Coherent Shared Memory on a Message Passing Machine

Roberto Bisiani, Andreas Nowatzyk and Mosur Ravishankar

December 1988

CMU-CS-88-204 3

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This research is sponsored by the Defense Advanced Research Projects Agency, DoD, through ARPA Order 5167, and monitored by the Space and Naval Warfare Systems Command under contract N00039-85-C-0163. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or of the United States Government. Parts of this report have been submitted for publication and will probably be copyrighted if accepted.

Table of Contents

List of Figures

List of Tables

Abstract

This report describes a new technique to maintain the coherency of replicated read/write data in large multiprocessors that do not have a global bus. The coherency maintenance technique makes it possible to efficiently support a shared memory programming model on a message passing machine. The report contains performance evaluation data obtained by executing both synthetic load and real applications on a detailed simulator.

1. Introduction

This report addresses some of the problems arising in the design of multiprocessors that have no physical shared memory but support a shared memory programming model at the architecture (instruction set) level.

There are many reasons why such a multiprocessor can be useful:

- *The widespread use of bus-based shared-memory processors has created a large base of existing software systems and applications that rely on the shared memory programming model.* On the other hand, bus-based architectures are limited to a few processors.
- *A shared-memory programming model can be more convenient.* For example, communication by critical regions and side-effects may be easier to understand than message passing when the structure of the problem and its parallel decomposition call for multiple processors accessing the same data.
- *An efficient shared memory model facilitates load balancing.* With message passing, work has to be explicitly assigned to a processor and, if necessary, rebalanced by explicitly redistributing the load to underutilized processors. In a shared memory model all processors can grab work as needed from one or more shared queues.
- *Distributed-memory machines are easier and cheaper to build.* Distributed-memory machines can be built with memory/processor modules that are interconnected by means of a small number of wires. On the contrary, the complexity of high performance buses requires shared memory machines to be engineered around a carefully designed backplane with a fixed number of slots.
- *Distributed-memory machines are more scalable.* Technology being equal, a distributed-memory architecture can support more processors than a machine with physically-shared memory.

The easiest way to support a shared memory model is to connect memories to processors by a single multiplexed connection -- a bus. The weakness of a bus lies in its limited bandwidth; its strength in the fact that all memory accesses are visible by all the processors in the system. The load on the bus can be lowered if data are replicated in local memories (caches) and local copies are kept coherent with the main memory content. There are a number of well-known, practical protocols that can maintain the content of the caches coherent if the architecture is bus-based. The effect of these protocols is such that the behavior of the machine (but not the speed!) is identical to the behavior of a machine with a single centralized shared memory and no caches. Most commercial systems are bus-based.

A bus limits the number of processors: the exact number of usable processors depends both on technology, e.g. the relative speed of bus and processor, and on application behavior. If we exclude "embarassingly parallel" applications (that would work well on any parallel architecture because of their very low communication rate) practical bus-based systems seem to be limited to a few 10s of processors. For example, by using the performance data from [1], a 10 MIPS processor will typically make one reference on the global bus every 15 instructions. If the bus can service 10 million references per second, 15 processors will use up all the available bus bandwidth.

There are only a few working systems that use processor-memory interconnections with a higher bandwidth than a bus. The major problem these systems face is keeping the memory system coherent. The solutions currently in use range from having no caches and waiting for each memory reference to be acknowledged before issuing the next one (Butterfly [7]), to caching only variables that are not shared (RP3 [14]). There are also some coherency protocols that do not require a bus because they maintain *directories* of the replicated data [1] at the main memory site. These protocols are largely untested in real systems.

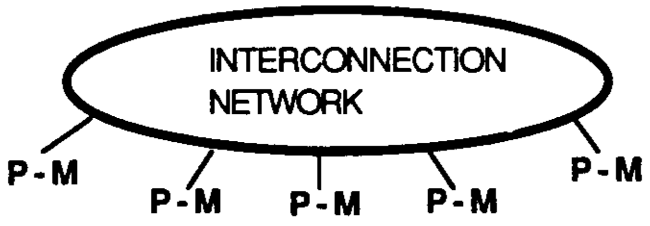

Figure 1-1: Generic Distributed-memory Architecture

This report describes a solution that does not require the use of a bus or a central directory to maintain the coherence of readable and writable replicated data. Although the coherency algorithm can be adapted to many architectures, we focus on distributed-memory architectures (Figure 1-1) in which each node has a general purpose processor, a fraction of the global shared memory and an interface that connects memory and processor to a communication network. The same data can be in more than one local memory and in more than one cache. The local memory and the cache are kept coherent with a local snooping protocol while the local memory of different nodes is kept coherent by the algorithm we will describe in the next Sections. In Section 2 we explain the coherency problem. In Section 3 we describe the coherency mechanism. In Section 4 we describe in detail the architecture that has been used for the evaluation of Section 5.

2. Shared Memory Coherence

Regardless of the language they are using, programmers commonly assume that every write operation is performed immediately and indivisibly. This implies that the value of a variable is always the value stored by the last write operation, i.e. a read operation always returns the last value written. Moreover, in a parallel program, it is common practice to assume that operations performed in sequence by one processor are also observed in sequence, albeit possibly with some delay, by all the other processors.

This assumption may be violated in two important cases:

1. If the memory system contains multiple memory banks, and processors do not wait for each access to be acknowledged before performing the next, then one processor could observe the actions of another processor as if they were performed out of sequence. For example, two write operations performed one after the other by one processor might be serviced at widely different times because of network delays. Another processor that

reads the same locations at about the same time might observe the effects of the second write operation but not the effects of the first one.

2. If the same variable is replicated in multiple memories, a write operation must update or invalidate the copies. If this operation cannot be performed indivisibly, some processors might observe the effect of multiple updates out of order.

Both cases prevent correct synchronization between processors and, therefore, make correct parallel processing impossible. The former situation can be avoided if all processors wait for an acknowledge from the memory system before performing another access. This is quite limiting since the latency of the interconnection network is typically much larger than the instruction time of processors of comparable technology. This seems to be the solution adopted in the RP3 multiprocessor ([14]).

The latter situation can be avoided if the hardware prevents all the processors from accessing a replicated value until the value in all the copies is *coherent*. There are quite a few coherence protocols but most of them require that all the processors be connected to all the memories through a single bus. If a bus is not available, *directory* schemes (see [1]) are a promising solution that is not used in any existing machine with more than a few processors and whose effectiveness remains mainly untested. In a directory scheme, the information on whether a block is cached or not is kept in a table that is associated with main memory. The performance of the best directory scheme seems to be comparable in number of communication cycles to the performance of bus-based schemes. For example, Hennessy et al. suggest that a directory scheme like the one described in [2] requires about four communication cycles for each memory reference. Since a communication cycle in a machine with many processors can be orders of magnitude larger than the latency of main memory access, the measurement of the performance of a directory scheme should take into account the impact of latency on the processors' execution time. Since we do not know of any such evaluation it is not possible at this point to do a fair comparison of directory schemes with any other solution.

All these coherency maintenance techniques guarantee what has been called by Dubois et al. in [9] *strong coherence*. Strongly coherent systems do not violate either of the two previous assumptions. Therefore, synchronization can be performed by simple read and write operations and atomic read/write sequences. Although strong coherence is sufficient to implement a usable shared-memory parallel-processing model, it is not necessary. In this report we argue that a less-restrictive form of coherence, called *weak-coherence*, is sufficient and easier to implement.

2.1. Weak Coherence

Typically, a parallel program alternates between a long sequence of normal read and write operations on shared data structures and synchronization operations (e.g. P and V). Enforcing strong coherence *among* the normal read and write operations is not necessary if the programmer understands that a synchronization operation should be used every time a specific order of access between two concurrent computations must be enforced. Assume that all synchronization accesses are strongly coherent among themselves and in relation to the preceding and following sequence of normal accesses. At the same time, do not make any assumption on the coherence of a sequence of normal accesses relative to other sequences of normal accesses. In this case the memory system is said to implement *weak coherence* (see the paper by Dubois for a formal definition of weak-coherence [9]). Our contention is that weak coherence is easier to implement on non-bus architectures and does not impair programming capabilities.

The only problem generated by weak coherence at the programming level is that synchronization operations must be explicitly flagged by the programmer so that the system can implement them correctly. For example, the code in Table 2-1 shows two ways of programming a producer-consumer synchronization. In a strongly coherent memory system all write operations are "visible" as soon as the processor has performed them and the **(a)** code achieves the desired effect. In a weakly-coherent

```
/* Shared data structure */          /* Shared data structure */
struct a                             struct a
{                                    {
 int a;                               int a;
 int flag = FALSE;                    int flag = 0;
} p;                                 } p;


/* Process 1 */                      /* Process 1 */
...                                  ...
p.a = ...;                           p.a = ...
p.flag= TRUE;                        V(p.flag);
....                                 ...

/* Process 2 */                      /* Process 2 */
...                                  ...
while (p.flag == FALSE)              P(p.flag)
    wait();
... = p.a;                           ... = p.a;
...                                  ...

    (a)                                 (b)
```

Figure 2-1: Wrong (a) and Correct (b) Code for a Weakly Coherent System

system, if *p.a* and *p.flag* are allocated in separate memory modules, it is possible for the write of *p.a* to be delayed after the write and subsequent read by Process 2 of *p.flag*. The **(b)** code in Table 2-1 is correct because the synchronization operation has been made explicit. We believe that the coding style exemplified by the **(a)** code of Table 2-1 should be avoided because it masquerades a synchronization operation as a normal access and may be misunderstood. We have examined the code of the Mach operating system kernel and found that a synchronization had been implemented as a normal access only in one case. We believe most programs would port to a weakly-coherent system without any change. In the next Sections we will show that weak-coherence can be easily implemented on large multiprocessors.

3. Maintaining Weak Coherence: the Gray Zone

Coherency maintenance is usually based on observing all the memory operations and preventing the occurrence of transactions that might put the system in an incoherent state. For example, writing to a variable that has multiple copies might cause all the copies to be invalidated before the write operation occurs. Our mechanism, instead, is based on performing all the operations on all the copies in parallel but making sure that each copy ends up with the same value. This requires a global clock to timestamp every request before it is sent to other nodes. Consider a number of nodes accessing a shared variable stored in the memories of some other nodes. Each node may arbitrarily issue *read* and *write* messages. Any network will have some delay associated with the transmission of messages and will have to deal with contention for communication resources. Both effects can perturb the sequence of messages that arrive at the nodes where the shared variable is stored, thus destroying coherency. If all messages were timestamped on creation, the sequence of *read/write* messages to a particular variable could be sorted at the receiving end and then performed correctly[1].

Synchronization operations also require that some special operation on the variable be performed in a specific order; this special operation computes a value that will be returned to the processor. For example, if multiple fetch-and-add operations are performed on the same variable, each operation must return a unique value that depends on the ordering of the requests. Sorting of timestamped requests achieves the correct result.

Unfortunately, the sorting step can only be performed when all the requests that might change the outcome of an operation have arrived at the node that contains the shared variable. The implementation of the sorting step is impossible unless the upper bound on the message propagation time in the network is known. The simplest solution is to approximate the upper bound with a fixed upper bound, discarding and retrying all messages that take more than the upper bound. In this case, a small upper bound would minimize the time to finalize the memory operations but would potentially cause a lot of messages to exceed the bound and be deleted. A large upper bound would have the opposite effect. A fixed bound could never be tuned correctly, because the optimal value of the bound depends on the delay through the network and on the load, both of which are varying continuously.

Our mechanism keeps a dynamic bound that is a good approximation of the real bound without requiring much network bandwidth. In the next two subsections we will describe how absolute time and communication-time upper bound can be maintained.

3.1. Maintaining Absolute Time

Providing each processor in the system with an accurate absolute time requires a counter that is reset at system initialization time and is continuously incremented by a clock signal that is common to all the nodes. For this to work properly two problems must be solved: the global distribution of a common clock signal and the detection of inconsistencies and subsequent recovery from them.

