INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

A Replicated Concurrent-Read Architecture for Scalable Shared-Memory Multiprocessing

by

Bahman Shahir Motlagh
B.S., Istanbul Academy of Sciences, 1977
M.S.Cp.E., University of Central Florida, 1993

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy in Computer Engineering
in the Department of Electrical and Computer Engineering
in the College of Engineering
at the University of Central Florida
Orlando, Florida

Spring Term
1997

Major Professor: Ronald F. DeMara

Abstract

A Replicated Concurrent-Read (RCR) architecture was developed which is cost-effective, yet adequately scalable. First, the role of the common memory space is re-evaluated from the viewpoint of actual multiprocessor memory reference characteristics. Second, the most frequent memory operations are optimized based on the availability of inexpensive storage technologies. Third, aggregrate storage requirements are minimized by devising a *spatial caching* technique by replicating only the current working set. The resulting design leverages reference behavior and component expense by using broadcasting to update replicated memories in $O(1)$ time while allowing read references to be performed locally without delay. Thus, W simultaneous writes require $\lceil \frac{W}{d-1} \rceil$ memory cycles using d-port memory components. However, read bandwidth of a full N words/cycle is obtained for N processors.

Analytical models were developed and simulations of memory latency were performed for Uniform Memory Access (UMA), Non-Uniform Memory Access (NUMA), Local-Remote-Global (LRG), and RCR architectures for hit rates from 0.1 to 0.9 in steps of 0.1, memory access times of 10 nsec to 100 nsec, proportions of read/write access from 0.01 to 0.1, and block sizes of 8 to 64 words. The RCR architecture provides favorable performance over UMA and NUMA architectures for all ranges of application and system parameters. RCR outperforms LRG architectures when the hit rates of the processor cache exceed 80% and replicated memory exceed 25%. Thus, inclusion of a small replicated memory at each processor significantly reduces expected access time since all replicated memory hits become independent of global traffic. For configurations of up to 32 processors, results show that latency is further reduced by distinguishing burst-mode transfers between isolated memory accesses and those which are incrementally outside the working set.

To my mother

Acknowledgements

I would like to take this opportunity to express my thanks to my advisor, Dr. Ronald F. DeMara, for his continuous guidance, encouragement and endless support. I also want to thank Dr. Brian Petrasko, who along with serving as a member of my advisory committee, guided me through my masters degree program with profound wisdom and direction. Deepest thanks also to the other members of my oral examination committee, especially Dr. Darrell Linton and Dr. Chris Bauer, who have offered me direction and encouragement when I needed it most. Thanks also to Dr. N. Lobo for his time spent on my advisory committee.

Many thanks also to Concurrent Computers, Inc. for their support of my research. A special acknowledgement goes to Dr. Al Romagosa who has been especially supportive of my work.

I want to sincerely thank my dear friends, Simin, Bardia, Soheil, Bahram, Mehdi, Majid and Ata, for their friendship, kindness and moral support throughout the years. Deepest thanks also to my friend and colleague, Ali, who has been a constant source of companionship and support.

I am, and always have been, grateful to my mother whose love and unconditional support has been a great source of encouragement and joy. To my sisters and relatives, Mina, Lila, Sima and her husband Scott, Behrooz, and Nima, my grateful thanks for years of unwaivering support in furthering my growth and education. Finally, I would like to express my deepest gratitude to my dear friend, Michele, for her friendship and wisdom.

Contents

List of Figures

List of Tables

1 Introduction

Rapid changes in the cost and density of semiconductor memory technology have made the previously preferred multiprocessor design approaches obsolete. In particular, traditional interconnection strategies between multiple processors and a common memory regard the storage space as a very scarce resource. These conventional approaches restrict scalability by requiring latency to transfer data whenever and wherever remote memory references occur. Previous designs have addressed this problem by including local caches, multistage combining networks, and elaborate referencing schemes, but require sophisticated hardware to maintain coherence between the physically-distinct memories.

We present a novel multiprocessor architecture which is cost-effective, yet sufficiently scalable. Our approach involves a complete re-evaluation of the common memory space based on actual multiprocessor memory reference behavior and the availability of inexpensive memory devices. Our technique leverages these characteristics by broadcasting memory updates in constant-time while allowing read references to be performed locally with zero access latency.

1.1 Overview of Parallel Computer Architectures

Historically, digital computer systems containing one or more processors have been classified into four categories according to the number of simultaneous instructions performed and data items which are operated on concurrently [FLYNN66]:

1. Single Instruction stream, Single Data stream (SISD),
2. Single Instruction stream, Multiple Data stream (SIMD),

3. Multiple Instruction stream, Single Data stream (MISD),

4. Multiple Instruction stream, Multiple Data stream (MIMD).

SIMD, MISD and MIMD machines are considered parallel machines, but since SIMD and MISD classes are mainly suitable for special purpose computations, MIMD machines are more popular.

MIMD machines, operate on multiple instruction streams and multiple data sets as shown in Figure 1. Every processor (P_i) is capable of fetching its own instruction streams (IS) and required data (DS) from shared memory and execute the program. Figure 1 illustrates the general architecture of a MIMD multiprocessor. We can divide this class of computers into two major groups:

1. *Shared-memory* multiprocessors

2. *Message-passing* multicomputers

The basic difference between these two groups lies in their memory architecture and interprocessor communication mechanisms. The processors in shared-memory systems communicate with each other through shared variables stored in a common memory space. If at least some physical memory exists which can be accessed by more than one processor then the system is classified as a shared-memory machine, regardless of the other communication facilities which may be provided. This characteristic is the rationale for referring to shared-memory multiprocessors as *tightly-coupled systems.*

In a *message-passing multicomputer*, communication is done by exchanging messages among the computer nodes. Each computer node (PE_i) has only private *local memory* which is not shared with other processors, a control unit (CU_i), a private local memory (LM_i), and perhaps even attached disks or I/O peripherals. Since total memory space is contibuted among all memory modules, processors do not have equal access time to a given word in memory. Figure 2 shows a typical architecture of message-passing multicomputers or *Loosely-Coupled Systems.*

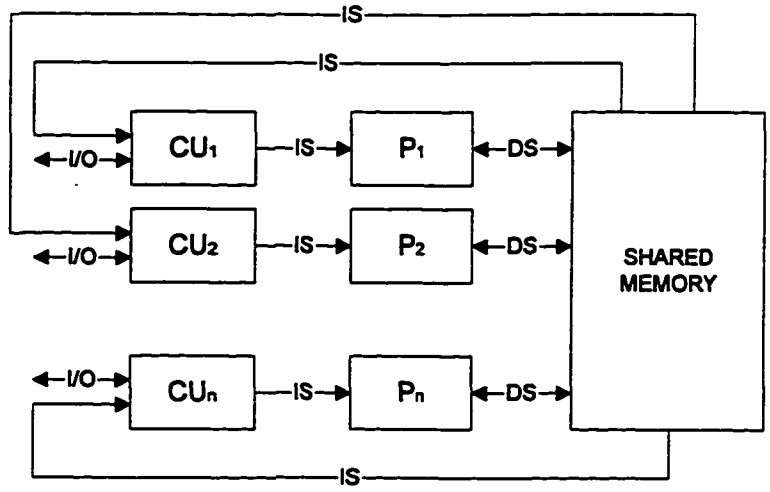

Figure 1: MIMD architecture.

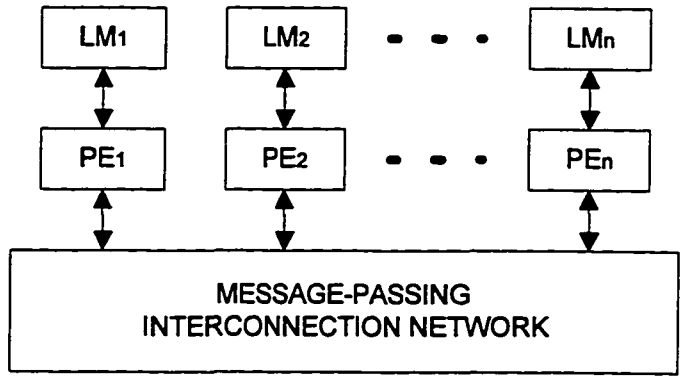

Figure 2: Message passing multicomputer.

A significant limitation of message passing is the transfer of large amounts of data between processes which require a number of message exchanges. This limitation not only degrades the system throughput, but also requires sophisticated processor interconnection. On the other hand, shared-memory can be considered a more flexible means of interprocessor communication, due to the ease at which communication of results generated by one task need not specify the destination of the recipients [KESSLER89]. Thus, the architectural trend for future scalable general-purpose computers is toward MIMD configurations with distributed memories as in message-

passing machines, yet providing a globally-shared virtual address space [HWANG93]. With respect to design of a shared-memory multiprocessor, the primary challenge is to avoid contention during access to the common memory space.

1.2 Desirable Multiprocessor Characteristics

For many years, designing a scalable, affordable, and programmable parallel computer which satisfies the needs of sophisticated computation problems has been an illusive goal. The majority of existing programs are still written in sequential languages. Their most direct and efficient conversion to a parallel form is via the *shared-memory programming model.* Consequently, there is a need to design architectural support for this model that possesses a reasonable balance between cost and performance. The design criteria should include the following principles:

1. The design should be simple and inexpensive to build,
2. Coherence among the distributed memories must be maintained, and
3. The design should be optimized for typical memory reference characteristics.

The first criteria can be met in part by using multiple physically-distinct memory units organized hierarchically. By adding small, fast cache memories, a considerable performance improvement is obtained for a negligible increase in system cost.

However, in the memory hierarchy of a multiprocessor computing system, data inconsistency may occur between adjacent levels of different processors or even within the same level of the memory hierarchy. As far as the second criteria is concerned, the main memory and caches may contain inconsistent copies of the same data, refered to as the *cache coherency problem.* In particular, multiple caches may contain different copies of the same memory block. This coherence problem generally arises during asynchronous and independent operations of multiprocessors having physically-distributed memories.

While the third criteria can have a very significant impact on system performance, it has essentially been neglected for many years because multiprocessor READ and WRITE latencies were identical, as in uniprocessors. In practice however, their difference in frequency of occurance turns out to be the key to solving the multiprocessor design problem. Specifically, READ operations on a uniprocessor typically constitute 90% of all memory references leaving only 10% of accesses as WRITE operations. Moreover, when the transition is made to a multiprocessor environment, this effect is further amplified by the nature of sharing which provides the opportunity to allow data written by *at most* one processor, to be read by *at least* one processor. In general, the data will be read by more than one processor.

1.3 Cost-Effective Design Criteria

Cost-effectiveness of an architecture concerns the relative benefits of tangible and intangible performance characteristics. The tangible characteristics of cost-effectiveness can be expressed in terms of MIPS obtained per dollar. For instance, many applications can be executed on an ensemble of slow processors more cost-effectively than on a single extremely fast uniprocessor.

Within the shared-memory multiprocessor domain, the best overall system price-performance will be determined primarily by the memory architecture. This is because powerful microprocessors have become off-the-shelf commodities. Thus, the memory system design is at the forefront of importance to obtaining a cost-effective system design.

1.4 Outline of Dissertation

This dissertation presents the memory system design for a novel shared-memory multiprocessor system. The *Replicated Concurrent-Read* (RCR) architecture takes advantage of a distributed multiported memory organization. It avoids coherence problems

by performing READ operations locally while broadcasting all memory updates. By design, this system places greater emphasis on optimization of READ memory references. The RCR memory system eliminates contention by providing independent local READ buses to every processor. This results in zero overhead for frequent READ references, at the expense of potentially slower yet typically infrequent WRITE access.

Chapter 2 investigates shared-memory multiprocessor organizations and their characteristics. In this chapter, currently existing shared-memory models are described and characterized. Diverse interconnection networks are examined and their advantages and disadvantages are also reviewed. We study the locality of memory references, and how private memories could improve overall average memory access time by taking advantage of this phenomenon. Finally, memory consistency models are discussed.

Chapter 3 describes a representative sample of previous scalable architectures. The Stanford DASH and Kendall Square Research (KSR-1) are covered as examples of directory-based schemes. The Harris Nighthawk is presented as an example of the snooping-based approach. Advantages and disadvantages of directories and snooping buses are discussed.

In Chapter 4, we analyze memory access behavior in shared-memory multiprocessors. We categorize memory references into global vs. local memory references, and READ vs. WRITE memory accesses.

Chapter 5 derives and evaluates the analytical model of RCR architecture. We define the RCR memory model, and discuss statistical distributions. This chapter estimates the probability of read and write occurrences.

Chapter 6 presents the hardware design of the RCR shared-memory system.

Chapter 7 describes the development of a software simulator for the RCR architecture. This chapter illustrates the result of simulations, validates the analytical models, and provides new insight into components of expected access time.

Chapter 8 discusses overall conclusions, and outlines topics for possible future work.

2 Shared-Memory Organizations and Their Characteristics

2.1 Overview

Distributed shared-memory multiprocessors communicate using mutually accessible stored data structures called *shared variables*, with READs and WRITEs of multiple CPUs capable of accessing the shared data. The memory design objective is to match the *effective memory bandwidth* with the peak processor throughput, so that the maximum demand for memory words can be satisfied. Ideally, the memory bandwidth would match the transfer rate demanded by each processor even after coherence and contention are taken into account.

The basic characteristics of shared-memory multiprocessors can be studied under two separate, though inter-related, organizations. The first takes a logical viewpoint called the *conceptual organization*, while the other deals with the *physical design* of the shared-memory system. Moreover, to design an efficient shared-memory system, it is necessary to study and quantify the behavior of memory modules as observed from the processor's point of view.

2.2 Conceptual Organizations for Multiprocessor Architectures

There are three primary conceptual organizations for shared-memory multiprocessors:

1. *Uniform Memory Access model,*
2. *Non-Uniform Memory Access model, and*
3. *Cache Only Memory Architecture.*

Each model offers distinct performance and design advantages, as discussed below.

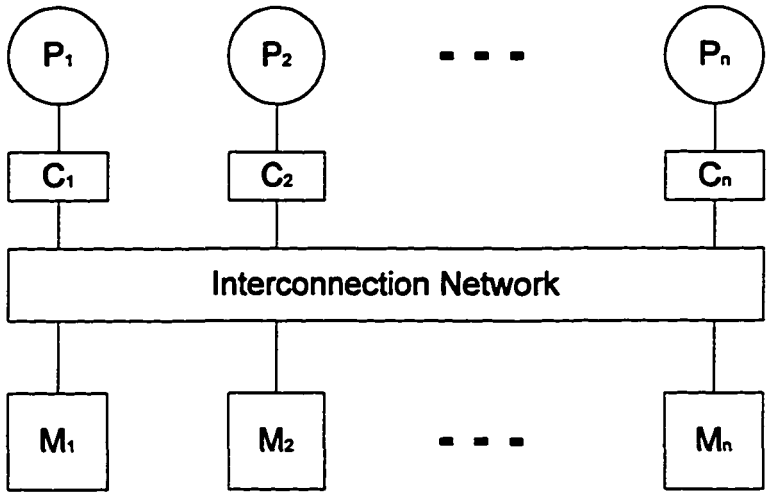

Figure 3: The UMA Multiprocessor Model.

2.2.1 Uniform Memory Access Model

In the Uniform Memory Access (*UMA*) multiprocessor model shown in Figure 3, all processors, P_i, experience an equal *expected access time* to all shared-memory modules, M_j, for all $1 \leq i, j \leq N$. In this model, each processor, P_i, is attached to a private cache, C_i, so that it can take advantage of *locality of reference* of the data and/or instructions which are currently used. All Processors may access their private cache in t_c time with hit rate h. In case of a miss, P_i may have to compete with other processors to access global memory. The waiting time to access shared-memory is dependent on the number of pending memory references and their characteristics. Processing elements also share peripheral devices in some fashion. When all processors equally share all peripherals, the system is called a *symmetric* multiprocessor. In such a system, every processor is capable of running operating system executives or I/O service routines.

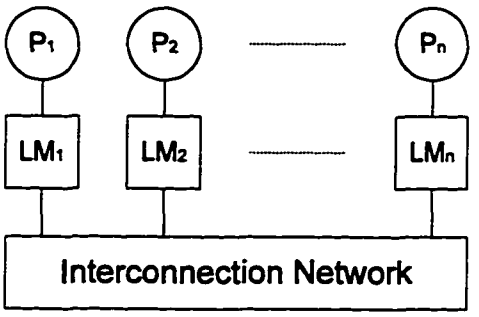

Figure 4: The NUMA Multiprocessor Model.

2.2.2 Non-Uniform Memory Access Model

In a Non-Uniform Memory Access (*NUMA*) multiprocessor model shown in Figure 4, The shared-memory is physically divided into smaller regions and each is assigned to a processor (P_i) forming an ensemble of local memories, LM_i, where $1 \leq i \leq N$. Together, these local memories (LM_i) comprise the global address space accessible to all processors. In a *NUMA* multoprocessor model the processor's access time to main memory varies with the location of the memory word within the global space. Every P_i is attached to a private cache, C_i. In case of a miss, then P_i may have to access local or remote memories. In this shared-memory model, access time of a local processor to a local memory is less than the same processor's access time to any remote memory. The increase in access time results from the added delay and possible contention within the interconnection network.

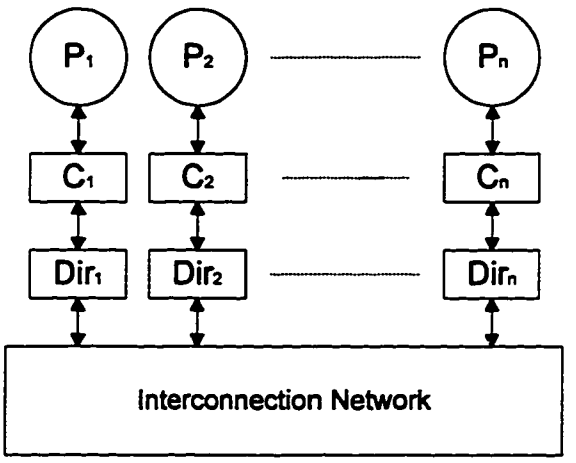

Figure 5: The COMA Multiprocessor Model.

2.2.3 Cache-Only Memory Model

In the Cache-Only Memory Model (*COMA*) multiprocessor model, shown in Figure 5, each processor (P_i) has its own cache (C_i) and it is the collection of these fast memories which enables them to form a global address space. In a *COMA* machine there is no memory hierarchy nor access by any processor to a remote memory device. We can consider a *COMA* model a special case of *NUMA*, in which the distributed main memories are converted into caches. Data which resides in remote caches is forwarded to local caches upon a miss from the local cache. Typically, distributed cache directories (Dir_i) are used to assist accessing the remote caches.

Besides these primary models, there are other models which can usually be considered some combination of *UMA*, *NUMA*, and *COMA* machines. Figure 6 exhibits a model which has combined the *NUMA* with a *UMA* model. In this model, a globally shared memory is added to a multiprocessor system. The processors are divided

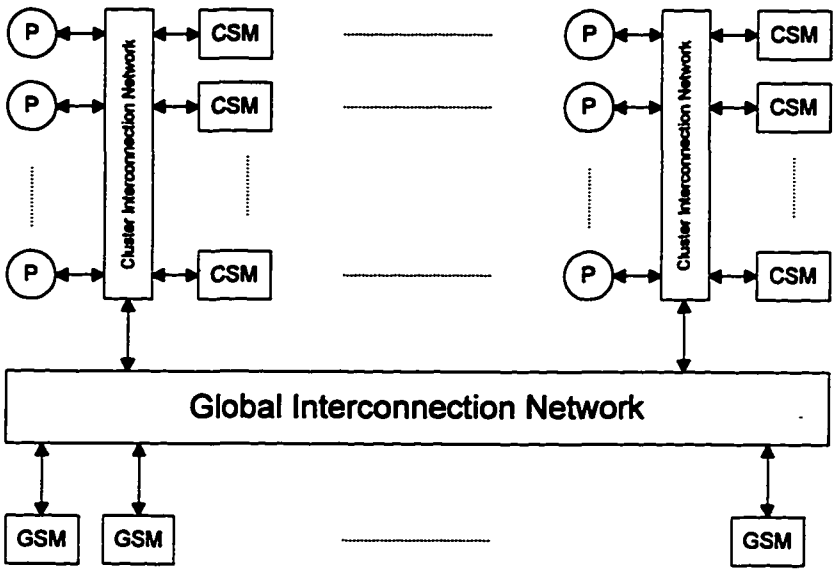

Figure 6: Hierarchical cluster model.

into several clusters. The clusters are connected to globally shared-memory modules. Each cluster could be designed as a *UMA* or *NUMA* machine while the system as a whole is considered a *NUMA* machine.

Regardless of the model used, each design possesses memory coherence and synchronization requirements. In the upcoming chapters, a new shared memory multiprocessor model, that solves the named problems using replicated shared memories together with distributed local memories, will be introduced.

2.3 Viable Physical Designs

The interconnection network between components of a multiprocessor have been constructed using several diverse designs. These networks provide the means for internal connections among processors, memory modules, and I/O devices. Thus, the communication efficiency of the network is critical to system performance. The ultimate goal

is to provide a low-latency network with a high data-transfer rate while providing a wide communication bandwidth to all devices simultaneously.

There are four basic organizations for the interconnection network:

1. *Bus-based* strategies,
2. *Static interconnection* networks,
3. *Dynamic interconnection* networks, and
4. *Multiport memory-based* schemes.

2.3.1 Bus-Based Interconnection Strategies

The simplest interconnection system for shared-memory multiprocessors employs a single common communication path called the *bus* which connects all devices. An example of a multiprocessor system using a bus interconnection is shown in Figure 7. The traffic generated per processor and the bus bandwidth provided, will determine the number of processors which can be adequately supported in such a shared-memory system. Since the overall transfer rate within the system is limited by the bandwidth and the speed of the bus, use of private memories is highly advantageous. Figure 8 shows such a multiprocessor system.

A bus-based memory system is the most frequently used interconnection in commercially available multiprocessors because it is a relatively inexpensive interconnection. However, a shared-bus interconnection exhibits two major deficiencies:

1. It provides low effective bandwidth available to each processor, and
2. It creates a single point-of-failure in the interconnection network.

Moreover, system expansion use of additional processors or memory modules, further increases the bus contention, which decreases system throughput and increases

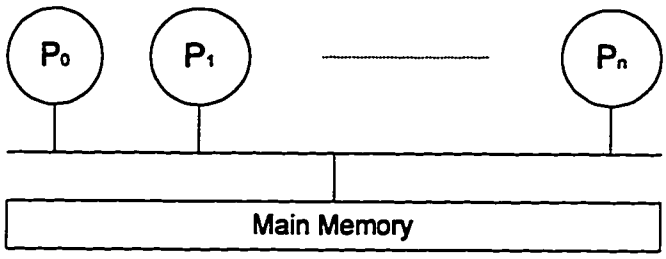

Figure 7: Single-bus multiprocessor.

arbitration logic. For this reason, bus-based interconnection alone is not considered scalable.

2.3.2 Static Interconnection Networks

Static networks consist of point-to-point dedicated links between individual processors or memories which remain fixed once the machine is built. Figure 9 shows such a system with 16 processors ($N = 16$). Each P_i is capable of sending data to any one of P_{i+1}, P_{i-1}, P_{i+r}, and P_{i-r} where $r = \sqrt{N}$ in one circulation step through the network. This example of static interconnection network topology (Wrapped-Around Mesh) is defined by four routing functions:

$$R_{+1}(i) = (i + 1) \ mod \ N$$

$$R_{-1}(i) = (i - 1) \ mod \ N$$

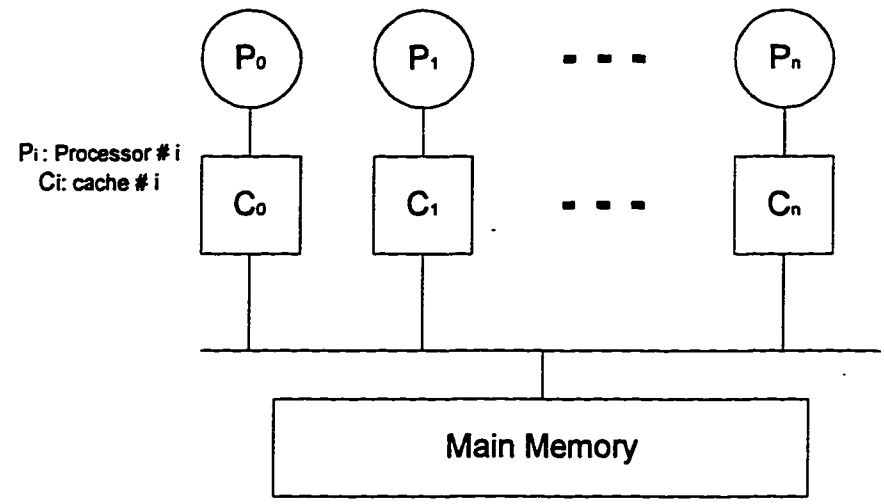

Figure 8: Single-bus multiprocessor with local caches.

$$R_{+r}(i) = (i + r) \ mod \ N$$

$$R_{-r}(i) = (i - r) \ mod \ N$$

where $0 \leq i \leq N - 1$. This type of network is more suitable for providing repeated regular connections among components. Static networks could be very expensive especially if the system employs a large number of processors and memory modules.

2.3.3 Dynamic Interconnection Networks

Dynamic interconnection networks are realized using switched channels, which are dynamically configured to match the communication requirements demanded by the concurrent tasks. Dynamic networks include *multistage networks* and *crossbar switches*, which are often used in shared-memory multiprocessors. Figure 10 depicts a shared-memory multiprocessor with a crossbar-based interconnection. As shown in Figure 10,

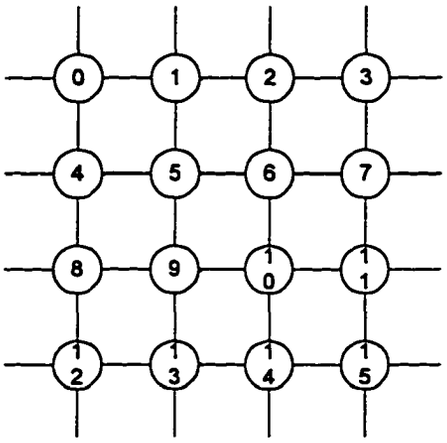

Figure 9: Static interconnection network.

a *n processor system with* n memory module has been implemented with $n \times n$ crossbar network which connects *n processors to* n memory modules. Although the highest bandwidth and interconnection capability are provided by crossbar network, it is the most expensive to build, due to the fact that its complexity increases (N^2) for a system containing N processors.

2.3.4 Multiported Memory Modules

Since the bandwidth of single-ported memory systems is limited, more scalable alternatives have been sought out. However, the recent availability of multiport memory chips has allowed multiple simultaneous access to individual memory devices [STODLECK89]. Figure 11 illustrates a multiport memory organization. By providing multiple simultaneous access, multiport memories can help mitigate the bandwidth mismatch between main memory and the processor by increasing the effective

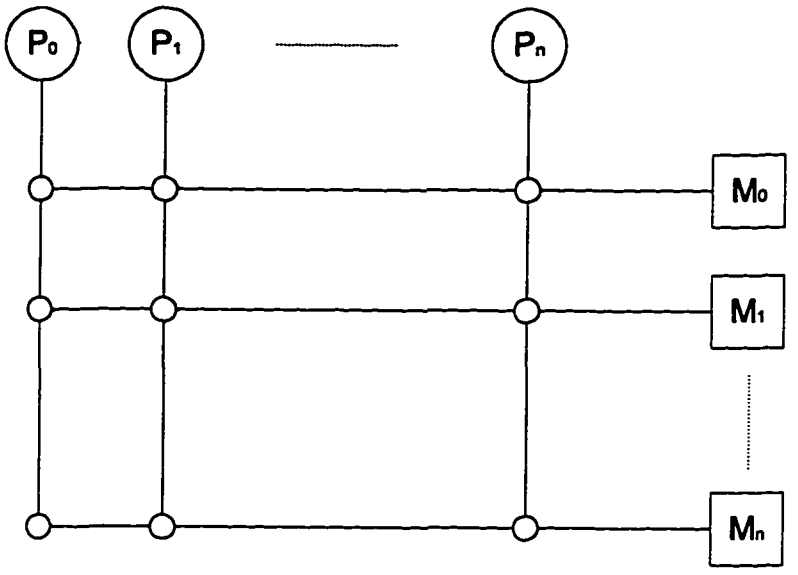

Figure 10: Crossbar-Connected Shared-Memory Multiprocessor.

memory bandwidth, and reducing contention.

As Figure 11 shows, multiple processors could access multiport main memory simultaneously. Multiport-memories are becomming popular among computer system designers because of their simplicity and low cost factor. In upcoming chapters, we introduce a computer design that takes advantage of multiport memories to improve total performance.

2.4 Private Cache Memories to Capture Locality

As previously mentioned, private cache memories can be employed to increase performance. Specifically, as a program executes its memory references over a given period, it tends to be confined to a few localized areas of the entire address space. This occurrence is referred to as the *locality of reference* property [MANO82]. Since

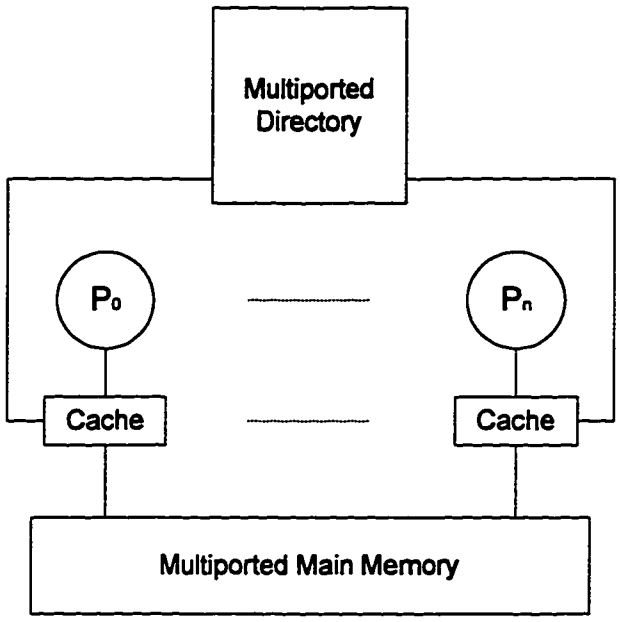

Figure 11: Multiport memory organization.

memory references are generated for both instruction and data fetches, a typical computer program may spend the majority of its execution time in only a fraction of the total code, such as a nested loop operation. This phenomenon is referred to as "90-10 rule" since 10% of the total code takes 90% of the execution time in a typical program [HENNESSY90]. Three types of locality of reference behavior have been identified [HWANG84]:

1. Temporal locality,
2. Spatial locality, and
3. Sequential locality.

Temporal locality refers to the inclination for programs to reference memory locations which have been accessed in the recent past. This is often caused by program

constructs such as loops, stacks, and temporary variables. By moving recently accessed memory blocks from remote memory into the fast memory, overall computer performance can be significantly improved. *Spatial locality* refers to the tendency of programs to access items whose addresses are near one another. For example, searching through a field of data, or operations on arrays. *Sequential locality* refers to a case when a typical program, fetches the instructions in sequential order unless branch instructions create out-of-order executions. The ratio of in-order execution to out-of-order execution is approximately 5-to-1 in typical parallel numeric programs [HWANG93]. Since 80 to 85 percent of the total code of a typical program executes in sequential order, then only 15 to 20 percent of the code contains branch instructions. As a result, by moving continuous blocks of memory space to local caches, a significant performance improvement can be observed.

