MCOMA: A Multithreaded COMA Architecture

Halima M. El Naga

CENG 99-03

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213-740-4484)
May 1999

MCOMA: A Multithreaded COMA Architecture

by

Halima M. El Naga

A Dissertation Presented to the
FACULTY OF THE GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(Computer Engineering)

May 1999

Dedication

To my family.

Acknowledgments

I wish to thank my thesis advisor, Dr. Jean-Luc Gaudiot for his continuous support and encouragement through my research. Only with his advisement and friendship have I been able to complete this work. My sincere thanks and deep gratitude to my committee member Dr. Alice Parker. She was always there for me at the hardest times. Without her continuous support, this work would have never been completed. I also wish to thank Dr. Mary Hall who contributed much time towards improving the final form of this thesis.

I also like to thank all the members of Parallel and Distributed Processing Center for many interesting discussions and suggestions that contributed to the improvement of this work. I am thankful to Joonho Ha for providing me with the compiled running copy of the applications that I used in this research.

On a personal note, I like to thank my family for their understanding during the course of my graduate study. In particular, I like to thank my husband, Nagi for everything. Without his encouragement, I wouldn't have started my study and without his continuous love, help, support, patience, understanding, ... I wouldn't have been able to survive my graduate school.

Abstract

MCOMA: A Multithreaded COMA Architecture

Shared memory multiprocessor architectures provide a flexible and powerful computing systems. In current technology, the performance of these systems is limited by the large gap between processors and the memory access speed. The interconnection network bandwidth is another performance limiting factor. Data allocation affects the memory access latency and the interconnection contention. Shared memory multiprocessor systems can be divided into two main dominating architectures: the non-uniform memory architecture (NUMA) and the cache only memory architecture (COMA). In NUMA, data is associated with a fixed physical address, which is known as the home node of the data, which is different from COMA that allows data migration and replication.

The cache only memory architecture, COMA, exploits data locality by supporting automatic data migration in hardware but suffers from long communication latencies. Multithreading is a latency hiding technique that improves the processor utilization by activating a new thread while other threads are waiting on events. By combining COMA and multithreading, multithreading will benefits from the dynamic data migration of the COMA and, at the same time, it will hide long communication latencies of COMA which will improve the overall system performance.

In this thesis, we describe new cache only memory architecture, MCOMA, whose performance supersedes any of the existing COMA architecture. The system consists of a number of nodes connected by a mesh interconnection network to move data between nodes. Each group of nodes has a group directory. The group directories are connected by a separate search interconnection to improve the data search.

To improve the performance further, the new architecture is combined with multithreading. This architecture benefits from the data locality of COMA and the latency tolerance technique of multithreading. We developed a simulator to evaluate and study the performance of the new system. The simulator is execution driven, which is developed on top of MINT, a MIPS interpreter. We evaluated the performance of the architecture using a subset of SPLASH benchmark suite of parallel application programs covering different problem domains.

The performance evaluation has been done for a wide variation of architecture parameters. The effect of multithreading on execution time, cache hit rates, attraction memory hit rates and group hit rates of the system have been evaluated and the results have been discussed. The effect of different cache sizes, as well as, the memory block length on system performance and different system parameters has also been evaluated, analyzed and presented. The simulation results show MCOMA has achieved a good overall speedup on all the used applications. All the applications performance improved by the use of multithreading on the MCOMA which proved that combining the benefits of dynamic data migration in COMA with multithreading improved the overall system performance

Table of Contents

List of Figures

List Of Tables

Chapter 1

Introduction

Shared memory multiprocessor architectures provide a flexible and powerful computing architectures. In these architectures, a large number of processors communicate and synchronize the execution of applications through shared memories. In general, the architecture consists of a number of high performances processing nodes that share a single global address space. These nodes are connected through an interconnection network. This allows all processors to access and modify shared memory locations, which simplifies intertask communication and eases the system programmability. In current technology, the performance of these systems is limited by the large gap between processors and the memory access speed. The interconnection network bandwidth is another performance limiting factor.

The simplest form of shared memory architecture is the uniform memory architecture (UMA). These architectures mainly use a single bus to connect a number of processing nodes and a global memory. All processing nodes have a uniform access time to all memory locations. This architecture provides an easy programming model to the user. However, since the bus has a limited bandwidth, the system can only support a small number of processing nodes. Examples of this type of architecture include Encore Multimax and Sequent Symmetry.

Larger multiprocessor systems are built using a distributed memory organization. Sets of processing nodes are connected through a scaleable interconnection network. Each processing node has part of the global memory that can be accessed locally. In general, the access time to different parts of the memory in distributed memory systems is non-uniform depending on the location of the data. Therefore, data placement and movement on the interconnection network can greatly affect the system performance.

Distributed memory systems can basically be divided into two main dominating architectures: the non-uniform memory architecture (NUMA) and the cache only memory architecture (COMA). In NUMA, the system consists of a number of processors; each processor has a portion of the shared memory. These processors are connected together through an interconnection network. Each part of the distributed main memory is associated with a fixed physical address, which is known as the home node of the data. Examples of this architecture are the DASH and the Alewife [32,3].

The cache only memory architecture (COMA) such as the KSR1 and the Data Diffusion Machine (DDM) [8,17] has a similar architecture to NUMA. However, in COMA the data location is detached from its physical address. The portion of the shared memory associated with each node acts as a large cache, which is called the attraction memory [33]. The attraction memory of each processor attracts all the data that are used by that processor, and it may contain some data from the shared address space. These data movement are supported by hardware. This system tries to relieve the burden of data partitioning by allowing the data to migrate and replicate in attraction memories depending on the data references.

1.1 Memory Latency

Remote memory access latency occurs when a processor accesses a data item in the shared memory space that is located on different processing nodes. The processor has to send data requests over the interconnection network to the memory module that has the requested data. This memory module sends the requested data back to the requesting node. The time interval between the sending of the request for the date until receiving the data is called the memory access latency. This remote memory access latency negatively affects the processor utilization since the processor would sit idle until the arrival of the requested data. For large distributed shared memory machines, this memory access latency can grow

to be hundreds of cycles, which increase the processor stall time. Different approaches have been proposed to reduce, avoid and tolerate large communication latencies in shared memory multiprocessors [11,13,17,20]. The two major approaches that are related to our work are data caching and multithreading.

1.1.1 Data Caching

Cache memories are used to reduce both the average memory access time and the access contention from different processors. It exploits the temporal locality of the application. Local processor cache maintains a copy of different memory locations accessed by the processor at the processor speed. Data is moved to the local cache of a processor where it can be reused and the processor does not need to access this data through a remote request again. The data stays in the local cache until invalidated or replaced. This reduces the need to traverse the network for remote data accesses. Coping shared data to different caches introduces the cache coherence problem. All copies of the same memory location must be consistent. The problem arises when a processor shares a copy of a shared data block with other processors caches. If this processor update its copy to a new value, it has to update or invalidate all the other copies of the block in the shared memory as well as in the other processors caches. A large number of schemes have been proposed to solve this problem. These schemes have wide variations of hardware and software supports.

A large number of hardware based cache coherence protocols choose to invalidate rather than update shared data items. These protocols require that invalidation be sent to all copies of data over the interconnection network. In bus based systems, these coherence messages are broadcasted over the bus. All processing nodes snoop on the bus to monitor any changes done to a local cache copy. A shared copy of a data item is invalidated when a write request for this item is observed on the bus. As the

number of processors in the system increases, contention on the bus would result in longer latencies. These protocols are called snoopy cache protocols.

In distributed-shared memory that uses general interconnection networks, consistency commands can no longer be broadcasted to all processing nodes. Directories are used to track different copies of a cache block. A directory keeps a record of all information about caches that has copies of the cache block. Consistency commands are sent only to those nodes that has copies of the data block rather than broadcasting them to all processing nodes. In a write invalidate protocol, when a processor issues a write request to a shared cache block, the request is sent to the directory of the portion of the shred memory that has the data block. The directory sends invalidation to all nodes that has a copy of the block and an exclusive copy is sent to the requester. Any request to the exclusive copy by any other node is forwarded by the directory to the cache that has in exclusive state. Different cache coherence models were developed to improve system performance [10, 44]. Relaxed memory consistency models reduces the latency of remote data requests by buffering writes and allowing the reads to bypass them.

In some existing NUMA architectures, each processor has an associated cache and a portion of the shared memory These architectures, such as DASH and Alewife [32.3], uses directory-based write-invalidate cache coherence protocols to maintain cache coherency. To handle a cache miss on NUMA architecture, the home node of a block has a directory where the status of its blocks is stored. Data is transferred to the local cache of the requesting processor and the home directory is adjusted accordingly. The way of handling a cache miss depends on the cache protocol used by the architecture.

The cache only memory architecture (COMA) the portion of the shared memory associated with each node acts as a large cache. Data is moved between attraction memories as cache lines and data coherence must be maintained between different copies of data in the attraction memories. Since the

data has to be coherent on both the attraction memories and the processor caches, this result in a coherence protocol that is more complicated than that used for NUMA architectures.

The main advantage of COMA is the dynamic data migration and replication in attraction memories at a fine granularity (cache line) which is supported by hardware. The user does not have to explicitly distribute and move data among physical memories and initial data allocation does not have the same impact on the system performance as in NUMA. As in any cache memory system, COMA exploits locality, which improves its performance. In COMA, as data is requested by a processor, it is copied to both the local processor cache as well as the local attraction memory. The size of the attraction memory is usually large enough to hold the working set of a processor. Therefore, if the data were replaced in the local cache because of the limited cache size, it would be less likely replaced in the attraction memory. This reduces the number of remote memory requests, which in turn reduces the average cache miss latency and moves data to the processor's local memory.

The performance of the system can also be improved by increasing the size of the attraction memory, which is usually a cheaper alternative than increasing the size of the cache, as in NUMA. In addition, the size of the working set does not have to be small to fit into the local cache. Also, any possible integration of the processor and the data memory will further improve the COMA performance.

However, COMA architectures have their disadvantages. First, they require a mechanism to locate the data on a miss. Most COMAs use a hierarchical directory structure and/or snooping, which causes large communication latencies. Second, it needs more complicated coherence protocols to ensure that at least one copy of a data item remains in the system and not replaced. It also needs to ensure that the data is coherent on the processor cache as well as the attraction memory. Third, a larger attraction memory is required to allow for data replication, which contributes to the larger percentage of the memory overhead required by COMA [24].

1.1.2 Multithreading

Multithreading is another technique that can be used to tolerate memory access latencies. It masks these latencies and reduces the cost of synchronization by utilizing the processor and its associated resources, while waiting on an event. A processor executes a program that consists of one or more instruction stream, which is, called a thread. Instead of allowing resources to be idle during a memory latency operation of a thread, these resources are shared by switching the processor to work on another ready thread, thus reducing the idle time of the processor. The processor usually holds multiple contexts, one per active thread. One context is active at a time. When a thread waits on a long latency operation, the processor context switches to the another ready thread. The context switching time, which is the lost processor cycles to switch between threads, should be minimized to improve the processor utilization.

