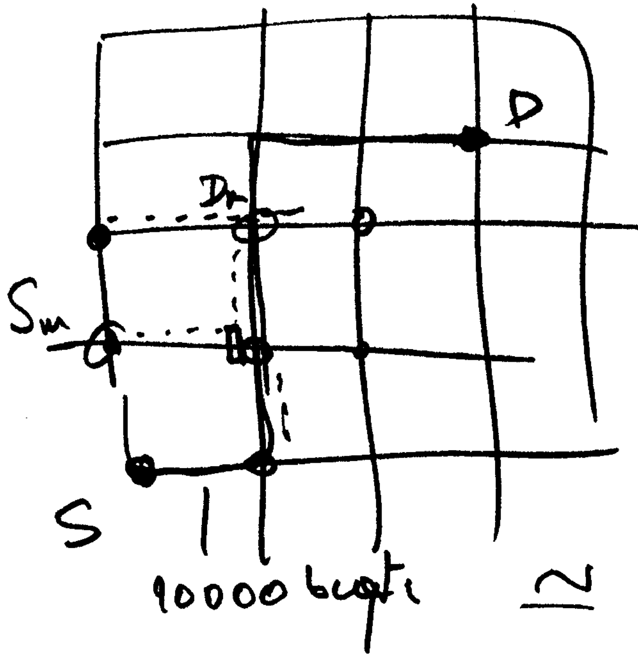
* Office hours 1:30 - 2:30 PM
* Projects
* Experiments with LOGPC
* Comparison SM - MP performance
* COMA
* Stanford Dash
  Book: Daniel Lenoski, Wolf-D. Weber
  SCALABLE SHARED-MEMORY MULTIPROCESSING



$\simeq$ 10 000 . G

$\boxed{20.000\ c}$, $G = 2c$

Note : one long message delays lots of short ones for a long time!

problem for small data items, but it can degrade performance if there is frequent communication or a large amount of data is exchanged.

To illustrate this performance difference between message-passing and shared-memory systems, consider the case where a producer process wants to send 10 words of data to a consumer process. In a typical message-passing system with a blocking send and receive protocol, this would be coded simply as

Producer Process | *Consumer Process*

```
send(Proc_i, Process_i, @sbuffer,
        num_bytes);

                                        receive(@rbuffer,max_bytes);
```

Since the operating system is multiplexing the network resources between users, this code would usually be broken into the following steps:

1. The operating system checks protections and then programs the network DMA controller to move the message from the sender's buffer to the network interface.
2. A DMA channel on the consumer processor has been programmed to move all messages to a common system buffer. When the message arrives, it is moved from the network interface to this system buffer, and an interrupt is posted to the processor.
3. The receiving processor services the interrupt and determines which process the message is intended for. It then copies the message to the specified receive buffer and reschedules the user process on the processor's ready queue.
4. The user process is dispatched on the processor and reads the message from the user's receive buffer.

On a shared-memory machine, there is no operating system involvement, and the processors can transfer the data using a shared data area. Assuming this data is protected by a flag indicating its availability and the size of the data transferred, the code would be

Producer Process | *Consumer Process*

```
for(i:=0; i<num_bytes; i++)
   buffer[i] := source[i];
flag := num_bytes;

                                        while (flag == 0) ;
                                        for (i:=0; i<flag; i++)
                                            dest[i] := buffer[i];
```

A comparison of the timing of these operations is given in Figure 1-3. For the message-passing case, the dominant costs are fixed and determined by the operating system overhead, programming the DMA, and the interrupt processing. For the shared-memory system, the overhead is primarily on the consumer reading the data since it is then that data moves from the global memory to the consuming processor. Thus, for a short message the shared-