Taking advantage of the principal of locality, by designing the memory of a multiprocessor as a memory hierarchy, results in a substantial reduction in average memory access time. Figure 12 depicts the basic structure of a memory hierarchy. Because of the differences in access time and cost, it is useful to build memory as a hierarchy of levels, with the faster memory close to the processor and the slower, less expensive memory below that, as illustrated in Figure 12. Caches are the fastest and closest memory modules to processors. A typical cache access time is in the range of 8 to 35 ns, while typical access time to the main memory is between 90 to 120 ns. Connecting a local memory directly to a processor will reduce the frequency of access to the shared memory space interconnection networks, and therefore, will decrease the total interconnection network traffic. As a result, caches increase the overall performance of a system, while reducing the interconnection network contention.

2.4.1 Data Updating and Coherence

Effective caching techniques are indispensable in distributed shared-memory systems, since providing high-speed caches for each processor significantly mitigates the overhead in referencing main memory. However, this improvement in performance is not

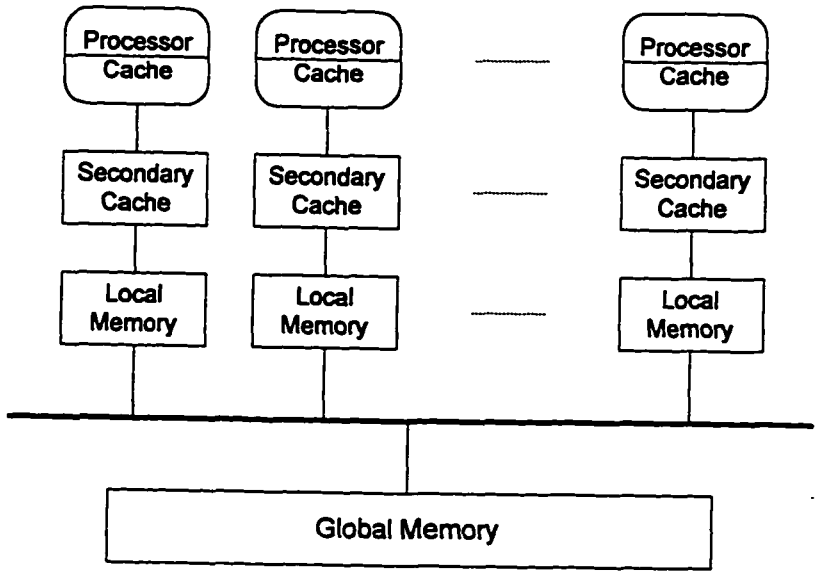

Figure 12: Basic Structure of a Memory Hierarchy.

cost free. Employing multiple caches gives rise to the problem of *cache coherence*.

The cache coherence problem is an undesirable, but surmountable side-effect which results from the use of a hierarchical memory design in a distributed shared-memory system [CRAWFORD94]. Since multiple caches are allowed to hold simultaneous copies of an assumed memory location's content, some techniques must exist to ensure consistency of all copies when one is modified. Simply, maintaining multiple copies representing the content of the same memory location implies the urgency to know which copy is valid.

Figure 13 illustrates a n processor distributed shared-memory system with coherent caches. Cache 1 and n each hold a copy of the shared variable, A. When Processor 1 updates its copy of variable A, the copy stored in Cache n is no longer valid, creating a cache coherency problem, as shown in Figure 14. To correct this inconsistency, Cache $n's$ copy must either be updated or marked as invalid.

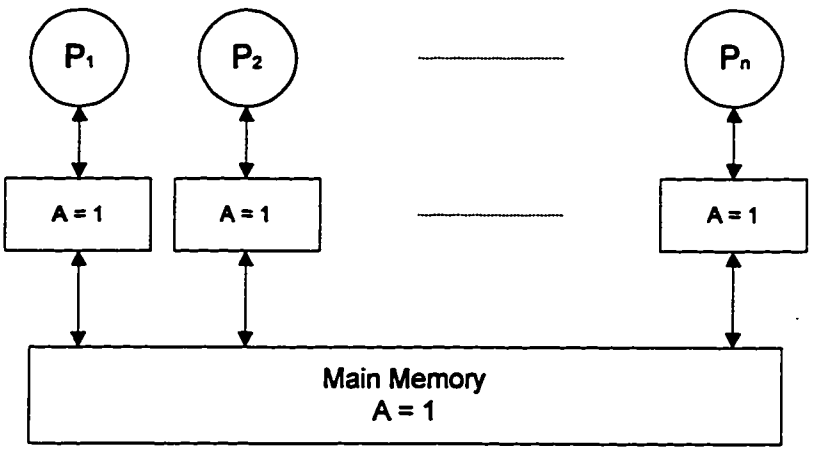

Figure 13: Coherent Caches Before Modification Occurs.

The performance of a cache design depends on two important factors:

1. *Cycle Count*: number of processor cycles required to locate, fetch, and deliver the data to the requesting PE, and

2. The *Hit Ratio*: an important factor in determining how effectively the cache can reduce the total memory-access time.

The cycle counts includes the number of total machine cycles needed for lower-level cache misses, cache updates, and consistency control.

The hit rate refers to a situation when a memory reference can be satisfied by the cache which is called a data reference *hit*. If the data is not found in the cache, it is refered to as a *miss*. Generally, a miss necessitates a search at the next higher level of memory hierarchy. The hit ratio between two adjacent levels is defined as the ratio of number of hits to number of references. Let H_{ratio}, h denote the hit rate, and the number of hits respectively. Let's assume A represents number of references then:

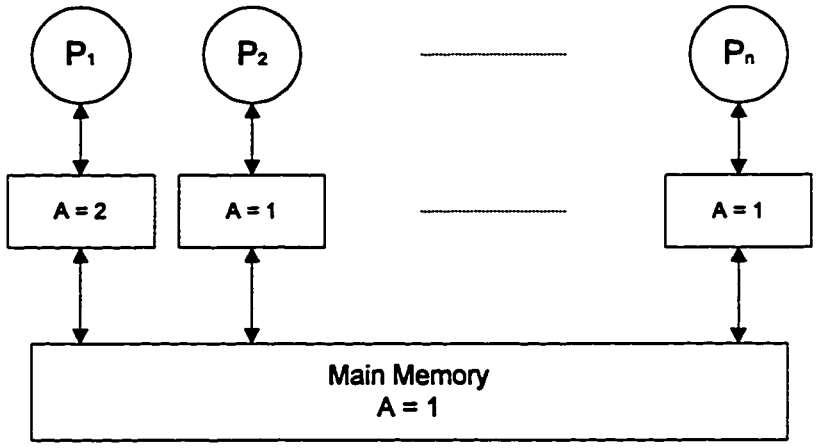

Figure 14: Inconsistent Caches after Modification by Processor 1.

$$H_{ratio} = \frac{h}{A} \tag{1}$$

where the number of references includes both hits and misses. If m denotes the number of misses then:

$$H_{ratio} = \frac{h}{h+m} \tag{2}$$

Hit ratios above 0.99 have been observed, with typical hit ratios above 0.9, validating the locality of reference principle.

2.4.2 Cache Updates In Distributed Shared-Memory Systems

A major characteristic of any multilevel memory system is the way it handles WRITE operations. Basically, there are two distinct alternatives to update memories:

1. *Write through* - whenever there is a need to update a memory block, the update will be done to blocks, both in the cache and in the lower level memory unit at the same time.

2. *Copy back* - the WRITE is performed only to the block in cache and whenever there is a need to replace that block then the modified block is written to the lower level memory.

Both of these update schemes are associated with certain advantages and disadvantages. The main advantages for write-through policy are:

1. READ misses never cause WRITEs to lower level memory, and

2. It is less complex to implement than the copy-back scheme.

The key advantages of copy-back policy are:

1. A single word can be written to the cache at processor speed rather than memory speed,

2. Several updates to the same block in the cache require only one WRITE operation to the lower level memory, and

3. Whenever blocks are written back, the system can take advantage of a wide lower level, since the entire block is being updated.

Generally, the copy-back policy requires a more complex and sophisticated hardware system, while the difference in the speed of processors and main memories, favors a trend towards the copy-back scheme. Very often, a write-through policy is adopted in the first level cache, and the copy-back scheme is used in the second level cache [Hwang93]. However, both of these update policies require hardware support, which is avoided totally in the Replicated Concurrent-Read (RCR) memory hiararchy system design. Since RCR performs all READ operations locally, while it broadcasts WRITEs globally, all caches will have a valid copy of data.

2.4.3 Coherence Strategies

Even though, the cache coherency problem has been the subject of much research, there is very little literature discussing cache coherency in multiport-based distributed shared-memory systems. Previously defined strategies to ensure cache coherency generally fall into two major categories:

1. hardware-based strategies, and
2. software solutions.

Hardware-based techniques were implemented without any support from software. These techniques maintain cache consistency by adopting one of the following protocols:

1. *Directory based* - The physical memory is divided into equal-size blocks. The sharing status of a block is kept in just one location, called the directory.
2. *Snooping* - In this protocol the sharing status of memory blocks is not kept in a centralized location. Snooping protocol is based on monitoring the shared-memory bus, since processors share a common memory via a single bus. This scheme could also be applied to mutiple buses in a system with moderate cost [O'KRAFKA90] [BERTONI91].

While hardware-based techniques are efficient, they add to system hardware complexity. Our proposed Replicated Concurrent-Read architecture requires no hardware support in order to maintain cache coherency. Software techniques, rely solely on the compiler or programmer to maintain cache coherency. These schemes could be categorized in two classes:

1. *Static placement* - at compile time, all shared writable data are tagged as non-cashable. Thus, any memory block that can be modified by more than one processor, will be stationary in the main memory. This protocol is considered the simplest to implement.

2. *Software-based* protocols - this scheme depends on a sophisticated compiler to insert cache control instructions at compile time to maintain data consistency.

Replicated Concurrent-Read architecture does not require any software support to eradicate the coherency problem because of its unprecedented design. There are advantages and disadvantages associated with each one of these coherence strategies. Crawford examined and analyzed the effect of four coherence schemes on multiport memory systems. Namely, *No-cache*, *Synapse (snooping)*, *Firefly*, and *Directory* schemes were analyzed and simulated for a multiport system. The result of the analytical model and simulation were in favor of Firefly techniques in terms of memory average access time [CRAWFCRD94]. However, this technique requires extensive hardware, since cache-to-cache transfers demand four dedicated paths between caches. Such hardware costs become enormous as the number of processors grows large, making the Firefly less scalable than other techniques. Our proposed Replicated Concurrent-Read architecture virtually requires no adoption of coherence strategies in any form. Broadcasting all WRITE operations will solve this unwanted problem at minimal cost.

2.5 Memory Consistency Models

The memory model characterizes the behavior of a shared-memory system as observed by the processors. The problem of *memory inconsistency* emerges when the memory-access order differs from the program execution order. A memory model specifies three fundamental perspectives:

1. Behavior of a shared-memory multiprocessor,
2. Coverage of all contingencies, and
3. Behavior of processors and shared-memory systems to ensure consistent adherence to the expected behavior of the multiprocessor.

In general, choosing a memory model is a process of making a compromise between a robust model minimally restricting the software or a relaxed model offering. The

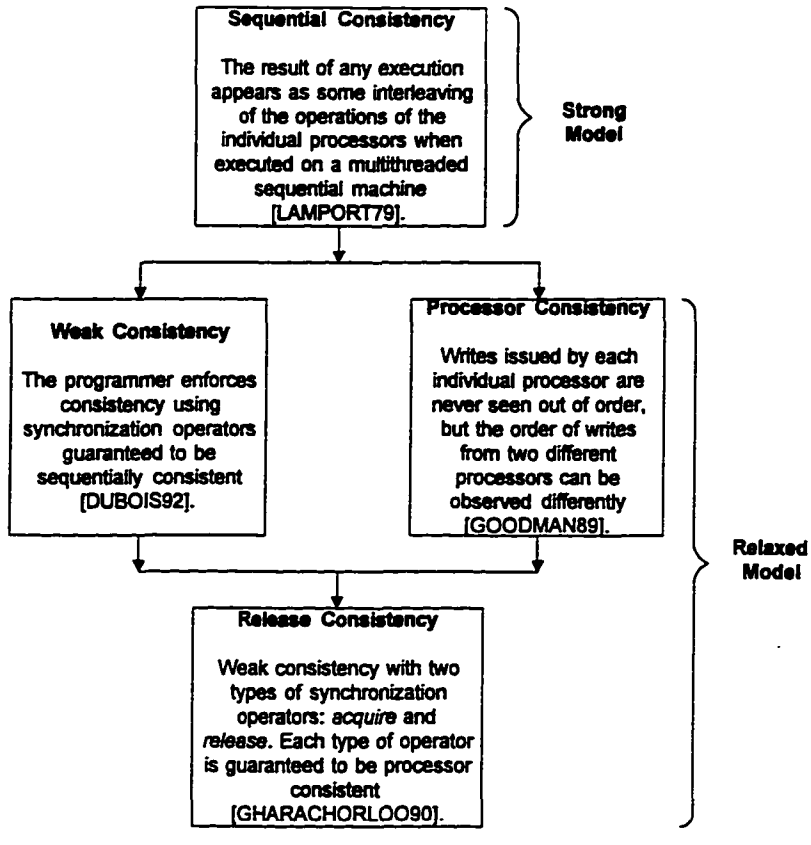

Figure 15: Intuitive Definition of Four Memory Consistency Models.

quality of a memory model is evaluated by software/hardware efficiency, simplicity, and bandwidth performance. Most multiprocessors have implemented the strong consistency model because of its simplicity. Memory accesses for this model are atomic and strongly ordered, and confusion can be avoided by having all processor/caches sufficiently wait during unexpected run-time events. However, the model may lead to poor memory performance due to the imposed strong ordering of memory events. This is especially true when the shared-memory system becomes very large. Therefore, a strong consistency model makes a scalable design more difficult to obtain. On the other hand, relaxed consistency models may offer increased performance by hiding as much write latency as possible. The main disadvantage is increased hardware complexity and a more complex programming model. Figure 15 summarizes four memory consistency models [HWANG93].

Shared-memory systems can be further classified into two behavioral categories:

1. Only atomic memory accesses are supported, and

2. Non-atomic memory accesses are also allowed.

A memory access is *atomic* if the memory updates are known to all processors at the same time, and it is *non-atomic* if coherence mechanism does not necessarily inform all processors at the same time. A strong consistency model can support atomic access and a relaxed consistency model usually conforms with non-atomic accesses.

Three primitive memory operations have been defined for the purpose of specifying memory consistency models [DUBOIS86]:

1. A READ by processor P_i is considered performed with respect to the processor P_k at a point of time when the issuing of a WRITE to the same location by P_k can not affect the value returned by the READ.

2. A WRITE by P_i is considered performed with respect to P_k at one time when an issued READ to the same address by P_k returns the value by this store.

3. A READ is globally performed if it is performed with respect to all processors and if the WRITE, that is the source of the returned value, has been performed with respect to all processors.

The Replicated Concurrent-Read (RCR) memory consistency model supports atomic memory accesses by broadcasting WRITE operations globally. Since READs are performed locally in one cycle, then an issued WRITE to the same address by any processor can not affect the value returned by the READ. The RCR memory consistency model is simple and efficient.

2.5.1 Sequential Consistency Model

Sequential Consistency is the most widely implemented model by multiprocessor designers. Figure 16 illustrates this memory model where the READ, WRITE, and

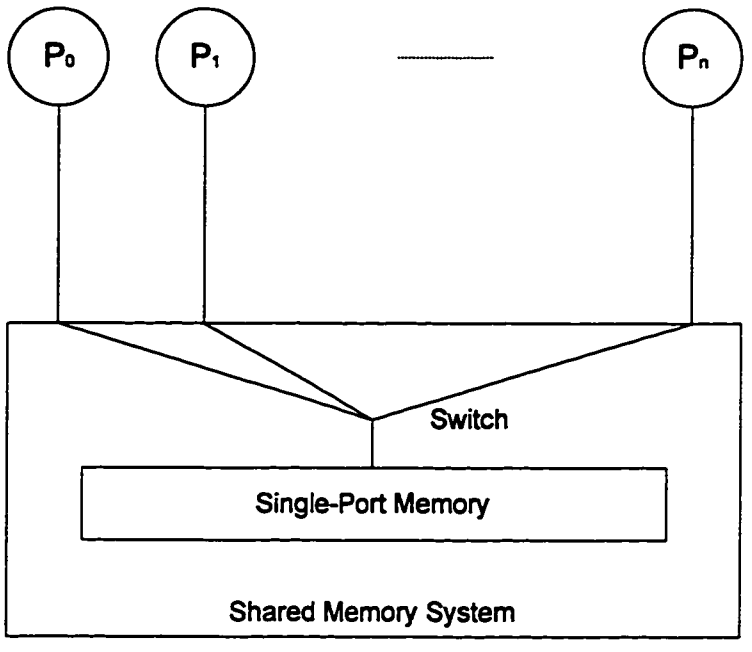

Figure 16: Sequential Consistency Memory Model

swap of all processors are performed serially in a single global memory order that conforms to the individual processor's program orders.

Lamport has defined Sequential Consistency as follows: a multiprocessor system is sequentially consistent if the result of any exection is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program [LAMPORT79].

Dubois, Scheurich, and Briggs have stipulated the following two sufficient conditions to reach sequential consistency in shared-memory access [DUBOIS86]:

1. Before a READ is allowed to perform with respect to any other processor, all previous READ accesses must be globally performed and all previous WRITE accesses must be performed with respect to all processors.

2. Before a WRITE is allowed to perform with respect to any other processor, all

previous READ accesses must be globally performed and all previous WRITE accesses must be performed with respect to all processors.

Sindhu, Frailong, and Cekleov have defined the sequential consistency memory model with the following five propositions [SINDHU92]:

1. A READ by a processor always returns the value written by the latest WRITE to the same location by other processors.

2. The memory order conforms to a total binary order in which shared-memory is accessed in real time over all READs and WRITEs with respect to all processor pairs and location pairs.

3. If two operations appear in a particular program order, then they appear in the same memory order.

4. The swap operation is atomic with respect to other WRITEs. No other WRITE can intervene between the READ and WRITE parts of a swap.

5. All WRITEs and swaps must eventually terminate.

The RCR memory consistency model is comparable to the sequential consistency model.

2.5.2 Relaxed Consistency Models

Memory consistency models, as stated before, can range from strong to various degrees of relaxed consistency. Figure 17 exhibits an example of a relaxed model, the *total store order* (TSO) Weak Consistency model developed by Sun Microsystems' SPARK Architecture Group (1990). Dubois, Scheurich, and Briggs have developed a relaxed consistency memory model by relating memory access request ordering to synchronization points in the program. This model indicated by the following three conditions [DUBOIS86]:

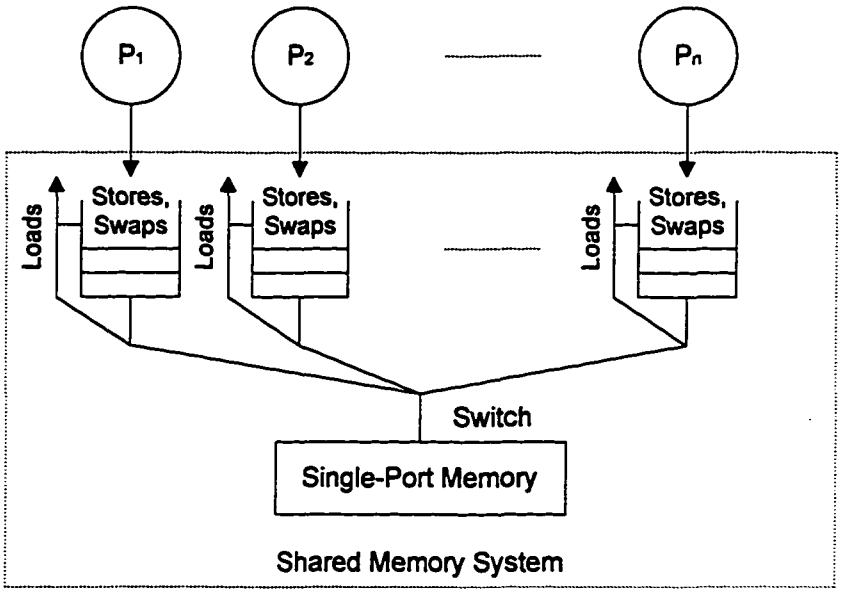

Figure 17: TSO Weak Consistency Model.

1. All previous synchronization accesses must be performed, before a load or a WRITE access is allowed to perform with respect to any other processor.

2. All previous READ and WRITE accesses must be performed, before a synchronization access is allowed to perform with respect to any other processor.

3. Synchronization accesses are sequentially consistent with respect to one another.

By applying different restrictions on memory-access ordering, we may define multiform weak memory models.

Sindhu, Frilong, and Cekleov have provided the TSO Weak Consistency (WC) model with six behavioral axioms [SINDHU92]. The following is an intuitive description of their abstracted axioms by Kai-Hwang [HWANG93]:

1. A READ access is always returned with the latest store to the same memory location issued by any processor in the system.

2. The memory order is a total binary relation over all pairs of WRITE operations.

3. If two WRITEs appear in a particular program order, then they must also appear in the same memory order.

4. If a memory operation follows a load in the program order, then it must also follow the READ in memory order.

5. A swap operation is atomic with respect to other stores. No other store can interleave between the READ and WRITE parts of a swap.

6. All WRITEs and swaps must eventually terminate.

The RCR memory consistency model, without relating memory access request ordering to synchronization points in the program, offers the same efficiency as the above memory model without any increase in hardware complexity.

The Processor Consistency (PC) model, introduced by Goodman (1989), notes that WRITEs issued by each processor, are always in program order. However, the WRITEs issued by two different processors can be out of program order. The order of READs issued by each processor is not restricted as long as they do not involve other processors. This model relaxes from the sequential consistency model by removing some restrictions on WRITEs from different processors. Thus, it creates more opportunities for the buffering and piplining of WRITEs. Two conditions must be required for insuring processor consistency:

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

2. Before a WRITE is allowed to perform with respect to any other processor, all previous READ or WRITE accesses must be performed.

Release Consistency (RC), introduced by Gharachorloo et al.(1990), is one of the most relaxed memory models available. In this consistency model, the synchronization accesses in the program must be identified as either *acquire* (e.g.,locks) or *release*

(e.g., unlock). An *acquire* is a READ operation that attains permission to access a set of data, while a *release* is a WRITE operation that grants such permission. There are three conditions that ensure release consistency:

1. Before an ordinary READ or WRITE access is allowed to perform with respect to any other processor, all previously acquired accesses must be performed.

2. Before a release access is allowed to perform with respect to any other processor, all previous ordinary READ and WRITE accesses must be performed.

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

Stanford researchers have reported results for evaluating these memory models under three applications [GUPTA91]. They include a particle-based three-dimentional simulator used in aeronautics(MP3D), a LU-decomposition program (LU), and a digital logic simulation program (PTHOR). Their research illustrates the breakdown of execution times under the sequential and relaxed models for the three benchmarks. Relaxed models remove all idle time due to write-miss latency. Based on the same model, the READ operation cannot be performed in parallel with a single port shared-memory holding the global data. Thus, the execution time has increased for benchmarks MP3D and LU, while, a very slight improvement of 52.9% to 49.0% is shown for PTHOR.

The key point being that the major processor's time is spent stalling for a READ miss.

2.6 Summary

The architecture of a multiprocessor refers to the relationship between processors, memory modules and I/O devices. In this chapter, UMA, NUMA and COMA, architectures were introduced. In UMA architecture all processors are attached to their private caches while sharing a global memory. The advantage of this architecture is the simplicity of design and ease of programming. The major drawback of the UMA machine is contention in global traffic, as the number of processors grows. The NUMA architecture is a distributed shared-memory machine. Every processor can address its own local memory as well as any remote memory module. The advantage of this machine is its ability to support a large number of processors. COMA architecture is very similar to NUMA machines except that distributed memory modules are replaced with caches.

The communication efficiency of a network is essential to system performance. There are four major organizations for the interconnection network:

1. Bus-based strategies,
2. Static interconnection networks,
3. Dynamic interconnection networks, and
4. Multiport memory-based schemes.

Memory references tend to be confined to a few localized areas of the entire address space As a program executes these references over a given period of time. This phenomenon is known as locality of references. Three types of locality of reference behavior have been identified:

1. Temporal locality,
2. Spatial locality, and
3. Sequential locality.

One of the major characteristics of any multiprocessor system is the way it handles WRITE operations.

The memory model characterizes the behavior of a shared-memory system as seen by the processors. The problem of memory inconsistency emerges when the memory-access order differs from the program execution order.

3 Previous Scalable Architectures

In this Chapter, we review some of the most recent approaches based on maintaining coherence through demand-driven movement of data when it is referenced. Several approaches have been used, including *directory structures* which maintain a list of which memories hold the valid copy and *snooping strategies* which avoid the bottleneck of a centralized directory, but require many-to-many monitoring of each processor's update traffic. We discuss advantages and disadvantages of each scheme to provide the basis for a totally new architecture.

3.1 Overview

Since we are primarily interested in designs which are scalable in the number of processors they can support, we present four examples of the most scalable designs which have been proposed. We show two directory-based architectures, one snooping-based architecture, and a rudimentary replicated-memory architecture.

3.2 The Stanford DASH Architecture

The DASH multiprocessor system has been under development by John Hennessy and co-workers at Stanford University since 1988 [HWANG93]. "DASH" is an acronym for Directory Architecture for Shared Memory. It combines the scalability of message-passing machines, by distributing the shared memory space among PEs with the programmability of a single address space through directory-based coherence protocols. Processing *clusters* share a single global address space interconnected by a scalable interconnection network.

The architecture is composed of two structural levels. The first level consists of a set of processing clusters, each set is a bus-based multiprocessor with primary and secondary caches. Coherence within this cluster is supported using a snoopy bus-based protocol. The second level is a mesh interconnected network connecting the clusters together. Within this level, inter-cluster cache coherence is maintained by a

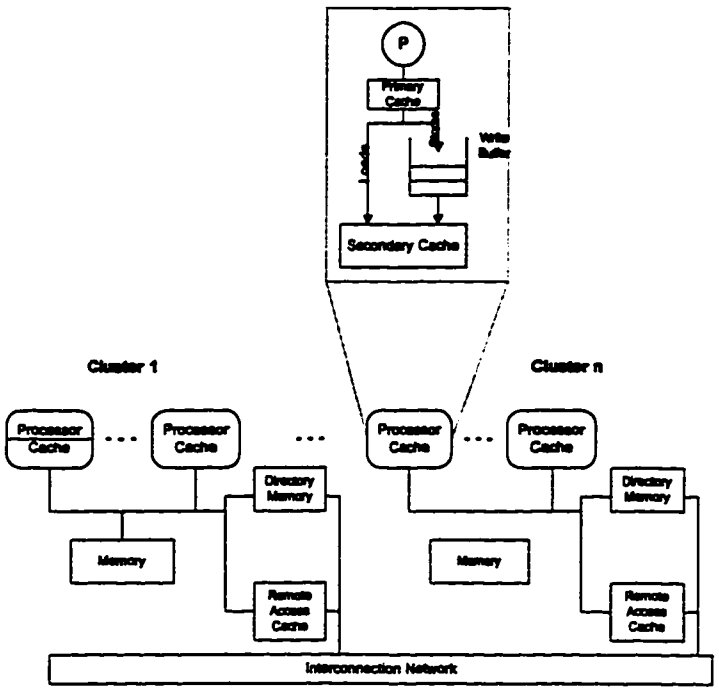

Figure 18: Stanford's DASH architecture.

distributed directory-based cache coherence protocol. The interconnection network among the 16 multiprocessor clusters is a pair of wormhole-routed mesh networks. One mesh network is used to request remote memory, and the other is a reply mesh. Figure 18 depicts at a high level organization of the Stanford's DASH architecture.

Caches within the clusters are designed to match the processor speed and support snooping from the bus to maintain the coherency. Intra-cluster coherence implements the Illinois or *modified, exclusive, shared, invalid* (MESI) protocol.

3.2.1 DASH Memory Hierarchy

The Stanford DASH implements an invalidation-based cache-coherence protocol. A memory location may be in one of three states:

1. *Uncached*: not cached by any cluster,

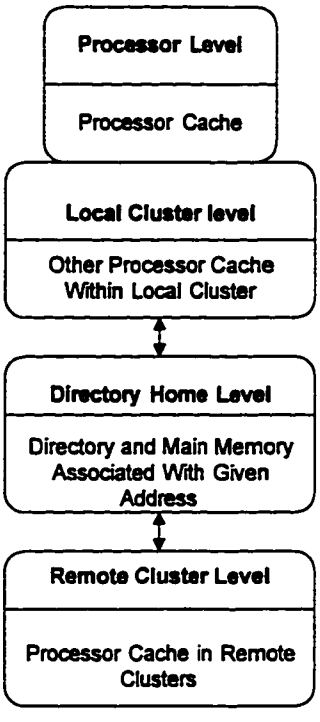

Figure 19: DASH memory hierarchy.

2. *Shared*: in an unmodified state in the caches of one or more clusters, or

3. *Dirty*: modified in a single cache of some cluster.

The directory keeps the summary information for each memory block, specifying its state and the clusters that are caching it. The four levels of the DASH memory hiererchy are shown in Figure 19.

The first-level cache is designed to match the processor speed. If any request is not satisfied at this level then it is routed to the second level. It takes 30 processor clock cycles to access this level. The second-level consists of other processor's caches within the local cluster. If the data is locally cached at this level, the request can be satisfied within the cluster, otherwise the request is sent to the *home* cluster level. The home-level consists of the cluster that contains the directory and physical memory for a given memory address. It takes 100 processor clock cycles to access the directory in this level.

For many accesses, most private data references, local and home clusters, are the same and the hierarchy can be collapsed to just three levels. In general however, a request will travel through the interconnection network to the home cluster. The home cluster can usually satisfy the request immediately, but if the directory entry is in a dirty state, or in a shared state when the requesting processor requires exclusive access, the fourth level must also be accessed which requires 135 clock cycles. The remote cluster level for a memory block consists of the clusters marked by the directory as holding a copy of the block.

3.2.2 Cache Coherence Method in DASH

Two levels of local cache are used per processing node as shown in Figure 18. One can assume a write-through primary cache and a write-back secondary cache. READS and WRITES are separated with the use of WRITE buffers for implementing weaker memory consistency models. The main memory is shared by all processing nodes in the same cluster. To facilitate prefetching and the directory-based coherence protocol, directory memory and remote-access caches are used for each cluster. The remote-access cache is shared by all processors in the same cluster.

The directory memory relieves the processor caches of snooping on memory requests by keeping track of which caches hold each memory block. Figure 20 illustrates a directory structure per cluster. As proposed by Censier and Feautrier, each entry contains one presence bit per processor cache. In addition, a state bit indicates whether the block is uncached, shared in multiple caches, or held exclusively by one cache; if the block is dirty. By inspecting the state and presence bits, the memory can become aware of which caches need to be invalidated when a location is written. This facilitates the scalability of the DASH by reducing the demand on available memory bandwidth.