Multithreaded architectures can be either fine grain or coarse grain depending on the context switching used. Typically, fine grain architecture switches threads every processor cycle. This allows the long latency operation to be completed before the rescheduling of the thread. This model requires a large number of threads to hide the latencies. It also requires a large amount of hardware support for these threads. The single thread performance is affected since each thread has a limited fraction of the processor time. In coarse grain multithreading, or block multithreading, a thread execute for many cycles before it is context switched when a multi-cycle operation is issued such as cache miss, memory miss or a synchronization failure. These architectures usually tolerate larger context switching penalty and they require less parallelism in the application. It requires a fewer threads to hide remote access latencies which reduces the amount of hardware required for multithreading.

Multithreading increases the processor utilization, however, the major concern of multithreading is that a large number of threads are needed to hide a long latency operation. This requires a considerable

amount of parallelism in the application. A scheduling mechanism is required to schedule different ready threads. A number of processor cycles would be lost while the processor switches between different threads. As the number of threads used per processor increases, the number of outstanding requests per processor in the system increases. This would increase the bandwidth requirement of the system [46]. For the systems that use caches, multithreading increases the cache conflict miss rate due to the sharing of the cache between different threads. As the cache miss rate increases, the number of remote data accesses increase and the number of threads needed to hide these long latencies also increase. This effect would limit the benefit of multithreading on the system overall performance.

1.2 Research Contribution

This thesis makes three important contributions. First, a new non-hierarchical cache only memory architecture, MCOMA, has been developed. Second, to improve the performance, the new architecture was modified and combined with multithreading. This architecture benefits from the data locality of COMA and the latency tolerance of multithreading. Third, a simulator has been developed to evaluate and study the performance of the new system. This simulator is an execution driven simulator, which is developed on top of MINT [48], a MIPS interpreter. A subset of SPLASH, a suite of parallel application programs, is used which covers different problem domains.

The existing COMAs either use a hierarchy of rings, such as KSR1 [8], a hierarchy of buses, such as the DDM [17] or assign a home node for the data as in COMA-F [23]. Hierarchical interconnections are used in COMA to trace copies of data blocks and to combine requests for the same data blocks. These hierarchies cause large communication latencies, which is the major disadvantage of the COMA. The size of the higher level directories in COMA grows larger since each directory contains the information of all the directories underneath it. This increases the directory search time, which results in longer latencies. The higher level of the hierarchy also tends to create a bottleneck since requests from lower

levels usually pass through the higher levels. There are different design variations of DDM under current research, all of which uses hierarchical network [35].

The new MCOMA architecture is different from all of the above mentioned architectures. On one hand, it overcomes most of the disadvantages of these existing architectures. On the other hand, it has hardware support for multithreading which none of these architectures offers. The new COMA is a non-hierarchical COMA architecture that has a search interconnection network, which is only used to search for and locate data. It also has an interconnection network that connects the processing nodes and the directory nodes. This interconnection network is used for data transfer. These two interconnections are non-hierarchical, and therefor, they have less communication latencies compared to hierarchical COMA architectures.

The new architecture also supports block multithreading to hide the remote data access latency. The use of caches helps avoiding remote memory latency. However, in a multithreading architecture, sharing the cache between different threads can contribute to a higher cache miss rate. In COMA, moving data to local memories would reduce the frequency of thread switching. This in turn means longer thread run length and a fewer number of threads are needed to hide remote access latency [46].

Since COMA has a large attraction memory, then switching between different threads in multithreading will not have the same negative effect as that on small local caches. The data used by an active thread will be migrated to the local attraction memory as well as the local cache. When this thread is switched on a long latency operation, if part of its data in the cache is replaced by a new active thread, this replaced part of data will probably still be in the local attraction memory. Therefore, when this switched thread resumes execution, it would have most of its data available locally.

In block multithreading, a thread executes until a long latency operation causes context switching. The use of block multithreading in COMA has its advantages. First, it does not greatly affect the single thread performance. The active thread can benefit from memory localities as long as it is not switched out. Second, since the attraction memory works as a large cache in COMA and data is migrated to the local memory as well as the local cache, the performance degradation due to sharing the same cache between different threads is reduced. This also reduces the number of remote memory access requests, compared to NUMA, which in turn reduces the need for frequent context switching and fewer active threads are needed to hide remote access latencies.

In this research, the impact of using multithreading on different architectural features such as private caches and the attraction memory is investigated. Such investigation is a good contribution towards a better understanding of the potential of multithreading. The results of our research have proven that combining the benefits of dynamic data migration in non-hierarchical COMA with multithreading to hide long latencies in COMA improves the overall system performance.

1.3 Thesis Overview

In Chapter 2, multithreading and COMA, architectures and the effect of combining both of them are discussed. It also presents some existing COMA and multithreading architectures. Chapter 3 introduces the new MCOMA architecture that combines both multithreading and COMA. The coherence protocol for the system is also presented in this chapter. Chapter 4 presents a qualitative comparison between the MCOMA and FCOMA architectures and discusses the expected performance of MCOMA. Chapter 5 describes the simulator developed to evaluate the new architecture performance and its parameters. It also presents the characteristics of the application programs used. Chapter 6 presents the performance evaluation of MCOMA and the effect of multithreading on the system performance. It also presents the

effect of different cache variations on the system performance. Finally, Chapter 7 concludes this thesis and discusses directions for future work.

Chapter 2

Multithreading and COMA Architectures

2.1 Multithreading

Multithreading is a technique to tolerate the long latencies inherent in large scale shared memory multiprocessors. These systems suffer from low processor utilization. Typically, a processor execute instructions until it encounters a long latency operation that may require many processor cycles depending on the type of the operation and the access time of the interconnection network. Multithreading can tolerate these long latency operations by allowing the processor to maintain multiple threads of execution. As one thread waits on a long latency operation, the processor is switched to execute another ready thread. Multithreading is efficient as long as there is enough parallelism in the application and the context-switching overhead between threads is kept to the minimum.

2.1.1 Multithreading effect on system performance

Multithreading improves the processor utilization as well as the interconnection network by hiding the remote access latencies. However, some factors limit this improvement. In a multiprocessor system that supports caching, the cache is shared between multiple active threads. This causes a higher cache miss rate due to the data conflict between threads using the same cache. This, in turn, increases the network traffic as well as the number of context switches. Since each processor would have multiple outstanding requests for multiple active threads, this increases the demand on the network bandwidth. Another

factor limiting the performance of multithreading is the cost of thread switching. Some processor cycles may be lost during the context switch, which should be smaller than the long latency operation that caused the switch.

Multithreading also requires applications that have sufficient parallelism to allow multiple active threads for each processor. Since some applications do not completely satisfy this requirement, high single thread performance should be maintained in multithreaded architectures. The block multithreading model allows for better processor utilization while keeping high single thread performance. To exploit the benefits of multithreading, the architecture design must address some issues such as the cache interference due to sharing the cache among threads, the context switching overhead, and the network bandwidth and network access overhead [4].

2.1.2 Multithreading Related Research

Multithreaded architectures are classified as fine grain or coarse grain, depending on the context switching policy. Fine grain multithreading has a small context interval, typically every cycle. This eliminates data dependencies between instructions in the processor pipeline by interleaving different threads. This model requires a large number of threads and a large amount of hardware to support them. A single thread is limited to a small fraction of the processing power. A coarse grain multithreading model executes a thread for many cycles and context switches only on long latency instructions such as memory access that does not hit in cache. It requires less degree of parallelism to tolerate latencies and can tolerate a longer context switch penalty [46]. This model does not greatly alter the architecture performance of the single thread model. It improves the processor throughput without relaying on the compiler to extract additional parallelism from the existing programs.

There are several projects researching multithreaded architectures. The TERA machine is an example of a fine-grain multithreaded multiprocessor [1,2]. It implements a shared memory programming model with up to 256 processors, each designed for heavy interleaving of memory operations. It has 128 contexts and switches every cycle with zero cycle penalties, ensuring that all instructions in the pipeline are from different threads. It has 16 protection domains with independent set of registers holding stream resource limits and accounting information, i.e. each processor can be executing 16 different applications in parallel. Tera's memory is distributed and shared. It has no cache and its memory system does not implement any data migration technique.

An example of a coarse-grain multithreaded architecture is the MIT Alewife machine [3,12]. It is also a distributed, shared memory machine. It uses a directory scheme to maintain coherence between processors caches. It switches context only to avoid processor idling when a long latency instruction is issued. Instructions from one thread keep executing until it encounters a long latency access operation. Stream scheduling is done in software. Its context switching time is longer than TERA. Its processor implements four hardware contexts with support for only one executing application at a time. It is a NUMA architecture where threads can lose some of its data while waiting on long latency event.

*T is another proposed multithreaded machine which has a separate synchronization processor and a separate remote memory processor in addition to the main computing processor [37]. These processors are extended RISC with instructions for rapid creation and consumption of network messages and for efficient synchronization and scheduling of lightweight threads. The synchronization unit is user-programmable. *T does not have effective support of cache coherence and does not discuss its implication on the design.

EARTH-MANNA (Efficient Architecture for Running THreads) is another multithreaded architecture that executes the synchronization operations and the computation operations using different functional

units [20]. The EARTH programming model is implemented on the MANNA multiprocessor. A ready queue and an event queue interface the execution unit (EU) and the synchronization unit (SU), which are implemented in local memory. The EU executes a thread to completion before moving to another thread. SU manages the synchronization and communication operations as well as thread scheduling. EARTH model does not support caching of globally shared data in the nodes and must rely on a software layer to maintain coherence of shared data.

2.2 COMA Architectures

Cache only memory architectures have the ability to hide remote memory access latency by reducing the number of these accesses. This is achieved by migrating or replicating the data locally into the local attraction memory such that the initial data placement has minimal effect on the execution time of the application. However, longer latency is introduced due to the more complicated data search and data replacement protocols than those of NUMA.

COMA architectures support automatic data migration in hardware. The attraction memory is organized as a large cache that attracts the data referenced by the processor. Data is not tied to a special physical location as in NUMA. Attraction memory is kept coherent using a directory-based coherency protocol. The directory contains the state of the coherence unit as well as an identifying tag for the data. The major difference between the NUMA and COMA coherence protocol is the way of handling data replacement in the attraction memory. In COMA, there is no specific location to write back the data upon replacement. To replace a data item, it has to make sure that it is not the last item in the system. If it is, it has to be moved to a different node. Coherence protocols can be implemented in either hardware or software. In the following discussion, only hardware COMA architectures are considered. None of the existing COMA architectures supports multithreading.

2.2.1 Existing COMA Architectures

One of the existing COMA architectures is the Data Diffusion Machine (DDM) [17]. Data migration and replication in attraction memories is supported in hardware at the granularity of a cache. The architecture is organized as a hierarchy of snoopy buses. The processing nodes are connected to the lower level of this hierarchy. A hierarchy of directories connected by buses is used to maintain memory coherence. Directories store the state information about all the data in the hierarchy below it. A data request traverses up the hierarchy until one of the directories that is snooping on the bus responds to the request and direct it to the node that has the requested item. This architecture suffers from contention at the root directory as well as the long latency for data search through the hierarchy.

Another COMA architecture is KSR1 [8.34], which uses a hierarchy of rings instead of buses as in DDM. The architecture can have up to two levels of rings. These rings have a higher bandwidth than that of the buses. It allows for simultaneous communication between the processing nodes on the same ring but it increases the remote memory access latency. The data request has to be sent around part of the ring then the data reply has to be sent around the rest of the ring to reach the requesting node. If the requested data were on another cluster, the remote memory latency would greatly increase. For the request to be satisfied, it has to travel across two lower level rings as well as the top level ring. Another disadvantage of this architecture is that the data allocation unit in the attraction memory is a 16K-byte page and the cache line is 128 bytes. This large allocation size results in inefficient use of memory. It also introduces false sharing as well as memory fragmentation.