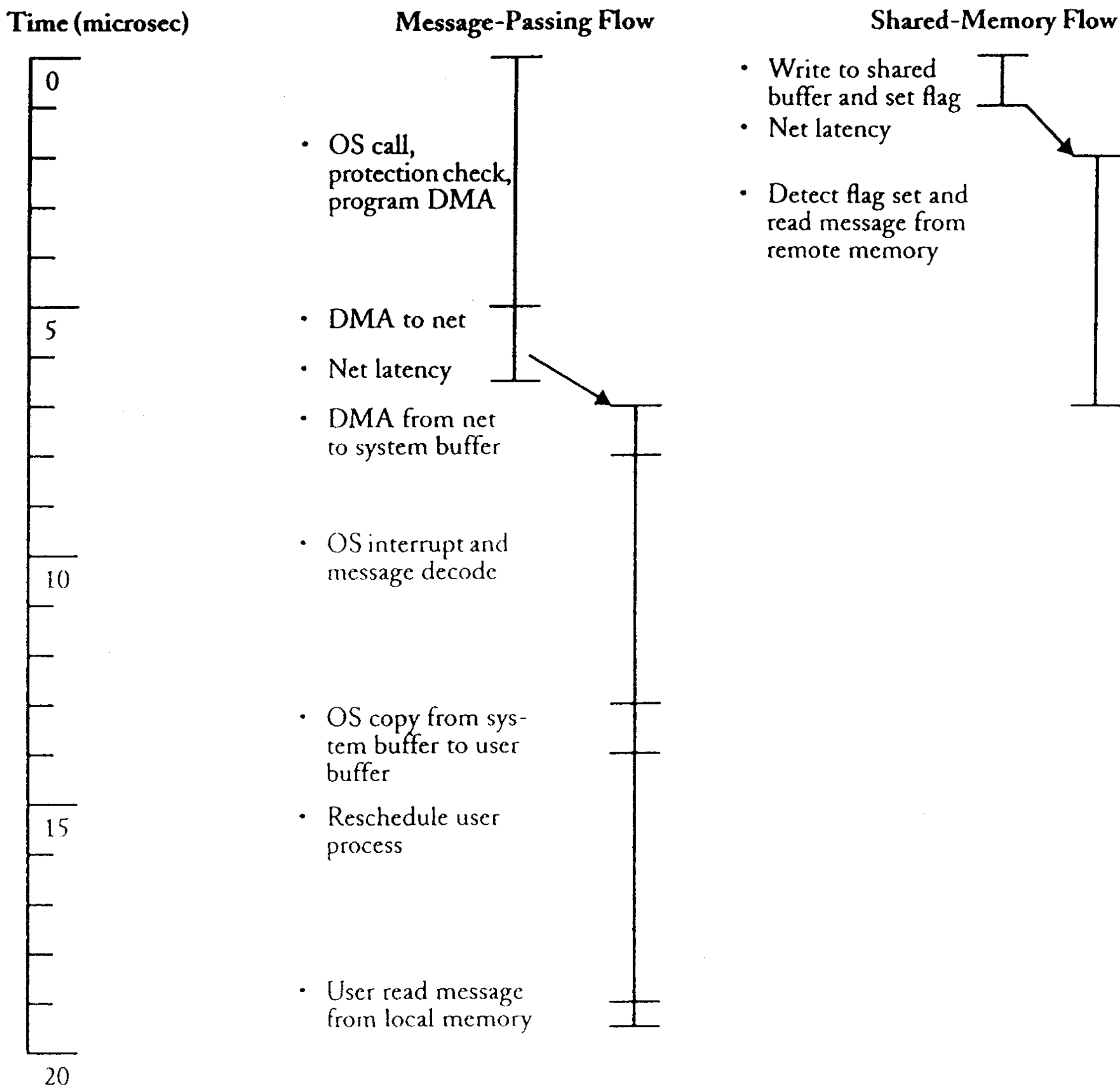


FIGURE 1-3 *Timing of a 10-Word Transfer on Message-Passing and Shared-Memory Systems.*

memory system is much more efficient. For longer messages, the message-passing system has similar or possibly higher performance, depending on the operating system overheads in the message-passing system and the efficiency of the shared-memory consumer reading the message.

Block diagrams of typical distributed-memory and shared-memory machines are given in Figure 1-4. The structure of the message-passing MIMD machine is the same as given in Figure 1-1(a). The network interface in the message-passing machine provides the DMA controllers and other hardware to send and receive messages from different processors. A simple MIMD shared-memory machine is shown in Figure 1-4(b). In most shared-memory machines the processors and memory are separated by an interconnection network. For small-scale shared-memory machines the interconnection network is a simple bus, while larger machines use multistage networks similar to the message-passing machines. Section 1.3.1 discusses the structure of the interconnection networks in more detail.

CHAPTER 6

DASH Prototype System

The DASH prototype system was an outgrowth of research into scalable shared-memory multiprocessing in the Computer Systems Laboratory at Stanford University. The primary goal of building the machine was a better understanding of the design issues and feasibility of this class of machine. The existence of a real machine would also aid in the development and characterization of parallel processing software including new applications, automatically parallelizing compilers and parallel languages, and multiprocessor operating systems. Another indirect benefit of building the machine was to provide a realistic set of performance metrics (e.g., obtainable latencies and bandwidths) for use in related architecture simulation studies.

Work on the prototype system began in the fall of 1988 and resulted in the initial 16-processor configuration in the spring of 1991 and a 48-processor system one year later. The design and implementation were carried out by a small group of graduate students (including the authors) under the direction of Professors John Hennessy and Anoop Gupta. Professors Mark Horowitz and Monica Lam were also part of the larger DASH project and gave valuable input in the prototype design.

This chapter summarizes the system-level organization and coherence protocol used in the prototype system. Details of the actual hardware structures and implementation costs are given in Chapter 7. While the prototype was not necessarily optimal in implementing all aspects of the DASH architecture, the description in this chapter is based on the prototype because it represents a complete and consistent design. A high-level description of a more ideal implementation of DASH is given in Chapter 9.

The discussion in this chapter begins with a description of the system-level organization of the DASH prototype. The discussion then moves down to the level of the individual clusters. The structure of a cluster is given with emphasis on the directory and network logic that execute the directory-based coherence protocol. The protocol is then discussed, starting with the basic invalidation-based coherence protocol. The protocols for the alternative memory operations (prefetch, update write, and synchronization) are then given. The chapter ends with a summary of the prototype organization and the protocol features.