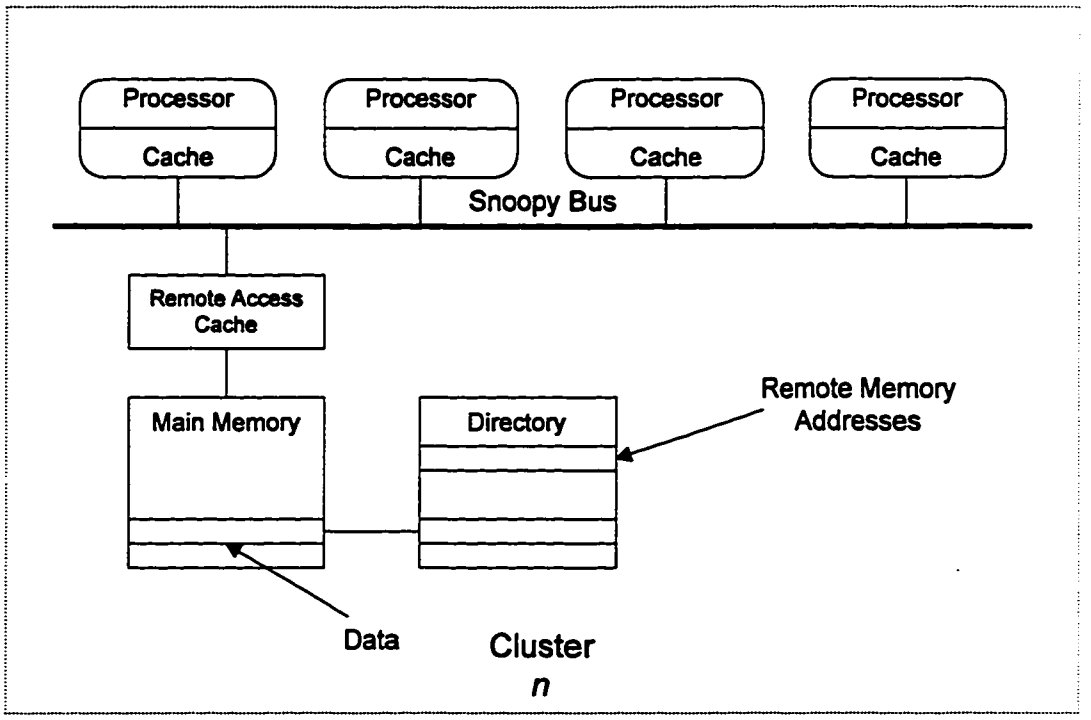

Figure 20: Directory structure per cluster.

3.3 The Kendall Square Research KSR-1

The Kendall Square Research KSR-1 is the first commercial attempt at building a COMA-style scalable multiprocessor. Scalability in the KSR-1 is obtained by connecting 32 processors to assemble a ring multi (search engine 0 in Figure 21) operating at 128 million accesses per second. As shown in Figure 21, the KSR-1 uses a two-level hierarchy to interconnect 34 Ring:0s by a top-level Ring:1. Each node consists of a 32-Mbyte primary high-speed memory, and a 64-bit superscalar processor. The superscalar processors are designed for both scalar and vector operations. Each processor comprises 64 floating-point and 32 fixed-point registers of 64 bits.

3.3.1 KSR-1 Memory Hierarchy and Coherence Strategy

The KSR-1 offers a single-level memory, named ALLCACHE by its designers. As a result of this, the KSR-1 eliminates the memory hierarchy present in conventional

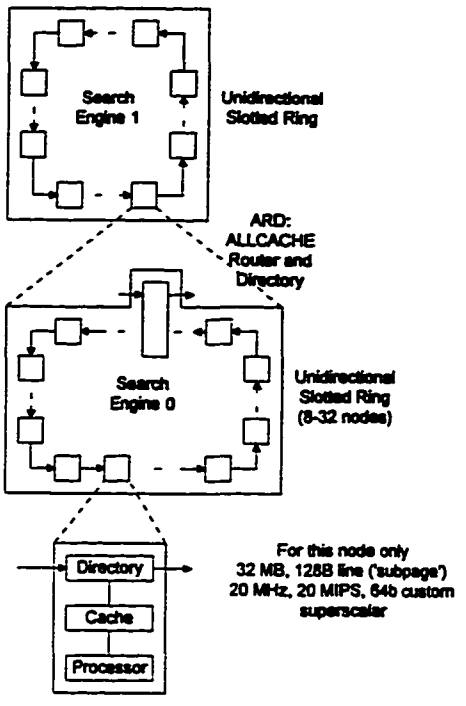

Figure 21: The Kendall Square Research KSR-1 architecture.

computers and the corresponding physical memory addressing overhead. The KSR-1 maintaines data consistency by replication of data throughout the distributed processor memory nodes.

3.3.2 KSR-1 Programming Model

The KSR-1 machine provides a strictly sequential consistent programming model and dynamic management of memory through hardware migration and replication of data throughout the distributed processor memory nodes using its ALLCACHE mechanism and a sequential consistency model. With ALLCACHE, an address becomes a data name, and this name automatically migrates throughout the system and is associated with a processor in a cache-like fashion as needed. Copies of a given cell are made by the hardware and sent to other nodes to reduce access time. A processor can prefetch data into a local cache and poststore data for other cells. The hardware exploits

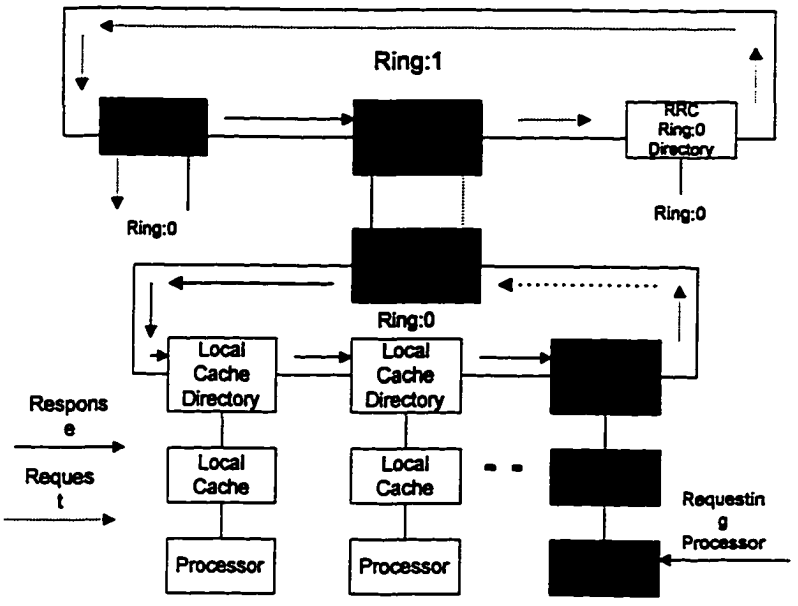

Figure 22: Remote cache access in KSR-1 architecture.

temporal and spatial locality.

Figure 22 illustrates the situation when the requester and responder reside in different ring:0s. The top level, Ring:1, consists entirely of Ring Routing Cells(RRCs), each containing a directory for the Ring:0 to which it is connected. Each RRC directory on Ring:1 is essentially a duplicate of the RRC directory on the corresponding Ring:0. When a packet reaches an RRC on Ring:1, it will be moved to the next RRC on the ring of the RRC directory indicating that the requested data is not on the corresponding ring. Otherwise, the packet is routed down to the RRC on Ring:0. Ring:0 has the capacity of processing 8 million packets per second, and Ring:1 could handle 8, 16, 32, or 64 million 1024-bit packets per second.

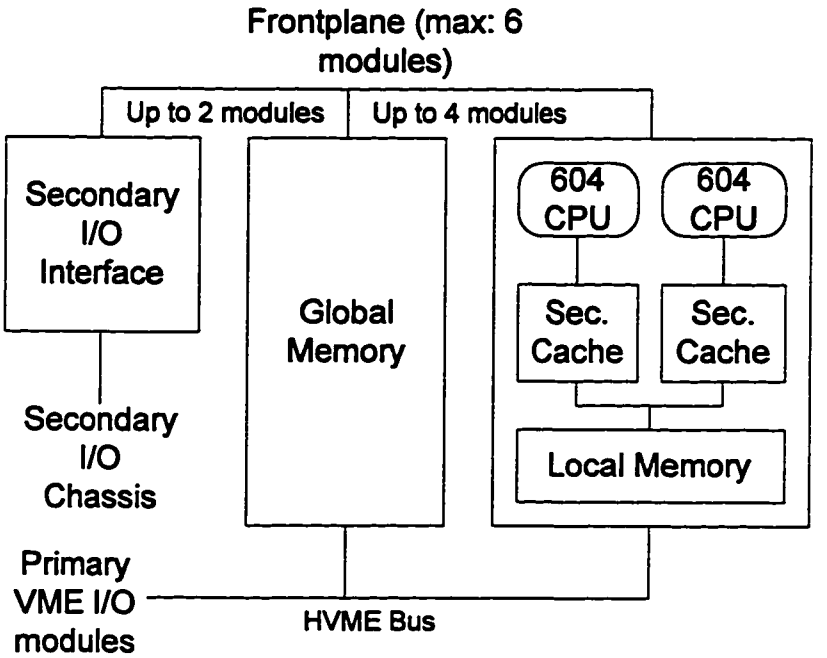

Figure 23: The Night Hawk System Block Diagram[Harris]

3.4 The Nighthawk 6800-Series Multiprocessor

The Nighthawk multiprocessor is a shared-memory system specifically for Real-Time applications. The system is composed of CPU modules, Global Memory modules and I/O Interface modules. A maximum system consists of 6 modules. Up to 4 modules can be CPU modules, and up to 2 modules can be Global Memory modules. One of the modules can be a secondary I/O Interface module. This system supports up to 8 processors as a tightly-coupled multiprocessor. Figure 23 depicts the Nighthawk simplified system block diagram.

3.4.1 Local-Remote-Global Memory Hierarchy

The Nighthawk memory system can be decomposed into four levels of hierarchy, as illustrated in Figure 24. The first level is the *primary cache* which is within the PE, and matches the processor speed.

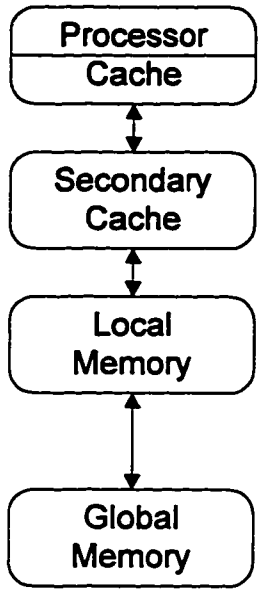

Figure 24: The Nighthawk memory hierarchy.

A request that cannot be serviced by the primary cache is sent to secondary cache. Designers have estimated a 95% hit rate at this level for typical real-time tasks. If there is a miss, then snooping the local bus and system bus takes place until the request is satisfied.

3.4.2 CPU-Level Local and Remote Memory Design

Each CPU board could have 1 or 2 IBM/Motorola 604 processors supporting 100 MHz and 150 MHz clock rates.

A separate direct-mapped secondary cache is used for each processor and these caches support copyback and write-through protocols. The processor bursts data (operands and instructions) to/from the secondary cache at the rate of 400 MB/sec. The CPU burst size is one cache line which consists of 4 (64-bit) double words. The write-through mode is used only by the operating system and is not available to the

users. The secondary cache RAMs are packaged as a single *daughter board* with the RAMs for both processors. The RAM module supports a 1 Megabyte direct-mapped secondary cache for each processor.

The Nighthawk 6800 includes a cache snoop filter at the frontplane interface. This filter consists of two cache tag sets (one for each processor) that keeps track of global memory reads into the processors on the board. The filter tags then indicate if a snoop request from other boards can possibly be in the caches on this board. If the snoop request address is not on the filter tags, then the filter responds to this miss status, of the frontplane, without bothering the caches or the processors. If the filter tags show a hit, then the snoop request is passed to the secondary caches and processors. The filter tags accommodate the secondary cache size defined above.

The Night Hawk 6800 supports two types of local memory, one using static RAM's for higher performance and one using dynamic RAM's for higher capacity. Local memories are packaged as daughter cards, and its size may be upgraded by replacing the daughter card.

3.4.3 Global Memory Design

The global memory system consists of one or two global memory boards and may contain a combined total of 1 gigabyte of storage. The memory board is a two ported dynamic RAM memory device that can service either the HVME backplane or the system *frontplane*. It consists of a mother board and a Memory Daughter Card (MDC). The daughter card allows for different memory densities using a common mother board and the capability of memory expansion.

The global memory board insures system-wide cache coherency by broadcasting primary I/O accesses to the frontplane for snooping.

3.5 Multiprocessors Employing Global Data Replication

Concept of data replication refers to a multiprocessor system where each processor has its own local memory and the shared data are replicated in each of the local mem-

ories. This scheme is somewhat orthogonal to the DASH and the KSR-1 architectures discussed earlier in this chapter. Although both architectures support data replication and process migration, they do not replicate the shared data over the processor memory pairs. It can be used to implement *Uniform Memory Access* (UMA) systems if all memory addresses are replicated, or it can be used to implement *Non-Uniform Memory Access* (NUMA) systems if only some addresses are replicated. However, this implies a cost disadvantage of requiring redundant memories that contain identical data.

Various aspects of data replication in multiprocessor systems have been described in United States patent documents, numbers 4928224, 5214776, 5247629, 5247673, and 5274789 of different applicants. Figure 25 illustrates an architecture (Patent No: 5214776) under the title name *"Multiprocessor System Having Global Data Replication"*. It is comprised of four identical control processing units, CPU 0 to 4. The detail is shown for CPU 0 only. All units communicate with each other through a system bus. They also share peripheral units and common memory resources through the system bus. Its designers claim this architecture provides remarkable advantages in terms of performance because each processor has access to global data for read operations without needing access to the system bus. However, this architecture requires that the global data be replicated in each of the several local memories.

Coherence problems could be solved using several techniques based on the type of replication. There are three types of replication that can solve this coherence problem:

1. Global data replication,
2. Dynamic data replication, and
3. Selective data replication.

3.5.1 Global Data Replication

The *global data replication* approach replicates shared data over entire local memories. This way each processor may read the global data in its related local memory without

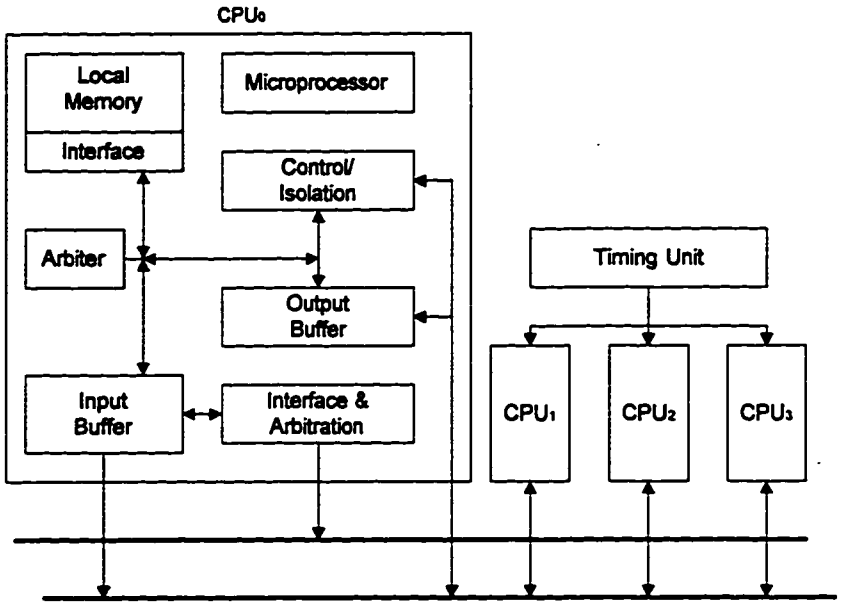

Figure 25: Multiprocessor system having global data replication.

accessing the system bus. In the event of a write operation on global data, access to the system bus is required to write the data in all local memories to assure consistency. This architecture demands a large amount of local memory, adequate for storing all the global data which may be required for parallel processing. To keep the size of the local memories economically feasible, it is important to keep the local data and replication at a minimal level.

3.5.2 Dynamic Data Replication

Dynamic data replication is performed at the page level, only when global data is requested by several processors. First, the data is treated as local then converted to global, and vice versa, to be replicated dynamically at run time. As long as the data is considered local, it is not necessary to allocate memory space in each and all of the local memories. This limits the bus access operations by maintaining coherence

among the copies of the data and size of the local memories required to run a particular application.

3.5.3 Selective Data Replication

Selective data replication uses local memories efficiently by further limiting the replication of the global data. It is necessary to perform replication according to selective criteria. This type of approach would require the development of complex and sophisticated replication mechanisms, which would permit control of WRITE operations in selected local memories. However, the drawback of costly memory replication, must be weighed against the resulting advantages. The outcome would limit the replication of data and the problems of global and dynamic type replication, and thus reduce the size of required local memories.

3.6 Summary

In this chapter, four scalable shared-memory multiprocessors were reviewed. DASH architecture implements an invalidation-based cache coherence protocol. Directory-based architectures, such as DASH, maintain the summary information for each memory block, specifying its state and the clusters that are caching it. KSR-1 architecture also relies on directory-based schemes to maintain data coherency. KSR-1 is an example of COMA architecture. The Nighthawk multiprocessor uses snooping strategies which avoid the bottleneck of a centralized directory, but require the monitoring of each processor's update traffic. Finally, a multiprocessor system having global data replication was introduced. This architecture requires that the global data be replicated in each of the several local memories. The multiprocessor system, having global data replication, maintains data consistency by employing several techniques based on the type of replication.

4 Analysis of Memory Access Behavior in Multiprocessors

4.1 Overview

Previous multiprocessor designs have viewed memory as a scarce resource which must contend for simultaneous READ access by storing only one valid copy of each updated data item. While this approach was economically necessary several years ago, it created a cache coherence problem. Advancement of memory technology has opened up new opportunities by optimizing memory system design for the most frequently used operations.

The principle of locality of references has demonstrated that, roughly 90% of memory accesses are local memory references. Also true is that, 80 to 90% of memory references are READ operations, thus, the concentration of our research is of these memory reference behaviors. In the next two sections, we will analyze these memory access behaviors.

4.2 Global vs. Local Memory References

A memory reference that can be satisfied by the local memory is called local memory reference. Likewise, an access to main memory is known as global memory reference. The principle of locality of reference states that, local memory references constitute 90% of all memory accesses. Since local memory access time is shorter than referencing global memory modules, then the emphasis should be placed upon the design of memory hierarchy systems, in order to minimize the total memory access time.

A memory hierarchy system consists of multiple levels of memory modules with different speeds and sizes. Fast memories are more expensive per bit than slower memories and are usually smaller. The ultimate design goal is to minimize the cost

Table 1: Memory Technologies.

Memory technology	Typical access time	$ per MByte in 1993
SRAM	8 - 35 ns	$100 - $400
DRAM	90 - 120 ns	$25 - $50
Magnetic disk	10,000,000 - 20,000,000 ns	$1 - $2

per bit of the total memory system while maximizing the speed of memory references. High speed memories are approximately four to ten times faster than main memories, while they are four to eight times more expensive than slow memories. Main memories use DRAM (dynamic random access memory), while caches are implemented from SRAM (static random access memory). Today, the three major technologies used to build memory units are DRAM, SRAM, and disk. Table 1 illustrates the access time and costs pertaining to these technologies [HENNESY94].

So far, by providing a local memory to every processor, average memory access time has been notably reduced. Generally, minimizing the number of global memory references while reducing the cost per bit of memory units will result in a more effective shared-memory system. By employing multiported memories in a replicated fashion, the total memory access time can be improved even further. Our proposed memory hierarchy system not only shows an improvement over existing systems, but that it is also free of cache coherency problems.

4.3 READ vs. WRITE Distribution of Memory Accesses

While the majority of all memory references are READ operations [HWANG84], it is surprising that no experimental studies, examining the impact of O(1) READ memory access without overhead cost, have been done. This reality has been neglected for many years, due to the fact READ and WRITE latencies are identical in uniprocessors. The ratio of READ/WRITE operations increases even more when the transition

Table 2: General Statistics on the Benchmark Applications.

Application	Instructions Executed (*millions*)	Shared Data References (*millions*)	percentages of shared data references
OCEAN	120	16.5	13.8%
PTHOR	86	15.8	18.4%
MP3D	209	22.4	10.7%
CHELOSKY	1302	217.2	16.7%
LU	50	8.2	16.4%
LOCUS	897	130.3	14.5%
BARNES	337	44.9	13.3%
WATER	2165	195.3	9%

is made to multiprocessors. Since in a shared-memory system, a WRITE operation will be followed by at least one READ memory reference, this actuality is further amplified. Table 2 shows the statistics taken from the execution of benchmark applications on the SPLASH architecture [GHARACHORLOO95]. These general statistics exhibit that only a small percentage of all memory references are shared data references. Let A_i and A_i^s denote the number of memory accesses and shared data references, respectively. The weighted average of shared references ($\bar{A}_i^s$) is:

$$\bar{A}_i^s = \frac{\sum A_i^s}{\sum A_i} \times 100 \tag{3}$$

By substituting the values of A_i and A_i^s in equation 3, we have:

$$\bar{A}_i^s = \frac{650.6M}{5166M} \times 100 = 12.6\%$$

These statistics amplify the fact that only small percentages of all memory accesses are shared data references.

Table 3: Statistics on Shared Data References and Their Characteristics.

Application	READs $X1000$	WRITEs $X1000$	R/W Ratio
OCEAN	12.280	4.255	2.9
PTHOR	14.516	1.316	11.0
MP3D	16.965	5.468	3.1
CHELOSKY	193.216	24.049	8.0
LU	5.478	2.727	2.0
LOCUS	117.440	12.847	9.1
BARNES	34.121	10.765	3.2
WATER	146.376	48.91	3.0

Table 3 shows statistics on shared data references and their characteristics [GHARACHORLOO95]. Let R_i^s and W_i^s denote the number of shared READs and shared WRITEs, respectively. Based on the statistics of table 3, $\sum R_i^s = 540.392$, and $\sum W_i^s = 66.318$ for an aggregate R/W ratio of 8.1 on shared memory references. These numbers imply that only a small (1.37%) percentage of all memory references are shared WRITEs. (89.1%) of shared references are READ accesses.

If 10% of all memory references are WRITE, which is likely the case for multiprocessors [HWANG84], then a greater focus must be placed on the READ capabilities of multiprocessor computers since it holds 90% of all memory references. Let us emphasize that 10% of WRITEs are composed of shared and local WRITEs. Typical multiprocessor applications have 1 to 5% shared WRITEs, and therefore, less emphasis can be placed upon WRITE operations.

By this analysis, we could say that the upper limit for the number of WRITE memory references in a typical application, is 50% of the total memory references, while the lower limit could be very close to zero. Since the total number of WRITEs can not exceed 50% of the total number of memory accesses, then, READ operations

have a lower limit of 50% with an upper limit of close to 100% of all memory accesses.

Let A_i denote the number of memory accesses by processor i in a given time interval. If W_i and R_i denote the number of READ and WRITE references respectively during that interval, then

$$A_i = W_i + R_i, \tag{4}$$
$$1 \leq i \leq N.$$

Thus, for all N processors in the system,

$$\sum_{i=1}^{N} A_i = \sum_{i=1}^{N} (R_i + W_i) \tag{5}$$

Thus,

$$\sum_{i=1}^{N} A_i = \sum_{i=1}^{N} R_i + \sum_{i=1}^{N} W_i \tag{6}$$

Theorem 1 : *Visibility of Shared Updates: For a shared-memory update to be visible to at least one other processor, the following inequality holds:*

$$\sum_{i=1}^{N} W_i \leq \frac{\sum_{i=1}^{N} A_i}{2} \tag{7}$$

Proof: (by contradiction)
Assume $\sum_{i=1}^{N} W_i > \frac{\sum_{i=1}^{N} A_i}{2}$. This implies that at least one superfluous WRITE occured then by equation 6 we have, $\sum_{i=1}^{N} R_i < \sum_{i=1}^{N} W_i$. This implies that data has been written unnecessarily since it is never read. □

Corollary 1 : *Cummutative READ vs. WRITE Ratio: Since theorem 6 dictates*

$$\sum_{i=1}^{N} W_i \leq \frac{\sum_{i=1}^{N} A_i}{2} \tag{8}$$

and substituting 6

$$\sum_{i=1}^{N} A_i = \sum_{i=1}^{N} W_i + \sum_{i=1}^{N} R_i$$

in equation 8, then

$$\sum_{i=1}^{N} W_i \leq \frac{\sum_{i=1}^{N} W_i + \sum_{i=1}^{N} R_i}{2} \tag{9}$$

$$2\sum_{i=1}^{N} W_i \leq \sum_{i=1}^{N} W_i + \sum_{i=1}^{N} R_i \tag{10}$$

Or

$$\sum_{i=1}^{N} W_i \leq \sum_{i=1}^{N} R_i \tag{11}$$

Thus, the number of READs is always at least as large as the number of WRITEs, yielding the READ vs. WRITE ratio,

$$\frac{\sum_{i=1}^{N} R_i}{\sum_{i=1}^{N} W_i} \geq 0.50 \tag{12}$$

Indicating that READs are more important to optimize, than WRITEs. □

Equation 12 expresses that the total number of WRITE references can not be greater than 50% of all references. Therefore, the number of READs will constitute at least above 50% of the total memory references. Because of the nature of sharing in multiprocessors, this lower limit of number of READs will be well above 90%. Thus, it is time to place greater emphasis on READ operations.

Our design objective which is ambitious though feasible, is to attain zero memory access overhead on all READ operations. Replication is the most feasible solution to this problem, however it is not an economical approach. To design a cost effective high performance multiprocessor, a combination of replication and multiport memories will offer an acceptable solution.

4.4 Summary

In depth research of memory reference behavior has inspired us with new ideas about multiprocessor architecture. WRITE operations can account for 0% to 50% of all memory references. Since READ operations constitute the majority of memory references, improvement of READ references has a great impact on overall multiprocessor performance.

5 A Scalable Replicated Concurrent-Read Memory Model

To evaluate the Replicated Concurrent-Read (RCR) architecture, an analytical model was developed. In order to compare the RCR approach with existing architectures, models were also developed for typical Uniform Memory Access (UMA), and Non-Uniform Memory Access (NUMA) machines along with the Local-Remote-Global (LRG) multiprocessor. Since hardware configurations of computer systems could vary among different models, consistent assumptions have been made.

5.1 Assumptions

It is assumed that the workload will be evenly distributed between processors in all models. This implies that each processor has the same chance to perform READ/WRITE to *shared/private* data. If there is a READ miss, a block of data will be transfered to the cache. In case of a WRITE request, *write-through* policy will be assumed. For the sake of simplicity, it is assumed that block sizes are the same between all levels of caches. The models do not employ the concept of finite population of Processing Elements. In other words, in these models, all of N processors could make N memory references at every clock cycle. Thus, no processor is idle at any given time due to competition for accessing memory. Developed models project the expected memory access time for each architecture.

5.2 RCR Analytical Model

In this architecture, every processor is attached to a private cache, a local cache, and a replicated memory module as shown in Figure 26.

A READ hit fetches data from the cache in t_c time or in a one clock cycle. Let h_c denote the probability of a cache hit. A READ miss with the probability of $(1 - h_c)$ will cause an access to replicated memory in search of a requested word. The time

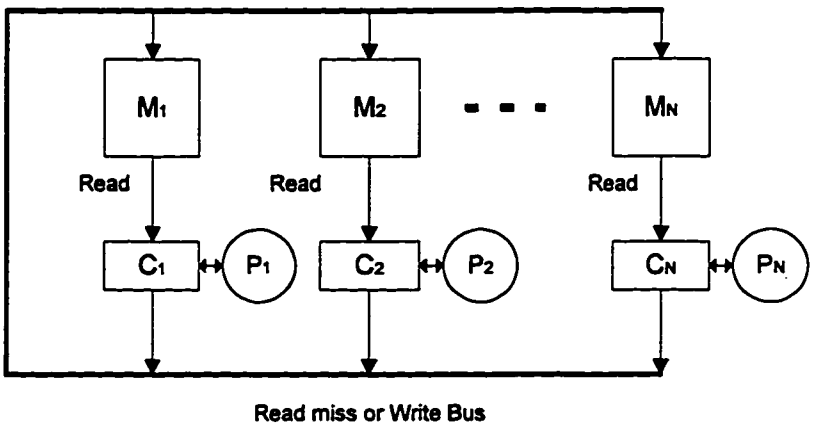

Figure 26: The RCR configuration.

required to access such a word is simply the replicated memory cycle time, t_M. Let h_R denote the probability of a replicated memory hit. In the case of a miss, the processor has to access the auxiliary memory in t_{aux} time. The auxiliary memory space is partitioned into addresses in the working set and addresses outside the working set. Replicated memories are images of the working set. A reference to auxiliary memory addresses are distinguished between isolated memory accesses and those which are incrementally outside the working set by some distance δ such that $L-\delta \leq X \leq H+\delta$ where L and H are the lowest and highest addresses currently stored in replicated memory, respectively. If an access is to an isolated memory address, then the requested word will be directly transfered from the auxiliary memory to the processor immediately. In the case of a reference being Incrementally Outside the Fence (IOF) such that $X \leq L-\delta$ or $X > H+\delta$, a quantity of D_{IOF} words will be transfered to the processor. Let P_{IOF} denote the probability of a memory access being incrementally outside the fence.

The time it takes to fetch a word from the auxiliary memory also depends on the global bus traffic. Every processor is equally likely to reference memory address space. The chance of a READ memory access is $(1-h_c)(1-h_R)$. A processor may have to wait for other READs or WRITEs to complete their execution. The time penalty associated with the duration of wait, depends on the number of pending READs and WRITEs in the bus queue. Let N and P_i denote the total number of processors and processor # i respectively. At any given time, the number of pending memory accesses is between $0-N$ inclusively. As a result, P_i may have to wait for $0-(N-1)$ memory references to be completed before its turn. Let $t^{RCR}_{wait\_global\_bus}$ denote the wait time. The WRITE access time depends on whether it is a shared or private WRITE. A private WRITE takes place in t_c time to the private cache of the writing PE. On the other hand, Shared WRITEs are broadcasted to all replicated memories via the global bus. A WRITE to memory takes place in t_w time. Let t^{RCR}_{read} denote the average READ time which be expressed as the sum of products for each access type and time mentioned above:

$$\begin{aligned} t^{RCR}_{read} &= t_c + (1-h_c)t_M B + (1-h_c)(1-h_R)[t^{RCR}_{wait\_global\_bus} \\ &\quad +(P_{IOF})t_{aux}D_{IOF} + (1-P_{IOF})t_{aux}] \end{aligned} \tag{13}$$

Equation 13 could be simplified as:

$$\begin{aligned} t^{RCR}_{read} &= t_c + (1-h_c)t_M B + (1-h_c)(1-h_R)[t^{RCR}_{wait\_global\_bus} \\ &\quad +t_{aux}[P_{IOF}(D_{IOF}-1)+1]] \end{aligned} \tag{14}$$

The value of $t^{RCR}_{wait\_global\_bus}$, depends on the number of pending accesses in the queue and the time penalty associated with it. This can be approximated by scaling the probability of each individual processor requesting global access by the expected number of requests pending from other processors which is $\frac{N-1}{2}$. The time penalty with $\frac{N-1}{2}$ accesses in the queue depends on the characteristics of each access:

1. whether the memory access is a READ or WRITE,

2. whether a requested word is in incrementally outside the fence or it is in isolated memory space.