FCOMA is a hybrid of the NUMA and COMA architecture [23]. It combines the notion of a home node location for directory information, as in NUMA, with the feature of replicating data at the main memory level, as in COMA. This reduces the effect of initial data placement but does not eliminate it. The home node still needs to be referenced and updated every time the data status is changed in any node's

memory regardless of the current location of the data. This would overload the home node since it has to manage data that doesn't necessarily reside in its attraction memory in addition to managing its own active data. It may also introduce more traffic on the network since a closer node, that has a copy of this data item, may have fulfilled the data request. The advantage of such an architecture is that data movement in it is more flexible than in the regular NUMA architecture, but it is not as flexible as in COMA. It also eliminates the need to search and locate data items over the network.

Another alternative architecture is the Simple COMA (S-COMA) [41]. It uses a software layer to manage the cache space allocation at the page granularity. The shared memory coherence is then maintained by hardware at the cache line granularity. Since the space of the attraction memory is managed at the granularity of a page, this architecture suffers from memory fragmentation. This is the main reason for the reported performance deterioration of S-COMA over NUMA and other traditional COMAs once the problem size increases. The long latency of page allocation using the operating system is another disadvantage of this system.

2.3 Combining Multithreading and COMA

The major concern in designing a new COMA architecture to improve the processor utilization by hiding the latency associated with local data misses that require remote data searches. Multithreading is a latency tolerance technique that improves processor utilization. A thread is executed until it encounters a long latency operation. The thread is then switched out and another thread starts execution.

Multithreading improves processor utilization by overlapping communication and computation. Caching of data also improves multiprocessor performance by eliminating large portions of remote memory access latency. However, the direct combination of both techniques, multithreaded and COMA architectures, causes them to interfere with each other. In a multithreaded architecture, sharing the cache

between different threads can contribute to a higher cache miss rate. An active thread may replace the data moved to the cache by previous threads. By the time a switched thread is ready to resume execution, most of its data in the cache has been replaced. This causes even more context switching and remote memory access requests.

COMA architecture mitigates this problem of multithreading. COMA has a large attraction memory that works as a large cache. Sharing it between different threads will have less negative effect compared to that of small caches. The data used by an active thread will be migrated to the local attraction memory as well as the local cache. When this thread is switched on a long latency operation and a new active thread replaces part of its data in the cache, this replaced data probably will still be available in the local attraction memory. Therefore, when this switched thread resumes execution, it will have most of its data available locally. This will reduce the number of remote memory accesses in multithreading.

Since COMA has a high local hit rate, less remote data access is needed. By increasing the number of local memory accesses, the frequency of context switching is minimized. This reduces the number of processor cycles wasted on context switching and improves the processor utilization in a multithreaded architecture. Since most of the thread is working set will be in the attraction memory where data can be easily cached, execution time of the thread is expected to be longer. This, in turn, reduces the number of active threads required to hide the remote data access latencies and the hardware needed to support these threads.

Another way to improve the performance of combining COMA with multithreading is by dividing the processor cache into smaller private thread caches which can be organized as either fully or set associative. Each thread cache is used by its thread and is preserved when the thread is switched on a long latency operation. When a thread resumes execution, it would have most of its data available in its cache as it was before thread switching. The only lost data will be due to the data conflict in the

attraction memory. The use of a smaller cache per thread would not degrade the thread performance since all data replaced in the cache will probably still be in the local attraction memory where it can be accessed locally without the need for more thread switching.

Chapter 3

MCOMA: A Multithreaded COMA

In this chapter, the main features of the new COMA architecture are presented. A general architecture of the system is first described, then one design variation is considered for this research. The combining of COMA and multithreading in the system design is presented. A description of the different features of the system is discussed. Finally, the coherence protocol is presented.

3.1 MCOMA Architecture

The MCOMA uses a scaleable non-hierarchical interconnection network to connect all processing nodes and supports coarse grain multithreading (block multithreading). Each group of these processing nodes duplicate its directories in one group directory node which is also connected to the same interconnection network. All group directories are connected together by a dedicated search interconnection network for fast data search. Only short data requests and coherence messages are sent on this interconnect, while all data movement is done on the processors interconnection network. This system reduces the long latencies caused by hierarchical search and reduces the amount of directory storage since multiple copies of directories are not needed. It also eliminates the need to assign the data to a home node as in NUMA and FCOMA. Since the directory size is smaller, the directory search time is less. There are no high-level directories to cause bottlenecks; instead, data information is distributed over the group directories. The overhead of duplicating the directory once is similar to the overhead of having to keep a directory in the home node of the data, as in FCOMA. However, the processing node only manages the data that resides in its attraction memory, and the entire data search is done in the group directory nodes.

A group directory node manages the directory of the data in its group, locates the requested data and directs each request to the proper node to answer it. This relieves the processing nodes from managing data that does not exist in its local memory. It also reduces the contention on the attraction memory, which improves its access time. It also reduces the effect of remote memory requests on the local node resources. The general MCOMA architecture is shown in Figure 3.1.

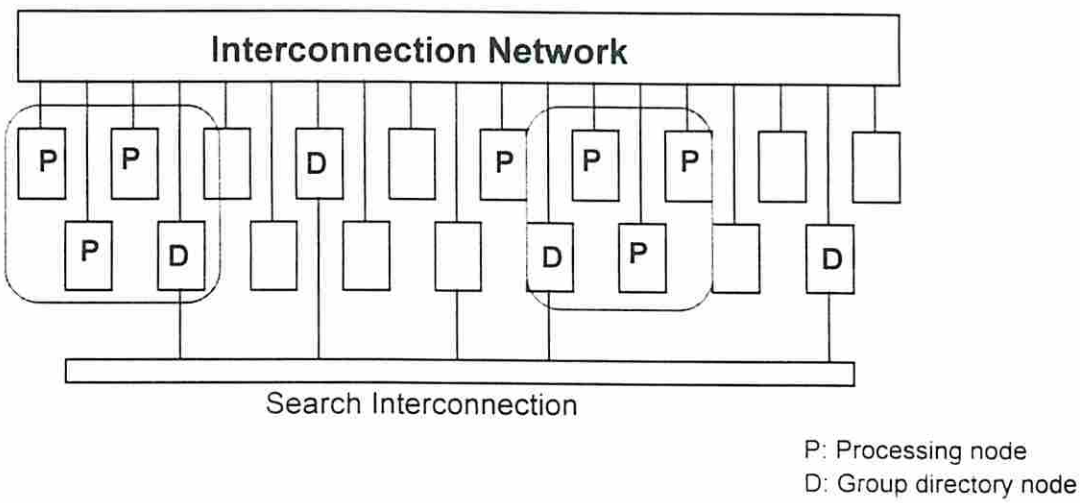


Figure 3.1 General Architecture of MCOMA

3.1.1 MCOMA on a 2-D Mesh Interconnection

In this section, the design of one possible implementation for MCOMA is discussed. This design chooses a two-dimensional mesh interconnection to connect the processing nodes and the directory nodes. All directory nodes are connected by a dedicated interconnect, in this case a bus, to allow for fast data search between directories. Each processing node consists of a processor (or a cluster of processors), attraction memory and its directory and a communication and memory management

processor. Each directory node contains all the information about data items in all the nodes of its group. Figure 3.2 shows the block diagram of the system. In this case, row nodes are grouped on the horizontal axis and the sum of its directories is stored in the group directory node that is connected to its row. Data requests are directed to the directory node to locate the data item. If the data does not exist on the group, the request is sent on the search bus to other directory nodes to satisfy it.

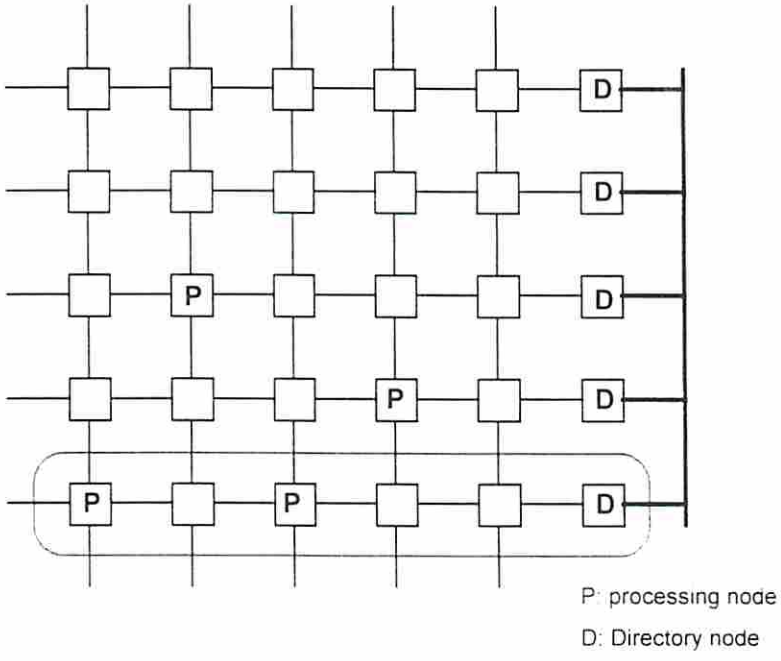


Figure 3.2 MCOMA Architecture

3.1.2 MCOMA Node Architecture

The MCOMA machine consists of a number of processing nodes and directory nodes connected by a mesh interconnection network. All processing nodes are identical. Each node consists of one high performance RISC processor that supports multithreading, a private cache, an attraction memory, a memory and communication management processor (MCMP) and a node controller. The node controller interfaces the node to the interconnection network. The MCMP is a separate coprocessor that

is used to handle the remote memory requests, manage the attraction memory and handle the incoming coherence and synchronization messages. It also manages different thread through the thread scheduler. It can also access the local memory directly. The basic architecture of the MCOMA node is shown in Figure 3.3.

The group directory nodes are also identical to each other. They are dedicated to data search. Each group directory node is connected to a row of the mesh interconnect. It is also connected to other directory nodes through a dedicated bus for faster communication between them. Each group directory node contains a directory for all data items that reside in the attraction memories connected to its own row as shown in Figure 3.4. It also has a directory controller that is interfaced to both the mesh and the bus interconnects. Each node snoops on the bus.