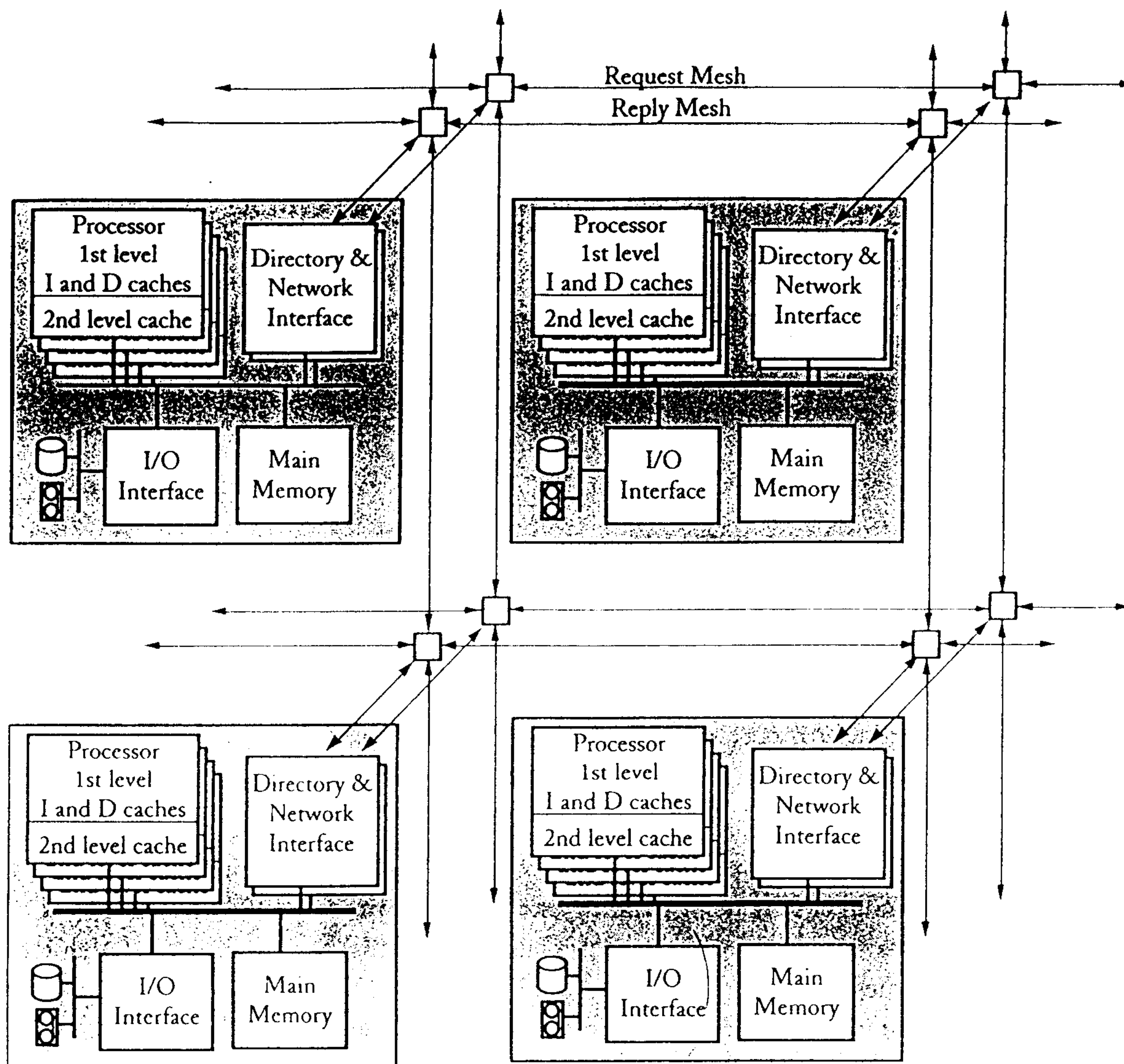


FIGURE 6-1 *Block Diagram of the DASH Prototype System*

6.1 System Organization

A block diagram of the DASH prototype system is shown in Figure 6-1. As a specific instance of the more general picture given in Figure 1-4(a), this diagram retains the three important attributes of the DASH architecture. First, the processing nodes and memory are interconnected by a scalable network. In the prototype, a pair of 2-D mesh networks are used. Second, the global shared memory is distributed among the processing nodes. Third, each processing node is a small-scale multiprocessor cluster. Each cluster contains four processors, a portion of global memory, and local I/O devices. The following sections examine the structure of the individual clusters and the interconnection meshes in more detail.

The prototype system is limited to a 4 × 4 configuration with 16 clusters and 64 processors. This limit was chosen due to the constraints in memory addressing of the base cluster hardware, which supports no more than 256 MByte of total memory. Thus, with 16 clusters the size of each cluster's local memory partition is only 16 MByte. While the system could

have been extended to support 32 or 64 clusters, the amount of memory per cluster would become too small. Support of 64 high-performance processors still gives the prototype much more power than would be possible on a single bus, and it provides a valuable platform for evaluating the architecture and experimenting with parallel software.

6.1.1 Cluster Organization

The individual shaded boxes in Figure 6-1 represent a single DASH cluster. Each cluster contains a set of processors, a section of the global memory, the directory and intercluster interface, and optional local I/O devices. These modules are interconnected by a bus supporting snoopy cache coherence. The directory tracks caching information at the cluster level, while bus snooping keeps the individual processor caches coherent.