The distribution of a global clock is perhaps an overrated problem as there are numerous large systems that operate on a single clock, e.g. [13], [3], [14] and [15]. A single clock source is not only necessary in order to maintain an absolute time, but it is also advantageous for interprocessor communication (eliminates the need for synchronizers, allows deterministic protocols that can be implemented with simple finite state machines and improves some routing and resource allocation strategies).

The implementation of a large (say 1000 processors), synchronous system with a single, global clock and tight bounds on the skew between any pair of processors is feasible with a conservative

[1]It is assumed that the global clock has a resolution sufficient to distinguish all requests. A clock that is advanced at the maximum message creation rate is sufficient if there is a way to distinguish multiple PEs issuing messages at the same time, for example fixed PE-id based priorities.

design by treating the signal as an analog, RF source. The continuously running clock from a crystal oscillator is amplified with a broadband power amplifier and, instead of using digital gates to divide the clock signal, passive power splitters are used.

However, given a good common clock distribution, the problem is only half solved. No serious digital design should be unprepared for a glitch in a critical component. If the proposed system were to rely on a large number of counters to advance in lock step fashion after reset - with no way of verifying consistency - it would be hard to recognize and diagnose errors. If one of the time keeping counters in a 1000 processor system were to be off by a small amount - say due to a power supply glitch, an alpha-particle hitting a critical gate - no immediate problem would be apparent. In the case of particular usage patterns that are sensitive to incoherent behavior of the machine, errors might occur that would be virtually impossible to diagnose due to the potentially long period between cause and effect. Even worse, in many applications it would be hard to recognize that the results are incorrect.

Note that the severity of the timer problem arises from its accumulative nature: any glitch would be preserved for as long as the system is running, potentially months or years. Thus, while the probability for a single counter failure is quite low, it is not likely that all 1000 of them will count correctly for a year at a high rate. Fortunately, periodic consistency checks and procedures to identify and correct transient errors can lower the probability system failure to a tolerable level. These procedures are facilitated by the fact that the communication network, as we will see in Section 4, is synchronous.

3.2. The Gray Zone

A coherent and reliable time distribution mechanism, such as the one outlined in the previous Section, is not sufficient to build a coherent memory system. Proper timestamping can resolve any ambiguity among competing operations but is unable to finalize any operation without additional knowledge about the behavior of the message passing layer. Typical communication systems for distributed memory multiprocessors have variable and potentially unbounded message latencies that depend on the traffic pattern, network load, routing and scheduling strategies. As a consequence, the receiver of a message has no way of knowing if there are messages still in transit with timestamps that are older than the timestamps of the message just received.

Assuming that the message passing layer is well behaved,[2] all operations will settle in a coherent final state in a finite time span. Let's define the upper bound of this time span:

Definition 1: The *Gray Zone* at time t is the smallest time interval $[t_{gz},t]$ that contains the timestamps of all outstanding messages at time t.

Note that t_{gz} is a function of t. Furthermore, it is a system-wide global quantity. We say that a message **M** has *matured* if its timestamp is smaller than t_{gz}; this implies that any message sent before or at the same time as **M** has been received. Operations that require the coherent value of a memory location will stall until the requesting message has matured, while all other operations proceed immediately.

[2]A message passing layer is considered well behaved if it is free of deadlocks, and if it is fair.

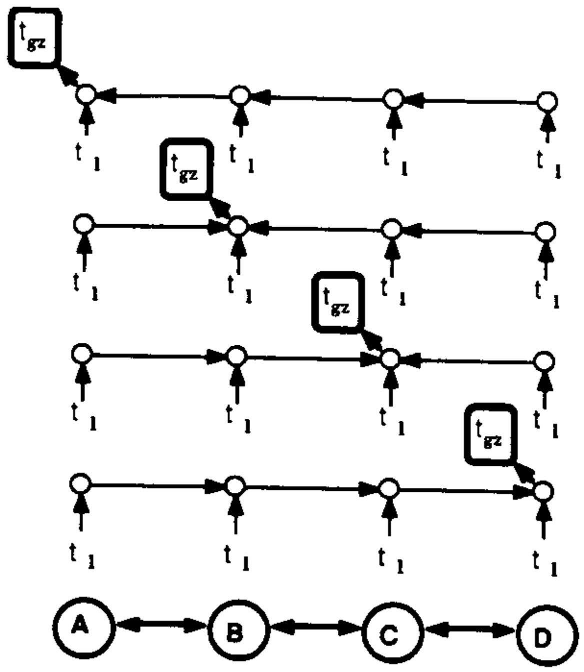

Figure 3-1: The Computation of the Gray Zone width in a 1-dimensional network

Let us call t_l the smallest timestamp of all the messages that are in the queues of a given node waiting to be transmitted (or retransmitted). Each node can easily compute t_l by updating it every time a new message enters or leaves a queue. Imagine four nodes connected by a 1-dimensional cube, for example the 4-ary cube in Figure 3-1. If node D sends t_l to the node on its left (C) then C can compute t_{gz} of the subnetwork C-D as the minimum of the t_l sent by D and its local value. If C sends this value to B and B also takes the minimum with its t_l and sends it to A then A can compute a system-wide approximation of t_{gz}. The value is approximate because it is based on the value of t_{gz} in B at $t-1$, the value of t_{gz} in C at $t-2$, etc. This procedure must be performed as often as possible to keep a good approximation.

It is easy to see that if A, B and C also compute the minimum of their t_l and the t_{gz} of their left subnetwork and send it to their right neighbors, then D will be able to maintain a system-wide approximation of t_{gz}. Let us now take node B. If B takes the minimum of t_l and the t_{gz} coming from its right and left subnetworks, then it can also compute the value of the system-wide t_{gz}. It might be useful to imagine each node in the system as the root of a tree that contains all the other nodes: local t_{gz}'s are propagated up the tree and, when they reach the root after depth-of-the-tree message cycles, the root is able to compute an approximation of the global t_{gz}.

As it is apparent from the simple case of Figure 3-1, it is not necessary to transfer the information relative to the trees rooted in each of the nodes separately: every cycle each node only needs to send one new approximation out on each of its output connections and receive one partial value from all of its input connections. Assume that the message system transfers fixed-length messages, one per time unit. If the network has diameter d then it will take d time units for the approximate t_{gz} to propagate, if every message carries this information.

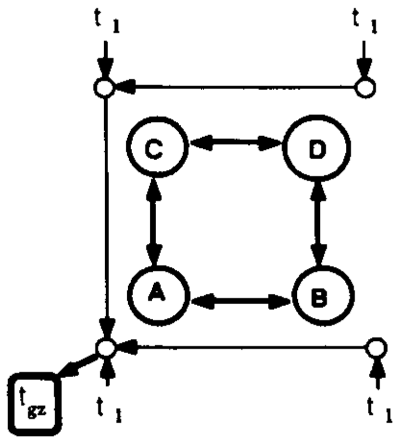

Figure 3-2: The Computation of the Gray Zone width in a 2-dimensional network

The Gray Zone computation that each node must perform is a function of the topology of the network and can be driven by a locally stored table once the configuration is known. In the case of binary n-cubes (see Figure 3-2) the derivation of the computation table is simple. Recall the construction of a boolean n-cube of degree d: first construct two subcubes of degree $d-1$ and then connect each node of one of the subcubes to one node of the other subcube. Now assume that each of the nodes in the two $d-1$ subcubes has the value of t_{gz} for its subcube. If each node sends this value to its corresponding node in the other subcube then that node can compute its view of the global t_{gz} as the minimum of the values of t_{gz} for the two subcubes. For example, see Figure 3-2, **C** first computes t_{gz} of the **C-D** subcube and then sends it to **A**, where it is combined with t_{gz} of the **A-B** subcube. For the binary n-cube topology the maximum time for a gray zone increment to reach all nodes is a function of the network diameter.

For any given topology, if a constructive mechanism for building the gray zone computation tables is not known, it is possible to use a minimum-depth spanning tree of the interconnection network as the basis for updating the Gray Zone information. In this case the maximum update time can be between d (the network diameter) and $2d$, depending on the topology (see [12] for more information).

In the following sections we describe the use of the Gray Zone concept in the implementation of weakly coherent memory operations in a system that has no physical shared memory. (Section 4 provides further implementation details.)

3.2.1. Read Operations

If a node performing a read operation has a local copy of the memory location, the read operation simply consists of a local memory access. Otherwise, a request message is transmitted to the nearest node that has a copy of the memory location, which returns the value. This is a weakly-coherent operation; if some other node had written a certain value to the memory location at some earlier time, there is no guarantee that the read operation would return that value. (The *coherent-read* synchronization operation does guarantees this, see Section 3.2.3 below.)

3.2.2. Write Operations

As in the case of read operations, write operations are performed locally or on the nearest node that has a copy of the memory location. In the latter case, the first node sends a request containing the address, data, and the timestamp of the operation to the remote node, after which it can continue normal operation. The write operation is then propagated to all copies of the memory location by *update request messages*. Both the original write request to the nearest copy (if necessary) and the write-update messages contain the *original timestamp* of the write operation, so that each node can sort all updates to memory in time order and perform them consistently. (If two nodes issue write operations to the same location with the same timestamp, the node-id is used to disambiguate these requests.) The sorting operation is implemented as follows. Each node has a *coherency buffer* that maintains the addresses and timestamps of the locations that have been changed recently on that node. Whenever the node receives a write or update request, it checks the coherency buffer to see if it contains an entry for the addressed location. If so, the request is allowed to proceed only if its timestamp is newer than that in the coherency buffer, and the timestamp in the coherency buffer is also updated. Otherwise, a coherency buffer entry is created. Whenever an entry in the coherency buffer has a timestamp older than t_{gz}, it can be discarded; this limits the size of the buffer.

Since the original timestamp is retained in all the update messages, the Gray Zone time t_{gz} is prevented from advancing until the update messages have all been received and processed. This affects the performance of synchronization operations, as explained below.

3.2.3. Synchronization Operations

Synchronization operations are handled similarly to write operations: the operation is handled locally or at the nearest copy, and then update messages containing the original timestamp are transmitted to all the remaining copies of the memory location, where they are sorted through the coherency buffer. But, in addition, the result of the synchroniztion must be returned to the node originating the operation.[3] This result is the value the memory location would contain if all preceding operations to that location had completed, and no succeeding one had. Furthermore, the strongly coherent nature of the synchronization operation also implies that when it completes, all preceding operations to memory by the node performing that operation have also completed. All this is handled, locally if there is a local copy or at the nearest copy, by the following mechanism.

For each location on which a synchronization operation is pending, the coherency buffer is extended by several fields: a value (initially the value obtained from memory), the timestamp corresponding to the value t_v (initially the value in the coherency buffer if any, or t_{gz}), and the timestamp of the operation t_{op}.[4] Whenever the node has to perform some other memory operation on that memory location, the value field is updated appropriately if the timestamp of that operation falls between t_v and t_{op}, and t_v is updated to the new timestamp. This extension to the coherency buffer is maintained as long as t_{gz} is less than t_{op}, at which point it *matures* and the value field is returned as the result of the synchronization operation, and the extension can be discarded. As a

[3]We have studied the implementation of four kinds of synchronization operations: *fetch-and-add*, *test-and-set*, *exchange*, and *coherent-read*. The last is similar to any other synchronization operation, but does not change the memory location in any way.

[4]Note that several synchronization operations may be pending simultaneously on a single memory location, so the coherency buffer should provide for several such extensions to exist simultaneously.

result, the performance of synchronization operations depends mainly on the Gray Zone width.