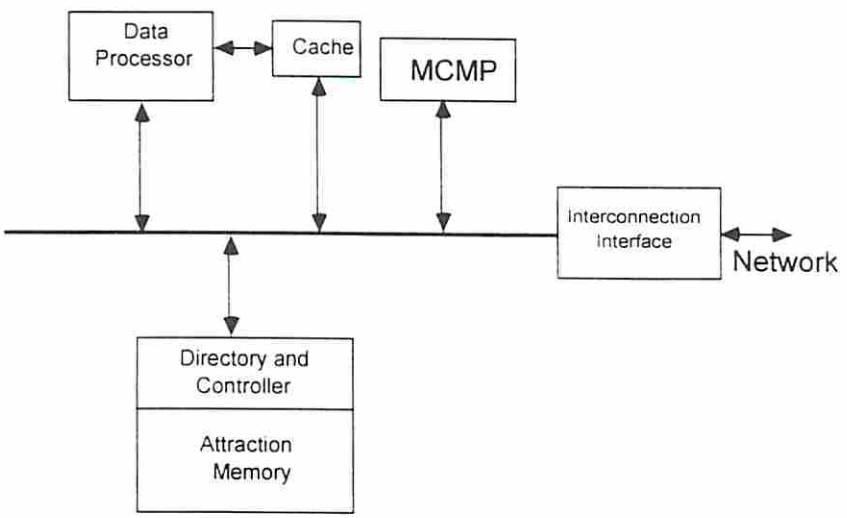


Figure 3.3 Processing Node Architecture

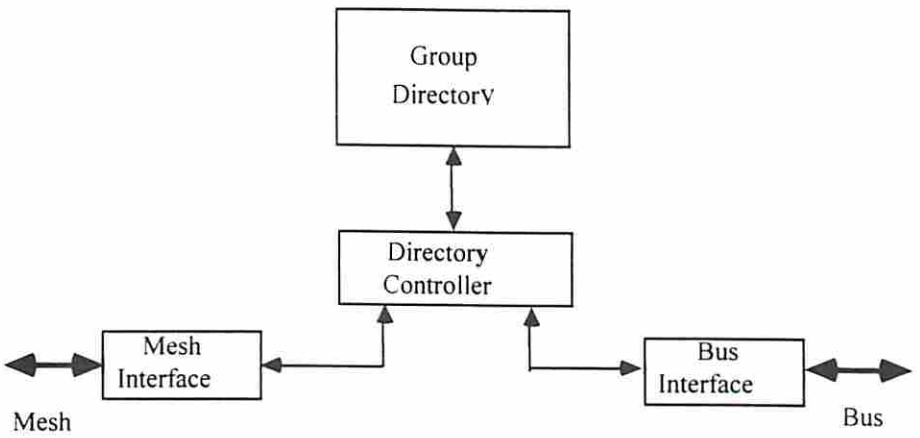


Figure 3.4 Directory Node Architecture

3.1.3 Thread scheduler

The main task of the scheduler is to determine which ready thread is to be executed. A thread is ready to continue execution when a long latency memory load or store, or synchronization event is satisfied. If multiple outstanding requests per thread are allowed, all these requests should be satisfied before continuing execution. A counter of these requests must be included to keep track of all outstanding requests. The scheduling policy should take into consideration any priority status of a thread. If more than one ready thread has the same priority, a simple first-in-first-out policy is used. A time out mechanism where a thread is forced to switch after holding the processor for a certain number of cycles can be used to prevent high priority threads from waiting long periods. The thread is switched out to the active ready queue and may start execution according to its priority. To minimize contention for critical operations, a thread should hold the lock for the shortest possible time. This thread should be given a higher priority such that it can execute and exit the critical region and release locks quickly. When a thread is switched out on a synchronization event, it is marked as a high priority thread and scheduled accordingly.

Threads can be in one of the various existing queues shown in Figure 3.5. At the start, all ready threads are in the ready new thread queue. There are two sets of ready queues, the Active Ready Queue set and the Non-Active Ready Queue set. Each set consists of more than one queue; each has a different priority. A maximum of four threads can be active at any time. Each of these active threads is assigned one of the thread caches and a register set. To make use of the locality formed for each active thread, the thread will stay active until it is completely executed or a point is reached when all active threads get into a waiting state. At that time, the last running active thread is transferred to the Non Active Waiting Buffer (ATWB) and the highest priority non active thread is transferred to the Active Ready Queue set out of the Non-Active Ready Queue set. This thread uses the thread cache and the register set of the last running read. This design choice was done assuming that the other active waiting threads have a great part of their outstanding long latency operation in progress and they are ready to run soon.

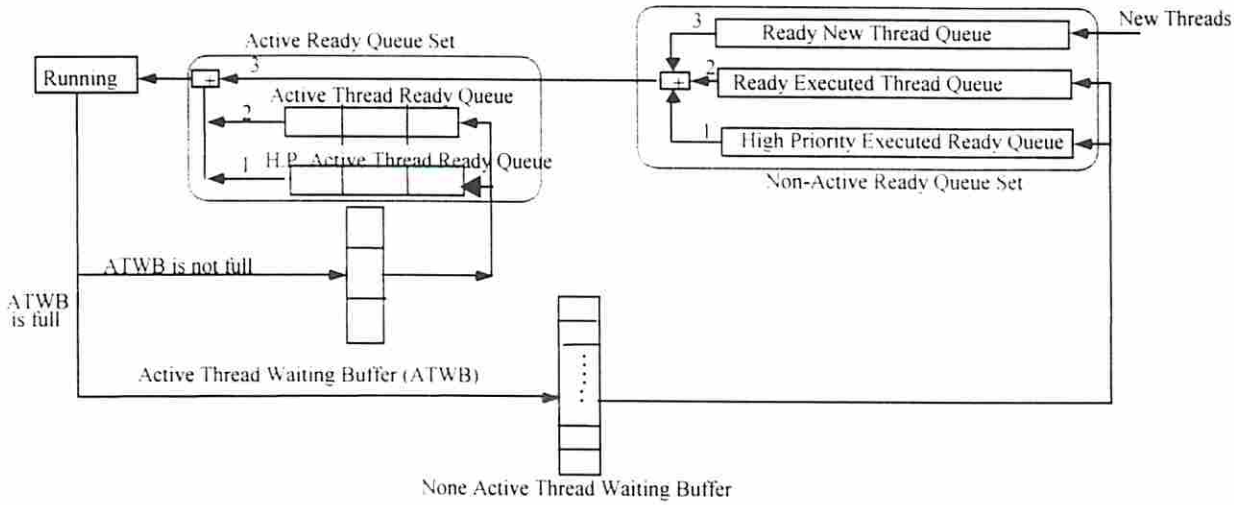


Figure 3.5 Scheduling Queues

3.1.4 Memory and Communication Management Processor

The memory and communication management processor is similar to the synchronization unit used in *T and EARTH-MANNA [37,20]. It is mainly for functionally separating the execution of threads from the synchronization and communication between different processing nodes as well as threads. It handles split phase transactions and the incoming coherence messages. It also manages thread scheduling and synchronization. It can receive the messages and load its value directly to its destination in the local attraction memory or the processor cache. When all messages for a certain thread are received, it queues the thread into the non-active ready queue. Since the network does not preserve the message order, a tag is attached to each message. This tag identifies the requesting thread number and the location to put the returned data.

3.1.5 Data Processing Node Architecture

The data processing node is the main processor. It executes threads from a ready queue. This processor can be regarded as a conventional RISC processor with a few modifications that supports multithreading. Since the processor is shared between different threads, all the processor resources are also shared including the processor cache. As was discussed earlier, sharing the cache between different threads reduces the cache hit rate. However, dividing the cache into separate parts, where each thread uses its own part, can solve this problem. When the thread is switched on a remote memory access, its associated part of the cache is preserved and the following thread uses a different part of the cache. This avoids losing the thread's data during the time when other threads are executing. The use of a smaller cache by a thread would not degrade the thread performance since all data replaced in the cache due to capacity will probably still be in the local attraction memory. In this case, there would not be a need to

fetch this data from another remote location across the network. This reduces the degradation of the cache performance because of sharing the cache between several threads.

Another concern of the multithreading is the overhead of a context switch. To reduce this cost, the register file can be modified to improve the switch time. It can be divided into separate register sets; each set is used by one of the active threads. This eliminates the need for saving the register file of the current thread and the loading of a new set of register file for the next thread. Other research suggests that hardware support for four to eight threads is sufficient to hide remote memory access latencies and improve processor efficiency [4,33,38]. COMA has long remote access latency because data items need to be searched on different directories. COMA also has high data hit rates, which improves thread run length, and fewer of threads are needed to hide the remote access latency. In this architecture, the processor hardware will support up to four active threads. If, at any time, all active threads are blocked, the highest priority thread from a non-active ready queue set, will replace one of these active threads. Threads from this non-active queue will continue executing until one of the active threads is unblocked.

The processor used is based on a modified RISC architecture. Using a modified version of an existing processor simplifies the design process and makes it easier to use the available technology. It also permits the use of available software and allows for faster development time for the new architecture. Coarse grain multithreading relies on the efficient execution of sequential code, which is, supported by the available RISC processors.

The processor has a modified RISC instruction set. Extra instructions may be added to support both COMA and multithreading. The processor supports four threads in hardware. The processor has four register sets and its local cache is divided into four sections. Each active thread uses a set of registers and a section of the processor cache. Each register set includes a program counter, instruction register

and a thread status registers. There is a frame pointer associated with the running thread that points to the active register set and a cache section.

Some instructions are added to the RISC instruction set to support multithreading. The instruction *next* terminates the current thread and loads a new thread from the current waiting queue [7]. If a thread is not available, the processor waits until one is available. Some instructions may also be used to manipulate the current frame pointer such as increment, decrement, and read or write the frame pointer. Some of the modification of the RISC processor is shown in the simplified diagram of Figure 3.6.

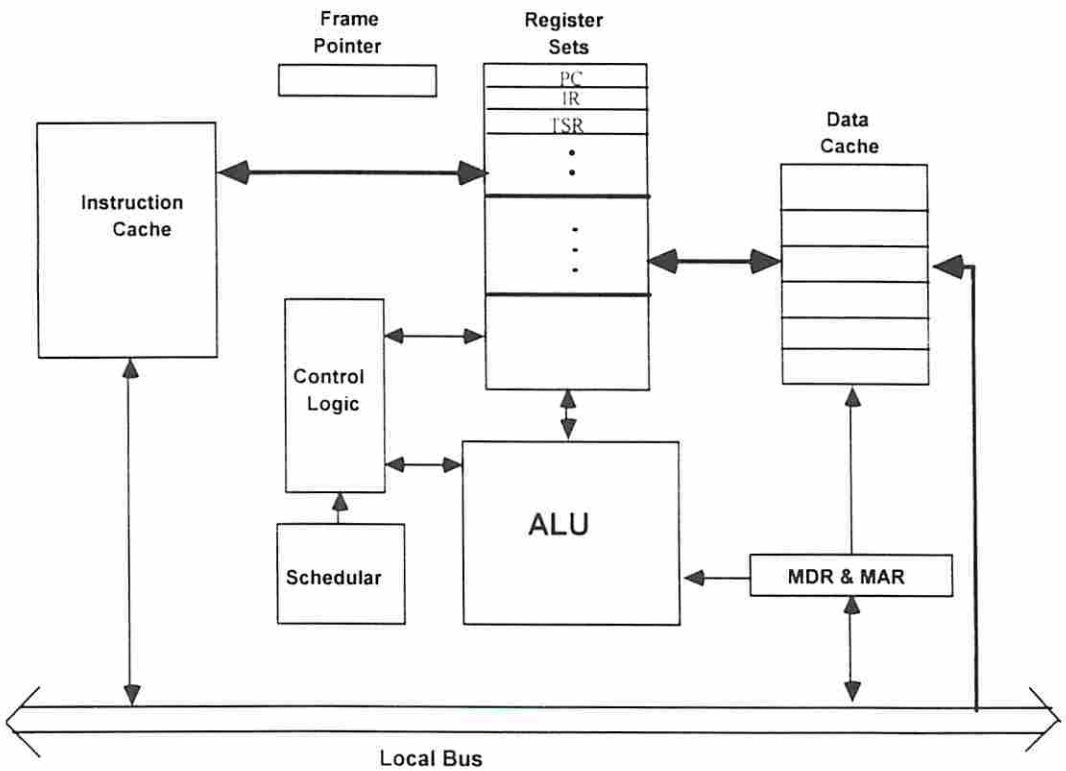


Figure 3.6 Data Path for a modified RISC Processor

3.1.6 Group Directory Node

A group directory node holds a directory for all the data items in the attraction memories of all nodes of its group. It only holds the state information of the data item, not the item itself. It is interfaced to the mesh interconnec as well as the common bus, which connects all group directories. It also has a directory controller that manages the directory and keeps track of the memory consistency transactions. Each node snoops on the search bus. If the request on the bus can be satisfied by the node, the node sends an answer to the requesting directory. It then forwards the request to the appropriate node in the group. This node, in turn, sends the requested data to the requester. To reduce network traffic, the group directory controller can combine all read requests for the same data item. If an item is requested by more than one node in the group, there is no need to send more than one request for the same data item. In this case, all requests are combined together and the reply is sent to all requesting nodes. The directory node can also send multiple invalidations to its group nodes.