The probability of a READ memory reference to be in the queue is $(1-h_c)(1-h_R)$. Let P_{shared_write} denote the probability of a shared-write memory access such that $P_{shared\_write} + P_{private\_write} + P_{read} = 1$. The value of $t^{RCR}_{wait\_global\_bus}$ may be expressed as:

$$\begin{aligned} t^{RCR}_{wait\_global\_bus} &= \frac{N-1}{2}\{P_{shared\_write}t_w + (1-h_c)(1-h_R)P_{read} \\ &\quad [P_{IOF}D_{IOF}t_{aux} + (1-P_{IOF})t_{aux}]\} \end{aligned} \tag{15}$$

Equation 15 may be simplified as:

$$\begin{aligned} t^{RCR}_{wait\_global\_bus} &= \frac{N-1}{2}\{P_{shared\_write}t_w + (1-h_c)(1-h_R)P_{read} \\ &\quad [t_{aux}(P_{IOF}(D_{IOF}-1)+1)]\} \end{aligned} \tag{16}$$

Hence, the t^{RCR}_{read} may be expressed as:

$$\begin{aligned} t^{RCR}_{read} &= t_c + (1-h_c)t_M B + (1-h_c)(1-h_R)[\frac{N-1}{2}[P_{shared\_write}t_w + \\ &\quad (1-h_c)(1-h_R)P_{read}[t_{aux}(P_{IOF}(D_{IOF}-1)+1)]] \\ &\quad + t_{aux}[P_{IOF}(D_{IOF}-1)+1]] \end{aligned} \tag{17}$$

The WRITE access time depends on whether it is a shared or private WRITE. A private WRITE takes place in t_c time while a shared WRITE is broadcasted to all replicated memories via the global bus. In the case of a shared WRITE, P_i may have to wait for $0-(N-1)$ memory references to be completed before its turn. The average WRITE time, t^{RCR}_{write}, may be written as:

$$t_{write}^{RCR} = P_{shared\_write}[t_{wait\_global\_bus}^{RCR} + t_w] + (1 - P_{shared\_write})t_c \quad (18)$$

By substituting the value of calculated $t_{wait\_global\_bus}^{RCR}$ in the equation 18, t_{write}^{RCR} becomes:

$$\begin{aligned} t_{write}^{RCR} &= P_{shared\_write}[\frac{N-1}{2}[P_{shared\_write}t_w + (1 - h_c)(1 - h_R)P_{read} \\ &\quad [t_{aux}(P_{IOF}(D_{IOF} - 1) + 1)] + t_w] + (1 - P_{shared\_write})t_c \end{aligned} \quad (19)$$

Let t_{ave}^{RCR} be the average memory access time. The overall expected memory access time depends on the percentages of READ/WRITE operations with respect to total memory references. By adding average READ/WRITE access time with respect to probability of an access being READ/WRITE, the overall expected memory access time becomes:

$$t_{ave}^{RCR} = P_{read}t_{read}^{RCR} + (1 - P_{read})t_{write}^{RCR} \quad (20)$$

5.3 UMA Analytical Model

In the UMA machine, all processors share a global memory module, while every processor is attached to a local cache and a seperate private cache. Every memory access starts with searching data in either a local cache or private cache, depending upon the address of the access. In the case of a miss, global memory is accessed. Let h_c denote the cache hit rate. A READ hit fetches data directly from the cache in t_c time, if there are no pending WRITEs since only one invalidation can occur at a time. The probability of having to wait for a pending write to complete, may be expressed as P_{shared_write}. The time penalty involved depends on the number of pending WRITEs in the queue. Let $t_{wait\_pending\_write}^{UMA}$ denote the time a processor may have to wait before accessing data from the cache. A READ miss requires checking the global memory for a copy of the data. The time required to access any such data is simply the global memory cycle time, t_m. The chance of having to wait for another memory reference to complete its execution is $(1 - h_c)$. The wait time also

depends on the number of pending memory accesses. Let $t^{UMA}_{wait\_global\_bus}$ denote the time a processor may have to wait for accessing global memory. The average READ time may be expressed as:

$$t^{UMA}_{read} = t^{UMA}_{wait\_pending\_write} + t_c + (1 - h_c)[t^{UMA}_{wait\_global\_bus} + Bt_m] \tag{21}$$

$t^{UMA}_{wait\_pending\_write}$ can be approximated by scaling the probability of each individual processor requesting global access by the expected number of requests pending from other processors which is $\frac{N-1}{2}$. Thus, the expected value of $t^{UMA}_{wait\_pending\_write}$ may be calculated as follows:

$$t^{UMA}_{wait\_pending\_write} = \frac{N-1}{2}[P_{shared\_write}t_c] \tag{22}$$

The waiting time for global bus is dependent on whether other processors are trying to access global memory. $t^{UMA}_{wait\_global\_bus}$ may be expressed as:

$$t^{UMA}_{wait\_global\_bus} = \frac{N-1}{2}[P_{shared\_write}t_m + (1 - h_c)P_{read}Bt_m] \tag{23}$$

A private WRITE takes place in t_c time plus waiting time for any other pending WRITEs. Any update to shared-data, also requires updating the global memory. As a result, any shared-write memory accesses may have to wait for global traffic. The average WRITE time may be expressed as:

$$\begin{aligned} t^{UMA}_{write} = {} & P_{shared\_write}(t_c + t^{UMA}_{wait\_pending\_write} + \\ & t^{UMA}_{wait\_global\_bus} + t_m) + (1 - P_{shared\_write}) \times \\ & (t_c + t^{UMA}_{wait\_pending\_write}) \end{aligned} \tag{24}$$

The overall memory access time is dependent on the probabilty of an access being READ or WRITE. Let t_{Ave} denote the overall memory access time. t_{Ave} may be expressed as:

$$t_{ave}^{UMA} = P_{read} t_{read}^{UMA} + (1 - P_{read}) t_{write}^{UMA} \quad (25)$$

As seen in equation 25, the expected memory access time is the total of average READ/WRITE time with respect to the probability of an access being READ/WRITE.

5.4 NUMA Analytical Model

Since global memory is distributed among processors, every processor is attached to its own private cache and a memory module, which is considered as local memory for the respected processor. If an access is not satisfied by the cache, then distributed shared-memory modules will be accessed. A READ hit is answered in t_c time plus the waiting time for pending invalidation WRITEs, if any. A READ miss requires checking the distributed memory modules. Let P_L denote the chance of having requested data in the local memory module. Fetching a word from local memory takes place in t_L time plus the time spent in local memory traffic. If the data is in a remote memory module, then the time required to fetch a word is t_{Remote} plus the waiting time in global traffic. Let $t_{wait\_pending\_write}^{NUMA}$ denote the time a processor may have to wait for accessing the cache. $t_{wait\_pending\_write}^{NUMA}$ depends on the number of pending invalidation WRITEs. Let $t_{wait\_local\_bus}^{NUMA}$ denote the time a processor may have to wait for accessing local memory. Finally, let $t_{wait\_global\_bus}^{UMA}$ be the time spent in global traffic. The average READ time may be expressed as:

$$\begin{aligned} t_{read}^{NUMA} &= t_{wait\_pending\_write}^{NUMA} + t_c + (1 - h_c)[P_L(t_{wait\_local\_bus}^{NUMA} + \\ &\quad Bt_L) + (1 - P_L)(t_{wait\_global\_bus}^{NUMA} + Bt_{Remote})] \end{aligned} \quad (26)$$

The waiting time for *pending_write* times depends if other processors are trying to invalidate a shared-word in the private cache. Also the number of *pending_write* accesses in the queue effects the duration of waiting time to access the cache. This can be approximated by scaling the probability of each individual processor requesting

global access by the expected number of requests pending from other processors which is $\frac{N-1}{2}$. Thus, $t^{NUMA}_{wait\_pending\_write}$ could be calculated as:

$$t^{NUMA}_{wait\_pending\_write} = \frac{N-1}{2}(P_{shared\_write}t_c) \tag{27}$$

The waiting time for local bus depends on the number of remote accesses by other processors and characteristics of those remote references. Thus, $t^{NUMA}_{wait\_local\_bus}$ may be expressed as:

$$t^{NUMA}_{wait\_local\_bus} = (\frac{N-1}{2})[P_{shared\_write}t_L + (1-h_c)(1-h_L)P_{read}Bt_L] \tag{28}$$

where h_L is the local memory hit rate.

The waiting time for global bus is the function of the number of remote accesses in the global traffic and also local traffic in a specified remote memory module. The characteristics of pending accesses in local and global traffic is also a factor in the calculation of waiting time for global bus. Thus, $t^{NUMA}_{wait\_global\_bus}$ could be calculated as:

$$\begin{aligned} t^{NUMA}_{wait\_global\_bus} &= (\frac{N-1}{2})[P_{shared\_write}t_{Remote} + (1-h_c)(1-h_L)P_{read}Bt_{Remote}] + \\ &\quad P_L(P_{shared\_write}t_L + (1-h_c)P_{read}Bt_L) \end{aligned} \tag{29}$$

A private WRITE access takes place in t_c time plus the waiting time in the cache queue. A shared-write also requires updating the distributed memory module. The average WRITE access may be expressed as:

$$\begin{aligned} t^{NUMA}_{write} &= P_{shared\_write}[t_c + t^{NUMA}_{wait\_pending\_write} + h_L(t^{NUMA}_{wait\_local\_bus} + t_L) + \\ &\quad (1-h_L)(t^{NUMA}_{wait\_global\_bus} + t_{Remote})] + (1 - P_{shared\_write}) \times \\ &\quad (t^{NUMA}_{wait\_pending\_write} + t_c) \end{aligned} \tag{30}$$

The overall memory access time may be obtained by combining expected READ and

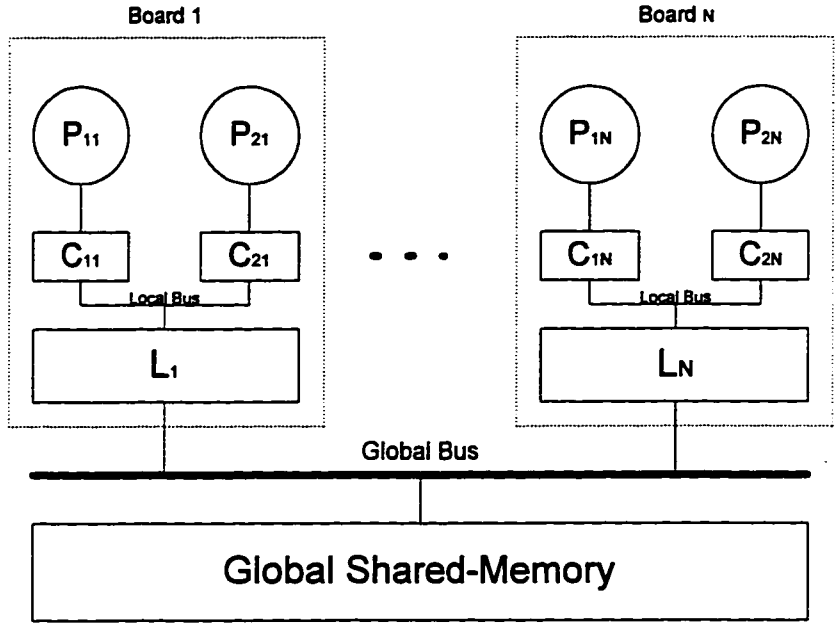

Figure 27: LRG architecture.

WRITE access times with respect to the probability of an access being READ or WRITE. The expected memory access time may be expressed as:

$$t_{ave}^{NUMA} = P_{read} t_{read}^{NUMA} + (1 - P_{read}) t_{write}^{NUMA} \tag{31}$$

As seen in the equation 31, the overall access time is a function of average READ and WRITE times plus the probability of an access being READ or WRITE.

5.5 Local-Remote-Global Analytical Model

Local-Remote-Global (LRG) architecture consists of several clusters with every cluster containing two processors and a shared local memory as shown in Figure 27. Each processor on the board is attached to a private cache. The LRG architecture also provides a global memory module which is uniformly accessible by all processors.

A READ hit is answered in t_c time. A READ miss requires checking the local memory for a copy of the data. The time it takes to fetch a word from local memory is t_L plus the waiting time for the local bus. Let h_c and h_L denote the probability of cache and local memory hit rates, respectively. The chance of having to access global memory is $(1-h_c)(1-h_L)$. Let $t^{LRG}_{wait\_global\_bus}$ denote the waiting time for the global bus. The average READ time may be expressed as:

$$\begin{aligned} t^{LRG}_{read} &= t_c + (1-h_c)(t^{LRG}_{wait\_local\_bus} + t_L) + \\ &\quad (1-h_c)(1-h_L)[t^{LRG}_{wait\_global\_bus} + Bt_m] \end{aligned} \tag{32}$$

The waiting time for the local bus is dependent on whether the other processor on the board is accessing the local memory and also on the characteristics of its access. A processor will access local memory if:

1. the processor is performing a *shared_write*, and
2. there is a cache *read-miss*.

$t^{LRG}_{wait\_local\_bus}$ may be expressed as:

$$t^{LRG}_{wait\_local\_bus} = \frac{1}{2}[P_{shared\_write}t_L + (1-h_c)P_{read}Bt_L] \tag{33}$$

The waiting time for global bus is dependent on the number of pending global memory accesses by other processors and characteristics of those accesses in the queue. A processor will access the global memory if:

1. the processor is performing a *shared_write*, and
2. there is a *read-miss* on local memory.

$t^{LRG}_{wait\_global\_bus}$ could be calculated as:

$$t^{LRG}_{wait\_global\_bus} = \frac{N-1}{2}[P_{shared\_write}t_m + (1-h_c)(1-h_L)P_{read}Bt_m] \quad (34)$$

The average WRITE time is dependent on whether a WRITE is on a private or shared word. A private WRITE takes place in t_c time. A shared WRITE requires also updating local and global memories. The average WRITE time may be expressed as:

$$t^{LRG}_{write} = P_{shared\_write}\{t_c + P_L(t^{LRG}_{wait\_local\_bus} + t_L) + (1-P_L)(t^{LRG}_{wait\_global\_bus} + t_m)\} + (1-P_{shared\_write})t_c \quad (35)$$

The overall memory access time is dependent on average READ and WRITE times with respect to the probability of an access being READ or WRITE. Let t^{LRG}_{ave} be the expected memory access time, t^{LRG}_{ave} may be expressed as:

$$t^{LRG}_{ave} = P_{read}t^{LRG}_{read} + (1-P_{read})t_{write} \quad (36)$$

Equation 36 reflects the fact that the average memory access time is directly function of number of memory accesses with respect to their characteristics at any given time.

5.6 Summary

By employing a preset distribution of memory accesses throughout the address space, analytical models were developed. The READ time in the RCR configuration is proportional to replicated memory hit rate. The READ time in a UMA architecture is strictly greater than that of the RCR architecture because UMA configurations must wait for global bus for all READs that are not satisfied by cache. The READ time in a NUMA machine depends upon the location of the requested word whether it is:

- in local distributed memory module, or
- in a remote memory module.

The expected READ time in the RCR architecture is less than NUMA configuration since the RCR performs majority of its READs locally. The READ time in LRG configuration is greatly dependent on local memory hit rate since READ misses are very costly. The RCR architecture with replicated memory hit rate of 80% and above has less READ time than LRG configuration with the same hit rate.

6 A Scalable Replicated Concurrent-Read Architecture

Demand to design a high performance computing system, capable of handling today's complicated engineering problems, continues to grow exponentially. In recent years, processing element's speed and overall performance has reached a point in which physical limitations restrict further improvements. This has captivated a great amount of attention to the re-evaluation of current memory hierarchy systems.

6.1 Overview

One of the major factors which effects the performance of a distributed shared-memory system is the strength of its memory hierarchy design. The primary memory hierarchy design goal is to increase the effective memory bandwidth so that more memory words can be referenced per unit time. Previous designs provided high-speed caches to every processor to alleviate processor memory bandwidth mismatches. While this approach was impelling, it had an undesirable side-effect of data inconsistency, which resulted in more complex hardware designs in order to secure correctness of execution. On the other hand, interconnection network systems started to experience a new problem called contention.

In this chapter, a new approach in multiprocessor design will be introduced. The Replicated Concurrent-Read (RCR) architecture demonstrates a unique characteristic in dealing with READ and WRITE operations. This design has not inherited the short comings of previous approaches while remaining simple, cost-effective, and a considerably scalable system.

6.2 Hardware Design

The foundation of the proposed design is based on extensive investigation of previously designed memory hierarchy architectures. The proposed concept increases efficiency of READ operations while decreases deficiencies of previous designs. To achieve this goal, two major components of multiprossesors were the center of our consideration, Memory units and interconnection network system. These components had to be designed in such a way that optimizes the READ operation while the probability of data inconsistency and contention is minimized.

6.2.1 Memory Units and the Interconnection Network

Multiported memories have been developed to support concurrent access to memory. Availability of these memory units would allow us to READ and WRITE the same memory unit simultaneously. By assigning each memory port to only one type of memory reference operation, it is possible to READ through one port while updating data through the other. In our proposed shared-memory multiprocessor system, each processor will be connected to its own dual-port memory unit. Thus, it is possible to read N different memory words by N processors simultaneously in one memory cycle with no delay. WRITE operations are accomplished by broadcasting over a system bus which will be connected to the other port of these memory units. This technique will allow us to WRITE to all memory units simultaneously, therefore, eliminating the cache coherence problem. Together, the dual-port high-speed memories form a global address space available to all processors simultaneously.

Since every memory unit will have a designated port for READ operations, then each processor could perform READ locally with zero overhead. All of processors will be connected to a system bus which is also connected to the other port of the memory units. In this fashion, we could broadcast all the WRITE operations. Therefore, this system will perform write operations in O(1) time while READs are done locally.

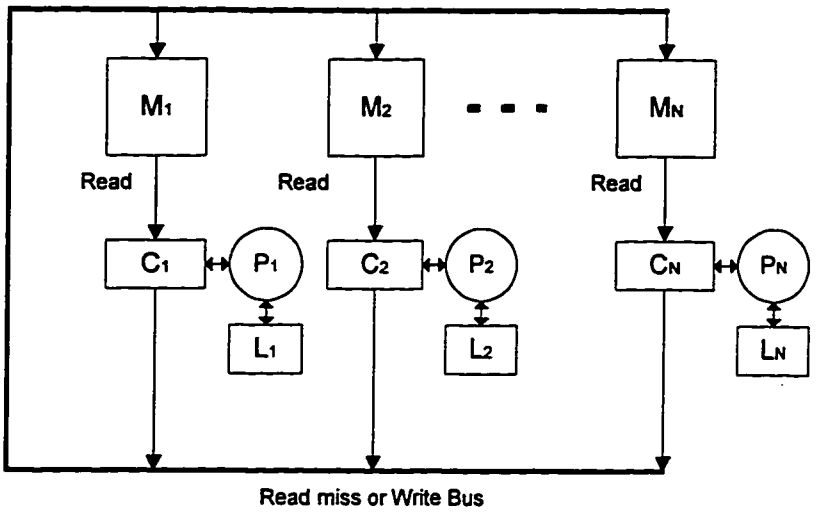

Figure 28: Basic RCR Architecture.

Thus, all memory units are replications of each other. Figure 28 illustrates the basic components of this design.

Multiported memories, currently available on the market, are an essential factor in the cost-effectiveness of this design. A dual-port Static RAM (SRAM), with an access time of 10 ns, provides two independent ports with separate control, address, and I/O pins that permit independent and asynchronous access for READ or WRITE to any location in the memory. Currently dual-port semiconductor memories are available in a wide range of speeds. Their cycle times range from a few hundred nanoseconds (ns) to less than 10 nanoseconds. These memory modules are in a wide range of sizes up to 1M. The RCR design employs $16K \times 32$ dual port static RAM module. This module holds $16K$ words of 32 bits. The replicated memory module is packaged in a ceramic 121 pin PGA (PinGridArray) 1.35 inches on a side.

6.2.2 Auxiliary Memory Unit

Since every memory unit is a replication of each other, the expansion of the shared memory space will be costly, and have an effect on the scalability of the system. Thus, by providing an auxiliary Dynamic RAM memory unit, which is part of general shared-memory space, we could increase scalability of the system. By having a controller chip monitoring the addresses that are being accessed, we could move blocks of data in or out of replicated memories based on the spatial locality of reference. Since more than 90% of all references will be done locally, then, the movement of these blocks of data will not cause or add to system bus contention. It only increases the effective utilization of the system bus. Figure 29 exhibits the memory configuration of the Replicated Concurrent-Read system. As shown in Figure 29, total global memory space is consist of replicated space and auxiliary space. Each processor is equally close to every memory word in global memory. Every processor is attached to a local memory, so that private data and programs could be stored.

Addition of auxiliary memory will effect two essential factors in the Replicated Concurrent-Read shared-memory system design:

1. It will increase scalability of the system, and

2. It will increase cost-effectiveness of the system.

Increasing size of replicated memories will effect scalability of the system, thus, by adding a spatial cache (auxiliary memory), the system maintains its scalability. Since replicated memories are expensive static RAMs increasing the size of these memory units is very costly. Thus, the addition of inexpensive auxiliary memory (DRAMs) is cost-effective, and at the same time, it will not significantly increase expected memory access time because of the spatial locality of reference, which will be shown later in this section.

Figure 30 exhibits the architecture of the Replicated Concurrent-Read system. A private READ/WRITE is done locally by a processor with no delay. A shared READ

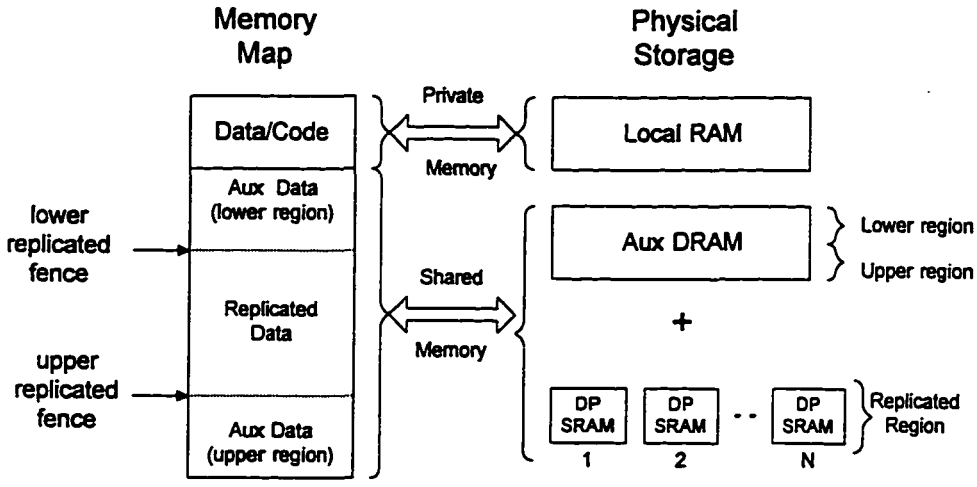

Figure 29: Memory Configuration of Replicated Concurrent-Read architecture.

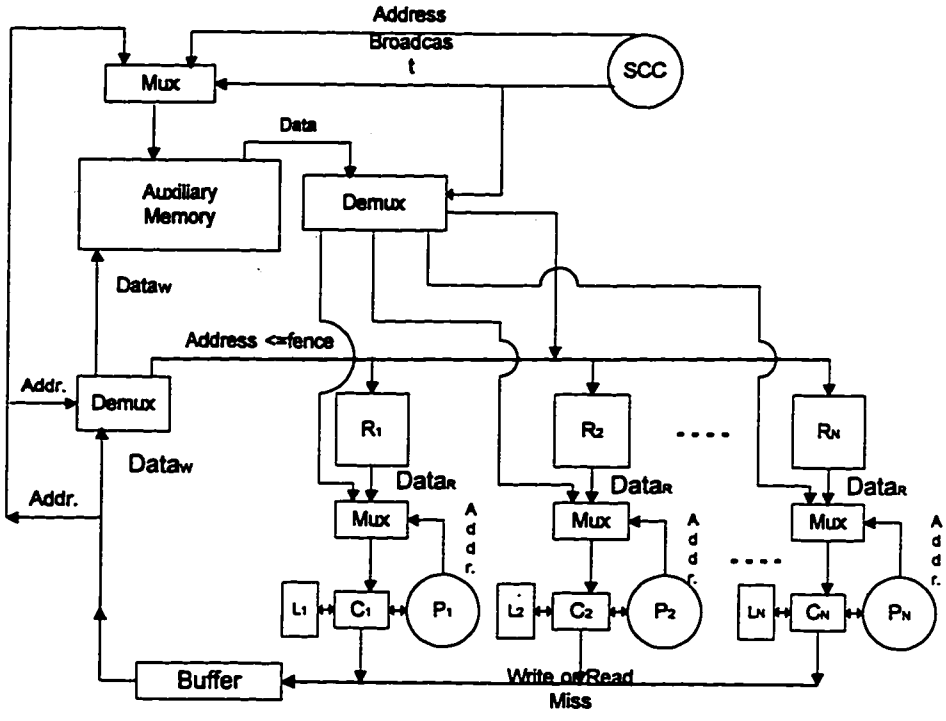

Figure 30: Replicated Concurrent-Read architecture.

will take place in one clock cycle since each processor is attached to a replicated memory module. A READ miss will cause an access to auxiliary memory through *write or read bus*. If the word is located incrementally outside the fence then a block of data will be transfered to all replicated memories, otherwise the intended word will be loaded to the requesting processor immediately. Simultaneously, the spatial cache controller (SCC) is monitoring all addresses being accessed, so that it can broadcast blocks of data to replicated memories in slow traffic time. In the case of shared WRITEs, all replicated memories will be updated cocurrently. A demux is attached to *write or read bus*, so that it determines whether the address is in auxiliary memory or it is in replicated meories as shown in in Figure 30.

In order to calculate the cost decrease by employing the spatial cache, we need to compute the total memory system cost with and without the auxiliary memory. Let M_{rep} denote the percentage of shared memory that is spatially cached. Let M, and N

denote shared memory size and number of processors respectively, and also, let C_S, and C_D denote cost in \$/word for SRAM, and cost in \$/word for DRAM respectively. Then, with a spatial cache, the memory system cost will be:

Memory System Cost = Cost Spatial cache + Cost all Replicated Memories

Let C_M denote the memory system cost then:

$$C_M = [M_{rep} \times M \times C_D] + [(1 - M_{rep}) \times M \times N \times C_S] \tag{37}$$

As illustrated in Figure 30, total memory includes N replicated memory units with size M plus spatial cache, thus the memory system cost is the total of the spatial cache and all replicated memories. If we do not employ the spatial cache, then the memory system cost will be:

$$C_M = M \times N \times C_S \tag{38}$$

As shown in Figure 28, the total memory consists of only high-speed replicated memory units. Since SRAMs are four to eight times more expensive than DRAMs (table 1), the memory system cost will be a function of shared memory size, number of processors, and percentage of shared memory that is spatially cached. By subtracting equation 37 from equation 38, we can calculate amounts of cost savings. Let C_{saving} denote the cost savings, then

$$C_{saving} = M \times N \times C_S - [M_{rep} \times M \times C_D + (1 - M_{rep}) \times M \times N \times C_S]$$

by factoring out the M, we have

$$C_{saving} = M \times [N \times C_S - M_{rep} \times C_D - (1 - M_{rep}) \times N \times C_S]$$

we can also factor out $N \times C_S$

$$C_{saving} = M \times [N \times C_S \times (1 - (1 - M_{rep})) - M_{rep} \times C_D]$$

by further simplification, we have

$$C_{saving} = M \times [N \times C_S \times M_{rep} - M_{rep} \times C_D]$$

Table 5: Cost Savings Factor For Various Numbers of PEs.

M_{rep}	$N = 4$	$N = 8$	$N = 16$	$N = 32$	$N = 64$	$N = 128$
0.1	9.6%	9.8%	9.9%	9.9%	10.0%	10.0%
0.2	19.2%	19.6%	19.8%	19.9%	19.9%	20.0%
0.5	47.9%	49.0%	49.5%	49.7%	49.9%	49.9%
0.9	86.2%	88.1%	89.1%	89.5%	89.8%	89.9%

by factoring out M_{rep}, we have

$$C_{saving} = M \times M_{rep} \times [N \times C_S - C_D] \tag{39}$$

Equation 39 calculates the amount of cost savings associated with the total memory system cost for a different percentage of shared memory that is spatially cached. As a result, we can design the memory system with minimal required expenses while optimizing the memory access operations. Since WRITE bus is not utilized 100% of the time, then copying blocks of data in and out of the auxiliary memory will not effect the overall performance of the memory system.

Let's assume the shared-memory size (M) is 1 Gbyte. Since the cost of SRAM is approximately 4 to 8 times higher than DRAM (table 1), then let's assume the cost of DRAM is 6 times higher than SRAM. If DRAM = 1, then SRAM = 6. Table 5 shows the cost savings for different percentages of shared memory that is spatially cached with respect to a different number of processors.

The Replicated Concurrent-Read (RCR) architecture is a hardware solution for Scalable multiprocessors, under the shared bus category. This system offers an effective memory reference mechanism. Any processor can broadcast the data through the bus and update the related block in the entire shared-memory space with a constant time complexity. With this capability of the system, the private memories become the exact replica of one another. Thus, consistency of the shared data is guaranteed at

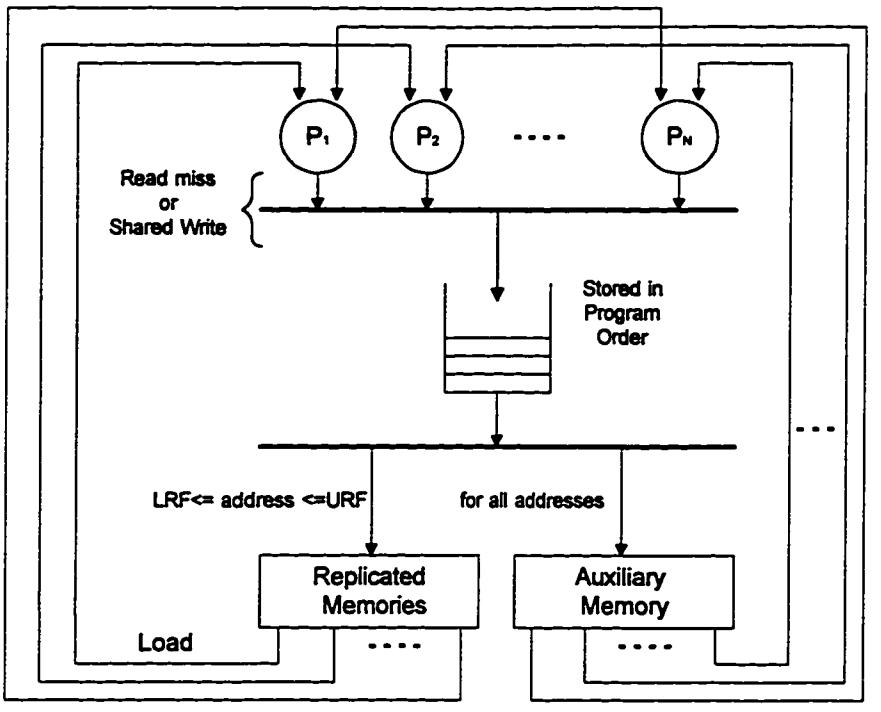

Figure 31: RCR memory consistency model.

any given time. Figure 31 shows the Replicated Concurrent-Read system's memory consistency model.

The Replicated Concurrent-Read architecture offers several distinct advantages over existing multiprocessors:

1. Cost-effectiveness and simplicity of design,
2. Zero overhead memory *read* operations, and
3. Data consistency by broadcasting updates globally with a constant time complexity.

6.3 Multiport Memory Replication Characteristics

Performance of a distributed shared-memory system greatly depends on the characteristics of its memory modules. Since advancement of memory technology has

provided designers with the option of fast multiported memories, concurrent memory references have been feasible. It is essential to analyze the performance of systems employing multiported memories with different numbers of ports.

Multiported memories provide a means to READ concurrently different or same memory locations by various numbers of processors. The maximum number of words that can be read simultaneously in one memory cycle is equal to the number of processors in the system. Thus, as the number of processors grows larger, the capability of referencing different memory locations increases linearly. The number of memory ports assigned for *read* operations has no effect on the total number of concurrent READs, since any processor can issue only one memory reference instruction per machine clock cycle.

The capability of performing more than one WRITE operation simultaneously in only one memory cycle, depends on the number of ports that are designated for WRITE references. The Replicated Concurrent-Read system utilizes dual-ported memory units. This architecture allows for only one WRITE operation without overhead in every memory cycle by one of the processors. Figure 32 illustrates an architecture using dual-ported memories.