A consequence of this mechanism is that an approximate result of the synchronization operation is available immediately. This can be advantageous if it has a high probability of being the correct outcome of the synchronization, as in the case of a lock for which there is little contention. Therefore, synchronization operations are implemented in two phases: the synchronization operation itself, that sets up the coherency buffer and its extension and returns an initial guess, and a subsequent *verify* operation that waits for the final result to become available. Other operations can be executed in between, thus absorbing some of the synchronization latency.

```
....
...
if (P(semaphore, tag))
...
/* instruction that do not depend on the outcome of P() or can be undone */
...
if(verify(tag))
...
/* instructions that depend on the outcome of P() */
...
.....
```

Figure 3-3: Delayed Synchronization

For example, the synchronization statement *P(semaphore,tag)* in Figure 3-3 can precede any instruction that can be executed independent of the outcome of the synchronization. (The initial guess returned by this synchronization can also be used, provided the actions can be undone if the guess proves to be wrong.) When the final result becomes necessary, the statement *verify(tag)* stalls execution until the outcome of synchronization operation is available. The variable *tag* makes it possible to have more than one pending synchronization.

4. A Distributed-memory Architecture

In order to be able to perform some initial evaluations we have assumed the availability of a specific architecture. This architecture has homogeneous nodes, see Figure 4-1, each node containing a general-purpose processor with a cache and memory mapping, some dynamic RAM memory and a *network interface* that handles coherency management, global memory mapping and routing.

Each node has a fixed number of communication channels that can be connected to channels on other nodes. No constraints are imposed on the connection pattern. In particular, it is possible to optimize the interconnection topology according to the characteristics of specific applications. The network interface is concentrated in a device that performs a number of different but closely related functions: i.e. memory coherence, system-wide memory management and routing. All these functions can be implemented by a number of independent state machines and a couple of small but functionally specialized memories. With current technology, it is reasonable to build an integrated circuit that contains all the circuits necessary to perform these functions. The functions of the network interface are defined below.

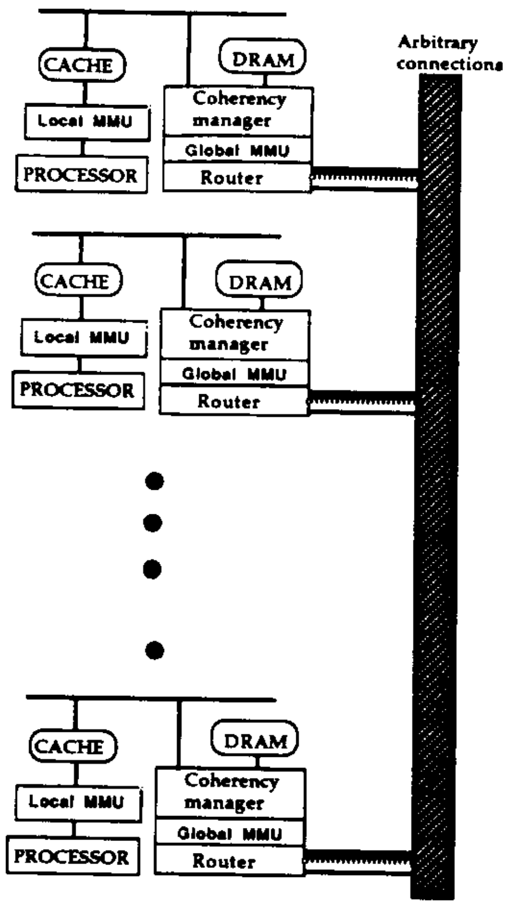

Figure 4-1: The Architecture Used in the Performance Evaluation

Memory Mapping. The memory mapping we have chosen is compatible with state-of-the-art general-purpose operating systems and processors. Coherency management, described below, does not require a virtual memory organization but software production of non-trivial applications depends on good memory management and full operating system support.

The system manages memory in pages. Processors manipulate per-process virtual addresses that are translated by a local memory mapping unit (MMU) into physical addresses (see Figure Figure 4-2). The local MMU could be a standard device like the MC88200 that includes a cache and translation look-aside buffer. Both virtual and physical addresses are 32-bit byte addresses.

Physical addresses are translated into *hardware* addresses for the local memory and/or messages that are sent to other nodes. These operations are performed by the *global* MMU that is managed by the operating system. The mapping tables of the global MMU are stored in the local DRAM memory.

There are four types of physical pages, distinguished by the two high-order address bits:

Local, private:	The page address directly references the local memory, the page can only be accessed on this node. There is no global mapping entry for such pages, and the memory reference can proceed at full speed. This kind of page is used for program code and stack, for example.

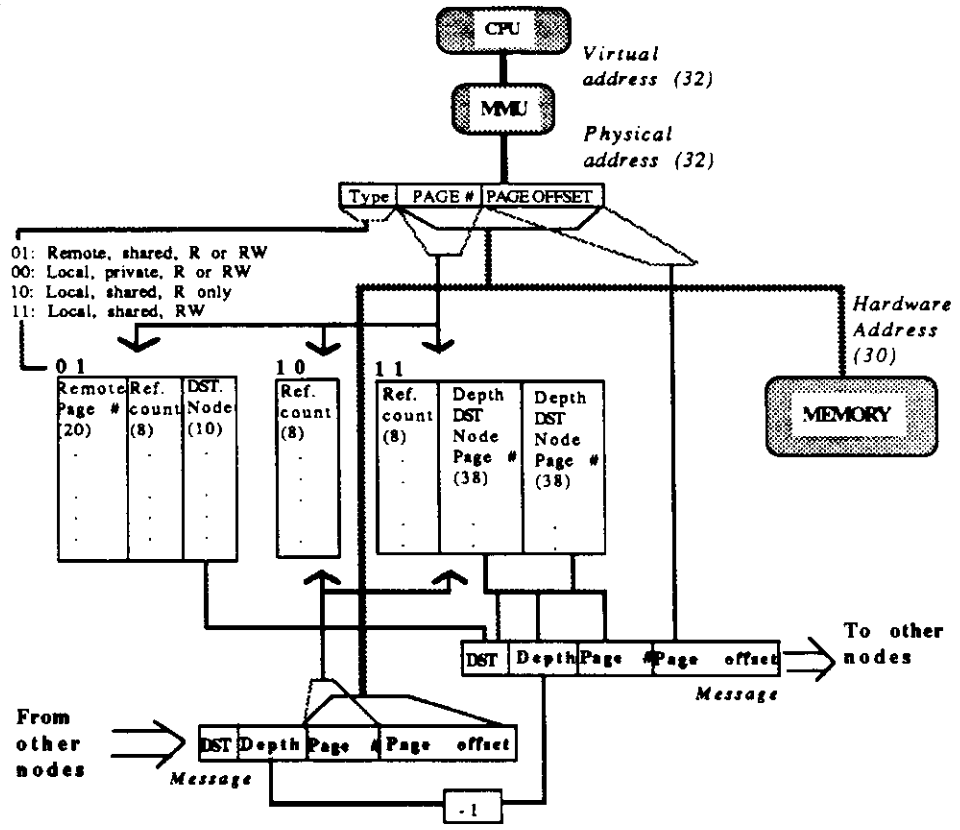

Figure 4-2: The Memory Mapping Structure

Local, shared, read only:
: The page address directly references the local memory. Other nodes have this page mapped in their hardware address space (i.e., the page is replicated on multiple nodes) but nobody may write it. This reference can also proceed at full speed but the global MMU maintains a reference counter that is incremented every time the page is accessed and can be used by memory management policies to improve memory allocation.

Local, shared, read/write:
: The page address directly references the local memory. Other nodes have this page mapped in their hardware address space (i.e., the page is replicated on multiple nodes) and the page is writable. Hence, any write or synchronization operation (other than *coherent-read*) must result in update messages being forwarded to all copies of the page. This is achieved efficiently by creating a minimum-depth binary multicast-tree from these nodes. The page table entry on each node contains two pointers to two nodes down the tree. The leaf nodes point back to the root node, so an update initiated at any level in the tree can eventually propagate to all the nodes in the tree; the *depth* field in the mapping table (see figure) is used to avoid cycling perpetually through the folded tree. The table also has a reference counter that is used by the memory allocation policies.

Remote, shared; The page is not available locally; a remote access is necessary. The page may be replicated on multiple remote nodes. The page address is used to access a table that identifies the nearest remote node containing the page, and the address of the page on that node. There is also a reference counter that is used by the memory allocation policies.

Notice that sharing or page replication occurs at the page level. However, remote accesses and update operations take place at the word level. That is, a remote read operation only results in the transfer of the desired word, and a write operation only updates the single word in all copies of the address page.

The physical-hardware address translation tables shown in Figure 4-2 are stored in DRAM memory and some overlapping between local memory and table access is possible. Each local shared page needs 1 byte and each read/write local page needs 11 bytes. Remote pages require 5 bytes per page.

From the user's point of view there is no difference between this memory organization and the organization of a uniprocessor or a bus-based multiprocessor. From the operating system's view, most memory management algorithms remain unchanged. The differences occur in memory management algorithms that deal with the migration or replication of pages between nodes. These must be performed only after having notified and received permission from the underlying hardware.

Coherency Management. Coherency management consists of Gray Zone computation, update management (of replicated memory pages) and synchronization management. The global Gray Zone computation is performed in the underlying interconnection network. The network is synchronous: a 64-bit fixed-length message packet is exchanged over each channel at every message cycle of 450ns. A message contains, among other fields, source and destination node addresses, a timestamp, and either a 32-bit address or 32-bit data. It also contains a field for exchanging Gray Zone information, as described in Section 3.2. In order to save network bandwidth, instead of transmitting the absolute value of t_{gz} we transmit an incremental value that indicates how much the new t_{gz} has changed from the old. We encode this value using two bits, so we can only represent the case of t_{gz} remaining stable or increasing by 1, 2 or 3. Since the absolute time also increases every message cycle, the encoding actually represents the case of the Gray Zone becoming wider, remaining constant, or narrowing by one or two message cycle units. If the Gray Zone changes by a larger value, the system will encode this in several consecutive increments. This method requires less than 4% of the network bandwidth.

Update management and synchronization management is performed by the coherency buffers and their extensions as described in Sections 3.2.2 and 3.2.3.

Routing. A non-deterministic, adaptive routing strategy is used to provide good resource utilization and dynamic behavior even under high load conditions. In particular, high dimensional topologies may be used to increase bandwidth without compromising latency because multiple alternate paths between two given nodes can be used concurrently. Implementation simplicity is achieved by supporting precisely one packet type with fixed length and format. A tightly synchronized store-and-forward strategy is used which fully utilizes the physical channel bandwidth and improves the efficiency of the routing heuristic (see [12] for an analysis of the characteristics of such a network).

The routing component is built around a register file with serial access capability. Essentially, each word in the register file is a shift-register that can store one entire packet plus some transient information (such as the subset of transmitters that can be used to advance the packet toward its destination). Serial access[5] is supported so that transmitters can operate directly out of the register file. This eliminates the need for special parallel to serial converters in each transmitter and their counterpart in each receiver. This also reduces the need for data transfers, if uniform access to the register file is provided. Multiple packets may be entered and/or removed into/from the network in one communication cycle.

5. Performance Evaluation

The validity of the coherency mechanism described in Section 2 depends not only on its correctness but also on its performance when implemented with current technology. Therefore, we have built a simulator (described in detail in [12]) to evaluate the performance of real programs on plausible system implementations.

The user of the simulator supplies a program in C language whose execution is simulated on each of the nodes. A library package provides functions to create shared memory and allocate it on the nodes specified by the user. When the program reads or writes data allocated in shared memory the simulator emulates the appropriate actions by the network interface and by the network. Coherence management, routing and memory access are simulated in detail. The time between memory references is computed by examining the instructions executed and computing an approximate execution time.[6] This is done during execution by instructions inserted by a preprocessor.

The network topology, its speed, the speed of the node processor and the page size can all be changed by the user. The simulator also includes memory management algorithms similar to the ones described in [6]. These algorithms attempt to replicate pages in order to improve performance; although we have implemented these algorithms in the simulator, they have been turned-off in the experiments described in this report in order to show the bare performance of the machine.