The directory is organized as a set associative memory. It is a full map directory where each node of the group is represented by one present bit in the status word of each data item. If the item is in Exclusive State, the owner bit and one presence bit in the status word are set. Other status bits represent different states of the item in different node group. To improve the system performance, this directory can be implemented using a fast lookup memory.

3.2 The Coherence Protocol

The MCOMA coherence protocol, as any other COMA coherence protocol, must maintain coherence between multiple copies of the same data item and must ensure that the last copy of the data item is maintained in the system. To insure coherence between multiple copies, a directory based write

invalidate combined with a snoopy protocol is used. Each processing node has a directory that stores the state information for the data blocks that reside in its local attraction memory. Directories reside in a reserved area of the attraction memory that can not be cached or replaced. The coherence unit is a cache line. The data item is kept coherent with its other copies, and all the copies in the system reflect the last modification to the item.

When the cache and the attraction memory need to replace an existing data item to allocate space for a newly requested item, a victim item is selected for replacement. In COMA, replacing a data item in the attraction memory may cause the data to be lost if it is the last copy of the data in the system. To ensure that the last copy of any item is never discarded out of the system, one copy of each item is marked as the *owner*. The *owner* copy is responsible for responding to requests for copies and for finding a new owner on replacement [8.45]. If this copy needed to be replaced at any time, a special replacement message that carries the data to a new location is issued. This copy keeps its ownership until it is transferred to a new copy of the data.

The coherence protocol keeps all the copies of the data item coherent by sending invalidation messages to all other copies whenever a copy is modified. The state of all other data items are kept in the local directories. For each group of nodes, the group directory node contains all the information about all the data items and its state that exists on the group processing nodes. Any change in the data item state will be reflected on the group directory node. The group directory node manages data requests and invalidation and sends the appropriate message to the corresponding node. It uses a snoopy protocol to search and locate the group that has the requested data item.

The coherence protocol tries to minimize the network traffic by keeping the data movement within the group as much as possible. The data request is sent to other groups only when it can not be satisfied locally from the group nodes. The group directory also combines all requests to minimize the network

traffic. One requesting node will receive the data then distribute it to the other requesting group nodes according to messages sent to it by the group directory.

In COMA, the processor's cache always contains a subset of what the attraction memory contains because of the inclusion property between the processor's cache and the attraction memory. If the data item in the attraction memory is also in the local cache, it is marked cached in the attraction memory. Since the processor cache is divided into four separate sections, one per thread, a presence bit is used for each of these sections. The data item in the attraction memory can be in one of the following states:

- *Invalid*: Data is not valid.
- *Shared*: There is at least one other copy of this data item. This data is readable.
- *Owner Shared* This node is the owner of the data item. The data item is readable and there might be another valid copy of the item in the system.
- *Owner Exclusive* This is the only valid instance in the system. It is readable and writable. It can be cached to local processor caches.
- *Pending* A read/write request is being sent for the item. A space is allocated for the item. The data is not valid yet.
- *Locked* Similar to Owner Exclusive. This is the only valid copy of the item. Data is requested by the lock instruction to obtain a lock.

The local thread cache block state is a subset of the attraction memory state of a data block. A cache block can also be in one of the following states:

- *Invalid*: Data is not valid.
- *Shared*: Data item is shared between this cache and any other cache or attraction memory according to the status of the item in attraction memory (owner shared or shared).

- *Exclusive* The item is readable and writable and has to be written through to the attraction memory when updated.

- *Pending* A read or write request is being sent for the item. A space is allocated for it.

The states in the group directory nodes reflect the state of the block in the processor attraction memory. The block can be in one of the following states:

- *Invalid*: Data does not exist.

-- *Shared*: One or more group node has a copy of this data item.

- *Owner Shared* One node in this group is the owner of the data item. It can also be shared by other group nodes.

- *Owner Exclusive* One node in the group has the only valid instance in the system.

- *Pending* A read/write request is being sent for the item. A space is allocated for the item in one or more of the group nodes.

-*Locked* A lock is held by one of the group nodes.

The MCOMA coherence protocol and the interaction between the different directories in the system is discussed in the following sections.

3.2.1 Attraction Memory Protocol

The attraction memory of a processing node is organized as a set associative cache with moderate assocciativity ($\geq$4). The attraction memory coherence protocol is based on distributed directories. It also uses snooping over the search interconnect to speed up the search for data and provide coherence and synchronization operations. The local directory for the data in the attraction memory resides in a reserved area and can not be cached or paged out to disk. Each directory entry has a status information about the state of the data item. To keep the inclusion property between the attraction memory and the local processor cache, an item in the attraction memory is marked cached to the processor cache. The processor cache is a write through cache. If the item is invalidated or moved from the attraction

memory, it will be also invalidated in the processor cache. On a cache miss, it is assumed that data items in local attraction memory are accessed without thread switching since the items can be cached and serviced locally.

In the following discussion of the attraction memory coherence protocol, the following conventions are used. A local node (LN) is the node that issues the data request. A local group node (LGN) is a node that belongs to the same group nodeas the local node belongs. Its directory is also contained in the same local group directory node as the local node. A local group directory (LGD) node is the group directory node to which the local node directory belongs. A remote node (group directory, processing or group node) is any node that is not local. These conventions are shown in Figure 3.7.

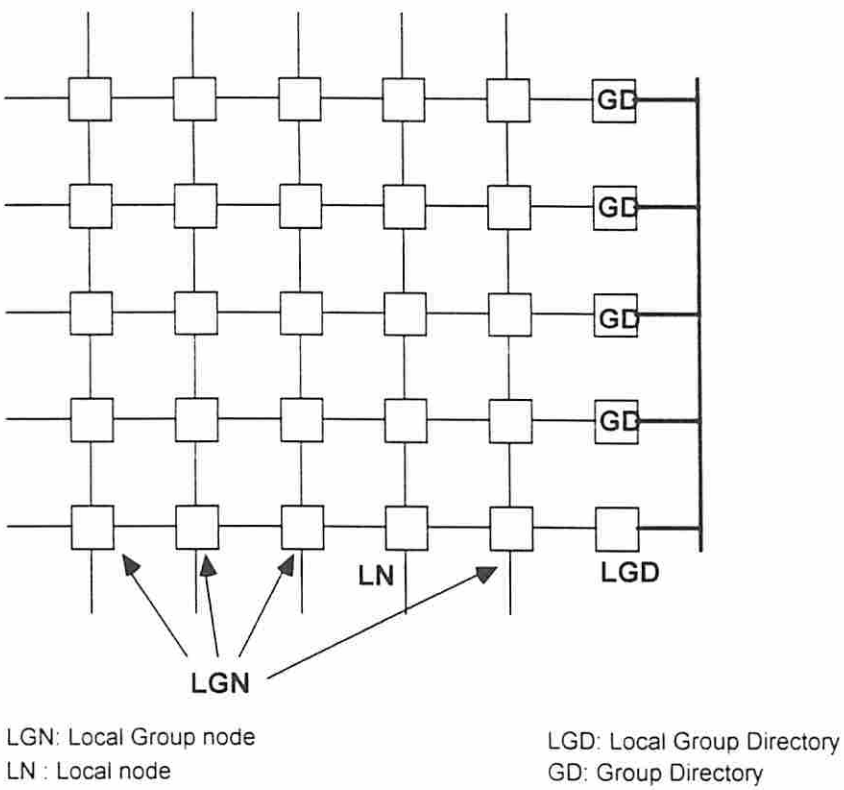


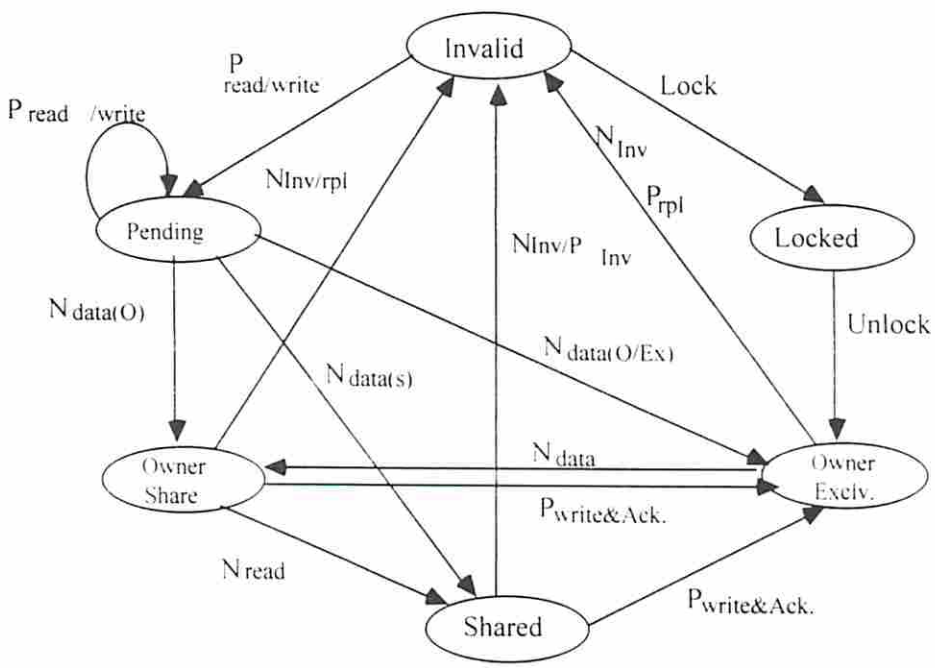
Figure 3.7. System Conventions

3.2.1.1 Read Requests

When a processor initiates a Read Share Request (load), the data item is first checked in the cache and the attraction memory. If a read-hit in the thread cache, the request is simply satisfied. If the data does not exist in the thread cache but exists in the local attraction memory, the request is completed on the local node and no state transition or network traffic is needed. If the data item existed as a Shared or Owner Shared in the attraction memory, the local directory is modified to show that the new requesting thread has a copy of the item. If the item was in Owner Exclusive State and it is cached to another thread on the same node, the state of the data item is changed in the corresponding thread cache to Shared. The data item is then cached to the new requesting thread in a Shared state. The attraction memory directory is modified to be holding the item in the exclusive state and it is cached to more than one thread.