A practical benefit of the prototype's structure is that a single cluster without its directory logic is a small-scale bus-based multiprocessor. This allows the cluster to be based on an existing commercial multiprocessor and helped reduce development time and effort. The clusters in the prototype are based on Silicon Graphics POWER Station 4D/340s [BJS88]. Although leveraging available hardware has constrained the coherence protocol and performance in some areas, the prototype retains the fundamental features of high-performance and scalable memory bandwidth.

The SGI 4D/340 system consists of four MIPS R3000 processors and R3010 floating point coprocessors running at 33 MHz. Each processor is nominally rated at 25 VAX MIPS and 4.9 DP LINPACK MFLOPS. Figure 6-2 shows a block diagram of a processor and its two levels of cache. The first-level caches consist of a 64-KByte instruction cache and a 64-KByte write-through data cache. The data cache interfaces to a 256-KByte, second-level write-back cache through a four-word write buffer. The write buffer allows the processor to continue executing instructions and accessing its first-level cache while writes are outstanding. Both the first- and second-level caches are direct-mapped and use 16-byte cache lines. The first-level caches are synchronous with their associated processor, and the second-level cache is synchronous to the 16 MHz cluster bus. The second-level caches are responsible for bus snooping and maintaining coherence among the data caches within the cluster. Coherence is maintained by an Illinois (MESI) protocol [PaP84, SwS86]. The Illinois protocol is especially useful in DASH since it specifies that processor caches should satisfy all reference possible (i.e., read requests if they have a shared copy of the requested line, and a read-exclusive request if they have a dirty copy). Such transfers do not reduce the latency of local memory, but they can short-circuit accesses to remote memory by sharing data between processor caches. Effectively, the set of processor caches act as a cluster cache for remote memory, similar to the shared multilevel caches proposed for other scalable systems [WWS+89, CGB91] (see Section 5.2 for details).

Local I/O devices within the cluster support scalable bandwidth to disks and communication channels. The I/O interface supports direct memory access (DMA) to memory. DMA access differs from normal processor access in that the DMA requestor does not contain a cache. Thus, DMA read operations must return coherent data, but this data is not subsequently kept coherent. Likewise, DMA write operations need to update all cached data, but they do not request exclusive ownership. The semantics of a DMA write amounts

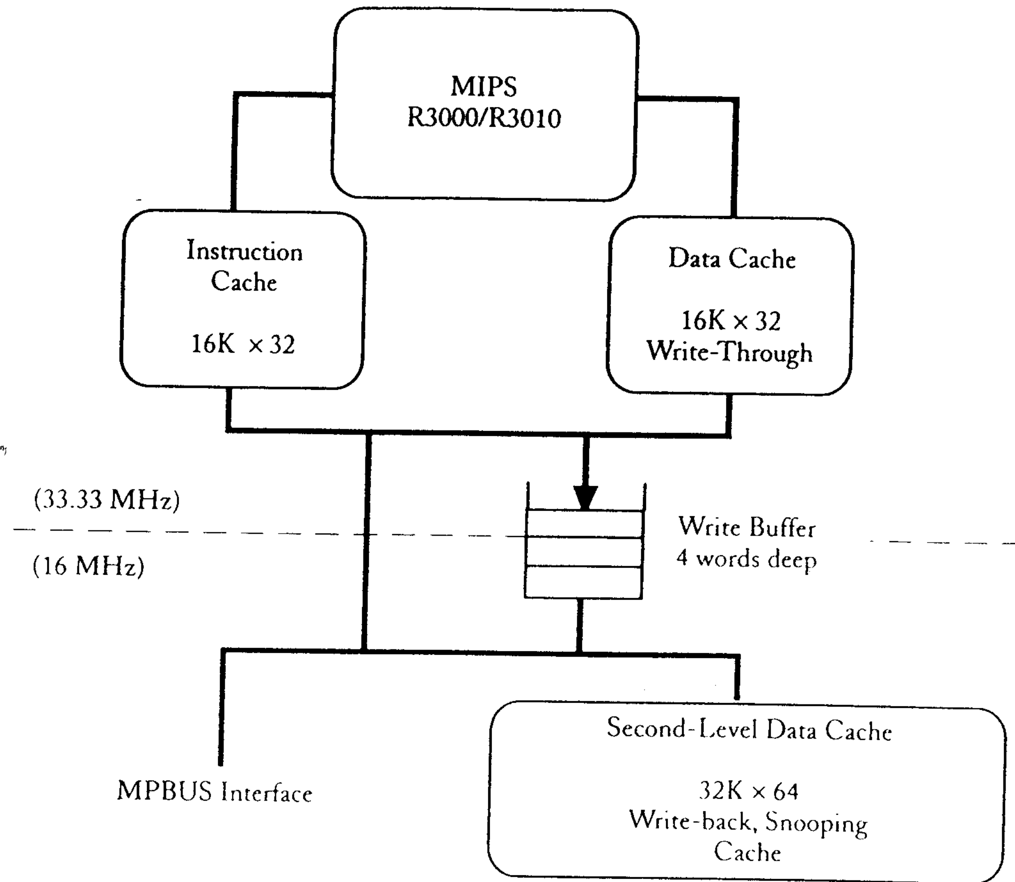


FIGURE 6-2 *Block Diagram of a Processor in the DASH Prototype.*

to an update coherence protocol. In the prototype, DMA transfers are supported across the system and are integrated into the support for processor update writes. Thus, support for DMA operations improves both I/O and interprocessor communication performance.