If we replace these dual-ported memories with 4-ported memory units, it will increase the number of concurrent WRITEs in one memory cycle by two. Figure 33 demonstrates a system with 4-ported memories. Since there will be only one port assigned to READ operations, the rest of ports will be exclusively for WRITE memory accesses.

Let d denote the number of ports in a multiported memory unit, then it will be possible to perform $(d-1)$ concurrent WRITEs without overhead. This increase in number of concurrent WRITEs is not cost free since the system will require more system buses to perform simultaneous WRITE memory references. A d-ported system will require $(d-1)$ system buses in order to perform $(d-1)$ simultaneous WRITEs

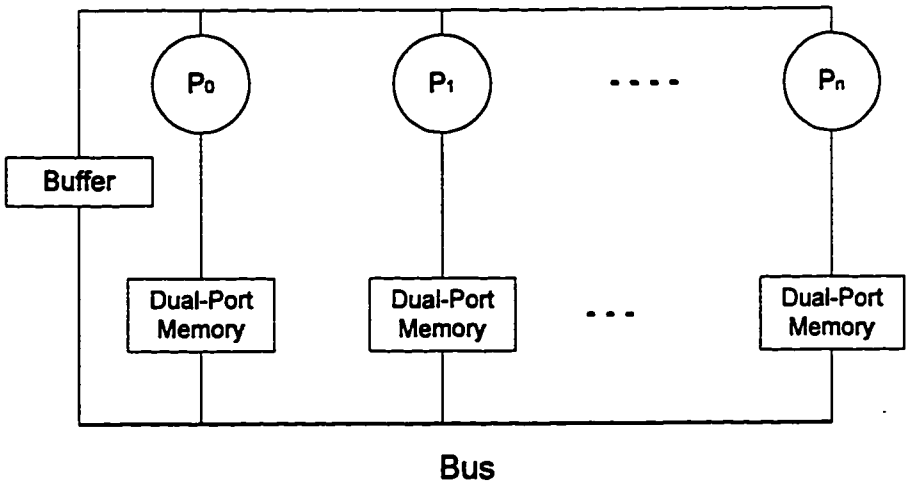

Figure 32: Dual-Port Memory.

as exhibited in Figure 34.

While increasing the number of ports illustrates an improvement in the number of concurrent WRITEs, it has a direct relationship with hardware complexity. Since the number of processors in the system is usually greater than the number of ports, this system will require demultiplexers to route the WRITE operations to different busses. This is because each bus is capable of accommodating only one WRITE in each cycle. This requirement is not necessary with the dual-ported memories.

Choosing the right multiported memory with respect to the number of ports greatly depends on the applications. An ordinary application's memory reference behavior demonstrates only 10% or less WRITE operations comparing to 90% READ operations. The speedup of different multiported systems will depend on the percentage of WRITE operations with the available number of ports and processors.

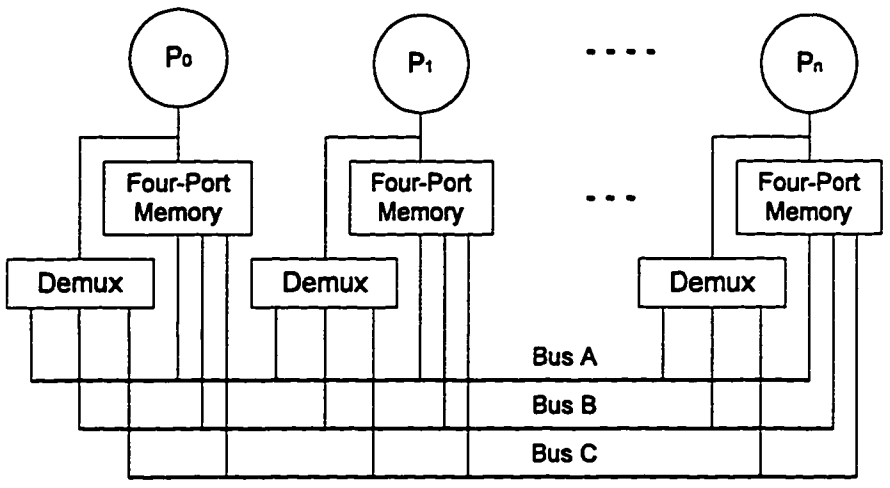

Figure 33: Four-Ported Memory RCR Architecture.

Therefore, it is possible to study scalability of such systems by the ability to calculate their speedup. Higher speedup will be observed when an application requires less WRITE operations in a system with a large number of ports.

6.3.1 Multiport Arbiter Design Characteristics

The number of concurrent WRITEs that can be performed in one memory cycle depends on the number of available ports in a multiport memory unit. Therefore, a d-ported system can accommodate only $(d-1)$ WRITEs simultaneously, where d is the number of ports in a multiported memory unit. Since a dual-ported system can grant bus access to only one processor at a time, there is a need to employ an arbiter to issue bus access grants to processors. Generally, as long as the number of processors in a system is greater than $(d-1)$ ports, there is a need for an arbiter to regulate the system bus accesses. The logical function of this arbiter can be realized using

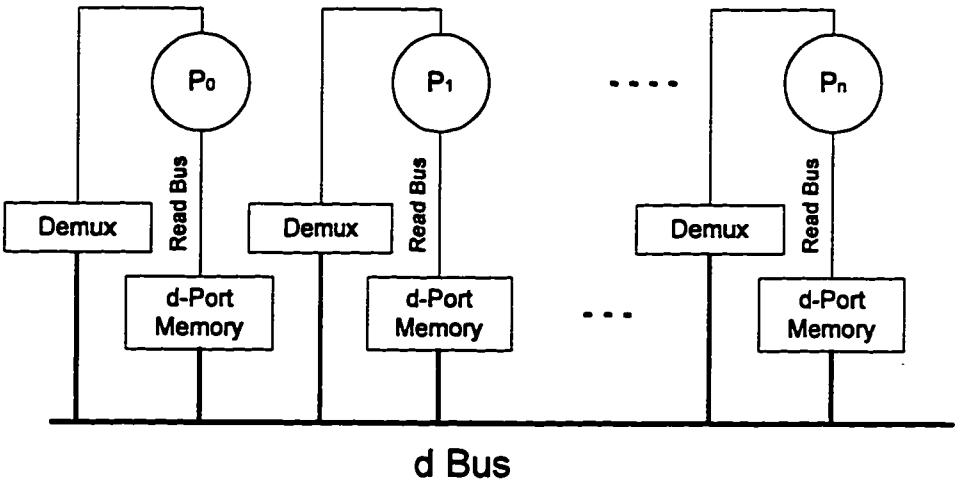

Figure 34: *d*-Ported Memory RCR Architecture.

off-the-shelf devices such as PLA, PAL, field-programmable ROM, or by designing a customized combinational logic system.

Any time a processor needs to write, it will have to request the arbiter for permission to access the WRITE bus. However, if there are no other requests in the queue, then permission will be granted immediately. Therefore, in a dual-ported memory system *expected delay* for bus access will be equal to the number of requests in the queue in terms of memory cycles. These requests can be handled based on first in first out (FIFO) or could be based on a priority scheme. If a WRITE request is on a shared data, then we may assign a higher priority to that request. The choice of a policy for the arbiter would greatly depend on the applications. In most applications, WRITE operations constitute only 10% of their total memory references, thus, a simple arbiter design is appropriate for a general purpose multiprocessor.

6.4 Multiport Memory Cycle Analytical Representation

The number of concurrent memory references, that can be performed simultaneously is directly dependent upon the number of ports in a multiported memory unit. In such a system, all processors can read simultaneously in one memory cycle with no delay. WRITE operations could experience some delay depending on the number of processors trying to access the WRITE bus and the number of ports designated for update operations. In the case of a dual-ported memory system, only one processor can perform a WRITE operation in each memory cycle. Thus, if two processors attempt to update their replicated memories, then one of them will experience one memory cycle of delay. Generally, the *expected delay* will depend on the exceeding number of WRITEs that can not be accommodated in one memory cycle. Thus, for a dual-ported memory, the number of needed cycles is equal to the number of WRITEs, and the *expected delay* would be the number of memory cycles left off after the initial execution.

By consideration of all these elements, we can formulate the needed memory cycles, and expected delays of WRITE operations using dual-ported memory units. Since, in a dual-port memory unit, there is only one port for WRITE purposes, the total number of required memory cycles is equal to the total number of WRITEs. Let M_c, and W denote the number of memory cycle and total number of writes respectively, then the total number of memory cycles needed for W WRITEs is:

$$M_c = W \tag{40}$$

where

$$W = W_0 + W_1 + W_2 + ... + W_n \tag{41}$$

where n is the number of processors that requesting WRITE access at any given time interval, $n = 0, 1, 2, ..., N$. Thus,

$$W = \sum_{i=0}^{n} W_i \tag{42}$$

where W_i is a WRITE by processor P_i, and N is the total number of processors in the system. Therefore, in a dual-ported memory system, the total number of memory

cycles needed to perform W WRITEs is:

$$M_c = \sum_{i=0}^{n} W_i \tag{43}$$

Therefore, at any give time, the total number of memory cycles needed to complete the total number of WRITEs is equal to the total number of WRITEs issued by all processors.

Since, in a dual-port memory only one WRITE access will be granted, there will be a delay in accessing the WRITE bus if there are more than one WRITE requests in one memory cycle. Let's assume that there are two WRITE requests, then one of the requests will be granted with no delay and the second one, with one memory cycle delay, will be granted. Thus, the expected *delay*, D, is equal to:

$$Delay = (0) \times D_0 + (1) \times D_1$$

where D_i is the delay related to W_i.
Thus, we can generalize this equation for W WRITEs at any given time. If there are n WRITEs, then there are $(n-1)$ memory clock cycles delay to accommodate all memory WRITE requests. Let D denote the expected delay, then

$$D = (0)D_0 + (1)D_1 + (2)D_2 + ... + (n)D_n \tag{44}$$

where

$$n = 0, 1, 2, ..., W$$

Equation 44 can be rewritten as

$$D = (1-1)D_0 + (2-1)D_1 + (3-1)D_2 + ... + (n-1)D_n \tag{45}$$

Thus, we can calculate expected delay for n WRITEs in terms of memory clock cycle as

$$D = \sum_{i=1}^{n} (i-1) \tag{46}$$

In case of a four-ported memory system, three WRITEs can be accommodated at each memory cycle, and an additional memory cycle will be required for every

multiple of three WRITEs thereafter. Therefore, we can formulate needed memory cycles, and expected delays for such a system in terms of memory cycles.

$$M_c = \lceil \frac{W_0 + W_1 + W_2 + ... + W_n}{3} \rceil \quad (47)$$

We can rewrite the equation 47 as

$$M_c = \lceil \frac{\sum_{i=0}^{n} W_i}{3} \rceil \quad (48)$$

Since $W = \sum_{i=0}^{n} W_i$, then

$$M_c = \lceil \frac{W}{3} \rceil \quad (49)$$

Equation 49, calculates the total number of memory cycles required to accommodate all memory WRITE access requests in a four-ported memory system.
Since three WRITEs can be performed in each memory cycle, the fourth WRITE request will experience one memory cycle delay. As the number of WRITEs grows larger, the number of delays will grow one cycle for every three WRITE requests. Therefore, we can formulate delays as below

$$D = (0)[D_0 + D_1 + D_2] + (1)[D_3 + D_4 + D_5] + ... + \lfloor (\frac{n}{3}) \rfloor [D_{n-2} + D_{n-1} + D_n] \quad (50)$$

We can rewrite the equation 50 as follow

$$D = \sum_{i=0}^{n} \sum_{j=0}^{2} \lfloor \frac{i}{3} \rfloor D_{3i+j} \quad (51)$$

Therefore, the expected delay time in terms of memory cycles at any given time interval is

$$D = \sum_{i=0}^{n} \sum_{j=0}^{2} \lfloor \frac{i}{3} \rfloor \quad (52)$$

As the number of ports increases, the ability to perform more concurrent WRITE operations grows larger. For d-ported memory systems, the maximum number of concurrent WRITEs is equal to $(d-1)$, the minus one indicates designation of one port to READ operations. If more than $(d-1)$ processors request to update memory, they will experience delays based on the number of requesting processors. We can

calculate the total number of required memory cycles to perform W WRITEs in a d-ported memory system as follows

$$M_c = \lceil \frac{W_0 + W_1 + W_2 + ... + W_n}{d-1} \rceil \tag{53}$$

We can rewrite equation'53 as

$$M_c = \lceil \frac{\sum_{i=0}^{n} W_i}{d-1} \rceil \tag{54}$$

Since $W = \sum_{i=0}^{n} W_i$, thus

$$M_c = \lceil \frac{W}{d-1} \rceil \tag{55}$$

Therefore, the number of required memory cycles to perform W WRITEs in a d-ported memory system can be calculated by using equation 55. Table 6 shows the required number of memory cycles for W WRITEs with different numbers of port memory systems.

In a d-ported memory system, $(d-1)$ WRITEs can be accommodated at each memory cycle, and an additional memory cycle will be required for every multiple of $(d-1)$ WRITEs thereafter. Therefore, if there are more than $(d-1)$ WRITEs at any given time, then there will be a delay time to accommodate all the WRITEs accesses.

$$\begin{aligned} D \;=\; & (0)[D_0 + D_1 + \ldots + D_{d-2}] + \\ & (1)[D_{d-1} + D_d + \ldots + D_{2d-1}] + \\ & \ldots + \lfloor (\frac{n}{d-1}) \rfloor [D_{n-d-1} + D_{n-d} + \ldots + D_n] \end{aligned} \tag{56}$$

We can rewrite equation 56 as follows

$$D = \sum_{i=0}^{n} \sum_{j=0}^{d-1} \lfloor (\frac{n}{d-1}) \rfloor D_{(d-1)i+j} \tag{57}$$

Therefore, we can calculate the expected delays in terms of memory cycles at any given time as follows:

$$D = \sum_{i=0}^{n} \sum_{j=0}^{d-1} \lfloor (\frac{n}{d-1}) \rfloor \tag{58}$$

Table 6: Memory Cycles Required for W Simultaneous Waits Using d-ported Memory Components

W	dual-port	four-port	eight-port	16-port	d-port
0	1	1	1	1	1
1	1	1	1	1	1
2	2	1	1	1	1
3	3	1	1	1	1
4	4	2	1	1	1
5	5	2	1	1	1
⋮	⋮	⋮	⋮	⋮	⋮
$N-2$	$N-2$	$\lceil \frac{N-2}{3} \rceil$	$\lceil \frac{N-2}{7} \rceil$	$\lceil \frac{N-2}{15} \rceil$	$\lceil \frac{N-2}{d-1} \rceil$
$N-1$	$N-1$	$\lceil \frac{N-1}{3} \rceil$	$\lceil \frac{N-1}{7} \rceil$	$\lceil \frac{N-1}{15} \rceil$	$\lceil \frac{N-1}{d-1} \rceil$
N	N	$\lceil \frac{N}{3} \rceil$	$\lceil \frac{N}{7} \rceil$	$\lceil \frac{N}{15} \rceil$	$\lceil \frac{N}{d-1} \rceil$

6.5 Performance Behavior and Metrics

In chapter 2 we defined four shared memory models ranging from sequential consistency to release consistency memory models. In this section, we will examine the effects of these memory models on execution time.

Our perspective is to balance software with hardware by the program's degree of parallelism. We have set our objective in the efficient utilization of the hardware. Several parameters have been defined for evaluating parallel computations by Ruby Lee [LEE80]. Let O(N) denote the total number of unit operations performed by a N-processor system and let T(N) denote the execution time in unit time steps. Let's assume $T(1) = O(1)$ in a uniprocessor. The *speedup factor* is defined as [HWANG93]

$$S(N) = \frac{T(1)}{T(N)}$$

then we can define the *system efficiency* for a N-processor system as

$$E(N) = \frac{S(N)}{N}$$

Where:

E = efficiency,

S = speedup, and

N = number of processors.

Amdahl's law defines *speedup* that can be gained by using a particular feature. Thus, speedup is the ratio

$$S = \frac{P_e}{P_{ne}}$$

Where:

P_e = performance for the entire task using the enhancement when possible, and

P_{ne} = performance for the entire task without using the enhancement.

Since:

$$performance = \frac{1}{T}$$

Where:

T = execution time.

Thus

$$S(N) = \frac{T(1)}{T(N)}$$

Therefore

$$E(N) = \frac{T(1)}{N.T(N)}$$

For RCR architecture, let:

t_{busy} = processor busy time in memory cycles,

t^R_{miss} = read-miss time in memory cycles,

t^W_{miss} = write-miss time in memory cycles, and

t_{sync} = synchronization time (time spent by a processor idling while waiting for the program order) in memory cycles.

We have:

$$E(N) = \frac{t_{busy}}{t_{busy} + t^R_{miss} + t^W_{miss} + t_{sync}}$$

Substituting from the equation above, we have:

$$S(N) = (\frac{t_{busy}}{t_{busy} + t^R_{miss} + t^W_{miss} + t_{sync}})N$$

If $t^R_{miss} = t^W_{miss} = t_{sync} = 0$, then we can scale linearly. But t^R_{miss}, t^W_{miss}, and t_{sync} increase with the number of processors.

In this model, we can eliminate t^R_{miss} and also relax t^W_{miss} by buffering an $O(1)$ WRITE. However, buffering can increase synchronization overhead time when multiple writes occur simultaneously. We can read with zero overhead, but in the case when the data to be read is still in the buffer, the processor(s) has to be idle for a fraction of the time (if the system does not support process migration and interleaving) which will add to the synchronization time. As t^W_{miss} and t_{sync} increase with the number of processors, the number of computing nodes that can be added to this system, in order to get the best possible performance, is crucial.

Calculation of *efficiency* and *speedup* for Replicated Concurrent-Read architecture based on needed memory cycles and expected delays are as follow:

$$M_c = t_{busy} + t^R_{miss} + t^W_{miss} + t_{sync}$$

Since $t^R_{miss} = 0$ for RCR, and $D = t^W_{miss} + t_{sync}$, then

$$E = \frac{t_{busy}}{t_{busy} + D}$$

$$E = \frac{M_c - D}{M_c}$$

Speedup for Replicated Concurrent-Read is

$$S(N) = \frac{M_c - D}{M_c}(N)$$

Table 7: RCR Speedup for Various Number of Ports in Multiported Memories.

N	$d = 2$	$d = 4$	$d = 8$
8	0	2.67	4
16	0	2.67	5.33
32	0	2.91	6.4

Speedup for RCR architecture is function of the number of WRITEs issued simultaneously by processors, since READs are done locally with no overhead. Table 7 shows speedup for RCR with dual-ported, 4-ported, and 8-ported memories when all processors attempt to perform WRITE simultaneously for various N.

Table 7 shows that if 100% of total memory accesses are WRITE operations, there is 0 speedup if dual-ported memories are used, since only 1 WRITE can take place in one clock cycle.

6.6 Summary

This architecture, allows READ operations to be performed locally with zero overhead while performing WRITEs globally. Since increasing the size of replicated memory modules affects the overall cost of the system, by adding auxiliary memory the scalability of the system is increased. RCR architecture by taking advantage of spatial caching, allows for greater performance with small sized replicated memories. Since RCR architecture performs the majority of memory references locally, this reduces system bus traffic while improving overall memory access time.

d-ported memories support $d-1$ WRITE operations at every single clock cycle. Thus, if there are more than $d-1$ WRITEs concurrently, D number of clock cycles *delay* will be experienced, where $D = \lceil \frac{N}{d-1} \rceil - 1$. Thus, by having larger number of ports on multiported memories, an overall improvement is expected. By using 4-ported memories instead of dual-ported memories, there is a 71% improvement in number of *delay* in the worse case when there are N WRITEs simultaneously. In order to experiment and compare the effectivenss of RCR design in the worse case, dual-ported memories are chosen.

7 Simulator Development and Performance Comparisons

A simulator has been developed in order to study and analyze the behavior of memory references in Replicated Concurrent-Read (RCR) Architecture. For comparable analysis purposes, simulators for Uniform Memory Access (UMA) Architecture, Non-Uniform Memory Access (NUMA) and Local-Remote-Global (LRG) Architecture, have also been developed.

The simulation code consists of a series of functions in C programming language which are included in Appendix section. The main program contains a FOR loop which allows for simulations in ten nano second iterations per cycle. Within the FOR loop, memory references are issued by calling related routines. Each memory reference could be a READ or WRITE access. As simulation progresses, the total memory access time is recorded and finally, expected access times are computed.

In the following sections, simulator design and analysis of RCR, UMA, NUMA and LRG architectures are discussed. In the final section, comparisons of these architectures will be presented. A few assumptions have been made in order to resume consistency in analyzing generated data from the simulators. If there is a READ miss, then a block of data will be copied to the cache. In the case of a *shared-write* then *write-through* policy is implemented.

7.1 Replicated Concurrent-Read (RCR) Architecture

In the RCR, a READ miss will cause an access to replicated memory in search of a requested word. If the search of replicated memory is unsuccessful, then the auxilary memory will be referenced. Let P_i denote processor # i. The probability of a cache hit is h_c and the replicated memory hit is h_L. P_i with $(1 - h_c)$ probability will face a cache miss and has $(1 - h_L)$ chance of replicated memory miss. Therefore, the chance of having to access auxilary memory is $(1 - h_c)\ (1 - h_m)$. P_i will have to wait until this data is transfered to C_i (P_i's private cache). During this time P_i will be inactive.

Table 8: RCR System Parameters.

Input Parameter	Value	Range
P_i	$10ns/cc$	N/A
P_{read}	0.90	$0.75 - 0.95$
P_{shared_write}	0.0164	$0.01 - 0.5$
h_c	0.50	$0.10 - 0.95$
h_L	0.50	$0.10 - 0.80$
Words per Block	8	$8 - 64$

P_i has to compete with other processors to access the global bus. As a result, P_i may have to wait for its turn to access the auxilary memory.

A WRITE access is treated differently in RCR architecture. Every *shared-write* access is broadcasted to all replicated memories. The simulator determines all memory reference characteristics:

- whether the memory access is a READ or WRITE,
- if memory access refers to shared data,
- if memory access is a cache hit or miss,
- is a replicated memory hit or miss.

The simulator also generates the number of other memory references pending for bus access in order to compute the wait time for P_i. table 8 lists the input to the simulator.

7.1.1 Varying Cache and Replicated Memory Hit Rate

Various cache hit rates, with respect to $10\%, 20\%, \ldots, 80\%$ replicated memory hit rates, have been studied. Figure 35 shows the average access time of RCR with a cache hit rate of 10% in conjunction with various replicated memory hit rates.

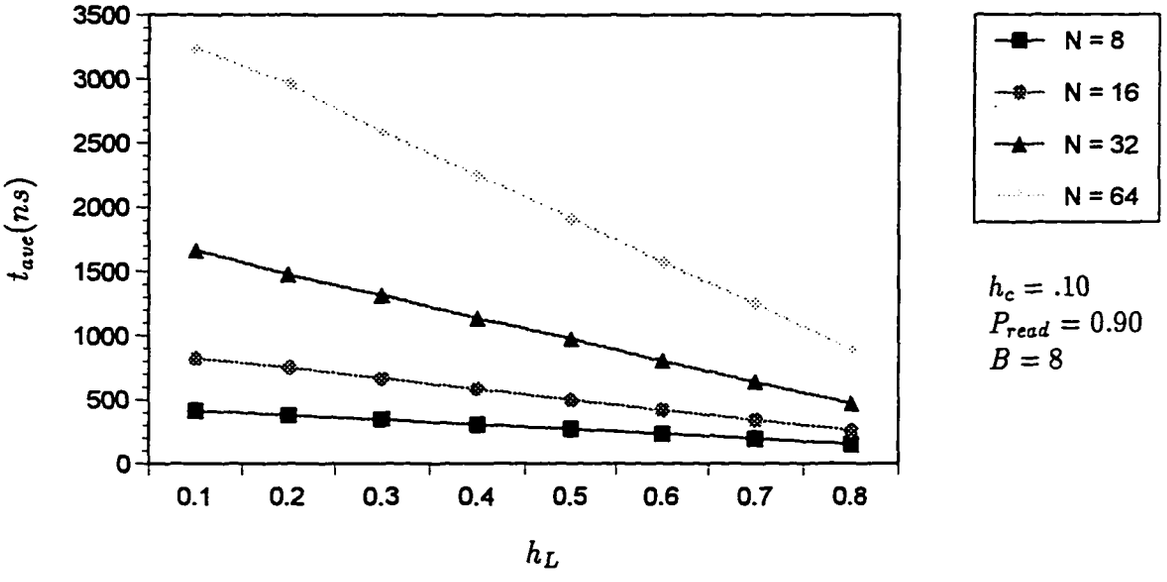

Figure 35: RCR architecture. Expected access time for various replicated memory hit rates.

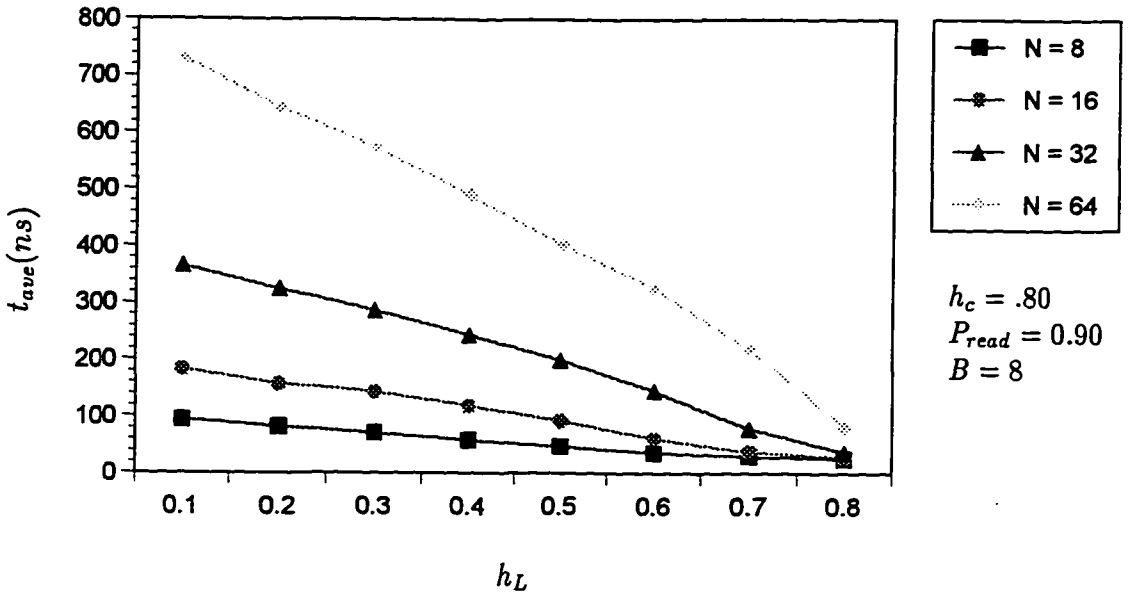

Figure 36: RCR architecture. Expected access time for various h_L with 80% h_c.

This experiment has been conducted for 8, 16, 32 and 64 processor systems. Average access time decreases as the replicated memory hit rate increases, as is shown in Figure 35. Let N denote the total number of processors. When N=8, there is more than a 64% improvement in access time as replicated memory hit rates increase from 10% to 80%. Systems with 16, 32 and 64 processors also demostrate an improvement in average access time by over 68%.

Figure 36 shows the expected access time for the cache hit rate (h_c) of 80% for various replicated memory hit rates. This figure shows expected access time improves over 72% as h_L increases from 10% to 80% for $N = 8$. For $N = 16, 32$, and 64, expected memory access time improves over 84%, 89%, and 89%, respectively. Comparing Figure 36 with Figure 35 shows the effect of h_c on average memory access time (t_{ave}). As more memory accesses are satisfied by cache and replicated memory, better average access time and more CPU utilization results.

Table 9: UMA System Parameters.

Input Parameter	Value	Range
P_i	$10ns/cc$	N/A
t_c	$10ns/cc$	N/A
t_m	$100ns/cc$	N/A
P_{read}	0.90	$0.75 - 0.95$
P_{shared_write}	0.0164	$0.01 - 1.0$
h_c	0.50	$0.10 - 0.95$
Words per Block	8	$8 - 64$

7.2 Uniform Memory Access (UMA) Architecture

In UMA architecture, all processors share a global memory which is equally close to all processing elements, while every processor is attached to a private cache. A READ hit feches data from the cache in t_c time. The probability of a cache hit is denoted as h_c. A READ miss will cause P_i to access the global memory in order to fetch data. The probability of having to access global memory is $(1 - h_c)$. P_i may have to wait to access the cache if there are a number of pending WRITEs since only 1 validation can occur at a time.

Since every processor in the system with a probability of $(1 - h_c)$ will have to access shared-memory, a delay in accessing the shared-memory will be inevitable. In UMA architecture, as the number of processors increases, undesirable delays will increase average memory access time. As a result, UMA architecture can support only a small number of processors. UMA simulators will generate access to memory and will also define the characteristics of all memory accesses.

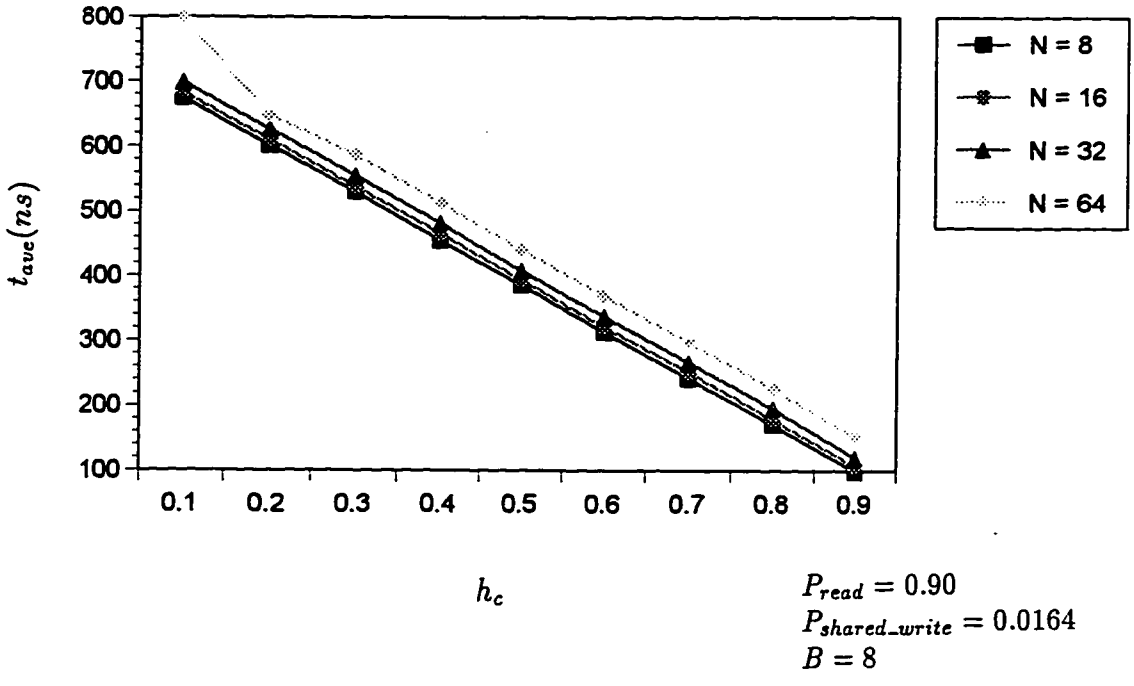

Figure 37: Expected access time vs. h_c for UMA configuration.