On a read-miss, space is allocated for the item in the attraction and processor cache memories and the location is marked Pending. Any other request by a different thread for the same item would allocate a pending space in its cache and wait until the first pending request is satisfied. When the item is received, both pending locations are updated accordingly. A request for the item is sent over the mesh to the local group directory node LGD. If the item is found on LGD, i.e. it is in one of the local group nodes, the LGD directs the request to the LGN that has the item and updates its directory. The LGN subsequently sends the requested data to the requesting node and updates the directory. If the item in the LNG is in the Sharded state, it is sent as a shared item to the requesting node. If the LGN has the item in Owner Shared, it changes its state to Shared and the ownership of the item is transferred to the requesting node. If the item exists as Owner Exclusive, its new state is Exclusive Shared and the requesting node gets the item in the Shared state.

If the item is not on the LGD, the LGD sends the request on the search interconnect to other GDs. The GD that has the item in owner status sends an answering reply to the LGD. It also forwards the request to the appropriate node that has the item. This node sends the data to the requesting node over the mesh. All directories are updated as necessary. The last node that gets the data item from the owner node will have the ownership of the data item, i.e. the requesting node will have the item in its attraction memory in owner shared state. This will improve the chance of the item replacement in its new owner memory. A simplified attraction memory protocol is shown in Figure 3.8. It includes the basic operations in the protocol.



P: local processor request
N: network request
Inv: invalidation

O: Owner
Ex: Exclusive
rpl: replacement

Figure 3.8 Attraction Memory Protocol

3.2.1.2 Write requests

When a processor initiates a Write-request, the presence of the item is checked in the thread cache and the attraction memory. If a write-hit occurs and the item is available locally in Owner-Exclusive State, the local node can write it and keep the coherence between its cache and attraction memory. The thread caches are write-through caches. If the item is cached to any other thread on the same node, the other copy is invalidated and the directories are updated before the write.

If the item is in either Shared or Owner Shared State, an invalidation of the item is sent to other shared copies and the pending write is performed on the cache and the attraction memory. The group directory nodes are responsible for sending invalidation to its local copies, if it has any, and returning the acknowledgments. The group directory node that has the item in one or more of its nodes, sends invalidation to them and send one acknowledgment back to the requesting directory node. The LGD collects all the acknowledgments from all group nodes and sends one acknowledgment to the requesting node. The state of the data item is changed to Owner-Exclusive when the invalidation acknowledgment is received.

If a Write-miss occurs and the data item is not in attraction memory, the write request is performed as a of Read-miss followed by a Write-hit. If there are multiple write requests for the same data item, the one that wins the bus of the search interconnect is the winner of the race. The other request has to be retried.

3.2.1.3 Replacement

When the attraction memory does not have any empty space in a certain set to receive a requested item, it selects an item to be discarded or displaced. The item to be replaced is selected based on its state. The

replacement priorities for different items are: Shared, Owner Shared then Owner-Exclusive. The first candidate item for replacement should be the one in shared state since at least one other valid copy which is the Owner Shared copy, exists somewhere in the system. In this case, the item is simply discarded. When the victim item is in Owner-Shared or Owner-Exclusive State, the item and its ownership must be displaced to a new location in the system since it may be the only copy of the item in the system. Swapping the replaced item with the requested item on the remote node would guarantee a space for the replaced item and would reduce the search time needed to find a space on a different attraction memory to inject the item. The data item is transferred holding its original state to its new location [23].

In MCOMA, when a data item is replaced, the protocol will try to place it in one of the attraction memories that belong to the local group nodes. This would minimize the network traffic due to replacement and the upgrading of more than one group directory. If the item were requested by the same node later, the item would be on one of the group nodes, which improves the local group data locality. If logically related threads of the application are assigned to the same group of nodes, this would keep the data used by these threads in the group and minimize the number of remote data requests out of this group.

3.2.3 Thread Cache Protocol

The thread cache protocol maintains the coherence between different thread caches in the same node and the attraction memory. Since the processor cache is divided between four different threads that are using the cache, the protocol is designed to maintain the coherence between different running threads. The data item state in a thread cache is a subset of its state in the local attraction memory. The thread cache protocol is shown in Figure 3.9.

When a thread cache accesses a data item, it should be moved to its cache in the appropriate state. For a read request, the thread cache should have the item in shared state whether it is Shared or Owner Shared. For a write request, the thread cache should have the item in Exclusive State. If the block exists in the attraction as Owner Exclusive and is requested to be shared by more than one thread cache on the same processing node, it is marked Exclusive and Shared in the attraction memory and can then be cached to more than one thread cache.

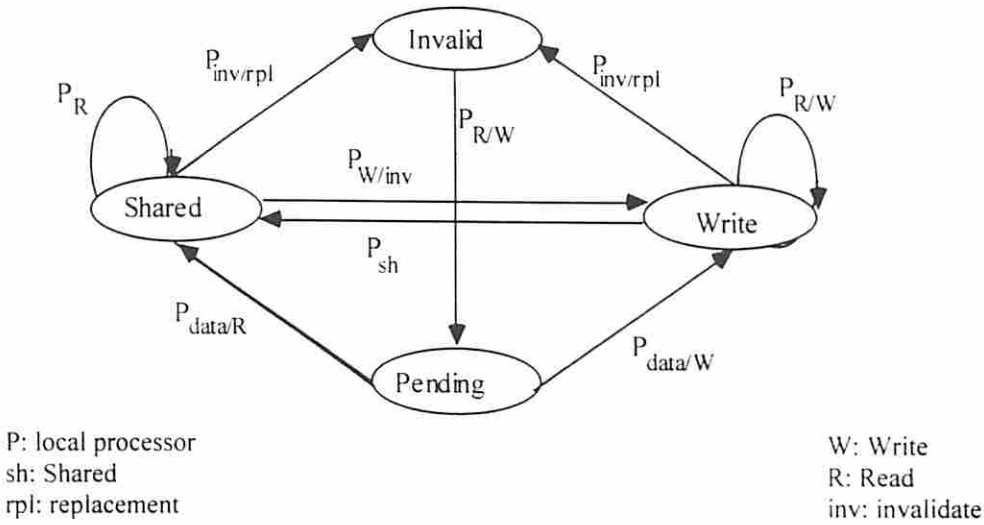


Figure 3.9 Thread Cache Protocol

3.2.2 Synchronization

Multithreaded processors use a non-spinning synchronization technique when requesting a lock [28]. When the lock is available, the processor is notified. When the lock is released, it should notify any waiting thread. Synchronization variables are accessed through synchronization instructions rather than regular load and store memory reference instructions. The MCOMA processor has a special instruction for synchronization support. The *lock* instruction gets the data in a locked form and sets the thread to a

high priority status. The *unlock* instruction releases the lock and changes its state to Owner Exclusive on the local attraction memory as well as the group directory node.

Since there is no home node for the data in MCOMA, the lock waiting list is implemented as a distributed list. One way of implementing this list is to allow only one pointer per node pointing to the next node requesting the lock. When a node requests a lock, it sends a lock request to its group directory node, which sends the request to other groups. If the lock is free, a remote node satisfies the request and the lock is granted to the requesting node. If the lock is not free, and there is no outstanding request for it on its group directory node, the request is attached to the lock at that node. When the lock is released, it is forwarded to the waiting requesting node by the directory node that has the lock. When there is at least one outstanding request for the lock, the pending request list is distributed over the directory nodes that are requesting it. To minimize the network traffic, each directory node satisfies all the requests of its processing nodes for a lock before it releases the lock to the following requesting group directory node.

Chapter 4

Qualitative Comparison between MCOMA and FCOMA

4.1 Introduction

In this chapter, an analytical comparison between MCOMA and FCOMA is presented. There are two main differences between MCOMA and other COMA architectures. First, MCOMA is a non-hierarchical COMA that does not require a home node to be assigned for the data. Second, MCOMA is the only COMA that uses multithreading as a latency tolerance technique. A comprehensive comparison between FCOMA and hierarchical COMA as well as NUMA architectures was done by other researchers [23,41,45], and it was concluded that FCOMA architecture outperformed both NUMA and hierarchical COMA architectures. In the following sections, a comparison between MCOMA and FCOMA architectures is presented. In this comparison, it is assumed that both architectures have the same common system parameters such as the memory overhead needed to realize the COMA architecture and the same memory organization which includes the size of both caches and attraction memory, the data block size and the cache line size.

4.2 Data Latency

In cache coherent shared memory multiprocessors that use write invalidation protocols, cache misses can be divided into three main types: cold miss, capacity miss and coherence miss. A cold miss is the cache miss that occurs when the processor references the data for the first time. Capacity miss is the data miss due to the finite size of the processor cache. Coherence miss is due to the loss of data because

of a coherence operation, such as the invalidation of the data item in one node when it is updated by another node

In NUMA architectures, each cache miss requires a remote memory reference to the home node of the data. The capacity miss is present since the processor cache is not usually large enough to capture the working set of an application. The effect of these types of misses almost vanishes in COMA architectures. In COMA, the attraction memory, which works as a large cache, is usually large enough to hold a large working set. The inclusion property between the attraction memory and the processor cache always holds true. If the data is replaced in the processor cache due to a capacity miss, it is very likely that, when the processor references this data again, the data will still be in the attraction memory and can be accessed locally. This is the main reason for the reduced number of remote memory accesses in COMA compared to that in NUMA. In the following section, the data latency in both FCOMA and MCOMA are compared.

4.2.1 FCOMA Latency

In FCOMA, a data request that misses in the local cache and attraction memory is sent to the home node of the data. At the beginning of the application execution, there is only one copy of each data item, which resides in the attraction memory of its home node. When the first read or write request is made to the data, the ownership of the data is transferred to the requesting node. Since the owner node is the node that answers any later request for the data, it is very unlikely to have the data request answered by the data home node after the initial request of the data. In an FCOMA that uses an N^2 mesh interconnection, the probability of such a condition is $1/N^2$. Thus, most of the data requests would be forwarded to a third node in the system; thus three hops on the network are expected to get the requested data.

For a load request, the processor cache is first checked for the data (T_{cache}), followed by a check of the attraction memory through the local bus ($T_{mem} + T_{bus}$). The request is then sent to the network through the local bus to the home node ($T_{net} + T_{bus}$). The home node checks its directory and sends the request to the owner node ($2T_{bus} + T_{mem} + T_{net}$). The request is satisfied by the owner node, and the data is sent to the requesting node [$T_{bus} + T_{mem} + (T_{net} + 2T_{bus})_{data}$]. The total data latency is [$T_{cache} + 5T_{bus} + 3T_{mem} + 2T_{net} + (T_{net} + 2T_{bus})_{data}$]. These steps are shown in Figure 4.1.