The cluster bus (MPBUS) of the 4D/340 is a synchronous bus that consists of separate 32-bit address and 64-bit data buses running at 16 MHz. While the MPBUS is pipelined, it is not split-transaction. The bus protocol is a problem for DASH because remote accesses must not occupy the bus while the request is outstanding. If such accesses did occupy the bus, then memory bandwidth would be reduced considerably, and deadlock could result. As shown in Figure 6-3, deadlock could occur when two processors in different clusters make accesses to the memory in the other's cluster (steps 1 and 2). If both processors continue to hold their local bus while attempting to acquire the bus in the other cluster (steps 3 and 4), then the system will deadlock.

The deadlock problem is solved in the prototype by adding a bus retry mechanism to the MPBUS, which effectively creates a split-transaction protocol for remote accesses. When a remote access is first made, the processor is forced to retry, and a request is sent to the remote cluster for service. To limit the loss in bus bandwidth while the remote request is outstanding, the bus arbiter is modified to accept a mask from the directory logic. The mask is set while the request is outstanding and keeps the processor from doing unnecessary retries. When the remote reply is received, the arbitration mask is released and the processor

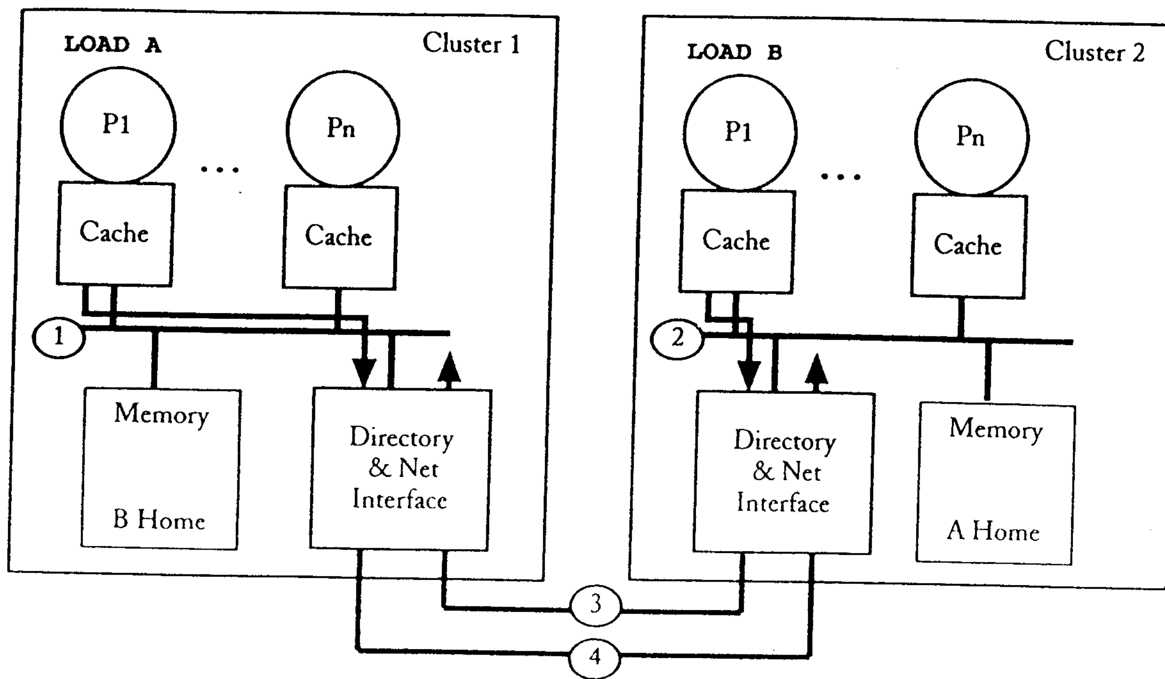


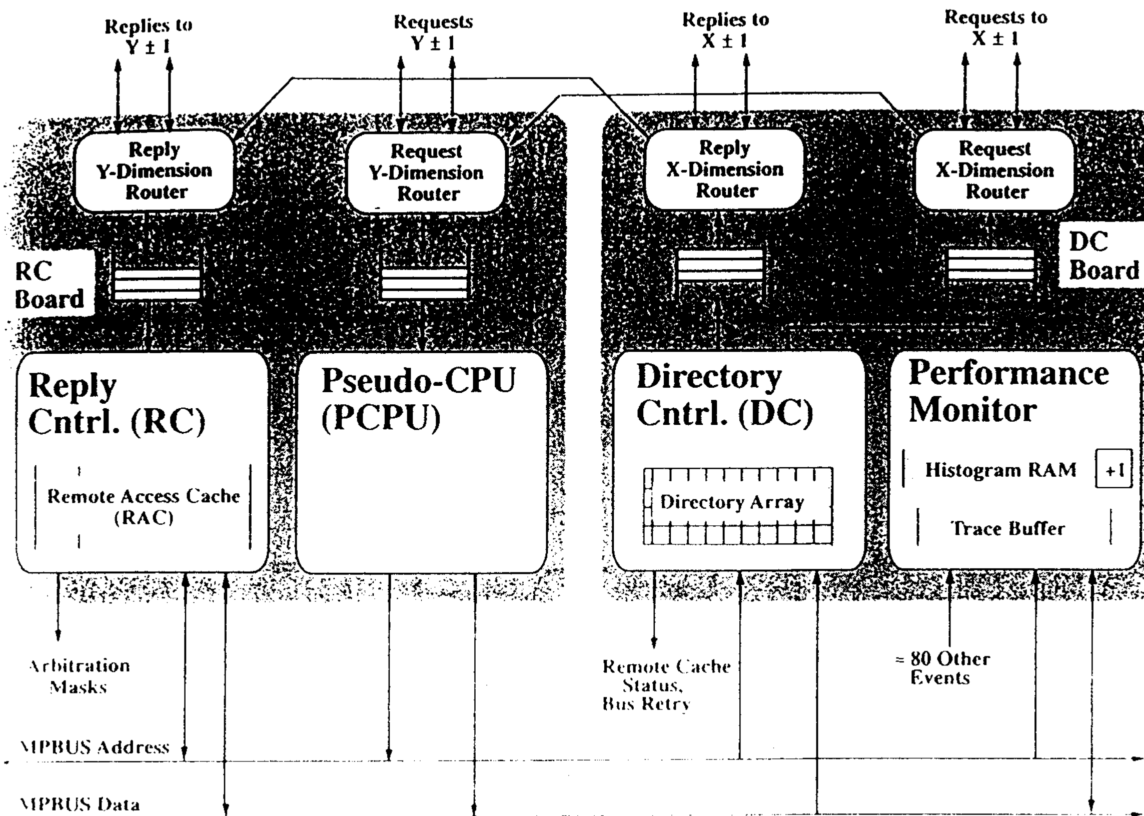
FIGURE 6-3 *Deadlock Example on Two Non-Split-Transaction Cluster Buses. (1) P1 in Cluster 1 loads A (generates local bus transaction). (2) P2 in Cluster 2 loads B (generates local bus transaction). (3) P1's read request can't be issued on Cluster 2 bus because of P2's outstanding request that is holding the bus. (4) P2's read request can't be issued on Cluster 1 bus because of P1's outstanding request that is holding the bus.*

retries the access. This time, the directory logic can satisfy the request, and the processor completes its memory access.

To use the 4D/340 in DASH, we have had to make minor modifications to the existing system boards and design a pair of new boards to support the directory and intercluster interface. The primary modification to the existing boards is support for the bus retry and arbitration masking outlined in the previous paragraph. Other minor modifications to the standard 4D/340 include changing the memory board to accept a local/remote decode signal from the directory logic and reducing the read miss fetch size from 64 bytes to 16 bytes. Reducing the fetch size to 16 bytes was done because the 64-byte fetch is actually done as four separate 16-byte bus transactions on the 4D/340. While this same technique could have been used across clusters in DASH, these accesses could not be pipelined. Thus, any hit rate improvement from the larger fetch size was likely to be nullified by the increase in miss penalty.

6.1.2 Directory Logic