All the experiments described in this report use the hypercube topology. The performance of the machine depends on the relative performance of the processors, the memories and the network. Therefore, we have chosen values that we believe are achievable with commonly used technology: the time to transfer one 64-bit packet (all packets are 64-bit long) between two adjacent nodes is 450 ns; the main memory access time is 150 ns; the state machines that implement the memory and network interface circuits cycle at 20MHz. Unless otherwise specified, we have assumed each node contains a processor similar in performance to a 20 MHz Motorola 68020.

Two kinds of experiments have been performed. First, the machine has been stimulated with a randomly generated artificial load to establish the basic performance of its components and the point of saturation of critical resources. Secondly, since the performance depends on the characteristics of the task the machine is running, we have evaluated three real applications: the matching part of an expert system, a single-point-shortest-path algorithm and the recognition part of a speech recognition system. In increasing order, the three tasks are examples of highly data dependent and

[5]Serial access does not imply bit-serial access: there can be more than one tap into the shift register so that 4 or 8 bits can be moved in/out in parallel

[6]The execution time is approximate since the internal behavior of the processor is not simulated and changes in performance due to pipeline stalls or overlapped instructions are not taken into account.

very low granularity tasks. One of the tasks has an inner loop which is only a few microseconds long and requires in average three synchronization operations for each inner loop execution: this is an order of magnitude more demanding than what is usually considered a low-granularity task. All the tasks are components of real applications and have been tried on real data.

5.1. Artificial Load

Our objective with these experiments is to measure latency and maximum frequency of shared memory operations (reads, writes and synchronization operations). Latency can be due to many reasons:

- A read operation did not find the page in local memory and caused a remote access. The processor and cache cannot proceed until the network interface has queried one of the remote nodes where the page is stored and the value has been returned.
- A write or a synchronization operation did not find the page in local memory and caused a remote write. There is no latency for this operation unless the network interface is saturated and cannot process the request.
- A read or write operation referenced a page that was not mapped in the same node. This is similar to a uniprocessor page fault and is not considered here, since its latency is mostly a function of the operating system overhead and secondary storage access time.
- A synchronization verification or a coherent read was issued by the processor. The latency is a function of the Gray Zone width.
- A local access conflicts with an access by the network interface on behalf of some other node.

5.1.1. Read Latency vs. Replication

We have performed two sets of experiments on binary hypercubes. The first maintains a constant average frequency of read/write operations (i.e., load) while varying the replication, and the second maintains a constant replication and varies the load. This section and the following describe the results of the first experiment.

The read latency can be separated into the following components: network transit time (t_n), memory read cycle time (t_m), queuing delay in the network (d_n), and queuing delay at the memory location (d_m). If the number of network hops between the source node and the node containing the addressed page is h, then $t_n = 2ht_h$. (Two packets are needed for a read access: address and data.) The queuing delays d_n and d_m are integral multiples of t_h and t_m, respectively. Since t_h is much higher than t_m (450ns vs. 150 ns), the latency of remote read operations is dominated by t_n and d_n.

Replicating a page in one or more nodes can reduce t_n by reducing h, since h is the number of hops to the *nearest* copy of the addressed page. In the best case of complete replication, h becomes 0. This is a good solution for read-only pages if there is enough memory. However, if the page can also be written, each write operation generates extra write-update messages to update all copies of the replicated page, thus increasing the network load and the network queuing delay d_n. In addition, the Gray Zone width increases, extending the synchronization time. The extent of such behavior is dependent on both the frequency of read/write operations and the degree of replication.

In this experiment, each processing element (PE) holds a page that is replicated in a sub-cube around that PE, and each PE performs read and write operations on randomly chosen pages in the system[7]. The read and write operations are directed to the nearest copy of the addressed page; write operations then propagate to all the remaining copies. This generates a load that is evenly distributed throughout the system, without any hot-spot location. The interval between the initiation of successive operations is distributed exponentially with a mean of 2.4µs, representing a moderate to heavy network load. Read operations are three times as frequent as writes.

Figure 5-1 shows the variation of the average read latency with replication for binary hypercubes of various sizes. The simulation results are shown by the solid lines. Since the simulator models both the network and the memory system at every clock cycle, the latency figures are quite accurate.

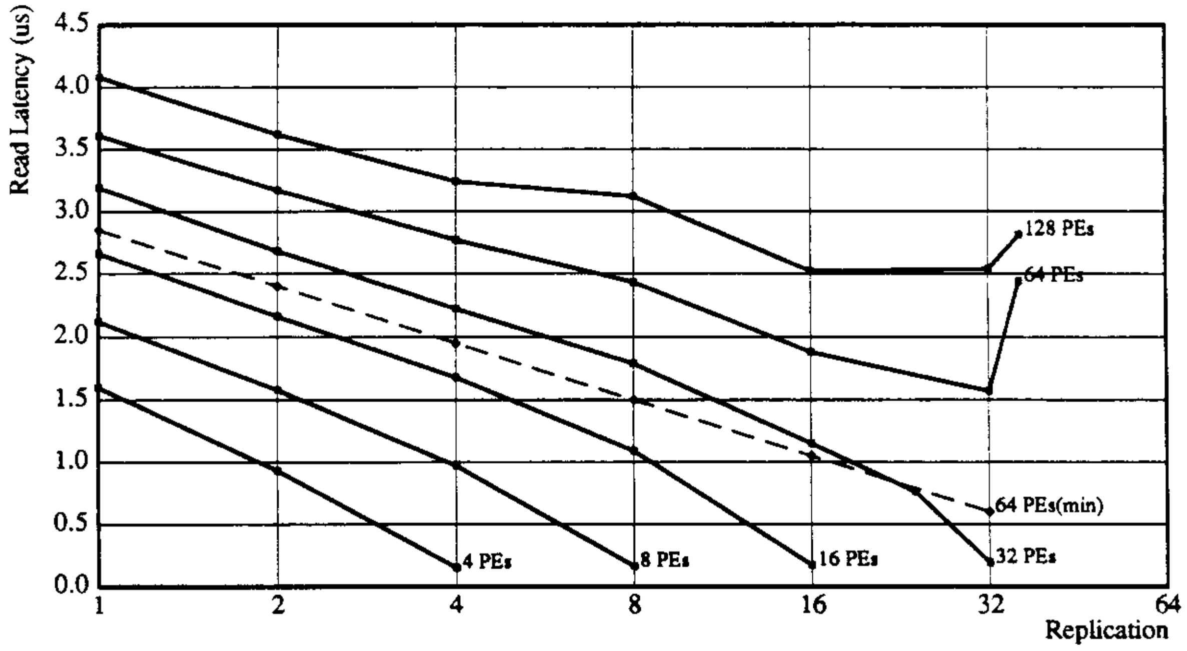

Figure 5-1: Read Latency vs. Replication, Uniformly Distributed Accesses Every 2.4µs

As expected, the read latency decreases with increasing replication, up to a certain replication factor. In this region, the latency is dominated by t_n. But with increasing replication the read latency rises, especially for larger configurations, indicating that the network queuing delay d_n is becoming significant. The lower bound on the average read latency, i.e. in the absence of any queuing delay, is given by:

$$(log_2N - log_2c)t_h + t_m$$

where N is the number of nodes, and c is the replication factor. This is obtained as follows: the average distance between the source node and the addressed page, if there is only one copy of the page, is $log_2N/2$ hops. With c copies of the page distributed in a subcube, this distance is reduced by $log_2c/2$ hops. The average read latency, in the absence of queuing delays, is the round-trip time plus the memory access time. This lower bound is shown by the dotted line in Figure 5-1 for the 64 PE case. The curve obtained through simulation shows a greater latency. Since simulation results indicate that the queuing delay for a memory cycle was negligible (less than 50 ns on the average),

[7]We are only modeling those references that miss the local PE cache, and cause a local or remote memory cycle.

the queuing delay in the network accounts almost entirely for the difference between the two curves. From the Figure, we see that this delay is only a small fraction of the total delay at low replication (about 20%), but grows to over 60% at 32-fold replication.

5.1.2. Gray Zone vs. Replication

Figure 5-2 shows the average Gray Zone width for the set of experiments described in the previous section. The Gray Zone width rises slowly with replication up to a certain point, after which it increases very sharply. This change in the slope of the curve marks the point where the network queuing delay starts becoming prominent.

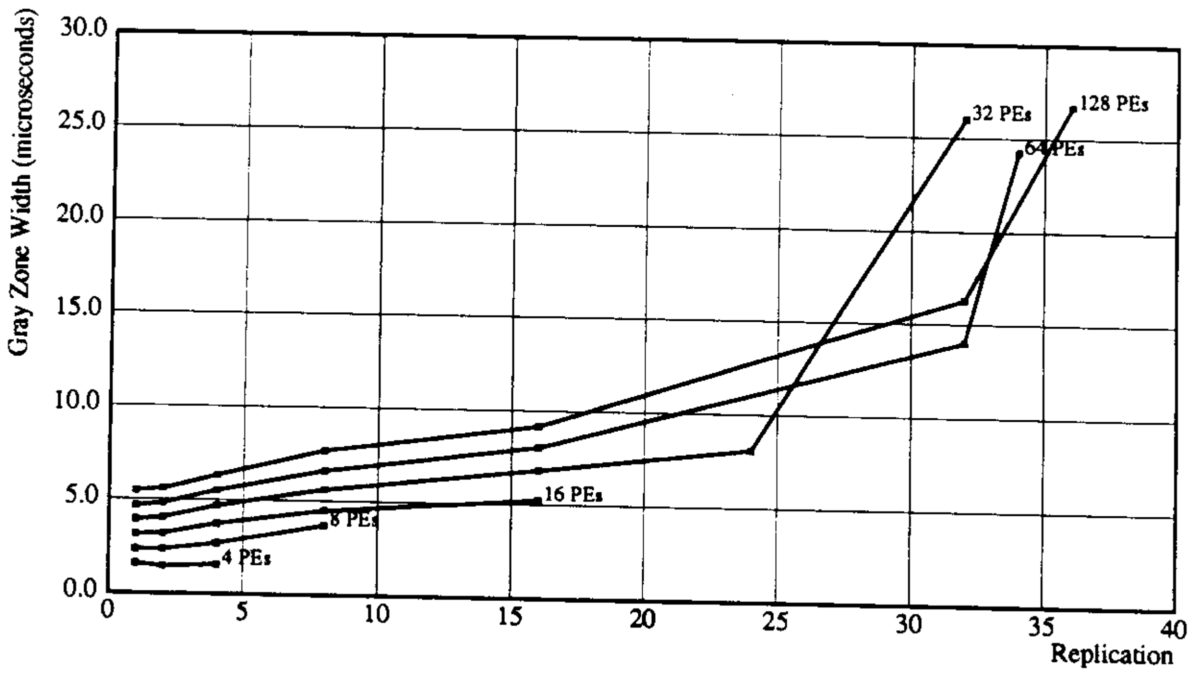

Figure 5-2: Gray Zone Width vs. Replication, Uniformly Distributed Addresses Every 2.4μs

The network delay is a function of the *offered network load*, which, in the above experiments, can be calculated as follows. The average distance between a source and the nearest copy of the addressed page is $(log_2N - log_2c)/2$. Each write operation results in $c-1$ additional update operations, requiring as many network hops. Since each operation requires the transmission of both an address and a data packet, the average number of network hops needed per PE per operation is given by:

$$n_{op} = log_2N - log_2c + 2f_w(c-1)$$

where f_w is the fraction of write operations. The offered network load per PE per network cycle is then:

$$\frac{n_{op}t_h}{\textit{period between operations}}$$

whereas the network capacity per node per network cycle is, approximately,

$$log_2N$$

These expressions provide us with guidelines for controlling replication, within the limits imposed by the assumptions made during the experiment. From these and other simulation experiments, we have noticed that the read latency and Gray Zone width behave reasonably as long as the average offered network load is within approximately 20% of the network capacity.