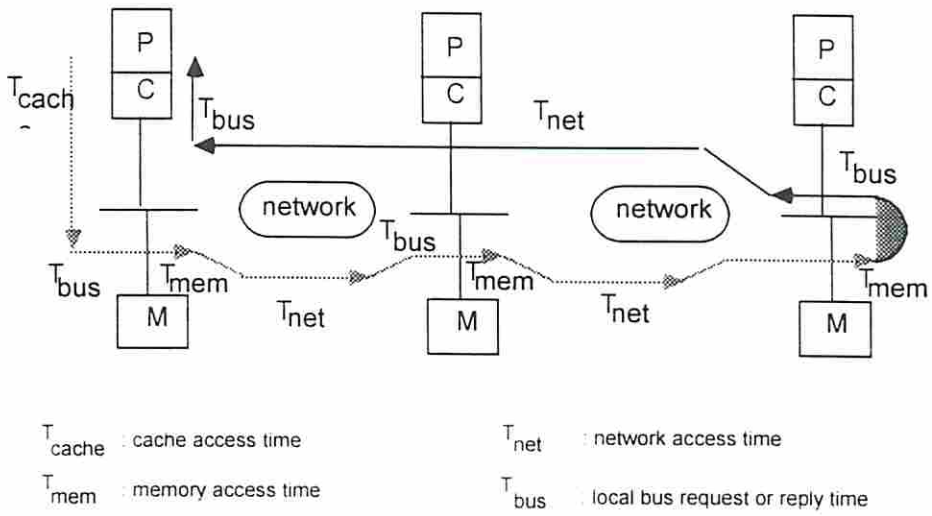


Figure 4.1 Data Latency in FCOMA

Since the probability that a data request needs two hops is $1/N^2$ and that the request requires three hops is $(1-1/N^2)$, then the latency for a load request in FCOMA is:

$$T_{FCOMA} = 1/N^2 [T_{cache} + T_{bus} + T_{mem} + T_{bus} + T_{net} + T_{bus} + T_{mem} + (T_{net} + 2T_{bus})_{data}] + (1-1/N^2) [T_{cache} + T_{bus} + T_{mem} + T_{bus} + T_{net} + T_{bus} + T_{mem} + T_{bus} + T_{net} + T_{bus} + T_{mem} + (T_{net} + 2T_{bus})_{data}].$$

Assuming that the time required to transfer a data request is similar to the time to transfer the data (this assumption is being used for both FCOMA and MCOMA where data is transferred on similar networks), then T_{FCOMA} can be reduced to

$$T_{FCOMA} = T_{cache} + (7 - 2/N^2)\, T_{bus} + (3 - 1/N^2)\, T_{mem} + (3 - 1/N^2)\, T_{net}$$

4.2.2 MCOMA Latency

The latency of a general model of MCOMA is shown in Figure 4.2. A load request is first checked in the processor cache as well as the local attraction memory ($T_{cache} + T_{bus} + T_{mem}$). If the data is not local, the request is sent to the group directory node ($T_{bus} + T_{Gnet}$). The local directory checks the availability of the data in any of the group nodes and if the data is found, the request is forward to the group node that has the data ($T_{dir} + T_{Gnet}$). The probability that a local group node has the data is 1/N for a system that has N^2 nodes. If the data is not in the local group, it is searched in the other group directories through the search network ($T_{Dbus} + T_{dir}$). The group that has the owner copy answers and forwards the request to its local node that has the data ($T_{Gnet} + T_{bus}$). This node would send the requested data to the requesting node through the network [$T_{bus} + T_{mem} + (T_{net} + 2T_{bus})_{data}$]. As for FCOMA, it is assumed that the time required to transfer a data request is similar to the time for data transfer. Then the total latency to satisfy this request is

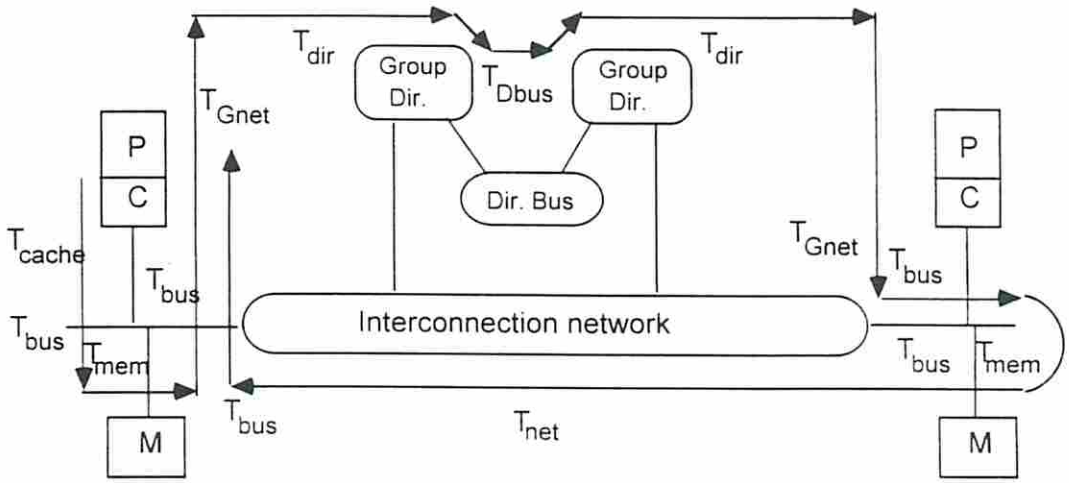
$$T_{MCOMA} = 1/N\,[T_{cache} + 5T_{bus} + 2T_{mem} + 2T_{Gnet} + T_{dir}] + (1-1/N)\,[T_{cache} + 5T_{bus} + 2T_{mem} + 2T_{Gnet} + 2T_{dir} + T_{Dbus} + T_{net}]$$

$$T_{MCOMA} = T_{cache} + 5T_{bus} + (1-1/N)T_{Dbus} + 2T_{mem} + (2-1/N)\,T_{dir} + 2T_{Gnet} + T_{net}$$

4.2.3 Latency Comparison

To compare between the two architectures, it is assumed that for a medium scale system, e.g. $N^2 = 64$ processing nodes, T_{FCOMA} can be simplified to:

$$T_{FCOMA} = T_{cache} + 7\,T_{bus} + 3\,T_{mem} + 3\,T_{net} \qquad (1)$$



T_{cache}	: cache access time	T_{net}	: network access time
T_{mem}	: memory access time	T_{Gnet}	: network access time for a group
T_{dir}	: directory access time	T_{bus}	: local bus request or reply time
		T_{Dbus}	: directory bus request or reply time

Figure 4.2 The Latency in MCOMA

In addition, T_{MCOMA} can be reduced to:

$$T_{MCOMA} = T_{cache} + 5T_{bus} + T_{Dbus} + 2T_{mem} + 2T_{dir} + 2T_{Gnet} + T_{net} \qquad (2)$$

This assumption favors FCOMA since its latency is reduced by factors of $1/N^2$ not factors of 1/N as in MCOMA. Comparing (1) and (2), FCOMA needs three memory accesses per request while MCOMA requires two memory accesses and two directory accesses. In most COMA implementations, such as DDM, the tag memory is implemented in SRAM, which reduces the time to check the tags in the directory to less than the time of a full DRAM memory cycle. For the existing technology, the fastest access time for a DRAM is 60 ns, while the access time for the SRAM is between 4 to 10 ns. Assuming a processor running at a 200 MHz (1 cycle = 5 ns), the difference in memory access time between FCOMA and MCOMA is, at least, 8 cycles.

FCOMA requires three hops on the interconnection network while MCOMA requires one hop on the interconnection and two hops on the group interconnection. These two hops on the group interconnection are, on the average, requires less time than the two hops on the interconnection network. The average distance between two arbitrary nodes for a two-dimensional mesh interconnection with N^2 processing nodes is (N-1). For the group nodes, grouped on the horizontal axis as in MCOMA, the average distance between two nodes in the same group is 1/2(N-1). This means that the time a message spends on the network of MCOMA is, on the average, less by one hop than the time of a message spends on FCOMA.

For a 200 MHz processor, the clock cycle is 5ns. Assuming that the access time for a DRAM is 60 ns (12 clock cycles), the SRAM access time is 10ns (2 clock cycles) and the processor cache access time is

1 clock cycle. The directory bus clock is assumed to be 50 MHz (4 cycles) and the local bus is assumed to be clocked at 100 MHz (4 clock cycles). Each mesh interconnection is assumed to cost 1 clock cycle.

The latency for a data load request in FCOMA T_{FCOMA} can be calculated from (1) to be, on the average, 72 clock cycles. The latency for MCOMA from (2) T_{MCOMA} is calculated to be, on the average, 57 clock cycles. From this comparison, it is shown that the data request latency on the MCOMA is better than that of FCOMA.

4.3 Memory Overhead

In COMA, the system requires additional physical memory to allow for data replication and migration. Each memory block also requires tag and state information in addition to the directory memory. Both FCOMA and MCOMA have the same requirement of the physical as well as the tag and state information. The two systems differ mainly in the directory requirement, which is discussed in the following section.

4.3.1 Directory overhead for FCOMA

In FCOMA, the home node of the data should have a set of status bits; a presence bit vector for all the system nodes and the ID for the processing node that has the master copy.

Number of presence bits $= N^2$

Number of bits for master copy ID $= \log_2 N^2$

The number of bits that is required per directory entry to store the information of one memory line is:

Directory bit per entry (FCOMA) $= N^2 + \log_2 N^2$

4.3.2 Directory overhead for MCOMA

Each directory entry in MCOMA should have a set of status bits; a presence bit vector and an ID of the processing node that has the master copy. Since the group directory node has the presence bits of the nodes in its group, then for a mesh interconnection that has N^2 nodes each directory entry has

Number of presence bits = Number of nods per group

= N

Number of bits for master copy ID = $\log_2 N$

The number of bits required per group directory entry to store the information of one memory line is:

Directory bit per entry (MCOMA) = $N + \log_2 N$

This analysis shows that the directory overhead for FCOMA is higher than that of MCOMA because the directory node of MCOMA has presence bits for the group nodes only. This reduces the size of the presence bit vector as well as the number of bits required to store the master copy ID. However, if the data item is shared by more than one group, the time required to access the directory would increase since the directory information is repeated through different directories, but its upper bound is $N^2 + N \log_2 N$.

4.4 Network Contention

The COMA performance is greatly affected by the cache as well as the attraction memory hit rate. The COMA architecture is mainly dominated by coherence misses since capacity misses are reduced to the

minimum by the large attraction memory and the inclusion property between this memory and the processor cache. As long as this property holds true, any data replaced in the processor cache because of its limited capacity, the data would most likely remain in the attraction memory. Since most data requests can be serviced locally, this would result in lower memory access latency and less network traffic.

Cold misses usually have little effect on the system performance assuming a good initial data placement for the application. These misses are very small, typically 0.1 - 0.3 % for different applications [45], and are not considered to have a large effect on the total system performance. FCOMA suffers from the extended effect of initial data placement as compared to MCOMA since the home node of the data is fixed through the execution time of an application. For MCOMA, initial data placement effects only last until the data is referenced for the first time by the node using it. The optimization of the initial data placement would not have any further effect through the execution of an application.

In MCOMA, the separation of the search interconnection from the interconnection network relieves the later from the data search traffic. The search interconnection deals with data requests, which are short messages that do not carry data. The group directory node would try to satisfy the data request first before passing it to the other groups. This reduces the number of requests serviced by the search interconnection. MCOMA, unlike FCOMA, can exploit request combining on the directory nodes. This would further limit the traffic on the search as well as the interconnection networks.