The directory logic in DASH is responsible for implementing the directory-based coherence protocol and interconnecting the clusters within the system. A block diagram of the directory boards is shown in Figure 6-4. The logic is partitioned between the two boards roughly into the logic used for outbound and inbound portions of intercluster transactions. The boards are called the directory controller (DC) and reply controller (RC) boards, respectively.



IGURE 6-4 *Block Diagram of Directory Logic*

The DC board contains three major subsections. The first section is the *directory con-* *ller,* which includes the directory memory associated with the shared memory contained thin this cluster. The DC logic is responsible for keeping the directory updated and send- g all outbound network requests and replies. The second section is the request and reply tbound *network logic* together with the horizontal dimension of the network itself. cluded in this logic are FIFOs that isolate the local bus from congestion on the global net ork. The final section of the DC board is the *performance monitor.* The prototype is tended as an experimental machine, and the performance logic aids in the analysis of the stem by tracing and counting a variety of low-level intra- and intercluster events.

The directory memory is organized as a simple bit vector with one bit for each of the 16 usters. In addition to the bit vector, each directory entry contains two state bits and two rity bits. One state bit indicates whether the memory block is held dirty in some remote uster. The other bit is the logical OR of directory vector and provides a quick indication of hether the block is cached in a remote cluster. While the bit vector structure is not scalable, memory overhead is similar to more scalable schemes given the limited size of the proto pe. In addition, the full bit map allows more direct measurements of the caching behavior the machine. Scalable extensions to this structure and their effects on the prototype proto are outlined in Chapter 9.

The directory memory is accessed on each bus transaction. The directory information is mbined with the type of bus operation, address, and the result of snooping the local caches

Assuming false sharing is minimized, coherence misses are affected primarily by the actual cache coherence protocol. For example, support of a *clean-exclusive* state (this state is entered on a cache read miss when the block is uncached at the time of the cache fill) in the protocol [PaP84] reduces cache misses in the common case of a processor updating a location through a read-modify-write sequence. When there is actual processor-to-processor communication through shared-memory locations, however, an invalidation-based coherence protocol must cause cache misses. Some techniques attempt to delay or combine the implied invalidations [DBW+91], but maintaining coherence with invalidations will inevitably cause coherence misses. Update protocols can be used to eliminate these misses, but at the cost of substantial interconnect bandwidth to propagate updates [GDF94]. Unfortunately, these updates may not even represent real communication if the updated processor does not reread the updated location.