7.2.1 UMA Simulation Results

In UMA architecture, since there is no local memory other than cache, the cache hit rate is a major concern in regard to its performance evaluation. Figure 37 shows the results of a simulation as the cache hit rate increases from 10% to 90%. The simulation has been repeated for a various number of processors in the system. The average access time shows an improvement of over 85% as h_c increases from 10% to 90%. The effects of other parameters of simulation will be discussed as RCR, UMA, NUMA and LRG simulation results are compared.

7.3 Non-Uniform Memory Access (NUMA) Architecture

In NUMA architecture, shared memory is distributed among all processors. Every processor can address its local memory or remote memories of other processors. Every P_i is also attached to a private cache. A READ hit fetches data from the cache in t_c time. The cache hit rate is denoted as h_c.

A cache miss with a probability of $(1 - h_c)$ will cause an access to local memory. Fetching data from local memory may be delayed if there are other pending READ or WRITES by other processors. In the case of a local memory miss, remote memories will be accessed. This NUMA simulator generates memory references as it defines their characteristics, whether the access :

- is a READ or WRITE;
- is a cache hit or miss,
- is a local memory hit or miss, or
- refers to shared or private data.

7.3.1 NUMA Simulation Results

The cache hit rate h_c and local memory hit rate h_L are two major parameters in the performance evaluation of NUMA machines. Figure 38 shows results of simulation for $h_c = 0.50$ and varying h_L percentages.

There is a direct relationship between average access time and h_L. An increase in h_L will decrease t_{ave} directly as shown in Figure 38. This experiment is repeated with $h_c = 0.90$ for varying percentages of h_L in order to study the effects of higher h_c. As shown in Figure 39 with $h_c = 0.90$, a 77% improvement, in average memory access time, is achieved over $h_c = 0.50$. The effects of other parameters of simulation will be discussed as RCR, UMA, NUMA and LRG simulation results are compared.

Table 8: NUMA System Parameters.

Input Parameter	Value	Range
P_i	$10ns/cc$	N/A
t_c	$10ns/cc$	N/A
t_L	$100ns/cc$	N/A
t_G	$200ns/cc$	N/A
P_{read}	0.90	$0.75 - 0.95$
P_{shared_write}	0.0164	$0.01 - 1.0$
h_c	0.50	$0.10 - 0.95$
h_L	0.50	$0.10 - 0.80$
Words per Block	8	$8 - 64$

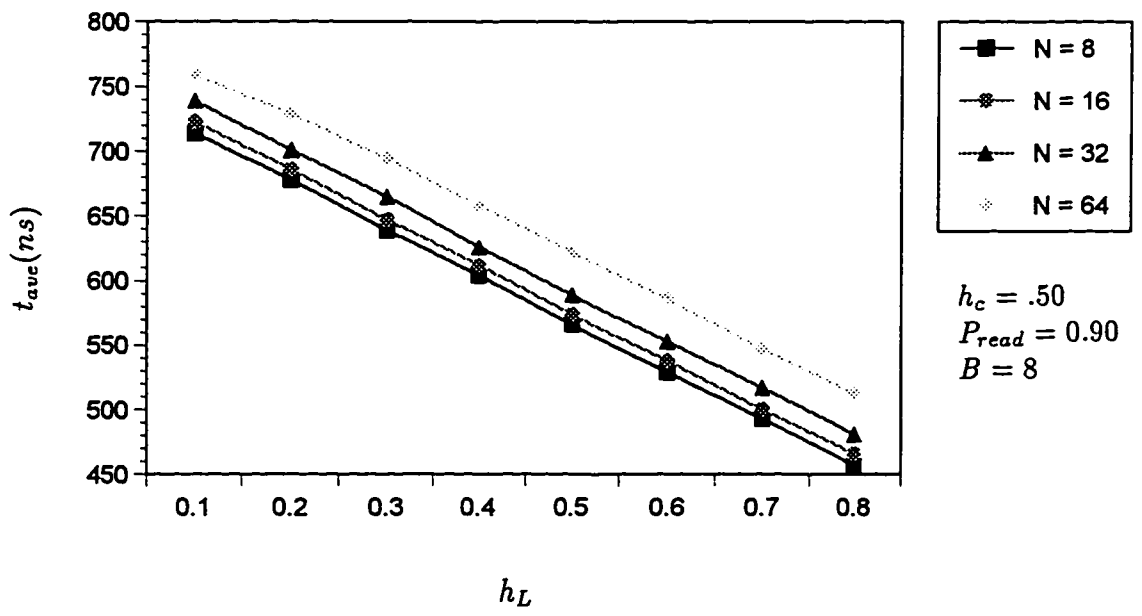

Figure 38: Expected access time vs. h_L when $h_c = 0.50$ for NUMA configuration.

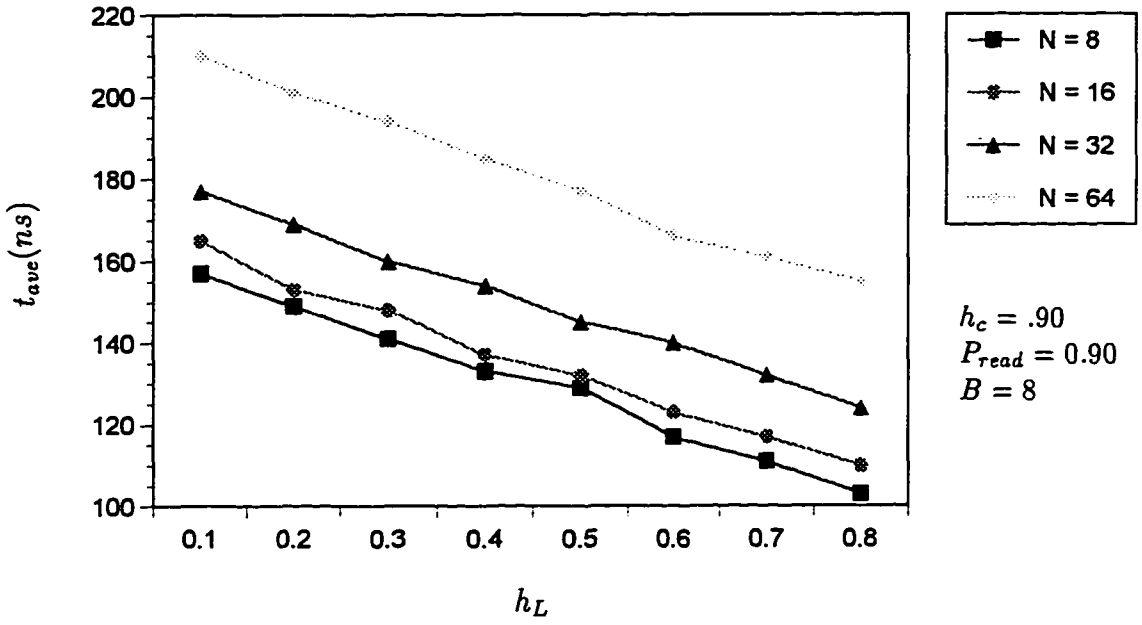

Figure 39: Expected access time vs. h_L when $h_c = 0.90$ for NUMA configuration.

7.4 Local-Remote-Global (LRG) Architecture

Local-Remote-Global Architecture is a combination of UMA and NUMA machines. Every cluster contains two processors and a shared local memory. Each processor on the cluster is attached to a private cache. LRG also provides a global memory accessible by all processors.

A READ hit with a probability of h_c fetches data directly from the cache in t_c time. In the case of a cache miss, the local memory is referenced. Let h_L denote the local memory hit rate. The probability of accessing global memory is $(1-h_c)(1-h_L)$. As a result, the average access time is a function of h_c and h_L. This simulator will study and analyze various percentages of h_c and h_L.

7.4.1 Local-Remote-Global (LRG) Simulation Results

For a complete analysis of the effects of h_c and h_L on expected memory access time, various percentages of h_c and h_L are studied. Figure 40 shows the average access time as h_L increases from 10% to 90% for various cache hit rates. The cache hit rate has a more drastic effect on expected access time than the local memory hit rate. As shown in figure 40, the local memory hit rate also has a significant effect on the expected memory access time. The effects of other parameters of simulation will be discussed as RCR, UMA, NUMA and LRG simulation results are compared.

7.5 Performance Comparisons

For the purpose of comparing these machines, the effect of numerous varying parameters will be examined. As shown in figure 41, the effect of various cache hit rates is illustrated. As expected, the NUMA machine has shown great improvement in average access time, with respect to varying cache hit rates. The reason being, that the delays caused by the interconnection network is decreased. All of the other

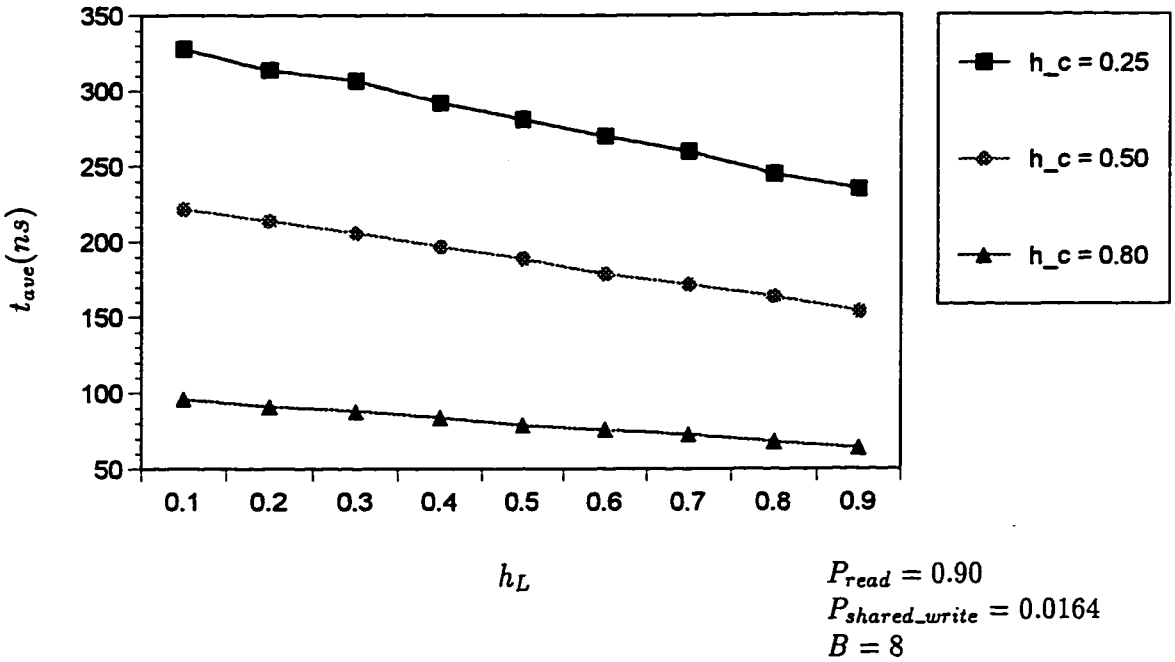

Figure 40: LRG architecture. Expected access time for various local memory hit rates.

Table 9: LRG System Parameters.

Input Parameter	Value	Range
P_i	$10ns/cc$	N/A
t_c	$10ns/cc$	N/A
t_L	$100ns/cc$	N/A
t_G	$200ns/cc$	N/A
P_{read}	0.90	$0.75 - 0.95$
P_{shared_write}	0.0164	$0.01 - 1.0$
h_c	0.50	$0.10 - 0.95$
h_L	0.50	$0.10 - 0.80$
Words per Block	8	$8 - 64$

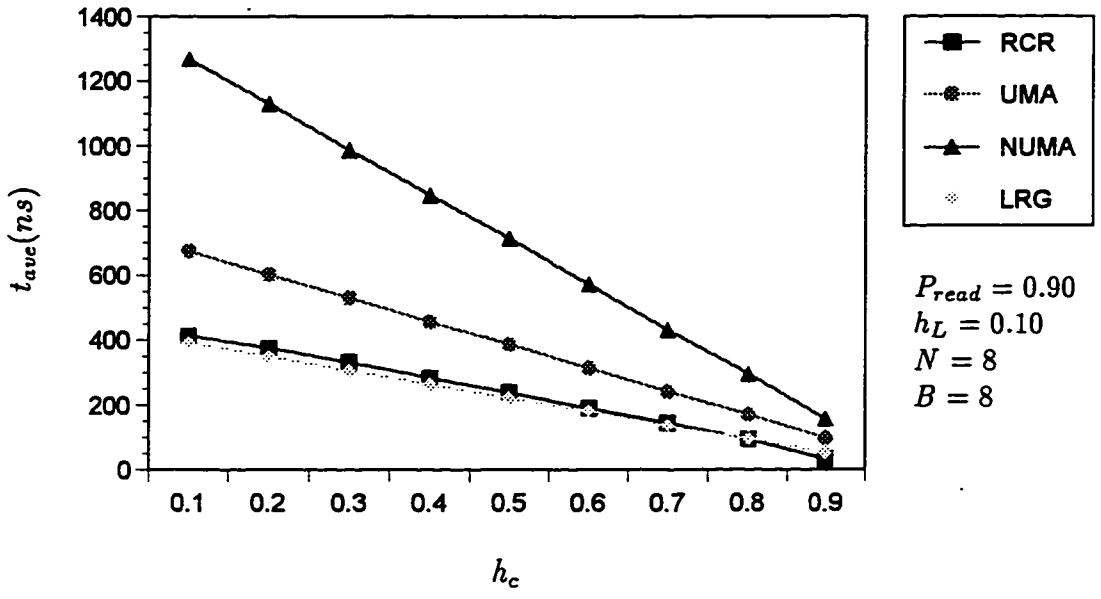

Figure 41: Expected access time for various cache hit rate percentages.

machines have shown improvement as h_c is increased from 10When h_c is above 75%, RCR delivers a lessened memory access time. When h_c is below 75%, RCR's memory access time is comparable to LRG's average access time.

Figure 42 shows the effect of varying local memory hit rates on expected memory access time. Since the UMA machine does not have local memory, its average memory access time does not vary. RCR demonstrates a direct effect as a result of increasing the replicated memory hit rate. The NUMA machine demonstrates a better performance as hit rate increases. The LRG machine is less affected by the local memory hit rate than the RCR and NUMA machines.

Figure 43 illustrates the effect of varying local memory hit rates when the cache hit rates increase. The effect of varying local memory hit rates, accompanied by higher cache hit rates, is more pronounced with the RCR and NUMA machines. The other machines did show improvement, but not as significantly as that of the rate of

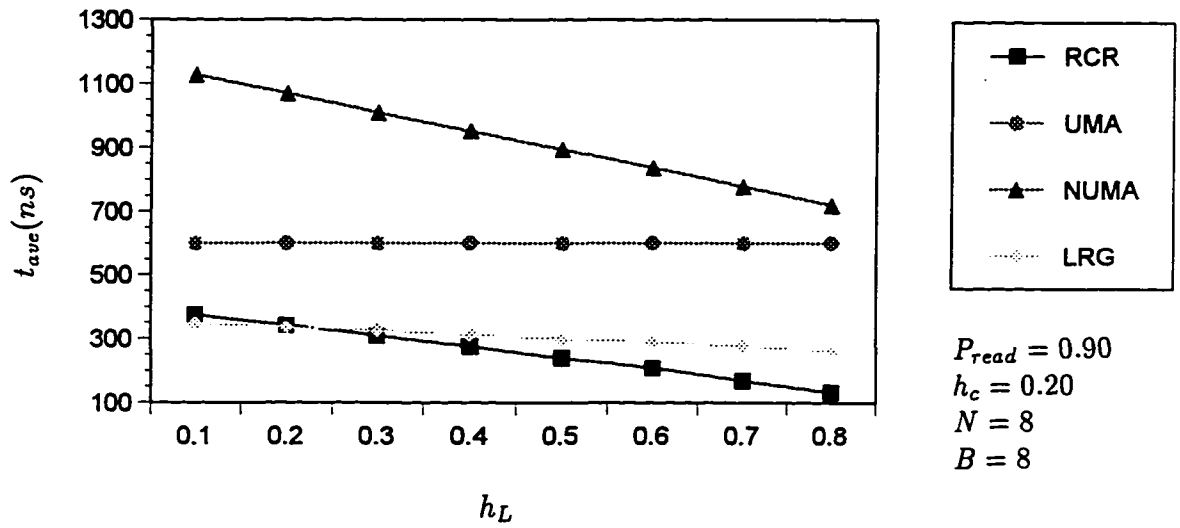

Figure 42: Expected access time for various h_L when $h_c = 0.25$.

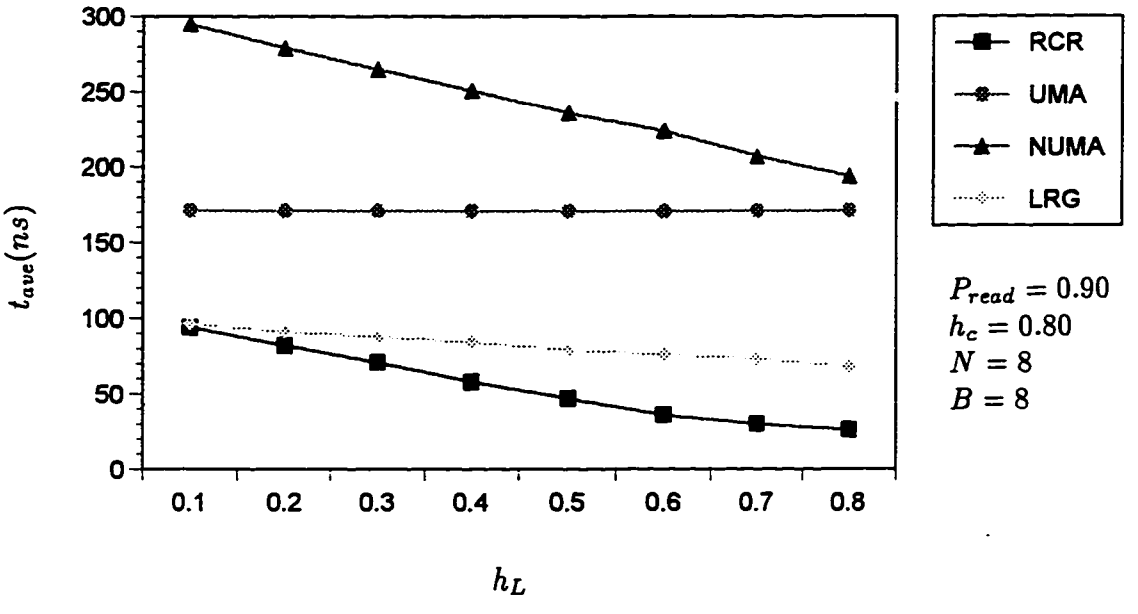

Figure 43: Expected access time for various h_L when $h_c = 0.80$.

the RCR machine.

Figure 44 also illustrates the effects of varying local memory hit rates in conjunction with a 90% cache hit rate. The RCR, LRG, and NUMA machines demonstrates an improvement as local memory hit rates increase.

The effect of varying shared-write percentages on these machines have been analyzed. Let P_{shared_write} denote the probability of a shared-write memory access such that $P_{shared\_write} + P_{private\_write} + P - read = 1$. The results, shown in figure 45, illustrate a slight increase in the average memory access time as the probability of shared-writes increases from 0.0 to 0.5.

Figure 46 shows the effect of varying block sizes on expected memory access time on the RCR, UMA, NUMA, and LRG machines. The NUMA machine demonstrates a drastic increase in memory access time as the block sizes increase from 8 to 64. The main reason for this significant increase in memory access time is due to the transfer

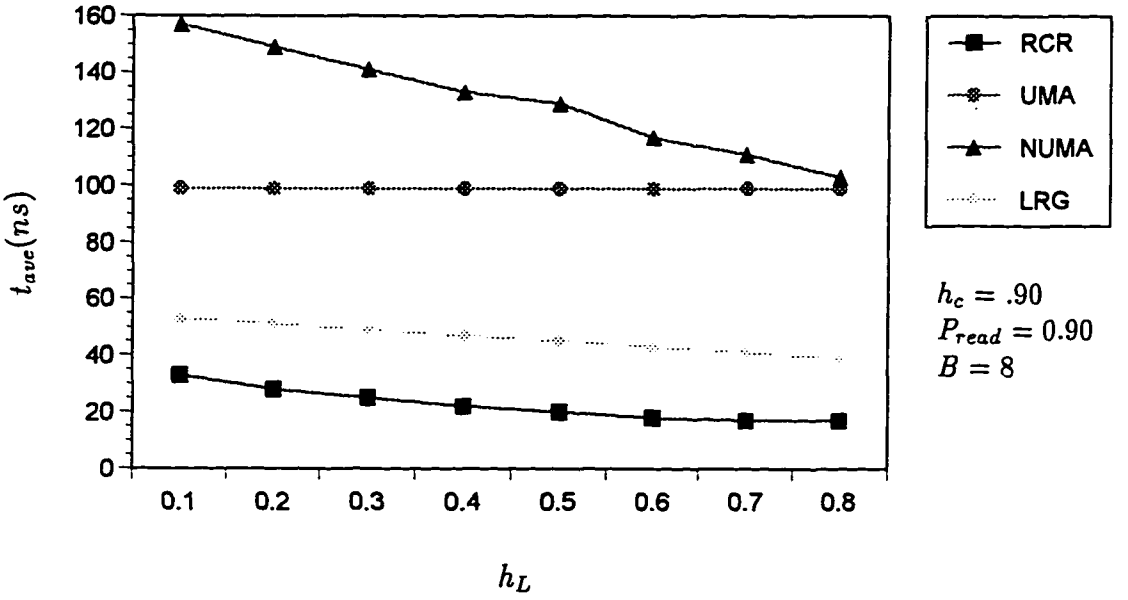

Figure 44: Expected access time for various h_L when $h_c = 0.90$.

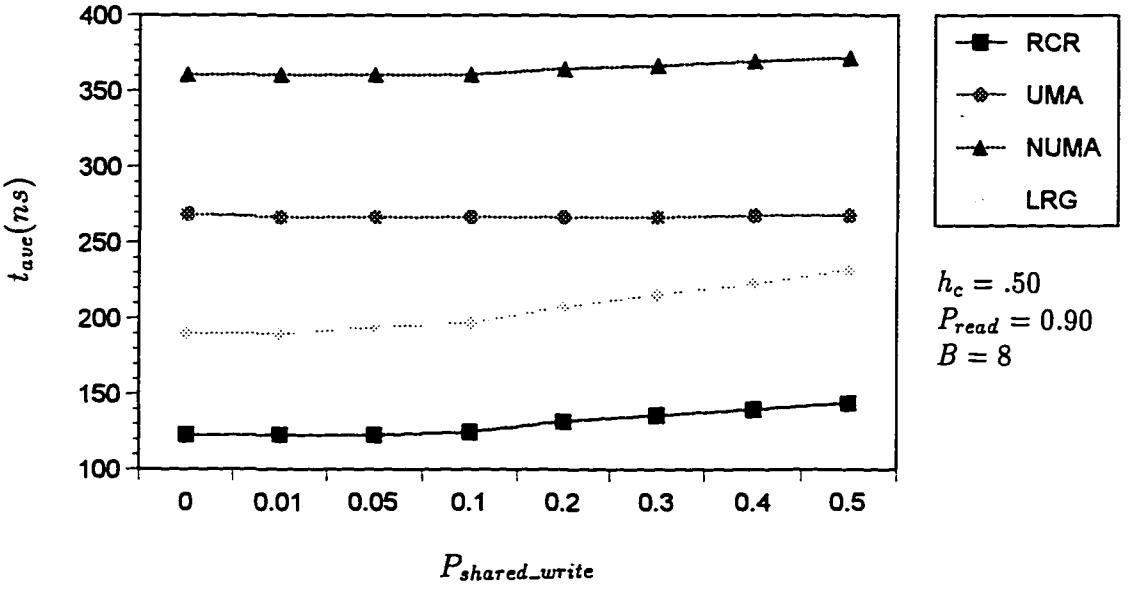

Figure 45: Expected access time for various shared-write percentages.

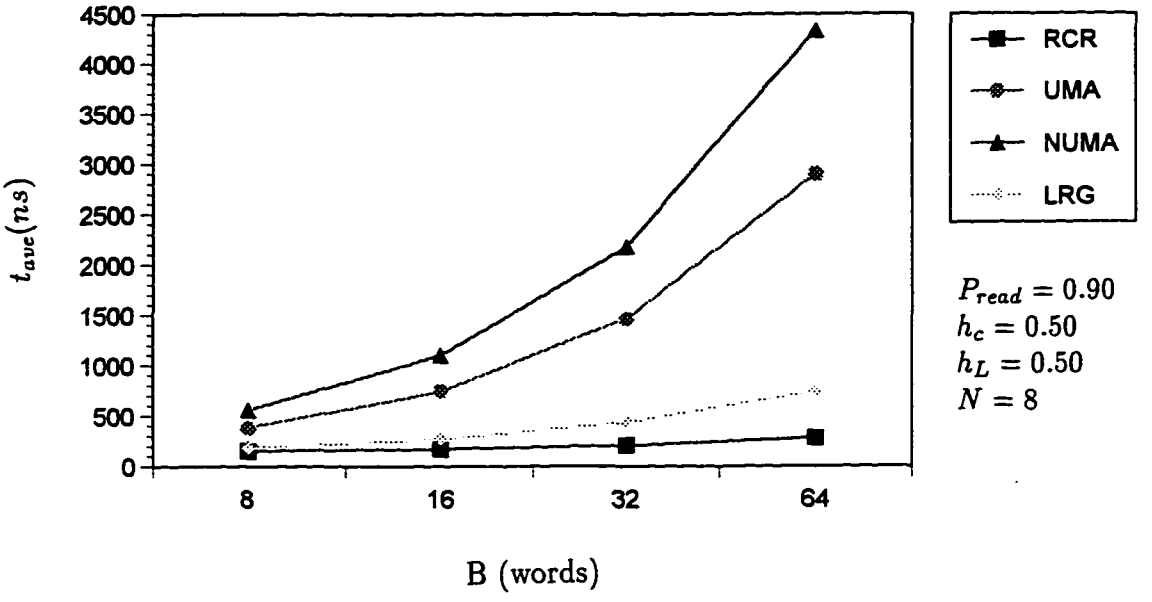

Figure 46: Expected access time for various block sizes.

of blocks of data from remote memories. The UMA machine also experienced a significant increase in memory access time due to the transfer of blocks of data from global memory. The RCR and LRG machines demonstrate a lesser effect as block size increases.

In conclusion, the varying probability of READs is examined. Let P_{read} denote the probability of READ. The results shown in figure 47 exhibit that the NUMA machine's performance decreases as P_{read} increases. The reason is that, the READ from the remote memories are costly. The UMA machine demonstrates a similar effect but at a lesser rate. The RCR machine shows the most desired result since it performs READs locally. The RCR configuration has less sensitivity to READ/WRITE percentage.

$P_{shared\_write} = 0.0164$
$h_c = 0.50$
$h_L = 0.50$
$N = 8$
$B = 8$

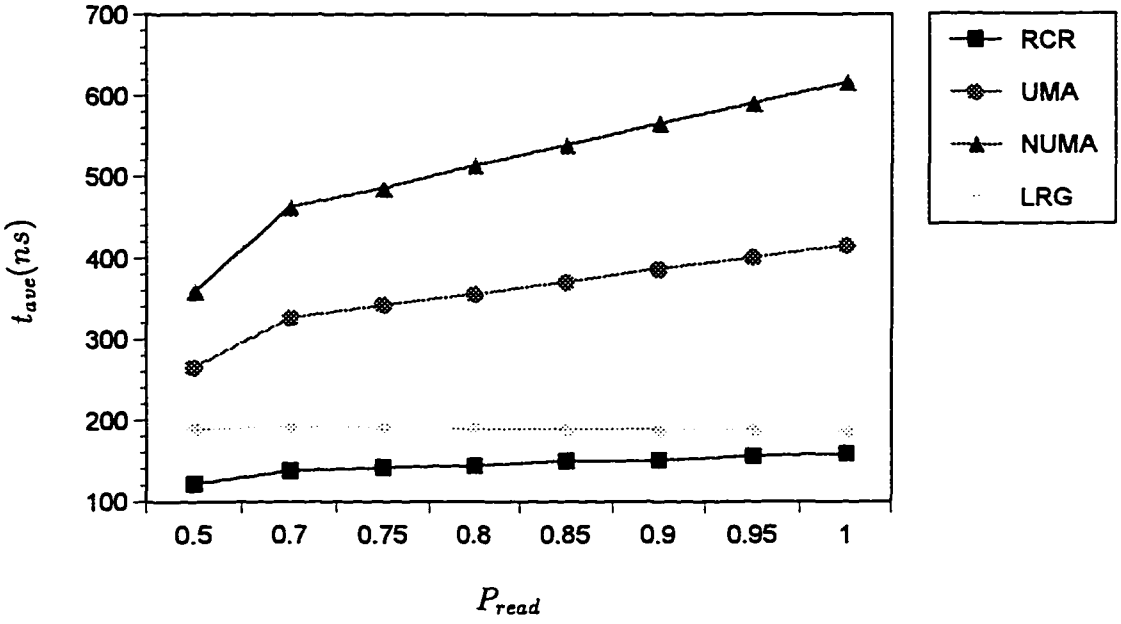

Figure 47: Expected access time for various P_{read} percentages.

7.6 Summary

In this chapter simulators for RCR, UMA, NUMA, and LRG architectures were developed and their performance results discussed. The simulation results proved that, for a wide range of system parameters, RCR outperformed the UMA and NUMA architectures. The RCR architecture outperforms LRG architecture when the hit rates of the processor's cache exceeds 80% and replicated memory exceeds 25%. Simulation results show that RCR architecture, with up to 32 processors, offers outstanding performance over existing multiprocessors.

8 Conclusion

Replicated Concurrent-Read architecture, with its novel design and functionality, offers favorable performance over UMA and NUMA architecture for all ranges of application and system parameters. RCR outperforms LRG architectures when the hit rates of the processor cache exceed 80% and replicated memory exceed 25%. The RCR architecture resulted from a complete re-evaluation of common memory space based on actual multiprocessor memory reference behavior. The resulting design leverages memory reference behavior and component expense by broadcasting memory updates in constant time while allowing READ references to be performed with zero access latency.

8.1 Cost-Effectiveness

Since the best overall system price-performance will be determined primarily by the mermory architecture, the RCR offers a cost-effective system by redesigning the memory space. The RCR, by not requiring complicated and expensive hardware, offers low cost, yet efficient design. Memory latency is the most significant issue in the design of a shared-memory multiprocessor. Memory latency could be improved by increasing cache hit rate in a uniprocessor. Unlike uniprocessors, increasing size of caches is not a dominant factor in multiprocessor hit rate. In a multiprocessor system, maintaining coherence between caches is a significant factor in memory latency. These coherence misses are independent of the cache size. Increasing cache sizes will not decrease the expected memory access time, since data sharing causes invalidations and extra misses because of coherence. The RCR architecture outperforms UMA architecure with same cache hit rate, while it is a cost-effective system.

8.2 Scalability

Unfortunately Designing an ideal scalable system is not possible. The RCR design, by inclusion of an auxiliary memory unit, has increased its scalability rather than

Table 12: RCR Speedup for Various Percentages of *shared-read.*

N	$P_{shared\_write} = 0.20$	$P_{shared\_write} = 0.40$	$P_{shared\_write} = 0.80$
8	5	2.5	1.25
16	5	2.5	1.25
32	5	2.5	1.25

the size of replicated memory modules. Table 7 and 12 illustrate that by increasing number of processors, as long as, adequate multiported memory units are used, RCR proves its scalability.