5.1.3. Read Latency and Gray Zone Width vs. Load

This section describes the results of another experiment in which the replication is a constant fraction of the number of PEs in the system, while the frequency of read/write operations is varied. The distribution of read/write operations over the various pages is uniform, the interval between successive operations is exponentially distributed, and the frequency of reads is three times as much as that of writes, as in the previous experiment.

Maintaining replication at a constant fraction of the number of PEs implies that the average read latency in the absence of queuing delays:

$$(log_2N - log_2c)t_h + t_m$$

is independent of N. Network queuing delays (ignoring memory queuing delays) account for most of the deviation from the latency computed with this formula. Table 5-1 shows the variation of read latency with load for binary hypercubes of various sizes. In the first part of the table, replication is kept at 1/4 the system size, and in the second at 1/2. The load is characterized by the mean interval between the initiation of successive operations.

Ops.	Int.-->	.8	1.2	1.6	2.4	3.6	4.8	9.6
Repl.	PE's							
1	4	1.60		1.62	1.60	1.58	1.57	1.51
2	8	1.68		1.60	1.58	1.59	1.52	1.48
4	16	1.74		1.74	1.68	1.66	1.63	1.63
8	32	1.89		1.81	1.79	1.72	1.66	1.65
16	64	2.10		2.04	1.88	1.80	1.74	1.72
32	128	2.94		2.92	2.54	1.98	1.87	1.78
2	4	0.99	0.95	0.97	0.93	0.91	0.89	0.96
4	8	1.07	0.98	0.95	0.97	0.92	0.96	0.94
8	16	1.21	1.18	1.11	1.09	1.03	1.02	1.01
16	32	1.57	1.50	1.28	1.15	1.08	1.05	1.02
32	64	2.62	2.60	2.50	1.57	1.19	1.14	1.05

Table 5-1: Read Latency vs. Load
(Mean Operation Interval in Microseconds)

We can see that the average read latency is fairly constant almost throughout the table, indicating a relative absence of network queuing delays. These become apparent only for large systems (when replication becomes high) operating at large loads.

Figures 5-3 and 5-4 show the Gray Zone widths for the same sets of experiments. Once again, the Gray Zone width increases fairly slowly except in the case of large systems operating at high loads.

In conclusion, page replication reduces the average read latency significantly, but only up to a certain point. Furthermore, the Gray Zone width always increases with replication, and it increases

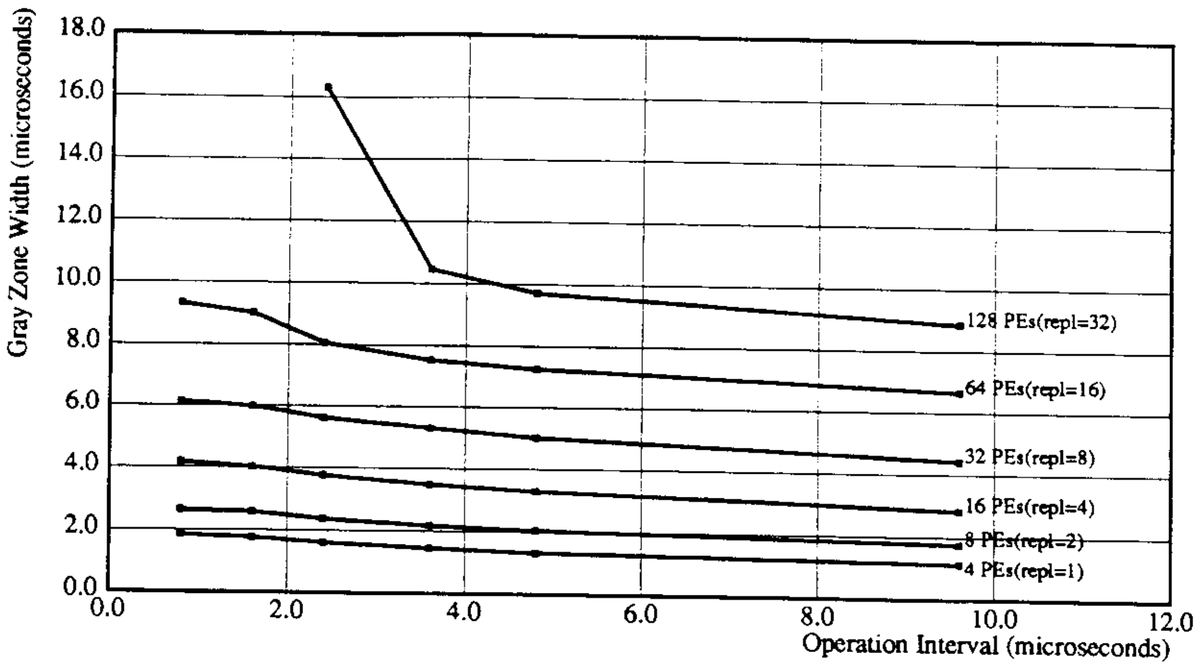

Figure 5-3: Gray Zone Width vs. Load, Replication Equal to 1/4 System Size

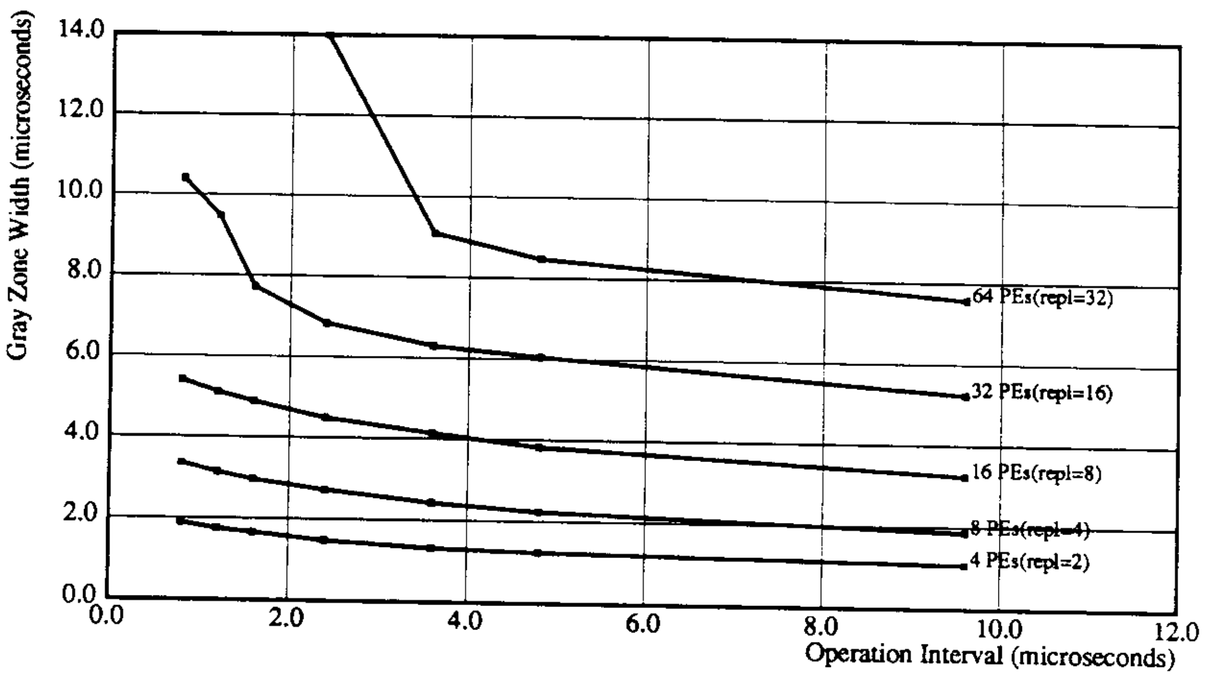

Figure 5-4: Gray Zone Width vs. Load, Replication Equal to 1/2 System Size

rapidly beyond a certain point. This implies that in order to obtain the full benefit of replication, it is important to limit the degree of data sharing among PEs, especially in large systems. For example, if certain data structures are shared between only 8 PEs in a 64 PE system, they can be replicated on all these PEs without causing the network to be overloaded. The experiments described in later sections make use of precisely this strategy.

5.1.4. Synchronization Performance

This set of simulation experiments was conducted to measure the average time taken for a *fetch-and-add-verify* synchronization operation with all PEs in the system contending for a single lock, and the effect of such a (potential) hot-spot on other background (remote read/write) operations. The results of such an experiment are dependent on several parameters: the network topology and size, the frequency of synchronization operations, the frequency of read/write operations, and the lock replication count, to name a few principal ones. The topology was restricted to binary hypercubes, and the load pattern was constrained as follows: each PE generates read and write operations directed to a randomly chosen remote PE node. In addition, it generates fetch-and-add-verify operations on the single lock variable, which may be replicated on several nodes. One write is generated every four reads and one fetch-and-add-verify is generated every 10 read or write operations. There is no delay between the completion of one operation and the start of the next; hence, this is the highest load that the system will sustain.

The system behavior under this load was determined by simulation. Figure 5-5 shows the average synchronization latency per PE vs. replication for various system configurations. The latency is measured from the time the fetch-and-add operation is issued until the time the corresponding verify operation completes.

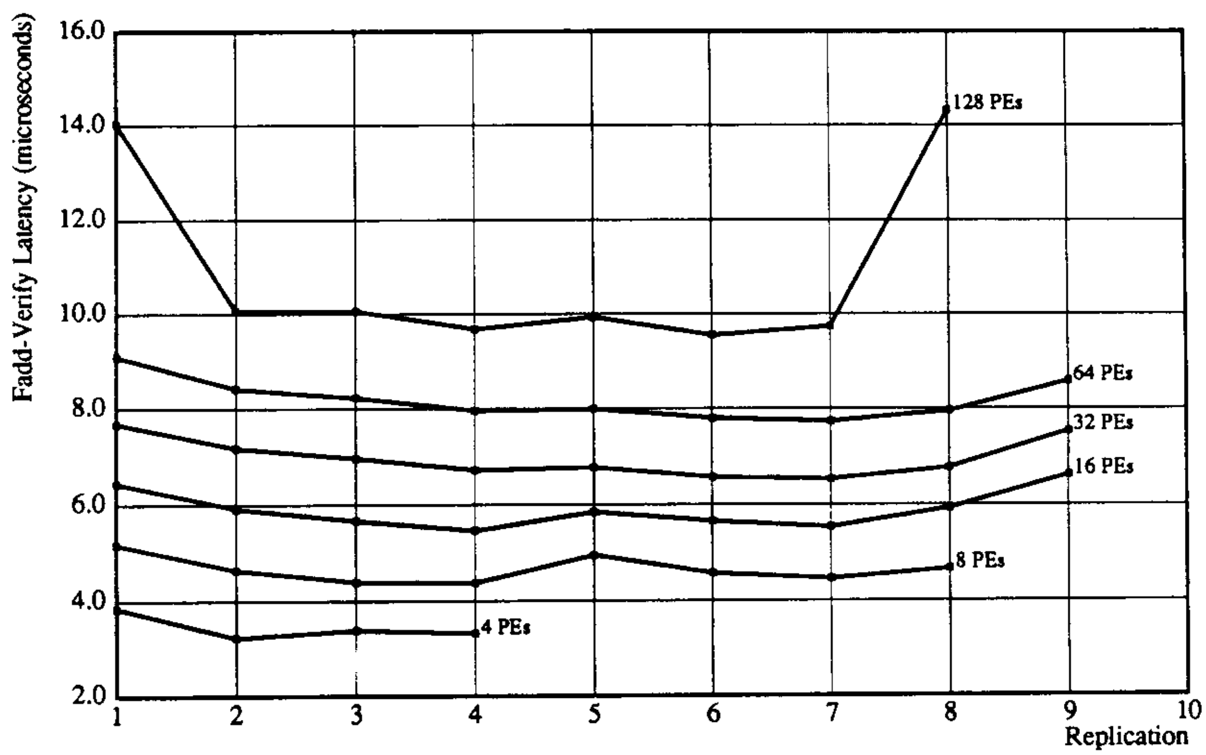

Figure 5-5: Fetch-and-add Latency vs. Replication

When the number of PEs is between 4 and 64, the synchronization latency is dictated essentially by the network performance; i.e., the time taken for synchronization requests to propagate to a node containing the lock, for the Gray Zone information to percolate through the system, and for the verified reply to return to the source PE. Every time the number of PEs is doubled, (i.e., the network dimensionality is increased by 1) the performance curve is shifted up by approximately 1.1μs. Most of this is accounted for by the extra distance the request and response packets must travel in the larger system. Specifically, the average distance to the lock increases by 0.5 hop, and four packets must travel this extra distance: address and data packets for the request, a data packet for the initial response, and one for the verified response. All of this adds up to an extra 0.9 μs.