After minimizing cache misses, the next important task is to reduce the penalty of cache misses (see Section 1.2.1). Reducing the latency of these cache miss requests requires reducing the latency of the lower levels of the memory hierarchy. Generally, the levels of hierarchy below the cache include the following:

1. Additional levels of caches close to the processor, including structures such as cluster caches or snooping on the caches of neighboring processors
2. The main memory array that holds the requested location
3. A remote processor cache that holds the most up-to-date (i.e., dirty) copy of the location

The latency in accessing these levels, especially levels 2 and 3, is highly dependent on the latency of the interconnect network. The most obvious way to reduce latency is to use aggressive technology and implementation techniques to reduce the absolute time of accessing each level and traversing the network between the levels. While this technique is always desirable, it is not likely to be sufficient because the processor usually uses the same implementation technology as the memory system. Thus, the speed of memory in CPU clock cycles is likely to be large if the system is large. Even worse, given the significant investment in a given system and interconnect implementation, many systems eventually use next-generation processors with the previous-generation memory system. Thus, technology alone is insufficient, and architectural techniques must be used as well. In the next sections, we outline a number of the architectural techniques to reduce memory latency.

3.2.1 Nonuniform Memory Access (NUMA)

NUMA architectures attempt to reduce the average latency of memory by partitioning system memory and allocating a portion of memory to each processor node. While the most general definition of a NUMA architecture is simply a system with varying delays to different portions of memory, NUMA usually refers to systems in which nodes of the system include both processors and memory, as shown in Figure 3-2. The NUMA structure has been used on a number of systems (e.g., BBN Butterfly, Stanford DASH, and Cray T3D). For small systems, the NUMA structure reduces latency even if random locations are

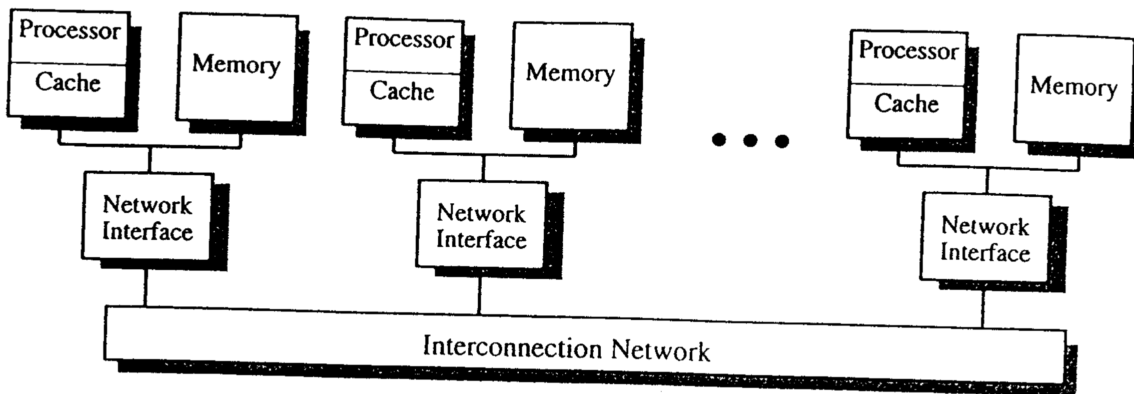


FIGURE 3-2 *Nonuniform Memory Access (NUMA) Architecture*

accessed because it will significantly reduce memory latency on the fraction of locations that are in the *local* memory (i.e., memory in the same node as the processor). For example, if there is a 3:1 ratio of remote to local memory delay, a NUMA architecture will reduce the average memory latency by 16% in a four-processor system, when compared with an architecture where all memory is remote.

On larger systems where local memory represents a small fraction of the system's total memory, software must properly allocate memory and limit process migration for the NUMA architecture to reduce latency significantly. One obvious area for optimization is data that is private to a given process. Likewise, if code is replicated, then code misses can be arranged to be satisfied from local memory. Finally, if shared data structures are accessed primarily by one processor, allocating these locations from the processor's local memory can reduce average latency.

3.2.2 Cache-Only Memory Architecture (COMA)

In a NUMA architecture every memory line has a fixed mapping from its address to the main memory of one node. In contrast, in a cache-only memory architecture (COMA) there is no fixed mapping. Even the system main memory space is treated as a cache, known as an *attraction memory* [HLH92] (see Figure 3-3). There are two major advantages to this arrangement. First, memory lines can move around to where they are needed. Second, shared copies of a line can exist in several nodes at once, which is beneficial for read-only or mostly-read data. These advantages help primarily by reducing the capacity misses of shared data, leading to lower average memory latencies.

The downside of COMA architectures is the additional hardware complexity required to treat main memory as a cache. This entails an additional level of mapping memory lines into attraction memories and finding a copy of a given cache line somewhere in the system. There is also overhead both in space (to store the tags) and time (to find a copy of a given line). In addition, the system must make sure that at least one copy of every memory line is retained in some node, which means that lines sometimes need to be moved around as a result of a replacement in the attraction memory.