8.3 Performance Prediction

The RCR design has a unique characteristic by performing majority of memory references locally with no delay. Thus, percentage of *shared-write* operations is the only factor influencing speedup in the RCR architecture. This is not possible for UMA, NUMA, and LRG architectures. speedup for RCR configuration is:

$$S(N) = \frac{M_c - D}{M_C}(N)$$

where

$$M_C = \lceil \frac{W}{d-1} \rceil,$$

and

$$D = \lfloor \frac{W}{d-1} \rfloor.$$

Thus,

$$S(N) = \frac{\lceil \frac{W}{d-1} \rceil - \lfloor \frac{W}{d-1} \rfloor}{\lceil \frac{W}{d-1} \rceil}(N)$$

Table 12 shows speedup for various percentages of *shared-write* for dual-port RCR configuration.

As seen in table 12, RCR speedup is only function of *shared-write.* Thus, it is possible to predict the performance of RCR for any application. By increasing

number of ports in d-ported memories, RCR shows an improvement in speedup as shown in table 7.

8.4 Hardware Feasibility

Since RCR hardware design is based on commonly available off-the-shelf parts, it does not require any customized complicated hardware. The RCR architecture utilizes currently available $16K \times 32$ dual-port memory modules as part of its global address space. Each processor is attached to a local memory for its private data and programs. Every processor is also connected to a dual-port replicated memory module. The RCR architecture provides an auxiliary memory unit which is equally accessible by all processors.

8.5 Future Work

To actually build the RCR will remain as future work. The expansion of RCR architecture to support a larger number of processors while maintaining scalability, also remains as future work.

A RCR Simulation Code

```
/*********************************************/
Simulation Code For Replicated Concurrent-Read
Architecture
Version 4.2
/*********************************************/

#include<stdio.h>
#include<math.h>
#define PE  8

/*********************************************/
Global Variables
/*********************************************/


FILE *fptr;
int N = PE ;
 /* Number of processing elements */
float h_c = .50 ;
   /* Primary cache hit rate */
float h_M = .50;
int t_p = 10 ;
      /* Processor cycle time */
int t_c_read = 1;
    /* Time it takes to read a word from primary cache
in terms of processor's clock cycle*/
int t_c_write = 1;
       /* Time it takes to write a private word to
primary cache in terms of processor's clock cycle*/
```

```
int t_M_read  = 1;
  /* Time it takes to read a word from replicated
    memory in terms of processor's clock cycle*/
int t_M_write = 2;
    /* Time it takes to write a word to replicated
     memory in terms of processor's clock cycle*/
int t_aux_read = 2;
     /* Time it takes to read a word from auxiliary
      memory in terms of processor's clock cycle*/
int t_aux_write = 2;
     /* Time it takes to write a word to auxiliary
       memory in terms of processor's clock cycle*/
float P_read = .90;
      /* Probability access is read (vs. write)*/
float P_write =1-P_read;
 /* Probability access is write */
int read;
 /* A flag to indicate an operation */
float P_IOF = .80;
/* Probability of an address being Incrementally
  Outside Fence   */
float P_shared_write = .0164;
float seed;
   /* A variable which holds a random number*/
int B = 8;
int D = 4;
      /* Distance outside fence */
int num_write_inpipe = 0;
int iof_adres_read = 0;
int iso_adres_read = 0;
```

```
int old_write_inpipe =0;
int old_iso_read = 0;
int old_iof_read =0;
int num_idle_pe = 0;
int active_N = N;
long i;
long count_iof_time[PE];
long count_iso_time[PE];
long count_write_time[PE];
     /* Maximum WRITEs in pipe can not exceed N */


int num_pending_access = 0;

int old_pending_access = 0;
long recal = 0;

unsigned long number_of_miss = 0;

            /*****************************************/
                        Function Prototype
            /*****************************************/

int PEi();
long access_dport();
   /* This function calculates the access
      time for dual-port memory*/
float rand(float);
   /* This function generates random numbers */
float get_prob_uniform();
       /* This function will provide a random address */
```

```
int read_write();
  /* This function sets access values based
     on read vs write */
int wait_for_bus();
void bus_arbiter();

            /*******************************************/
                        main function
            /*******************************************/

void main()
{
int n;
int counter;
int accesstime_dport = 0;
long total_access_time = 0;
long num_of_access = 0;
float Expected_access_time = 0.0;
int y,j,k;
for(y=0 ; y<=(N-1);y++)
{
count_iof_time[y] = 0;
count_iso_time[y] = 0;
count_write_time[y]= 0;
}

fptr = fopen("rcr_test.cpp","a");

printf("\nPlease provide one random number
      seed for this experiment");
printf("\nPlease enter an odd 6 digit
```

```
       number not ending in 5:");
scanf("%f",&seed);

active_N = N;
for ( i=1;i<=100000; i++)
 {
 read = read_write();

 if(active_N >= 1)
     {
      num_of_access +=1;
      accesstime_dport = PEi();
      total_access_time += accesstime_dport;
     }

 if (iof_adres_read > 0)
 {

 for (j = 0; j<iof_adres_read ;j++)
{
 if (i+1 == count_iof_time[j])
    {
     for(k = j; k<iof_adres_read-1 ; k++)
     count_iof_time[k] = count_iof_time[k+1];
     iof_adres_read -=1;
     break;
    }
}
 }
  if (iso_adres_read > 0)
```

```
 {

 for (j = 0; j<iso_adres_read ;j++)
{
 if (i+1 == count_iso_time[j])
      {
       for(k = j; k<iso_adres_read-1 ; k++)
       count_iso_time[k] = count_iso_time[k+1];
       iso_adres_read -=1;
       break;
      }
}
 }
if (num_write_inpipe > 0)
 {

 for (j = 0; j<num_write_inpipe ;j++)
{
 if (i+1 == count_write_time[j])
{
 for(k = j; k<num_write_inpipe-1 ; k++)
     count_write_time[k] =
 count_write_time[k+1];
     num_write_inpipe -=1;
     break;
 }
}
 }

old_iof_read = iof_adres_read ;
```

```
old_iso_read = iso_adres_read ;
old_write_inpipe = num_write_inpipe;
num_pending_access = old_iof_read + old_iso_read +
old_write_inpipe ;
old_pending_access = num_pending_access;
num_idle_pe = num_pending_access ;
active_N = N - num_idle_pe;
 }
 Expected_access_time =( (float)total_access_time
 /(num_of_access)) * t_p;
 printf("\nExpected Access Time = %5.2f",
   Expected_access_time);

 fprintf(fptr,"\nProbability of read");
 fprintf(fptr,"\nExpected Access Time = %5.2f",
  Expected_access_time);
 fprintf(fptr,"\nCache hit rate = %f",h_c);
 fprintf(fptr,"\nReplicated memory hit rate = %f",h_M);
 fprintf(fptr,"\nP_read = %f",P_read);
 fprintf(fptr,"\nNumber of blocks = %d",B);
 fprintf(fptr,"\nNumber of processor = %d",N);
 fprintf(fptr,"\nProbability of shared write = %f",
       P_shared_write);

/* printf("\n active N = %d",active_N);

 for (n = 0;n<iof_adres_read; n++)
 printf("\ncount_iof_time[%d] = %ld",n,count_iof_time[n]);
 for (n = 0;n<iso_adres_read; n++)
 printf("\ncount_iso_time[%d] = %ld",n,count_iso_time[n]);
```

```
 for (n = 0;n<num_write_inpipe; n++)
 printf("\ncount_write_time[%d] = %ld",n,count_write_time[n]);
/* } */
 printf("\n Number of cache miss = %ld",number_of_miss);
 fclose(fptr);
 }

            /*******************************************/
               This function simulates PEi's memory access
            /*******************************************/

int PEi()
{
int time = 0;
float write_is_shared ;
float hit_or_miss;       /* Cache hit or miss */

hit_or_miss = get_prob_uniform();

if (hit_or_miss <= h_c)
  {
    if (read)
      /* Access is a READ (rather than a WRITE) */
    time = t_c_read;
    /* Processor clock cycle time */
    else
       /* Access is WRITE under write-thru policy */
  {
  write_is_shared = get_prob_uniform();
  if (write_is_shared <= (P_read + P_write)*
 P_shared_write )
```

```
    {
     time = wait_for_bus() + t_M_write ;
     num_write_inpipe +=1;
bus_arbiter();
       }
     else
time = t_c_write ;
     }
  }
else
  {
  number_of_miss +=1;
  time = access_dport();
  }
return(time);
}

            /*****************************************/
                This function calculates the access time for
                          dual-port memory
            /*****************************************/

  long access_dport()
  {
  long time = 0;
  float prob_in_replicated;
  float IOF ;
/* ckeck to see if the intended word is
  in replicated memory */
  prob_in_replicated = get_prob_uniform();
```

```
  if (prob_in_replicated <= h_M)
{
 if(read)
     time = B * t_M_read;
 else
     {
      time = wait_for_bus()+ t_M_write;
      num_write_inpipe +=1;
      bus_arbiter();
     }
 }

  else   /* there are some read/write misses */
{

if(read)
    {
    IOF = get_prob_uniform();
    if (IOF <= P_IOF )
      {
      time = wait_for_bus() + D * t_aux_read;
      iof_adres_read += 1;
      bus_arbiter();
      }
    else
      {
      time = wait_for_bus() +  t_aux_read;
      iso_adres_read += 1;
      bus_arbiter();
      }
```

```
    }

else
    {
     time = wait_for_bus() + t_aux_write;
     num_write_inpipe +=1;
     bus_arbiter();
    }
}


  return(time);
  }
```

/***/

This function calculates the wait time

for a global bus

/***/

```
 int wait_for_bus()
 {
 int it_is_read;
 int n,wtime = 0;
 float P_wait_global ;
 float IOF;

 for(n=1 ; n <= (active_N-1) ; n++)
{
P_wait_global = get_prob_uniform();
if (P_wait_global <= (1 - h_c)*(1 - h_M))
  /* Other processors are using the bus at this time*/
```

```
{
it_is_read = read_write();
if (it_is_read)
  /* If another processor is performing READ operation */
    {
    IOF = get_prob_uniform();
    if (IOF <= P_IOF )
    iof_adres_read += 1;
    else
    iso_adres_read += 1;
    }
else
  /* Another processor is performing WRITE operation */
    {
    num_write_inpipe +=1;
    }
}
}

 if (iof_adres_read + iso_adres_read +
      num_write_inpipe == 0)
 wtime = 0;
 else
 wtime =iof_adres_read * D * t_aux_read +
iso_adres_read * t_aux_read +
       num_write_inpipe * t_aux_write;
 return(wtime);
 }
```

```
/*****************************************/
```

Bus Arbiter

```
/*****************************************/

void bus_arbiter()
{
int x,y,z,v;
int n,k;
long index;
if (i>= recal)
    index = i;
else
    index = recal;
num_pending_access = iof_adres_read +
 iso_adres_read + num_write_inpipe;

x = iof_adres_read - old_iof_read;
v = iso_adres_read - old_iso_read;
y = num_write_inpipe - old_write_inpipe;


  for (z = 0; z <(num_pending_access -
 old_pending_access); z++)
   {
   if (x > z)
    {
    count_iof_time[old_iof_read+z] = index +
  D * t_aux_read ;
    index = count_iof_time[old_iof_read+z];
```

```
   recal = index;
   }
   if (v > z)
   {
   count_iso_time[old_iso_read+z] = index
    + t_aux_read ;
   index = count_iso_time[old_iso_read+z];
   recal = index;
   }
  if (y > z )
   {
   count_write_time[old_write_inpipe+z] = index +
t_M_write ;
   index = count_write_time[old_write_inpipe+z];
   recal = index;
   }
  }
}
```

/***/

Random number generator

/***/

```
float rand(float x)
{
int i;
i = 997.0 * x / 1.e6;
x = 997.0 * x - i * 1.e6;
return(x);
}
```

```
/*******************************************/
```

This function computes a probability value that
is uniformly distributed on the interval [0,1]

```
/*******************************************/

float get_prob_uniform()
{
seed = rand(seed);
return(seed/1.e6);
/* In order to have a value between 0 and 1 */
}

/*******************************************/
```

This function decides if an operation is
write vs read then sets the effected values
of access time.

```
/*******************************************/

int read_write()
{
int read_chance;
float P_read_write;
P_read_write = get_prob_uniform();
if (P_read_write <= P_read )
   read_chance = 1;
else
   read_chance = 0;
return(read_chance);
}
```

2 NUMA Simulation Code

```
/*****************************************/
        Simulation Code For a Typical
              NUMA Machine
              Version 2.2
/*****************************************/


#include<stdio.h>
#include<math.h>
#define PE 8

/*********************************************/
                Global Variables
*********************************************/


int N= PE;
 /* Number of processing elements */
int active_N;
   /* Number of active processors */
float h_c =.50;
   /* Hit rate at level 1 */
float h_L = .50;
   /* Hit rate at level 2 */
float P_read = .95;
 /* Probability access is read (vs. write)*/
float P_write =1 - P_read;
   /* Probability access is write */
int read;
```

```
      /* A flag to indicate an operation */
int t_0 = 10 ;
  /* Processor cycle time */
int t_c = 10;
  /* Access time to private cache */
int t_L = 100;
   /* Access time to local memory */
int t_G = 200;
  /* Access time to remote memory */
int B = 8;
   /* ++ Burst size in words is 4 */
float seed;
/* A variable which holds a random number*/
int cache_miss = 0;
/* Level 1 cache missess */
int local_mem_miss = 0;
/* Level 2 cache missess */
float P_shared_write = .0164;
long c ;
int waiting_for_cache = 0;
int old_waiting_cache = 0;
int num_local_read = 0;
int old_local_read = 0;
int num_local_write = 0;
int old_local_write = 0;
int num_global_read = 0;
int old_global_read = 0;
int num_global_write = 0;
int old_global_write = 0;
int total_local_pending = 0;
```

```
int old_local_pending = 0;
int total_global_pending = 0;
int old_global_pending = 0;
int num_global_access = 0;
int old_global_access = 0;
int num_local_access = 0;
int old_local_access = 0;
long count_idle_time[PE] ;
long count_local_time[PE];
long count_global_time[PE];
int count = 1;                    /* A flag  */
int dontcount = 0;                /* A flag  */
FILE *fptr;
int num_cache_miss = 0;

            /*******************************************/
                        Function Prototype
            /*******************************************/


int PEi();
  /* This function simulates the microprocessor */
int f_distributed_mem();
/* This function simulates references to
  distributed memories*/
float rand(float);
/* This function generates random numbers */
float get_prob_uniform();
/* This function will provide a random address */
int wait_for_cache(int);
/* This function calculates wait time to
```

```
       access cache */
int local_bus_manager();
/* This function calculates wait time
    for a local bus */
int global_bus_manager();
/* This function calculates wait time for
       a global bus */
int read_write();
/* This function sets access values
    based on read vs write */

            /*****************************************/
                        main function
            /*****************************************/

void main()
{
int A;
int accesstime = 0;
long tot_access = 0,n,k;
float av_access;
long num_of_access = 0;
char answer;

for (n = 0 ; n < PE ; n++)
  {
  count_idle_time[n] = 0;
  count_local_time[n] = 0;
  count_global_time[n] = 0;
/*   count_local_read[n] = 0;
  count_local_write[n] = 0;
```

```
  count_global_read[n] = 0;
  count_global_write[n] = 0; */
  }

fptr = fopen("numa_test.cpp","a");

printf("\nPlease provide a random number seed
   for this experiment");
printf("\nPlease enter an odd 6
 digit number not ending in 5:");
scanf("%f",&seed);
active_N = PE;

for ( c=1;c<=100000;c++)
{
 if (active_N > 0)
  {
  num_of_access += 1;
  read = read_write();
  accesstime = PEi();
  tot_access += accesstime;
  }

for (n = 0; n <waiting_for_cache; n++)
   fprintf(fptr,"\ncount_idle_time[%d] =
      %ld",n,count_idle_time[n]);


    for(n = 0; n < waiting_for_cache  ; n++)
{
```

```
 if (c+1 == count_idle_time[n] ||
      c+1 >count_idle_time[n])
  {
  for (k = n ; k<waiting_for_cache
    - 1 ; k++)
     count_idle_time[k] =
  count_idle_time[k+1];
  waiting_for_cache -=1;

  }
}
    for(n = 0; n< num_local_access ; n++)
 {
 if ( c+1 == count_local_time[n] ||
     c+1 >count_local_time[n])
   {
   for (k=n ; k<num_local_access
    - 1; k++)
      count_local_time[k] =
  count_local_time[k+1];
      num_local_access -=1;
   }
 }

      for(n = 0; n< num_global_access ; n++)
 {
 if ( c+1 == count_global_time[n] ||
     c+1 > count_global_time[n])
   {
   for (k=n ; k<num_global_access
```

```
      - 1; k++)
      count_global_time[k] =
count_global_time[k+1];
      num_global_access -=1;
   }
 }
 old_waiting_cache = waiting_for_cache;
 old_local_read = num_local_read;
 old_local_write = num_local_write;
 old_global_read = num_global_read;
 old_global_write = num_global_write;
 old_local_access = num_local_access ;
 old_global_access = num_global_access ;

old_global_pending = old_global_read +
       old_global_write;

active_N = PE - (old_waiting_cache +
    old_local_access + old_global_access);
}
av_access = (float) tot_access / num_of_access;

fprintf(fptr,"\n___________NUMA_____________");
fprintf(fptr,"\n.........P_read......");
fprintf(fptr,"\nExpected Access Time = %5.2f ns."
      ,av_access);
fprintf(fptr,"\nCache hit rate = %f",h_c);
fprintf(fptr,"\nLocal memory hit rate = %f"
  ,h_L);
fprintf(fptr,"\nP_read = %f",P_read);
```

```
fprintf(fptr,"\nNumber of blocks = %d",B);
fprintf(fptr,"\nNumber of processor = %d",N);
fprintf(fptr,"\nProbability of shared write =
    %f",P_shared_write);
fprintf(fptr,"\n active_N = %d",active_N);
fprintf(fptr,"\n ****************************");

fclose(fptr);
}

            /***********************************************/
                This function simulates processors request for
                                memory access
            /***********************************************/

int PEi()
{
int time = 0;
float write_is_shared ;
float hit_or_miss;
      /* Cache hit or miss */
hit_or_miss = get_prob_uniform();
if (hit_or_miss <= h_c)
     {
     if (read)
     /* Access is a READ (rather than a WRITE) */
       time = wait_for_cache(count) + t_c ;
     else
     /* Access is WRITE under write-thru policy */
      {
      write_is_shared = get_prob_uniform();
```

```
if (write_is_shared <= P_shared_write)
   time = wait_for_cache(dontcount) +
       t_c + local_bus_manager() + t_L;
else
time =  wait_for_cache(count) + t_c ;
      }
     }
else
     {
     num_cache_miss +=1;
     time = f_distributed_mem();
     }

return(time);
}
```

```
/*****************************************/
```

This function simulates references to
distributed memories

```
/*****************************************/

int f_distributed_mem()
{
int time = 0;
cache_miss++ ;
float hit_or_miss;
  /* Cache hit or miss */
hit_or_miss = get_prob_uniform();
if (hit_or_miss <= h_L)
 /*    If it is in local distribted memory */
   {
    if(read)
time = wait_for_cache(dontcount) +
t_c + local_bus_manager() + B * t_L ;
    else
time = wait_for_cache(dontcount) +
t_c + local_bus_manager() + t_L ;
   }
else
    /* It is in remote distributed memory */
   {
    local_mem_miss++;
    if (read)
      time = wait_for_cache(dontcount) + t_c
      + global_bus_manager() + B * t_G ;
    else
```

```
        time = wait_for_cache(dontcount) + t_c
        + global_bus_manager() + t_G ;
    }
return(time);
}
```

```
/*****************************************/
```

This function calculates the wait time
to access the cache if there are other
processors trying to invalidate shared data

```
/*****************************************/

int wait_for_cache(int W)
{
 int n,wtime = 0;
 int num_invalidate = 0;
 float P_wait_for_cache ;

 for(n=1 ; n <= (active_N-1) ; n++)
    {
    P_wait_for_cache = get_prob_uniform();
    if (P_wait_for_cache <= P_shared_write)
/* Another processor is using the bus at this
   time to invalidate shared data  */
    num_invalidate +=1;
     }
wtime = num_invalidate * t_c;

if (wtime > 0 && W)
  {
  waiting_for_cache += 1;
   count_idle_time[old_waiting_cache] =
    c + wtime ;
   fprintf(fptr,"\n c = %ld",c);
   fprintf(fptr,"\n************************");
   for(n = 0; n< waiting_for_cache; n++)
```

```
 {

 fprintf(fptr,"\nwaiting for cache = %d",
  waiting_for_cache);
 fprintf(fptr,"\n count_idle_time[%d] =
  %ld",n,count_idle_time[n]);
 }
 fprintf(fptr,"\n**************************");
 }
return(wtime);
}

            /*****************************************/
                    This function calculates the wait
                          time for a local bus
            /*****************************************/

local_bus_manager()
{
 int n,wtime = 0;
 int it_is_read ;
 int num_local_read = 0;
 int num_local_write = 0;
 int x,y;
 float P_local_waiting ;
 long index;

 for(n=1 ; n <= active_N-1 ; n++)
  {
  P_local_waiting = get_prob_uniform();
  it_is_read = read_write();
```

```
   if ( it_is_read && P_local_waiting
   <= (1 - h_c))
 /* Another processor is accessing the
 local memory  */
    num_local_read += 1;
 else
    if (P_local_waiting <= P_shared_write)
 num_local_write += 1;
 else ;
 }

  wtime = num_local_read * B * t_L +
num_local_write * t_L ;

  if (wtime > 0)
   {
    num_local_access +=1;
    count_local_time[old_local_access] = c + wtime;
   }

  return(wtime);
 }
```

/**/

This function calculates the wait time

for a global bus

/**/

```
 global_bus_manager()
 {
 int it_is_read;
```

```
 int global_read_counter = 0;
 int global_write_counter = 0;
 int index,x,y;
 int n,wtime = 0;
 float P_wait_global ;
 for(n=1 ; n <= active_N-1 ; n++)
   {
   P_wait_global = get_prob_uniform();
   if (P_wait_global <= (1 - h_c)*(1 - h_L))
/* Other processors are using the bus
     at this time*/
   {
  it_is_read = read_write();
  if (it_is_read)
/* If another processor is performing
     READ operation */
     /*   num_global_read +=1;  */
   global_read_counter += 1;
    else
/* Another processor is performing WRITE
  operation */
      /*    num_global_write +=1; */
   global_write_counter +=1;
    }
   }

wtime = global_read_counter * B * t_G +
  global_write_counter * t_G ;

  if (wtime > 0)
```

```
   {
   num_global_access +=1;
   count_global_time[old_global_access]
 = c + wtime;
   }
return(wtime);
}
```

```
/*****************************************/
```

This function decides if an operation is
write vs read then sets the effected
values of access time.

```
/*****************************************/

int read_write()
{
int access;
float P_read_write;
P_read_write = get_prob_uniform();
if (P_read_write <= P_read )
   access = 1; /* access is READ */
else
   access = 0; /* access is WRITE */
return(access);
}
```

```
/****************************************/
```

Random number generator

```
/****************************************/

float rand(float x)
{
int i;
i = 997.0 * x / 1.e6;
x = 997.0 * x - i * 1.e6;
return(x);
}

/*********************************************/
```

This function computes a probability value
that is uniformly distributed on the interval

$$0, 1$$

```
/*********************************************/

float get_prob_uniform()
{
seed = rand(seed);
return(seed/1.e6);
/* In order to have a value between 0 and 1 */
}

/*********************************************/
```

END OF FILE

```
/*********************************************/
```

3 UMA Simulation Code

```
/*******************************************/
        Simulation Code For a Typical
              UMA Machine
              Version 2.2
/*******************************************/


#include<stdio.h>
#include<math.h>
#include<stdlib.h>
#define PE 8

/*******************************************/
              Global Variables
/*******************************************/

int N= PE;
    /* Number of processing elements */
float h_c = .50;
/* Hit rate at level 1 */
float P_read = .95;
 /* Probability access is read (vs. write)*/
float P_write =1 - P_read;
   /* Probability access is write */

float P_shared = 0.0164;
       /* Fraction of shared data */
int read;
     /* A flag to indicate an operation */
```

```
int t_0 = 10 ;
/* Processor cycle time */
int t_c = 10 ;
       /* Access time to private memory (cache) */
int t_g = 100;
       /* Access time to global memory */
int B = 8;
 /* ++ Burst size in words is 4 */
float seed;
  /* A variable which holds a random number*/

float P_local_global;
  /* Probability of being in local or global
   memory */
int cache_miss = 0;
   /* Level 1 cache missess */
int active_N ;
long count_idle_time[PE];
long Pi_idle_time[PE] ;
int Pi_waiting_global = 0;
int old_pi_waiting = 0;
long c;
int old_waiting_local = 0;
int Pi_waiting_local = 0;


FILE *fptr;
int count = 1;                    /* A flag */
int dontcount = 0;
```

```
/************************************************/
                Function Prototype
/************************************************/

int f_PE();
  /* This function simulates the microprocessor */
int f_shared_mem();
   /* This function simulates the shared-memory */
float rand(float);
      /* This function generates random numbers */
float get_prob_uniform();
/* This function will provide a random address */
int local_bus(int);
/* This function calculates wait time for a
     local bus */
int global_bus();
/* This function calculates wait time for a
   global bus */
int read_write();
/* This function sets access values based
     on read vs write */

/************************************************/
                  main function
/************************************************/

void main()
{

int accesstime = 0;
long tot_access = 0;
```

```
int k,n;
float av_access;
char answer;
long num_of_access = 0;
for (n = 0; n<N;n++)
  {
   count_idle_time[n]= 0;
   Pi_idle_time[PE] = 0;
  }

fptr = fopen("uma_test.cpp","a");

printf("\nPlease provide a random number seed
    for this experiment");
printf("\nPlease enter an odd 6 digit number
    not ending in 5:");
scanf("%f",&seed);
active_N = N;

for ( c=1;c<=100000;c++)
 {
 read =  read_write();
 if (active_N >= 1)
 {
 num_of_access +=1;
 accesstime = f_PE();
 tot_access += accesstime;
 }
 else
   fprintf(fptr,"\nAll processors are idle
```

```
and c = %ld",c);
 if (Pi_waiting_local > 0)
 {
 for (n = 0;n<Pi_waiting_local;n++)
   {
    if(c+1 == count_idle_time[n] || c+1 >
     count_idle_time[n])
    {
     for (k = n ; k<Pi_waiting_local-1 ; k++)
       count_idle_time[k] = count_idle_time[k+1];
     Pi_waiting_local -=1;
     break;
    }
   }
  }
 for (n = 0; n<Pi_waiting_global; n++)
   {
    if(c+1 == Pi_idle_time[n] || c+1 >
       Pi_idle_time[n])
    {
     for (k = n;k<Pi_waiting_global -1;k++)
       Pi_idle_time[k] = Pi_idle_time[k+1];
     Pi_waiting_global -=1;
    }
   }

 old_waiting_local = Pi_waiting_local;
 old_pi_waiting = Pi_waiting_global;
 active_N = PE - (old_waiting_local +
old_pi_waiting);
```

```
 }
av_access = (float)tot_access/(num_of_access);


fprintf(fptr,"\nAverage access time = %5.2f ns.",
      av_access);
fprintf(fptr,"\n Probability of shared
= %f",P_shared);
fprintf(fptr,"\n Cache hit rate
  = %f & N = %d",h_c, N);
fprintf(fptr,"\n Probability of READ = %f",P_read);
fprintf(fptr,"\n B = %d",B);

fprintf(fptr,"\nActive_N = %d",active_N);
fprintf(fptr,"\nPi_waiting_local = %d",
Pi_waiting_local);
fprintf(fptr,"\nPi_waiting_global
 = %d",Pi_waiting_global);
for (n=0;n<=Pi_waiting_local-1;n++)
fprintf(fptr,"\n count_idle_time[%d]
     = %ld",n,count_idle_time[n]);
for (n=0;n<Pi_waiting_global;n++)
fprintf(fptr,"\n Pi_idle_time[%d] =
 %ld",n,Pi_idle_time[n]);

printf("\nNumber of memory access = %ld",num_of_access);

fclose(fptr);
}
```

```
/*********************************************/
```

This function simulates processors request for memory access

```
/*********************************************/

int f_PE()
{
int time = 0;
float hit_or_miss;
 /* Cache hit or miss */
hit_or_miss = get_prob_uniform();
if (hit_or_miss <= h_c)   /* It is a hit */
     {
      if (read)
      /* Access is a read (rather than a write) */

  time = local_bus(count) +  t_c ;


      else
      /* Access is write under write-thru policy */
  time = local_bus(dontcount) + t_c +
    global_bus() + t_g;
     }
else
     {                    /* It is a miss */
     cache_miss += 1 ;
     time = f_shared_mem();
     }
```

```
return(time);
}

            /*************************************************/
                    This function simulates the shared-memory
            /*************************************************/

int f_shared_mem()
{
int time = 0;
    if(read)
time = local_bus(dontcount) + t_c +
 global_bus() + B * t_g ;
     else
time = local_bus(dontcount)
      + t_c + global_bus() + t_g;
return(time);
}

            /*************************************************/
                    This function calculates the wait time for
                                a local bus
            /*************************************************/

 int local_bus(int W)
 {
  int n,wtime = 0;
  int pending_write = 0;
  float P_wait_local ;

  for(n=1 ; n <= (active_N-1) ; n++)
```

```
      {
       P_wait_local = get_prob_uniform();
       if (P_wait_local <= (P_read +
   P_write)*P_shared)

  /* Another processor is using the bus at this
     time to invalidate shared data  */

     pending_write += 1;
}

  wtime = pending_write * t_c;
  if (wtime > 0 && W)
   {
    Pi_waiting_local += 1;
    count_idle_time[old_waiting_local] = c + wtime ;
   }
 return(wtime);
 }
```

/**/

This function calculates the wait time
for a global bus

/**/

```
 global_bus()
 {
 int it_is_read;
 int  read_waiting_global= 0;
 int write_waiting_global= 0;
 int n,wtime = 0;
```

```
 float P_wait_global ;
 for(n=1 ; n <= active_N -1 ; n++)
    {
    P_wait_global = get_prob_uniform();
    if (P_wait_global <= (1 - h_c))
/* Other processors are using the bus at this time*/
      {
       it_is_read = read_write();
       if (it_is_read)
/* If another processor is performing
   READ operation */
       read_waiting_global +=1;
       else
/* Another processor is performing WRITE operation */
    write_waiting_global +=1;
}
}
 wtime = read_waiting_global * B * t_g +
 write_waiting_global * t_g;

 if (wtime > 0)
  {
   Pi_waiting_global +=1;
   Pi_idle_time[old_pi_waiting] = c + wtime;
  }
 return(wtime);
 }
```

```
/*************************************************/
```

This function decides if an operation is write vs
read then sets the effected values of access time.

```
/*************************************************/

int read_write()
{
int access;
float P_read_write;
P_read_write = get_prob_uniform();
if (P_read_write <= P_read )
   access = 1; /* access is READ */
else
   access = 0; /* access is WRITE */
return(access);
}

            /*********************************************/
```

Random number generator

```
            /*********************************************/

float rand(float x)
{
int i;
i = 997.0 * x / 1.e6;
x = 997.0 * x - i * 1.e6;
return(x);
}
```

```
/*********************************************/
```

This function computes a probability value that
is uniformly distributed on the interval [0,1]

```
/ /*********************************************/
```

```
float get_prob_uniform()
{
seed = rand(seed);
return(seed/1.e6);
/* In order to have a value between 0 and 1 */
}
```

```
/*********************************************/
```