When the number of PEs increases to 128, the memory bandwidth to the single lock becomes the bottleneck. With a cycle time of 150 ns and two cycles needed for each synchronization operation, a single lock can sustain at most 3.33 synchronization operations/μs. Table 5-2 shows the maximum number of synchronization operations per μsecond that was sustained by the lock in this experiment. Clearly, the memory module holding the lock is close to saturation in the last two cases. With 256 PEs, the fetch-and-add-verify latency is over 30μs for this kind of load.

Number of Processors	2	4	8	16	32	64	128	256
ops/μsec	0.12	0.20	0.34	0.58	1.02	1.78	2.90	2.95

Table 5-2: Peak Synchronization Rate
(Total Number of Synchronization Operations per Microsecond)

Replicating the lock decreases the fetch-and-add-verify latency up to a point. This is partly due to the shorter average distance between any given node and the nearest copy of the lock, and partly due to the increase in effective memory bandwidth. In larger configurations, where the memory bandwidth is the bottleneck, replication improves the fetch-and-add latency significantly. For instance, with 128 PEs and two copies of the lock, each copy handles only half the total number of requests. Each copy must also process update requests from the other copy, but the cost of these is one memory cycle, as opposed to two for the original requests. Hence, the memory bandwidth requirement per copy is reduced by about 33%, eliminating the bottleneck.

Replication does not, however, reduce synchronization time very significantly. In fact, beyond 8-fold replication the synchronization time begins to increase. This is in contrast to the read latency performance described in the previous experiment. One reason is that the lock is a hot-spot, unlike any given page in the previous experiment. With higher replication, the update requests that keep all copies coherent begin to flood the network, increasing both the message latency and the Gray Zone width.

With limited replication, the fetch-and-add latency is remarkably insensitive to the total load in the system, as well as to the relative ratio of synchronization operations and background traffic (provided the lock memory bandwidth is not a bottleneck). Varying the former between 20% and 100%, and the latter between 0.03 and 0.33 affects the performance only by about 15%. This demonstrates that the synchronization time consists almost entirely of the time for request and response packets to simply cover the distance between the source and the lock, with very little queuing delays.

The latency of background read operations is also quite insensitive to wide variations in the total load and increases linearly with the network dimension: from 2.18 μs with 4 PEs to 4.66 μs with 256 PEs. By varying the total load between 20% and 100% and the fraction of synchronization operations between 0.03 and 0.33 the read time is only affected by about 6%. This holds true even when the synchronization traffic is almost saturating the lock, which implies that the network is able to route read requests around the hot spot quite effectively.

5.2. Real Load

5.2.1. Production System Match

The matching process is one of the bottlenecks of production system computations. In this evaluation we have followed the model used by Gupta in [10]. The production systems class of applications can be modeled abstractly as follows. An application consists of a global hash table and a set of dynamically created tasks. The execution of each task requires access to (and modification of) one entry in the hash table and generates zero or more new tasks. The application runs as a succession of cycles; a cycle terminates when there is no task pending. In a parallel implementation, several processors can be working on independent tasks concurrently as long as there is no conflict for a hash table entry.

The average granularity of each task is between 50 and 100 μs, during which both the shared task queue and the hash table have to be updated. The size of the affected data structures is between 10 and 20 32-bit words. During this update two or more synchronization operations are needed: one for taking a task off the queue, one to lock the hash table entry for this task, and the remaining to insert new tasks into the queue. Furthermore, in each cycle the processors need additional synchronization to determine if the end of the cycle has been reached. The cost of these synchronization operations must be low in comparison with the task granularity. Finally, the available parallelism in these applications is limited by the average number of tasks active at any time; in the cases studied the number of active tasks is between 15 and 20.

In an ideal implementation, there would be a single task queue shared among all the processors (which requires that the hash table also be shared globally). This ensures the best possible load distribution among the processors and, provided the cost of global memory access is no different from that of local memory access in a uniprocessor, gives the best possible performance. In an actual implementation on our machine, however, it is not advisable to share the task queue and hash table globally, since these data structures are accessed and updated relatively frequently. The reason is the following. In order to be competitive with a uniprocessor implementation, each processor in this machine must have a local copy of all the data it needs for processing each task. Otherwise, reading 10-20 words from a remote location (at 2 μs per access *vs.* 50 ns for each local-cache access) is too costly considering the small granularity of the task. At the same time, it is not feasible to replicate all data at every processor when the number of processors is more than 8 or so, because the need to update all the copies begins to saturate the network.

An alternative solution is to partition the hash table among the processors, and maintain a separate task queue per processor, so that each processor will receive only those tasks that need access to its local hash table partition. This not only eliminates most of the remote accesses and the need for replication, but also some of the synchronization overhead: a processor no longer needs to lock its task queue to read a task and no longer has to lock the hash table entry for that task. The disadvantage of such a static allocation is the potential loss of performance due to a non-uniform distribution of tasks among the processors.

The compromise solution is to maintain a separate task queue for each small *cluster* of processors.

The cluster should be small enough so that all the data needed for that task queue can be replicated everywhere in that cluster without incurring too high a cost for the update of all copies. At the same time, the total number of queues should be small enough for the load distribution not to become too uneven.

A number of experiments were run on the simulator using traces generated from an actual production system (Rubik). Figure 5-6 shows the performance for various data allocation strategies. Separate curves are shown for an (ideal) totally-partitioned case (a separate task queue per processor), for different cluster sizes (number of processors sharing a given task queue), and for the totally shared case (a single task queue shared among all processors). The speedup values are relative to an efficient uniprocessor implementation. The experiments on a totally shared task

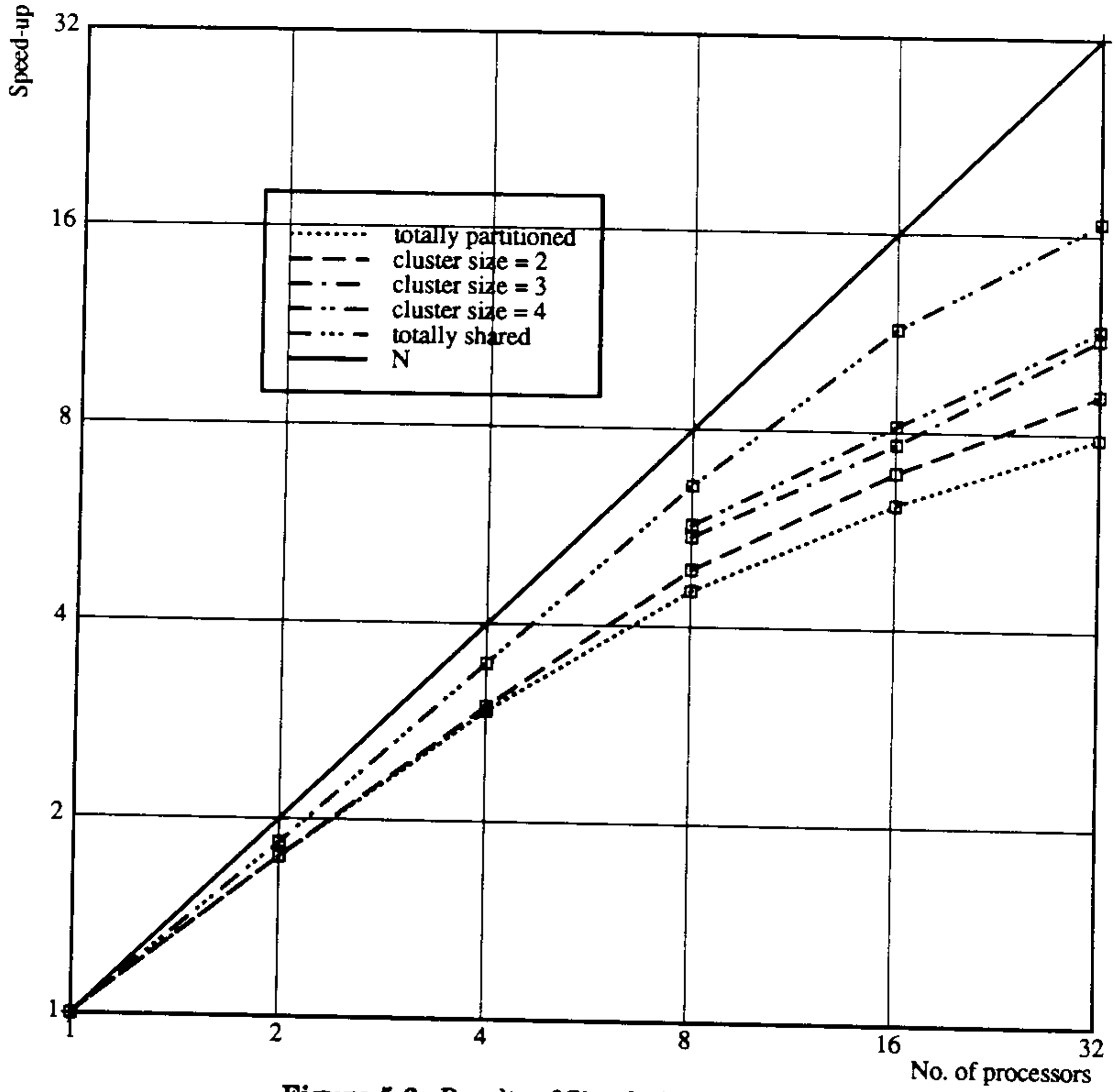

Figure 5-6: Results of Simulation of Rubik Traces

queue were conducted to place an upper bound on the available parallelism in the application. For this purpose, the extra computational cost arising from replicating the shared data structures was hidden from the simulator and the computational cost per task was made to be identical to that of an efficient uniprocessor implementation. Since the hash table was hidden from the simulator, we did not account for the time required for hash table conflicts. As a result, the performance figures of the totally shared case are overestimated. On the other hand, the synchronization cost was retained in the simulation and it does show up as an overhead in the performance figures.

The differences in the above performance curves can be understood by looking at the breakdown of the total processing time per processor, which consists of three major components: the actual computation time, the synchronization overhead, and the idle time waiting on an empty task queue. Table 5-3 summarizes these.

Number of Processors		2	4	8	16	32
totally shared (ideal)	compute	106.2	53.1	26.6	13.3	6.6
	sync	10.4	8.4	5.6	3.6	2.2
	idle	0.1	0.2	0.6	2.0	4.1
totally partitioned	compute	106.2	53.1	26.6	13.3	6.6
	sync	9.4	6.7	4.4	2.7	1.6
	idle	3.0	11.0	14.8	17.8	18.5
clustered	compute	109.4	53.6	27.2	13.8	7.0
	sync	11.0	7.2	5.4	3.1	1.8
	idle	1.2	9.7	3.9	8.8	9.6
	size	2	2	4	4	4

Table 5-3: Peak Synchronization Rate

The biggest factor determining the shape of the performance curve is the idle time. The totally shared case (with a single global task queue) has the best load distribution, and hence has the least idle time. The totally partitioned configuration has a separate queue per processor, which leads to a much worse load distribution among processors (i.e., a processor remains idle if its input task queue is empty, even though there may be tasks pending in other queues). Consequently, it has a much higher idle time. The clustered configuration reaches a better load distribution (and lower idle time) as expected, although these results are still inferior to the ideal case of a single task queue. The synchronization overhead is fairly steady between 10 and 20% of the computation time.