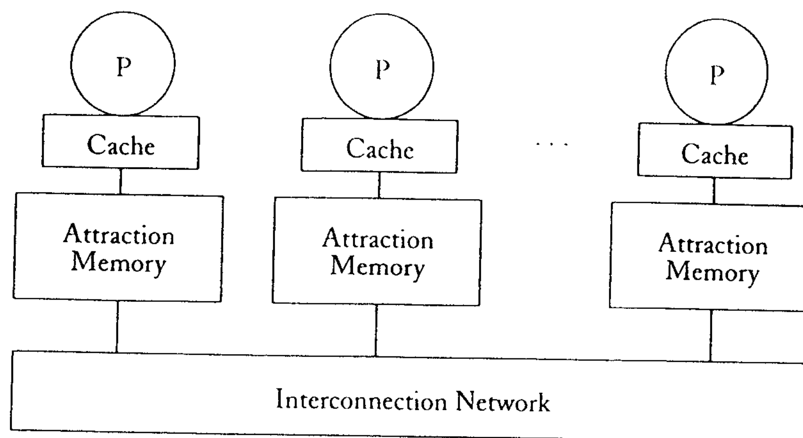


FIGURE 3-3 *Cache-Only Memory Architecture (COMA).*

Comparing the performance of COMA and NUMA architectures, we find that, on one hand, COMA machines have additional levels of caching, which reduce the average memory latency for large working sets that can be contained in the attraction memories. On the other hand, misses in the attraction memories are more expensive to service than processor cache misses in a NUMA machine, so if there is a lot of communication (which leads to attraction memory misses), the COMA machine sees larger memory latencies. Table 3-1 summarizes the comparison, which was inspired by the analysis presented by Singh et al. [SJG+93].

For each architecture we show the performance in four regimes of application behavior. The four regimes are characterized by the combinations of working set size (small or large) and communication-to-computation ratio (low or high). If the working set is small and the communication-to-computation ratio is low, both architectures do well because most references are satisfied out of the processor caches. If working sets and communication are large, both architectures do poorly: NUMA because of the poor cache performance, and COMA because of the high attraction memory miss latencies. If the working set is large and communication is low, then COMA does well because of the large attraction memories. In this regime, NUMA does medium well, unless the data can be placed locally with each processor, in which case NUMA does as well as COMA. In COMA architectures, data placement is achieved in hardware at a cache line granularity. For NUMA architectures, data placement can be attempted in software at a page granularity, which is successful for some applications, but not for all. Finally, if the working sets are small, but the communication-to-computation ratios are high, the caches of neither architecture help much. However, NUMA performs better due to the lower communication latencies.

Thus, there are different regimes of application behavior that favor either the NUMA or COMA architecture, and there is no obvious answer as to which architecture is better. It is clear, however, that COMA architectures are more expensive. In addition, COMA is relatively new, and there have been several proposals on how the overhead and complexity of the basic COMA architecture can be reduced [SJG92, HSL94]. This area of research is both active and interesting.

TABLE 3-1

Performance of NUMA and COMA Architectures

COMMUNICATION-TO-COMPUTATION RATIO	NUMA		COMA	
	SMALL WORKING SET	LARGE WORKING SET	SMALL WORKING SET	LARGE WORKING SET
Low	Good	Medium	Good	Good
High	Medium	Poor	Poor	Poor

3.2.3 Direct Interconnect Networks

Direct and indirect networks were introduced in Chapter 1. Here we look at them again in the context of latency reduction. A *direct interconnect network* [AtS88] is a switching network in which processing and memory nodes are allocated at the switch points. Indirect networks put the nodes at only the inputs and outputs of the network. Direct networks reduce latency because an average request only traverses a small number of the network switch points to reach its destination; in an indirect network every request must travel across the entire network. The disadvantage of a direct network is that distributing the network across the nodes makes the switch physically bigger, potentially slowing down the network. Furthermore, if the direct network is implemented as part of the individual nodes, it becomes difficult to change the network characteristics (e.g., additional dimensions in a mesh) as the system grows.

Direct networks can also be viewed as another level of NUMA, which can be exploited in software by allocating memory and processors that must communicate near to one another. This arrangement also permits the direct network to have increased aggregate bandwidth when communication is primarily with neighboring nodes. In many physical problems and their corresponding computational models, much of the interprocessor communication is between neighboring nodes, and a direct network can be very useful in decreasing latency and increasing bandwidth. In actual machines, both direct and indirect networks have been used. The MIT Alewife, Intel Paragon, Stanford DASH, and Cray T3D all use direct networks; the IBM RP3, the CM-5, and the University of Illinois Cedar machine use indirect networks.

3.2.4 Hierarchical Access

Since much interprocessor communication and data sharing is restricted to a subset of the processors, a hierarchical search can reduce memory latency if the locality of data sharing matches the structure of the hierarchy. Such a hierarchical search typically employs an interconnection network that maintains information about the outstanding requests or caching state of neighboring processors. A hierarchical search can reduce latency for both locally shared and widely shared data items. For local communication, searching the nearby caches retrieves data directly from the writing processor and avoids accessing a remote memory. For widely shared items, interaction between the local processors occurs indirectly when the