END OF FILE

```
/*********************************************/
```

4 LRG Simulation Code

```
/***********************************************/
                Simulation Code For
              Local-Remote-Global
                  Architecture
                  Version 4.3
/***********************************************/

#include<stdio.h>
#include<math.h>
#define PE 8

/***********************************************/
                Global Variables
/***********************************************/


int N = PE;
      /* Number of processing elements */
int NL=2;
     /* Number of processing elements on board */
float h_1 = .50;
  /* Hit rate at level 1 */
float P_local= .50;
/* Fraction of shared data in local memory */
float P_global;
/* Fraction of shared data in global memory*/
float P_read = .95;
/* Probability access is read (vs. write)*/
float P_write =1-P_read;
```

```
   /* Probability access is write */
int read;
/* A flag to indicate an operation */
int t_1 ;
/* Access time to promary (internal) cache */
int t_local ;
   /* Access time to local memory */
int t_global;
  /* Access time to global memory */
int B = 8;
   /* ++ Burst size in words is 4 */
float seed;
   /* A variable which holds a random number*/
float hit_or_miss;
     /* Cache hit or miss */
int t_burst_L = 20;
/* Time it takes to operate on words
when burst*/
int t_burst_FL = 20;
/* Time it takes to operate on words when burst*/
int t_burst_G = 40;
/* Time it takes to operate on words when burst*/
int snoop_time_local = 0;
/* Snooping time */
int snoop_time_global = 0;
/* ++ local and global snoop times are different */
float P_snoop_system = 0.05;
/* Probability of snoop hit on system bus */
float P_snoop_local = 0.05;
/* Probability of snoop hit on local bus */
```

```
float P_single = 0.01;
/* Probability that access is single */
int single;
/* This variable holds value of 1 or 0 to indicate
     the type of operation as single or burst */
float P_shared_write = .0164;
float P_local_global;
/* Probability of being in local or global memory */
int copyback=0;
/* This variable holds 1 or 0 to indicate the mode */
int level_1_misses = 0;
    /* Level 1 cache missess */
int level_2_misses = 0;
     /* Level 2 cache missess */
int active_PE;
int num_local_busy = 0;
int old_local_busy = 0;
long count_local_time[PE];
int num_global_busy = 0;
int old_global_busy = 0;
long count_global_time[PE];
long c;
long num_of_access = 0;
FILE *fptr;
```

/**/

Function Prototype

/**/

```
int fprocessor();
```

```
/* This function simulates the
  microprocessor (88110) */
int f_cachectrlr();
/* This function simulates the
cache controller (88410) */
int f_local_global();
   /* This function simulates Addr ASICs */
float rand(float);
/* This function generates random numbers */

float get_prob_uniform();
/* This function will provide a
  random address */
int local_bus_time();
/* This function calculates wait time
 for a local bus */
int global_bus_time();
/* This function calculates wait time for
    a global bus */
int pick(int,int);
/* This function picks the largest n
 umber among two */
void read_write();
/* This function sets access values
 based on read vs write */
void burst_vs_single();
/* This function makes decision whether
      operation is burst */
```

```
/*****************************************/
                main function
/*****************************************/

void main()
{

int accesstime = 0;
long tot_access = 0;
int n , k;
float av_access;
char answer;

fptr = fopen("L_R_G.cpp","a");


printf("\nPlease provide a random number
  seed for this experiment");
printf("\nPlease enter an odd 6 digit number
  not ending in 5:");
scanf("%f",&seed);


for ( n = 0 ; n < N ; n++)
 {
 count_local_time[n] = 0;
 count_global_time[n] = 0;
 }

active_PE = PE;
```

```
for ( c = 1; c <= 100000; c++)
 {
  if (active_PE > 0)
  {
   num_of_access +=1,
   read_write();
   accesstime = fprocessor();
   tot_access += accesstime;
  }
 for(n = 0; n < num_local_busy  ; n++)
{
     if (c+1 == count_local_time[n] ||
 c+1 > count_local_time[n])
  {
  for (k = 0 ; k<num_local_busy -
  1  ; k++)
    count_local_time[k] =
      count_local_time[k+1];
  for (n = 0; n<num_local_busy ;n++)
  num_local_busy -=1;
  }
}

 for(n = 0; n < num_global_busy  ; n++)
{
     if (c+1 == count_global_time[n] || c+1 >
 count_global_time[n])
  {
  for (k = 0 ; k<num_local_busy - 1  ; k++)
```

```
  count_global_time[k] =
  count_global_time[k+1];
  for (n = 0; n<num_local_busy ;n++)
  num_global_busy -=1;
  }
       }
old_local_busy = num_local_busy;
old_global_busy = num_global_busy;
active_PE = N - (old_local_busy + old_global_busy);

}
av_access =(float) tot_access/(num_of_access);

fprintf(fptr,"\n%%%%%%%%%%%%P_read%%%%%%%%%%%%%%");
fprintf(fptr,"\n*********************************");
fprintf(fptr,"\nExpected Access Time =
     %5.2f ns.",av_access);
fprintf(fptr,"\nCache hit rate = %f",h_1);
fprintf(fptr,"\nLocal memory hit rate
   = %f",P_local);
fprintf(fptr,"\nP_read = %f",P_read);
fprintf(fptr,"\nNumber of blocks = %d",B);
fprintf(fptr,"\nNumber of processor = %d",N);
fprintf(fptr,"\nProbability of shared write = %f",
  P_shared_write);
fprintf(fptr,"\n*********************************");

fclose(fptr);
}
```

```
/*********************************************/
        This function simulates processors request
                    for memory access
/*********************************************/

int fprocessor()
{
int time = 0;
float is_it_private;
hit_or_miss = get_prob_uniform();
if (hit_or_miss <= h_1)
   {
    if (read)
     /* Access is a read (rather than a write) */
      time = t_1;
    else
    /* Access is write under write-thru policy */
     {
      P_local_global = get_prob_uniform();
      is_it_private = get_prob_uniform();
      if(is_it_private <= (P_read + P_write)*
  P_shared_write)
      {
       if(P_local_global <= P_local)
       time =  t_1 + local_bus_time() + t_local;
       else
       time = t_1 + local_bus_time() + t_local +
       global_bus_time() + t_global;
       }
  else
```

```
        time = t_1;

 }
        }
else
        time = f_local_global();

return(time);
}
```

/***/

This function simulates local and global

memory references

/***/

```
 int f_local_global()
 {
  float P_snoop_hit;
  int wait_for_bus,waittime,time = 0;
  level_1_misses ++;
  /* To count number of cache miss */
  P_local_global = get_prob_uniform();
  if (P_local_global <= P_local)
  /* If the location is in local memory */
{
wait_for_bus = local_bus_time();
waittime = wait_for_bus;
time =  t_1 + waittime + t_local;
}
  else
  /* If the location is in global memory */
```

```
    {
    wait_for_bus = global_bus_time();
    waittime = wait_for_bus;
    time = t_1 + local_bus_time() + t_local
    + waittime + t_global ;
    }
  return(time);
  }

/*********************************************/ This function calculates the
                                    wait time
                                  for a local bus
                 /*********************************************/

 local_bus_time()
{
  int n,wtime = 0;
  int local_busy_single = 0;
  int local_busy_burst = 0;
  int t_local_0 ;   /* overlapped access time */
  float P_wait_local ;
  t_local_0 = t_burst_L*3 ;
  for(n=1 ; n <= (NL-1) ; n++)
    {
     P_wait_local = get_prob_uniform();
     if (P_wait_local <= P_local*(1 - h_1))
/* Other processor is using the bus at this time*/
  {
  if (single)
       local_busy_single =1;
  else
```

```
      local_busy_burst =1;
 }

 wtime += local_busy_single * t_local_0 +
 local_busy_burst * t_local_0 +
(B-1)*t_burst_L;
  }
 if (wtime > 0 )
   {

   num_local_busy +=1;
 /*  if (num_local_busy > N/2)
    num_local_busy = N/2;   */

   count_local_time[old_local_busy] = c + wtime;

   }

return(wtime);
}
```

/**/

This function calculates the wait time

for a global bus

/**/

```
global_bus_time()
{
int n,wtime = 0;
int global_busy_single = 0;
int global_busy_burst = 0;
```

```
 int t_global_0 ;
 float P_wait_global ;
 t_global_0 = t_burst_G*2 ;
 /* overlapped access time */
 for(n=1 ; n <= (N-1) ; n++)
    {
    P_wait_global = get_prob_uniform();
    if (P_wait_global <= P_global*(1 - h_1))
/* Other processors are using the bus at this time*/
       {
if (single)
  global_busy_single +=1;
else
   global_busy_burst +=1;
}
}
 wtime = global_busy_single * t_global_0 +
global_busy_burst * ( t_global_0 +
(B-1)*t_burst_G );
  if (wtime > 0)
    {

     num_global_busy += 1;
     count_global_time[old_global_busy] = c + wtime;
    }

 return(wtime);
 }
```

```
/*****************************************************/
```

This function picks the largest number

```
/*****************************************************/

 int pick(int x,int y)
 {
  if (x > y)
return(x);
  else
return(y);
 }

/*****************************************************/ This function decides
```

if an operation is write vs read then sets the effected values of access time.

```
/*****************************************************/

 void read_write()
 {
 float P_read_write;
 P_read_write = get_prob_uniform();
 if (P_read_write <= P_read )
{

 /* If operation is read then these time
  variables will be set as follows in ns.*/

t_1 = 10;
t_local = 100;
t_global = 200;
read = 1;
}
```

```
 else
{

 /* If operation is write then these time
  variables will be set as follows in ns.*/

t_1 = 10;
t_local = 100;
t_global = 200;
read = 0;
}
 }
```

/****(*****/

This function makes decision if an operation
is single or burst

/*****/

```
 void burst_vs_single()
 {
 float P_burst_single;
 P_burst_single = get_prob_uniform();
 if ( P_burst_single <= P_single )
single = 1;
 else
single = 0;
 }
```

```
/*********************************************/
```

Random number generator

```
/*********************************************/

float rand(float x)
{
int i;
i = 997.0 * x / 1.e6;
x = 997.0 * x - i * 1.e6;
return(x);
}

/*********************************************/
```

This function computes a probability value that
is uniformly distributed on the interval

$$0, 1$$

```
/*********************************************/

float get_prob_uniform()
{
seed = rand(seed);
return(seed/1.e6);
/* In order to have a value between 0 and 1 */
}

/*********************************************/
```

END OF FILE

```
/*********************************************/
```

5 RCR Cost-Effectiveness Code

/**/

Code For Calculation of Cost Savings

Factor For RCR Architecture with Various

Number of PEs

Version 2.0

/**/

CODE FOR CALCULATION OF COST SAVINGS FACTOR For RCR ARCHITECTURE WITH VARIOUS NUMBER OF PEs.

```
#include<stdio.h>
#include<math.h>


void main()
{
FILE *fptr;
int N,n ;
int M = 1;
int C_D = 1;
int C_S = 6;
float M_rep ;
float sav_4 = 0.0;
float sav_8 = 0.0;
float sav_16 = 0.0;
float sav_32 = 0.0;
float sav_64 = 0.0;
float sav_128 = 0.0;
float cost_without_cache = 0.0 ;
float cost_savings = 0.0 ;
```

```
float cost_factor = 0.0;
fptr = fopen("Scost_r.cpp","a");
fprintf(fptr,"\n\\begin{table}");
fprintf(fptr,"\n\\caption{Cost Savings
  Factor For Various Number of PEs.}");
fprintf(fptr,"\n\\label{costfact}");
fprintf(fptr,"\n\\begin{center}");
fprintf(fptr,"\n\\begin{tabular}{|c|c|c|c|c|c|c|}
       \\hline");
fprintf(fptr,"\n$M_{rep}$ & $N=4$ & $N=8$ &
$N=16$ & $N=32$ & $N=64$ & $N=128$ \\\\
\\hline\\hline");
for ( n=1;n<10; n++)
{
for (N = 4 ; N<= 128; N= N*2)
 {
  M_rep = n/10.0;
  cost_without_cache = (float)M * N * C_S ;
  cost_savings = M * M_rep * (N * C_S - C_D);
  cost_factor = cost_savings/cost_without_cache;
  if (N == 4)
sav_4 = cost_factor * 100;
  else if (N == 8)
sav_8 = cost_factor * 100;
  else if (N == 16)
sav_16 = cost_factor * 100;
  else if (N == 32)
sav_32 = cost_factor * 100;
  else if (N == 64)
sav_64 = cost_factor * 100;
```

```
  else
sav_128 = cost_factor * 100;
 }
 fprintf(fptr,"\n %4.1f & %4.1f & %4.1f & %4.1f
 & %4.1f & %4.1f & %4.1f \\\\ \\hline", M_rep, \
 sav_4, sav_8, sav_16, sav_32, sav_64, sav_128);
}
fprintf(fptr,"\n\\end{tabular}");
fprintf(fptr,"\n\\end{center}");
fprintf(fptr,"\n\\end{table}");
fclose(fptr);
}
```

6 List of Symbols, Abbreviations, and Nomenclature

Symbol	Represents	Relationship
A_i	Number of Memory Accesses	$A_i = R_i + W_i$
A_i^s	Number of Shared Memory Accesses	
$\bar{A}_i^s$	The Weighted Average Shared References	
B	Block Size	
C_i	Cache i	$i = 0, 1, 2, .., N$
$COMA$	Cache Only Memory Architecture	
C_D	Cost per Word for DRAM	
C_S	Cost per Word for SRAM	
C_{saving}	Cost Savings	
CSM	Cache Shared Memory	
CU	Control Unit	
CU_i	Control Unit i	$i = 0, 1, 2, ..., N$
D	Delay in Terms of Memory Cycles	
d	Number of Ports in a Multiport Memory Unit	
δ	an Arbitrary Distance from Lowest or Highest Addresses Currently Stored in Replicatred Memory	
D_i	Delay by P_i	
D_{IOF}	the Distance Outside the Fence	
Dir_i	Directory i	$i = 0, 1, 2, ..., N$
DS	Data Stream	
$E(N)$	System Efficiency for an N-Processor System	$E(N) = \frac{S(N)}{N}$

Symbol	Represents	Relationship
GSM	Global Shared Memory	
H	Highest Address Currently	
	Stord in Replicated Memory	
h	Hit Ratio	
h_L	Hit Ratio on Local Memory	
h_R	Hit Ratio on Replicated Memory	
IS	Instruction Stream	
L	Lowest Address Currently	
	Stord in Replicated Memory	
LM	Local Memory	
LRF	Lower Replicated Fence	
LRG	Local-Remote-Global	
M	Shared-Memory Size in Words	
M_c	Number of Memory Cycle	
M_i	Memory Unit i	$i = 0, 1, 2, ..., N$
MM	Main Memory	
MM_i	Main Memory i	$i = 0, 1, 2, ..., N$
M_{rep}	Percentage of Shared-Memory	
	That is Spatially Cached	
N	Total Number of Processors	
	in the System	
$NUMA$	Non-Uniform Memory Access Model	
P	A Processor	
PC	Processor Consistency Model	
PE	Processing Element	
p_e	Performance for the Entire Task	
	Using Enhancement	
P_i	Processor i in the System	$i = 0, 1, ..., N$

Symbol	Represents	Relationship
p_{ne}	Performance for the Entire Task Without Enhancement	
P_{IOF}	Probability of a Word Being Incrementally outside the Fence	
P_L	Probability of Data Being in Local Memory	
P_{shared_write}	Probability of a Shared WRITE	
R	Number of READ References During Interval	$\sum_{i=1}^{N} R_i = R$
R_i	Number of READ References by P_i	$\sum_{i=1}^{N} R_i = R$
R_i^s	Number of Shared data READs	
RC	Release Consistency Memory Model	
RCR	Replicated Concurrent-Read	
t_{miss}^R	Read-Miss Time	
S	Speedup	
$S(N)$	Speedup Factor for an N-Processor System	$S(N) = \frac{T(1)}{T(N)}$
t_{ave}	the Average Memory Access Time	
t_{sync}	Synchronization Time	
T	Execution Time	
t_{busy}	Processor Busy Time	
t_{read}	Average READ Time	
t_{miss}^W	Write-Miss Time	

Symbol	Represents	Relationship
t_w	Time it Takes to Write a Word to Replicated Memory	
t_G	Time it Takes to Access Global Memory	
t_c	Cache Clock Cycle Time	
t_{aux}	Time it Takes to Access Auxiliary Memory	
t_M	Time to Access Replicated Memory	
t_L	Time it Takes to Access Local Memory	
t_{Remote}	Time it Takes to Access Global Memory	
t_{read}^{LRG}	the Average READ Time for LRG Configuration	
t_{write}^{LRG}	the Average WRITE Time for LRG Configuration	
$t_{wait\_local\_bus}^{LRG}$	the Waiting Time for Local Bus for LRG Configuration	
$t_{wait\_global\_bus}^{LRG}$	the Waiting Time for Global Bus for LRG Configuration	
t_{ave}^{LRG}	the Expected Memory Access Time for LRG Configuration	
t_{read}^{NUMA}	the Average READ Time for NUMA Configuration	
t_{write}^{NUMA}	the Average WRITE Time for NUMA Configuration	

Symbol	Represents	Relationship
$t^{NUMA}_{wait\_local\_bus}$	the Waiting Time for Local Bus for NUMA Configuration	
$t^{NUMA}_{wait\_global\_bus}$	the Waiting Time for Global Bus for NUMA Configuration	
t^{NUMA}_{ave}	the Expected Memory Access Time for NUMA Configuration	
$t^{NUMA}_{wait\_pending\_write}$	the Time a Processor may have to Wait to Access Local Memory (NUMA Machine)	
t^{RCR}_{read}	the Average READ Time for RCR Configuration	
t^{RCR}_{write}	the Average WRITE Time for RCR Configuration	
$t^{RCR}_{wait\_global\_bus}$	the Waiting Time for Global Bus for RCR Configuration	
t^{RCR}_{ave}	the Expected Memory Access Time for RCR Configuration	
t^{UMA}_{read}	the Average READ Time for UMA Configuration	
t^{UMA}_{write}	the Average WRITE Time for UMA Configuration	
t^{UMA}_{ave}	the Expected Memory Access Time for UMA Configuration	
$t^{UMA}_{wait\_global\_bus}$	the Waiting Time for Global Bus for UMA Configuration	

Symbol	Represents	Relationship
$t^{UMA}_{wait\_pending\_write}$	the Time a Processor may have to Wait to Access Local Memory (UMA Machine)	
$T(1)$	Execution Time Steps in a Uniprocessor	$T(1) = O(1)$
$T(N)$	Execution Time Steps for N-Processor System	
TSO	Total Store Order Weak Consistency Model	
UMA	Uniform Memory Access Model	
URF	Upper Replicated Fence	
W	Number of WRITE References During Interval	$W = \sum_{i=1}^{N} W_i$
W_i	Number of WRITE References by P_i	$W = \sum_{i=1}^{N} W_i$
W^s	Number of WRITE References to Shared Data	
W_i^s	Number of WRITE References to Shared Data by P_i	

7 References

[ADVE91] Adve, Adve, Hill, Vernon, "Comparison of Hardware and Software Cache Coherence Schemes," *Proceedings of the 18th International Symposium on Computer Architecture*, 19(3):298-308, May 1991.

[AGARWAL88] Agarwal, Simoni, Hennessy, Horowitz, "An Evaluation of Directory Schemes for Cache Coherence," *Proceedings of the 15th International Symposium on Computer Architecture*, 16(2):280-289, May 1988.

[ALLEN78] Allen, *Probability, Statistics, and Queueing Theory With Computer Science Applications*, Academic Press, Inc., 1978.

[ARCHIBALD84] Archibald, Baer, "An Economical Solution to the Cache Coherence Problem," *Proceedings of the International Symposium on Computer Architecture*, pages 355-362, 1984.

[ARCHIBALD86] Archibald, Baer, "Cache Coherence Protocols: Evaluation Using a Microprocessor Simulation Model," *ACM Transactions on Computer Systems*, 4(4):273-298, November 1986.

[BAGNOLI93] Bagnoli, Casamatta, Lazzari. "Multiprocessor System Having Global Data Replication," *U.S. Patent Documents*, US005214776, May 1993.

[BARROSO93] Barroso, DuBois, "The Performance of Cache-Coherent Ringbase Multiprocessors," *IEEE Computer*, pages 268-277, March 1993.

[BELL92] Bell, "Ultracomputer: A Teraflop Before Its Time," *Communication ACM*, 35(8):27-47, MONTH 1992.

[BERTONI91] Bertoni, Wang, "Multiple Interleaved Bus Architectures," *Proceedings of the 1991 International Conference on Parallel Processing*, pages 1-8, August 1991.

[BUCHER90] Bucher, Calahan, "Access Conflicts in Multiprocessor Memories Queing Models and Simulation Studies," *Proceedings of the International Conference on Supercomputing. Computer Architecture News*, 18(3):428-438, September 1990.

[CENSIER78] Censier, Feautrier, "A New Solution to Coherence Problems in Multicache Systems," *IEEE-Transactions on Computers*, 27(12):1112-1118, December 1978.

[CHAIKEN90] Chaiken, Fields, Kurihara, Agarwal, "Directory-Based Cache Coherence in Large-Scale Multiprocessors," it IEEE Computer, 23(6):49-58, June 1990.

[CHEONG88] Cheong, Veidenbaum, "A Cache Coherence Scheme With Fast Selective Invalidation," *Proceedings of the 15th International Symposium on Computer Architecture*, 16(2):299-307, May 1988.

[COSTA93] Costa, Leonardi, "Multiprocessor System Having Distributed Shared Resources an Dynamic Global Data Replication," *U.S. Patent Documents*, US005247673, September 1993.

[CRAWFORD94] Crawford, DeMara, "Cache Coherence in a Multiport Memory Environment," *Proceedings of the 1st International Conference on Massively Parallel Computing Systems*, May 1994.

[DEMARA95] DeMara, Crawford, "Cache Coherence Strategies for Multiported Shared Memory Architectures," In Preparation for Submission to *Journal of Parallel and Distributed Computing.*

[DEMARA93] DeMara, Moldovan, "The SNAP-1 Parallel Prototype," *IEEE Transactions on Parallel and Distributed Systems*, 4(8):841-854, August 1993.

[DEMARA94] DeMara, Motlagh, Lin, Kuo, "Barrier Synchronization Techniques for Distributed Process Creation," *IEEE International Parallel Processing Symposium*, Cancun, Mexico, April 1994.

[DEMARA92] DeMara, *Parallelism, Design, and Performance of a Marker- Propagation Reasoning System*, Ph.D. Dissertation, University of Southern California, Department of EE-Systems, 1992.

[DUBOIS82] DuBois, Briggs, "Effects of Cache Coherency in Multiprocessors," *Proceedings of the 9th International Symposium on Computer Architecture*, 10(3):299-308, April 1982.

[DUBOIS90a] DuBois, Briggs, "Tutorial Notes on Shared-Memory Architectures for Multiprocessors," *Proceedings of the 17th Symposium on Computer Architecture*, Seattle, WA, 1990.

[DUBOIS92a] DuBois, "Delayed Consistency," in DuBois and Thakkar (eds.), *Scalable Shared-Memory Multiprocessors*, Kluwer Academic Publishers, Boston, MA, 1992.

[DUBOIS90a] DuBois, Thakkar (eds.), *Cache and Interconnect Architectures in Multiprocessors*, Kluwer Academic Publishers, Boston, MA, 1990.

[DUBOIS92b] DuBois, Thakkar (eds.), *Scalable Shared-Memory Multiprocessors*, Kluwer Academic Publishers, Boston, MA, 1992.

[DUBOIS86] DuBois, Scheurich, Briggs, "Memory Access Buffering in Multiprocessors," *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 434-442, 1986.

[DUBOIS88] DuBois, Scheurich, Briggs, "Synchronization, Coherence and Event Ordering in Multiprocessors," *IEEE Computer*, 21(2), 1988.

[DUBOIS91] DuBois, Wang, "Shared Block Contention on a Cache Coherence Protocol," *IEEE Transactions on Computers*, 40(5):640-644, May 1991.

[EGGERS89] Eggers, Katz, "The Effect of Sharing on the Cache and Bus Performance of Parallel Programs," *ASPLOS-III Proceedings, Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 257-270, April 1989.

[FLEISCH88] Fleisch, "Distributed Shared Memory in a Loosely Coupled Distributed System," *IEEE Computer*, pages 182-184, 1988.

[GHARACHORLOO92a] Gharachorloo, Adve, Gupta, Hennessy, Hill, "Programming for Different Memory Consistency Model," *Journal of Parallel and Distributed Computing*, August 1992.

[GHARACHORLOO92b] Gharachorloo, Gupta, Hennessy, "Hiding Memory Latency Using Dynamic Scheduling in Shared-Memory Multiprocessors," *Proceedings of the 19th Annual Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.

[GHARACHORLOO92c] Gharachorloo, Gupta, Hennessy, "Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors," *Proceedings of the fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1992.

[GHARACHORLOO90] Gharachorloo, Lenoski, Laudon, Gibbons, Hennessy, "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors," *Proceedings of the 17th Annual International Symposium on Computer Architecture*, June 1990.

[GHARACHORLOO91] Gharachorloo, Traub, "Multithreading: A Revisionist View of Dataflow Architecture," *Proceedings of the 18th Annual International Symposium on Computer Architecture*, May 1991.

[GIANCOLI84] Giancoli, *General Physics*, Prentice Hall, Inc., 1984.

[GUPTA91] Gupta, Hennessy, Gharachorloo, Mowry, Weber, "Computative Evaluation of Latency Reducing and Tolerating Techniques," *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 243-256, April 1989.

[GUPTA89] Gupta, Weber, "Analysis of Cache Invalidation Patterns in Multiprocessors," *ASPLOS-III Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 243-256, April 1989.

[HAMMING91] Hamming, *The Art of Probability for Scientists and Engineers*, Addison-Wesley Publishing Company, 1991.

[HENNESSY96] Hennessy, Patterson, *Computer Architecture A Quantitative Approach*, Morgan Kaufmann Publishers, Inc., 1996.

[HWANG93] Hwang, *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, McGraw-Hill, Inc., 1993.

[HWANG84] Hwang, Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, Inc., 1984.

[IRANI88] Irani, Naji, "Performance Study of a Clustered Shared-Memory Multiprocessor," *Proceedings of the International Conference on Supercomputing*, pages 304-313, July 1988.

[JAIN91] Jain, *The Art of Computer Systems Performance Analysis*, John Wiley & Sons, Inc., 1991.

[JAGADISH89] Jagadish, Kumar, Patnaik, "An Efficient Scheme for Interprocessor Communication Using Dual-Port RAMs," *IEEE Micro*, pages 10-19, 1989.

[JAMES90] James, Laundrie, Gjessing, Sohi, "Scalable Coherent Interface," *IEEE Computer*, 23(6):74-77, June 1990.

[KATZ85] Katz, Eggers, Wood, Perkins, Sheldon, "Implementing a Cache Consistency Protocol," *Proceedings of the 12th International Symposium on Computer Architecture*, 12(3):276-283, June 1985.

[KESSLER89] Kessler, Livny, "An Analysis of Distributed Shared-Memory Algorithms," *IEEE Computer*, pages 498-505, June 1989.

[LENOSKI95] Lenoski, Weber, *Scalable Shared-Memory Multiprocessing*, Morgan Kaufmann Publishers, Inc., 1995.

[LENOSKI93] Lenoski, Laudon, Joe, Nakahira, Stevens, Gupta, Hennessy, "The DASH Prototype: Logic Overhead and Performance," *IEEE Transactions on Parallel and Distributed Systems*, 4(1):41-61, January 1993.

[LENOSKI90] Lenoski, Laudon, Gharachloo, Gupta, Hennessy, "The Directory- Based Cache Coherence Protocol for the DASH Multiprocessor," *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148-159, 1990.

[LENOSKI92] Lenoski, Laundon, Gharachorloo, "The Stanford DASH Multiprocessor," *IEEE Computer*, pages 63-79, March 1992.

[MANO82] Mano, *Computer System Architecture*, Prentice-Hall, Inc., 1982.

[MILTON86] Milton, Arnold, *Probability and Statistics in the Engineering and Computing Sciences*, McGraw-Hill, Inc., 1986.

[MIN90] Min, Baer, Kim, "An Efficient Caching Support for Critical Sections in Large-Scale Shared Memory Multiprocessors," *Proceedings of the International Conference on Supercomputing*, 18(3):34-47, June 1990.

[NITZBERG91] Nitzberg, Lo, "Distributed Shared Memory: A Survey of Issues and Algorithms," *IEEE Computer*, 24(8):52-60, Augustt 1991.

[O'KRAFKA] O'Krafka, Newton, "An Empirical Evaluation of Two Memory- Efficient Directory Methods," *Proceedings of the International Symposium on Computer Architecture*, 18(2): 138-147, June 1990.

[OLKIIN94] Olkin, Gleser, Derman, *Probability Models and Applications*, Macmillan College Publishing Company, 1994.

[OWICKI89] Owicki, Agarwal, "Evaluating the Performance of Software Cache Coherence," *ASPLOS-III Proceedings, Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 230-242, April 1989.

[PATTERSON94] Patterson, Hennessy, *Computer Organization & Design: The Hardware/Software Interface*, Morgan Kaufmann Publishers, Inc., 1994.

[PFEIFFER73] Pfeiffer, Schum, *Introduction to Applied Probability*, Academic Press, Inc., New York, 1973.

[SCHEURICH89] Scheurich, DuBois, "Dynamic Page Migration in Multiprocessors with Distributed Global Memory," *IEEE Transactions on Computers*, 38(8), August 1989.

[SINDHU92] Sindhu, Frailong, Cekleov, "Formal Specification of Memory Modules," in DuBois and Thakkar (eds.), *Scalable Shared-Memory Multiprocessors*, Kluwer Academic Publishers, Boston, MA, 1992.

[SINGH92] Singh, Weber, Gupta, "SPLASH: Stanford Parallel Applications for Shared-Memory," *Computer Architecture News*, 20(1):5-44, March 1992.

[SKILLICORN92] Skillicorn, "Architecture Independent Parallel Computation," *IEEE Computer*, pages 38-49, December 1992.

[STENSTRÖM90] Stenström, "A Survey of Cache Coherent Schemes for Multiprocessors," *IEEE Computer*, 23(6):12-24, June 1990.

[STODLECK89] Stodleck, "The IDT FourPort^{TM} RAM Facilitates Microprocessor Designs," *Application Note AN-43*, pages 1-13, 1989.

[STUNKEL92] Stunkel, Fuchs, "An Analysis of Cache Performance for a Hypercube Multicomputer," *IEEE Transactions of Parallel and Distributed Systems*, 3(4):421-432, July 1992.

[TI93] Texas Instruments, *MOS Memory Commercial and Military Specifications Data Book*, Texas Instruments, Inc., 1993.

[THAKKAR90] Thakkar, DuBois, Laundrie, Sohi, "Scalable Shared-Memory Multiprocessor Architectures," *IEEE Computer*, 23(6):71-74, June 1990.

[THAPAR90] Thapar, Delagi, "Stanford Distributed-Directory Protocol," *IEEE Computer*, 23(6):78-80, 1990.

[THAPAR91] Thapar, Delagi, "Cache Coherence for Large Scale Shared-Memory Multiprocessors," *Computer Architecture News*, 19(1):114-119, March 1991.

[VERNON88] Vernon, Lazowska, Zahorjan, "An Accurate and Efficient Performance Analysis Technique for Multiprocessor Snooping Cache Consistency Protocols," *Proceedings of the 15th Annual Symposium on Computer Architecture*, 16(2):308-315, May 1988.

[WEISS91] Weiss, "Multiple-Port Memory Access in Decoupled Architecture Processors," *Proceedings of the 18th International Symposium on Computer Architecture*, pages 373-376, August 1991.

[ZULIAN90] Zulian, "Multiprocessor System Featuring Global Data Multiplation," *U.S. Patent Documents*, US4928224, May 1990.