Since these simulations were trace driven, we have not been able to take advantage of an important optimization feature of this architecture. This is its ability to initiate and complete synchronization operations in two phases, while carrying on other computation in between. From the above table, we can see that the synchronization overhead is between 10 and 20%. Therefore, this application can potentially run that much faster than the above graph and table indicate.

5.2.2. Single Point Shortest Path

The Single Point Shortest Path problem is a good example of a problem requiring many synchronization operations. The problem involves finding the minimum cost to traverse a graph from one vertex to any other vertex. Both sequential and concurrent algorithms for this problem work by propagating the distance cost from one vertex and updating it until no more updates are possible (see Dally's thesis [8] for a description of the algorithms). Each arc contains the cost to traverse it and each vertex contains the cost to reach it from the starting point. The former is set at

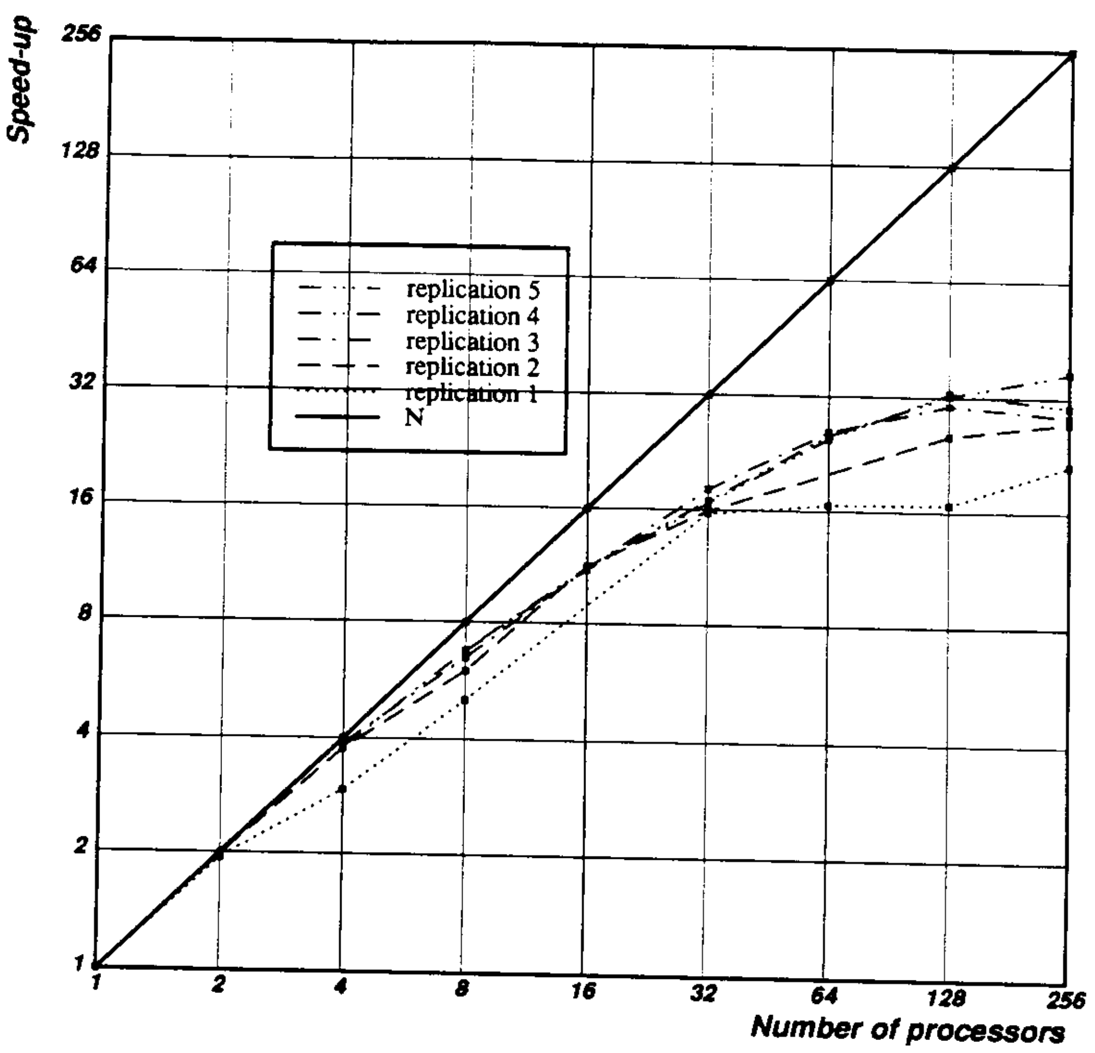

Figure 5-7: Speed-up for the Single Point Shortest Path Problem

initialization time and the latter is dynamically computed.

Number of Processors	2	4	8	16	32	64	128	256
best case (some replication)	99.8	99.53	97.34	91.62	83.37	66.89	51.62	32.62
worst case (no replication)	97.4	76.34	70.16	72.3	71.55	41.68	23.00	16.02

Table 5-4: Average Processor Utilization.

The basic step of a concurrent implementation involves choosing a vertex and computing the cost of moving to each of its neighbors. If the new cost is better than the cost stored in the vertex, the cost is updated and the vertex is queued for further expansion. When there are no more vertices to expand, the algorithm terminates. Each step requires three kinds of synchronization operations: extracting a vertex from the queue, locking a vertex in order to update its cost atomically and inserting a vertex into the queue. We used a fetch-and-add primitive to implement the queue and a semaphore to lock each vertex. Each step takes about 80 μs of processing time (if all synchronization-related instructions are not counted) and requires an average of two fetch-and-add and four fetch-and-set operations.

The implementation uses multiple queues since a single queue introduces serialization and requires long remote accesses. The vertices are evenly distributed among the nodes and there is one queue on each node. If a processor extracts work only from its local queue, it is possible for some processors to remain idle for part of the time, especially if the ratio of number of vertices to the number of processors is low. The shared memory model and the possibility of replicating data are very helpful in this case. We have replicated queues and vertices on more than one processor and found a substantial performance increase due to better load balancing. In this case, a processor looks at each of the queues starting with the queues that are replicated locally.

Figure 5-7 shows the speed-up of the average time to process one vertex for different levels of replication. The speed-up drop for 256 processors is mainly due to the fact that the graph has too few nodes (1024) compared to the number of processors; often the sum of the number of the vertices in the queues is less than 256 and it is impossible to rebalance the load. The task size creates a load balancing problem with more than 32 processors. This is indicated by the utilization (ratio of the average processor idle time to the elapsed time) shown in Table 5-4. With no replication, the utilization decreases substantially when more than 32 processors are used. When replication is used, utilization and speed-up remain reasonable up to 128 processors.

The network and the coherency mechanism are much less loaded than in the case of the artificial-load experiments described in the previous sections: the average read time is about 200 ns and the average synchronization latency is less than 2.5 μseconds in the case of 256 processors. The synchronization latency experienced by the program is less than 2.5 μseconds since some of the locking operations are partially overlapped with useful computation.

5.2.3. Beam Search

The SPHINX speech recognition system (Kai-Fu Lee et. al. [11]) is a state-of-the-art speaker-independent continuous speech recognition system which currently achieves in excess of 95% accuracy on a 1000 word vocabulary with loosely constrained grammars. The system has two components: a signal processing component that can be executed in real time (real time is the length of the utterance to be recognized) by a signal processing processor like the Texas Instruments TMS32030, and a beam search [4] component that requires about 7 times real time on a Sun 4/260 for a medium-difficulty task. The search component is a good example of a real-world non-numeric application since it exhibits the following characteristics:

1. No floating point arithmetic or integer multiplications; no more than 25% of the instructions are arithmetic operations.
2. Negligible amount of I/O (about .03 bytes per instruction).
3. Control-flow depends on the input data (speech input).
4. Poor locality of data reference.

5. Memory aliasing which creates dependencies that cannot be avoided by compilation analysis.

The inner loop of this algorithm can be coded in about 70 RISC instructions and requires about 10 memory references per iteration. A medium-complexity 1,000 word task requires the execution of about 10^6 inner loop iterations per second. Larger lexicons and more complex grammars require substantially more computation.

The beam search algorithm of SPHINX searches a Hidden Markov Model representation of the speech process and returns the most likely sequence of words. Beam search requires a very fine-grain parallel decomposition and a substantial amount of synchronization. Typically, a processor must dequeue one Markov state from the list of states to be processed, lock all the states that follow it and finally queue a new state.

Queuing (dequeuing) states in (from) a central queue causes too much serialization. The solution is to split the queue into local queues, one for each processor, so that queue accesses can proceed in parallel. In this case, because of the highly data-dependent behavior of beam search, it is likely that some queues will become empty before others and some processors will remain idle and create a load imbalance. This load imbalance can be limited by associating each queue with a few processors instead of one.

The algorithm requires a large memory bandwidth that cannot be fully obtained by means of caches since the search proceeds in many different parts of the graph at the same time and the locality of data references is limited.

We have implemented the search algorithm on different machines and on the simulator. The same C code has been used for all existing machines. Because of memory limitations (the algorithm requires about 8 Mbytes of memory) we have driven the simulator with a detailed trace collected from one of the C-code programs instead of the program itself. Both execution time and memory references were accurately modeled and we have validated the simulator version by comparing the execution time of a real machine with the execution time on a single node of the simulated system.

Figure 5-8 compares the performance of a few commercial machines with the simulated performance of our multiprocessor. We have chosen the speed of the processors so that the speed of a single processor system is similar to the speed of a single processor Encore Multimax. The absolute performance of such a one-processor system is about 27 times real time (real-time is the length of the utterance). The line labeled BEAM shows the performance of a shared-memory accelerator called BEAM [5] that uses three 10-MIPS Weitek-8032 and is currently the fastest machine running SPHINX. The Figure also shows the relative speed of a Sun 4/260 and a Sun 3/60. In comparing the values one should remember that the performance of the algorithm depends not only on the speed of the processor but also on the performance of the memory system, e.g. on the size of the cache.

The speed of the bus-based Encore peaks at about 12 processors, but there are no substantial speed improvements with more than 8 processors because the combination bus/shared memory system saturates. The distributed-memory multiprocessor has close-to-linear speed-up until 16 processors and then becomes less effective because of the synchronization overhead. One should also notice that the performance is as good as or better than the performance of the bus-based machine

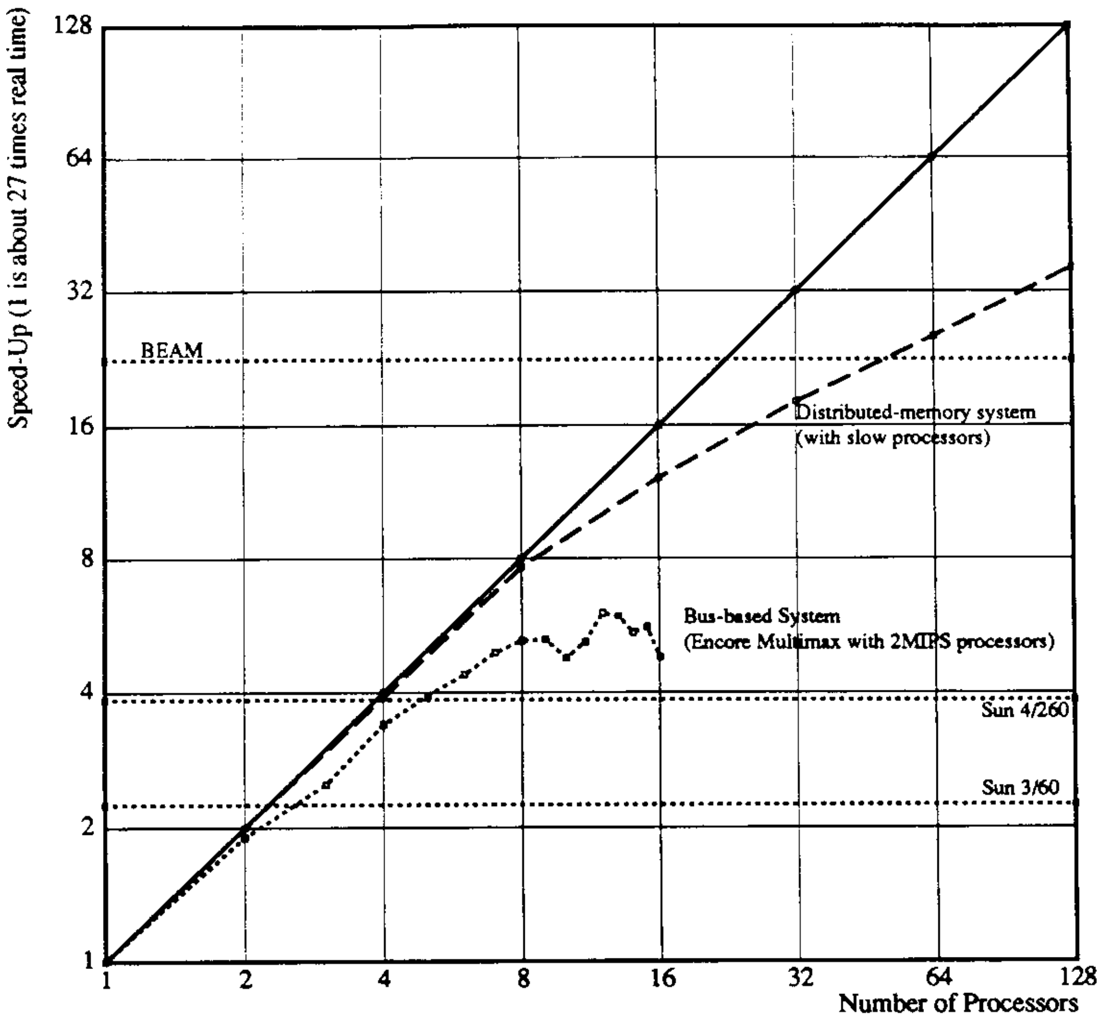

Figure 5-8: Beam Search Speed-up Relative to the Performance of a One-processor Encore Multimax.

even in the range in which the latter performs well. This indicates that a distributed memory architecture can be advantageous even with a small number of processors.

Figure 5-9 shows the simulated performance with a much faster processor. The single processor speed has been calibrated to be about 1.5 times the speed of a single Weitek 8032 processor as it was used in the BEAM accelerator [5]. We estimated that this speed is a conservative approximation of the speed attainable by a Motorola 88000 running at 20MHz. The performance of the single processor system is 1.83 times real time. In comparing Figure 5-8 to Figure 5-9 one should bear in mind that there is a factor of 15 performance difference between the single processor case in the two Figures. The inner loop of the search takes less than 4 μseconds and three synchronizations are necessary for each iteration.

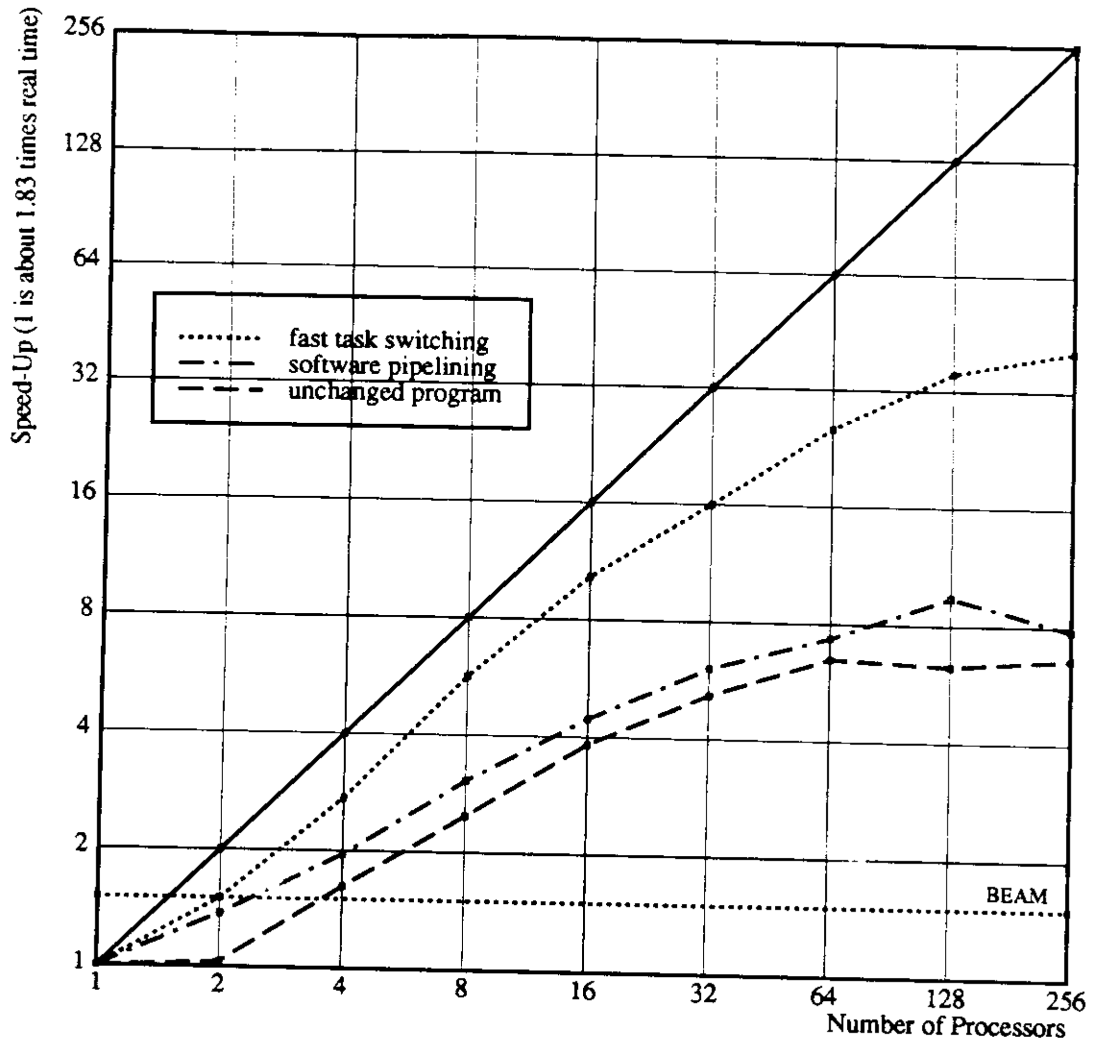

Figure 5-9: Beam Search Speed-up Relative to the Performance of the Best Sequential Implementation.

The dashed curve shows the performance with exactly the same program that was used to derive the measurements of Figure 5-9: the speed-up starts deteriorating after 32 processors. The dot-dash curve shows the performance of a different version of the program in which part of the synchronization latency has been overlapped with computation. For example, the program assumes that lock operations are always successful and performs the instructions that depend on the lock without waiting for the synchronization outcome. Only at the end of this processing it checks the outcome of the lock operation and repeats the computation if necessary. The absolute performance is improved because part of the synchronization delays are hidden, but the slope of the curve is similar to the slope of the previous case.

The dotted line shows the performance of a very different version of the same program: the loop is programmed as three independent tasks that are associated with the completion of a synchronization operation. Each of the tasks can be repeatedly executed and can issue a number of synchronization operations which, when completed, will trigger the execution of other tasks. Switching to a different task instead of waiting for the outcome of a synchronization is very useful since it increases utilization. If the overhead of switching is too high, though, the advantage is lost. In programming this specific program we have been able to keep the overhead within 10 instructions per context switch. Although we believe that a general purpose package with similar performance could be programmed, we have not done it. Since the slope of the curve is dominated by the increasing cost of synchronization, better speed-up is achieved in this last case because a larger percentage of the synchronization latency is overlapped with computation.

6. Summary

We have argued that a form of coherence (weak coherence) that is less restrictive than the one supported on current shared memory multiprocessors is sufficient and easier to implement on large multiprocessors. We have presented a new technique to support weak coherence in multiprocessors that do not have a physical shared memory. The technique is based on keeping a dynamic approximation of the timestamp of the oldest message that is still in transit.

We have simulated a system that implements our coherence mechanism at a very detailed level and found that the system can be more scalable than a bus based system using the same technology. Moreover, we found that replicating writable data can improve load balancing and therefore improve performance.

One of the experiments with a fine-grain real application has also shown that a distributed-memory multiprocessor of this kind is competitive with a shared-memory system even if only a few processors are used.

Acknowledgments

We would like to thank Duane Adams, Lawrence Butcher, Siddhartha Chatterjee, Duane Northcutt and Raj Reddy for comments on the manuscript and helpful discussions.

References

[1] Agarwal,A., Simoni,R., Hennessy,J. and Horowitz,M.
An Evaluation of Directory Schemes for Cache Coherence.
In *15th ISCA*, pages 280-289. IEEE, May, 1988.

[2] Archibald,J. and Baer,J.L.
An Economical Solution to the Cache Coherence Problem.
In *12th ISCA*, pages 355-362. IEEE, June, 1985.

[3] Beetem, J., Denneau, M. and Weingarten, D.
The GF11 Supercomputer.
In *12th Ann. Intl. Symp. on Computer Architecture*, pages 108-115. IEEE Computer Society, June, 1985.

[4] Bisiani,R.
Beam Search.
Encyclopedia of Artificial Intelligence.
John Wiley & Sons, 1987.

[5] Bisiani, R.
BEAM: An Accelerator for Speech Recognition.
In *IEEE International Conference on Acoustics, Speech and Signal Processing*, pages . May, 1989.

[6] Black,D.L., Gupta,A. and Weber,W.
Competitive Management of Distributed Shared Memory.
In *Compcon '88.* IEEE, Spring, 1988.

[7] Crowther, W., Goodhue, J., Starr, E., Thomas, R., Milliken, W., Blackadar, T.
Performance Measurements on a 128-node Butterfly Parallel Processor.
In *Parallel Processing Conference.* IEEE, 1985.

[8] Dally,W.J.
A VLSI Architecture for Concurrent Data Structures.
PhD thesis, California Institute of Technology, 1986.

[9] Dubois,M., Scheurich,C. and Briggs,F.
Memory Access Buffering in Multiprocessors.
In *13th ISCA*, pages 434,442. IEEE, June, 1986.

[10] Gupta, A., Forgy, C. L., Kalp, D., Newell, A., and Tambe, M.
Parallel OPS5 on the Encore Multimax.
In *Proceedings of the International Conference on Parallel Processing*, pages 271-280. August, 1988.

[11] Lee,K.F.
Large-Vocabulary Speaker-Dependent Continuous Recognition: The SPHINX System.
PhD thesis, Carnegie-Mellon, 1988.

[12] Nowatzyk,A.
Performance Analysis of Hypercube Based Ensemble Machine Architectures.
PhD thesis, Carnegie-Mellon, In preparation, 1989.

[13] Palmer, J. F.
A VLSI Parallel Supercomputer.
In *Hypercube Multiprocessors* , pages 19-26. SIAM, 1986.

[14] Pfister, G. F. et al.
The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture.
In *Proc. of Intl. Conf. on Parallel Processing*, pages 764-771. IEEE Computer Society, August, 1985.

[15] Russell, R. M.
The CRAY-1 Computer System.
Comm. ACM 21(1):63-72, January, 1978.