Reducing the potential traffic on the search interconnection allows for the use of a fast and wide shared bus to implement it. For larger systems, as the machine scalability is limited by bandwidth of the shared bus, the search interconnection can be implemented using a higher bandwidth interconnection such as a crossbar interconnection. This search network would allow for searching the groups closer to the requesting processor first. In this case, the searches of the group directory nodes are done sequentially

for different directory nodes, but more than one request can be served simultaneously by the search interconnection. For shared data requests, the protocol can be modified such that the first node that locates any shared copy would answer the data request. This will reduce the search time and the node closer to the requesting node would answer the request, which would reduce the traffic on the interconnection network even more. For larger system, more aggressive interconnections, which can afford a high bandwidth, such as optical interconnection, can be used to implement this search interconnection.

The network throughput can also be limited by hot spots resulting from concentrated data accesses for heavily shared data items. MCOMA does not suffer from hot spots as FCOMA. Since all requests of a data item must go through the home node of the data, FCOMA can suffer from the hot spot effect for certain data items. The home node of the data has to receive the entire data request then forward them to the master node that has the data. This is not the case in MCOMA since data is not tied to any specific location in the system and the node that has the master copy is the node that responds to the data request. The master copy status is then forwarded to the new node that requested the data. This new node handles the following request for the data. MCOMA can also use request combining on the directory nodes, which further reduces the effect of the hot memory spots.

The above discussion shows that FCOMA can suffer from potential interconnection contentions due to the unnecessary access to the home node of the data. Unlike MCOMA, its performance can also suffer from the hot spot effects. On the other hand, the separation of the search traffic from the data traffic in MCOMA relieves the interconnection network from possible traffic contention. The search interconnection that deals only with short messages would facilitate a fast data search without the tie up of any processing nodes. It also allows request combining to reduce the amount of network traffic. This search interconnection should be, depending on the size of the system, fast and efficient to reduce the possibility of contentions.

4.5 Summary

In this section, it was argued that the expected performance of the MCOMA architecture could exceed that of FCOMA due to the reduced data latency and the possible reduction of contention on the interconnection network. Since FCOMA outperformed other existing architectures such as DDM and the DASH architectures, MCOMA has a good potential for good system performance.

The improved data latency of MCOMA is mainly due to the reduction of the number of hops on the interconnection, which is done by using the search interconnection to locate the requested data. The search interconnection mainly supports global communication between different groups, which is lacking in the mesh interconnection. This support of global operations reduces the data search time, the data search traffic on the mesh, as well as the effect of memory hot spots in the system. To avoid the saturation of this search interconnection, the processing nodes in one group can be assigned related tasks. This increases the probability of having local communication within the group nodes and reduces the communication between the different groups in the system. This, in turn, would decrease the data latency in the system since data requests would be answered locally by the group nodes. The traffic on the interconnection network would also be reduced.

Chapter 5

Methodology

Architecture simulation has been chosen as a method for evaluating the performance of MCOMA. The system simulator is tested using the SPLASH benchmarks, which provides broad coverage of scientific and engineering applications [42]. This chapter discusses the important details about the developed simulator and its use for the study conducted in the following parts. The benchmark applications used for the evaluation of the system model are also presented.

5.1 The Simulation System

As the complexity of multiprocessor architectures has increased, it has became more difficult to model their behavior analytically. Simulation is an essential tool for the design and evaluation of new multiprocessor architectures and their performance. It can help to predict the performance and identify the bottleneck of unavailable hardware. This enables the evaluation and the behavior of applications running on more powerful architectures than currently available. Architecture simulation can basically be divided into trace driven simulation and execution driven simulation. Trace-driven simulation allows a timing simulator to estimate the timing of a program, presented by a trace of execution-events, on the target machine. Execution-driven simulation allows realistic simulation to be performed. It allows for the actual direct execution of real application programs. It also speeds up the experimental process by allowing rapid feedback from simulation results. The next three sections describe the MINT multiprocessor memory hierarchy simulation environment, based on execution-driven simulation, that is

used to develop the MCOMA simulator and the main simulation parameters used to simulate the MCOMA.

5.1.1 Simulator Development

Execution driven simulation is divided into two main parts: the front end, which is the memory reference generator, and the back end, which is the target system simulator. The front end executes the application on a number of processors. When the program executes a memory reference instruction, an event is generated and sent to the back end, which models the system memory hierarchy and the interconnection. When the event is performed, the back end signals the front end to continue executing the application.

The MCOMA simulator represents the back end for the front end MINT (MIPS Interpreter) hierarchy simulator [48]. MINT is an execution-driven simulator for modeling the memory hierarchy in multiprocessor systems. It provides a set of simulated processors that run standard UNIX executable files compiled for a MIPS R3000 based multiprocessor and has some support for the MIPS R4000 instruction set. It uses a novel hybrid technique of native execution and software interpretation to minimize the overhead of processor simulation. The target processor state is kept in memory as much as possible. It uses a form of code synthesis to encapsulate a block of code that does not generate events (no branches or memory references). This reduces the cost of loading registers from memory on every instruction. The synthesized function loads the necessary operand register and executes the target instruction natively then stores the register modification into memory. This allows MINT to interpret the instruction at native speed and eliminate unnecessary context switching overhead.

MINT simulates a collection of processors and provides support for spinlocks, semaphores, barriers, shared memory and most of UNIX system calls. It runs on Silicon Graphics computers, DEC stations

and SPARC workstations. Each instruction of the target machine can be executed in a single simulated cycle. When the front end performs a memory reference operation, it sends the event to the MCOMA simulator and the processor that generated the memory reference is stalled. The MCOMA simulator then determines the latency required in performing the memory reference operation. When it completes the generated event, it signals the MINT to continue the execution of the stalled processor.

The application program to be run on the simulator does not need to be modified. It is compiled separately, as it would be for a real parallel processor, and then linked with any libraries that it uses. The linked object code module is treated as input data to the simulator. The block diagram of the MCOMA simulator using MINT is given in Figure 5.1.

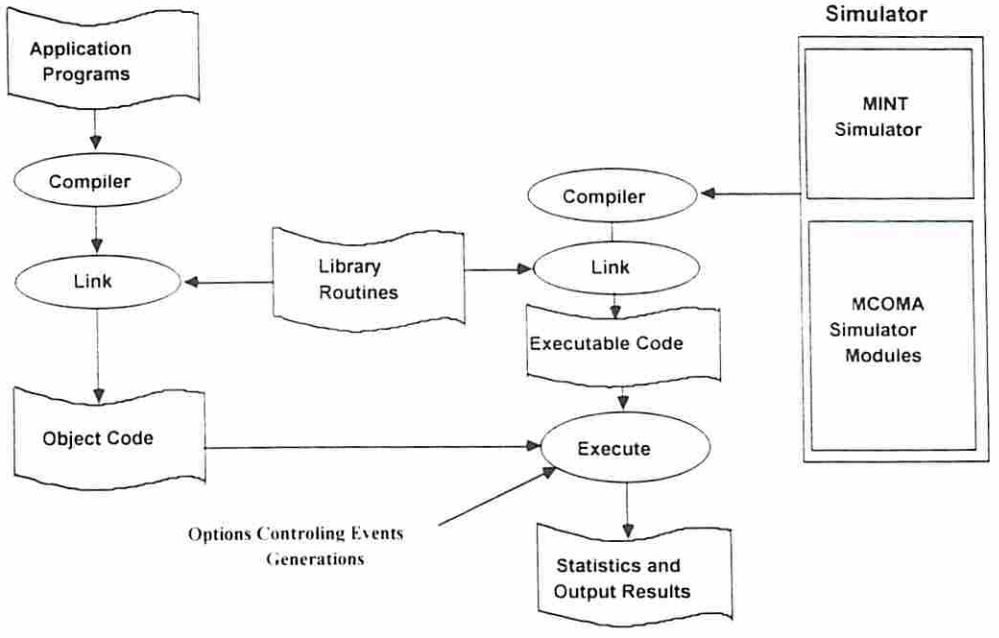


Figure 5.1. MCOMA Simulator Using MINT

5.1.2 The Architecture Model

The MCOMA architecture being modeled for this study simulates a moderate sized machine that runs applications chosen from benchmark the SPLASH suite [42]. The basic machine consists of 32 processing nodes and four directory nodes. The processor is assumed to be a high performance microprocessor based on the MIPS instruction set [36]. The processor clock rate is assumed to be 200 MHz (5ns clock cycle). Each processor has an 8 Kbyte of processor write-through cache per thread, with one cycle access time for read hits and two cycles access time for write hits. The cache per thread is assumed to be small since it only captures the thread locality. The attraction memory is modeled as 2 Mbytes DRAM with an access time of 60nsec (12 clock cycles). A tag memory is modeled as SRAM with access time of 10ns (2 clock cycles). The directory is assumed to be accessed in parallel with the attraction memory so it does not increase the memory access latency. The local bus is assumed to be clocked at 100 MHz and the group directory bus is clocked at 50 MHz. All latency numbers assumed for modeling this architecture are based on currently available technology and are consistent with other parameters used by other research groups [27]. The interconnection network is modeled as a pair of wormhole routed meshes each with 32-bit wide channels. The mesh interconnection is assumed contention free.

The use of a second level cache should not have a great impact on the present architecture. As this architecture intends to study the impact of combining multithreading and the new COMA organization, this relies mainly on the organization of the attraction memories and their coherence protocols. Hence, no second level cache is assumed.

5.1.3 The Simulator

The MCOMA simulator is developed to study the performance of the MCOMA architecture and the effect of using multithreading to tolerate latency on the total system performance. Different system parameters are used as simulation options to enable the study of different design variations on the overall system performance. The processor used is assumed to have a set of instructions similar to that the MIPS R3000. The register files are assumed to be replicated, and each active thread is using its own set of registers. It is also assumed that each thread has its own cache such that the thread would not loose its data locality while switched on long remote data access. These allow for a short context switch time, which assumed to be four processor cycles.

The interconnection network consists of a pair of wormhole routed meshes, each with 32-bit wide channels. One mesh is dedicated to request messages and the other to replies. Each routing node in the interconnection has 5 ports: 2 ports to connect nodes on the x-axis, 2 ports to connect nodes on the y-axis and one port connected to the processing node.

There are two types of messages: request messages (such as read and invalidation requests) and reply messages (such as read reply and invalidation acknowledgments). Each message consists of a message header and message trailer. The header bits represent the routing information of the message and are deleted from the message upon arrival to the destination node. Messages are divided into flits to be sent on the network. Each flit is 32-bit long, which is the width of data path of the network. The header occupies one flit. In case of a request message, the message address, the message type, the requesting thread number and other information requires another two flits. For reply messages that include data, an additional flit is required for every 32 bits of data. This depends on the length of the cache line. The time delay to send one flit between two adjacent route points is assumed to be one cycle. The search interconnection is modeled as a 64 bit wide bus with a clock rate of 50MH. The contention on the

directory node bus is simulated. Requests are schaduled on the bus only when the bus is free. If the bus is busy, the request is retried.

Threads are switched on an attraction memory miss. Threads are scheduled using a round-robin policy. The context switching time is assumed to be four cycles. Once a thread is switched out on a data miss, it is put in a blocked state. When the requested data becomes available, the thread is put in a ready state and added to the ready queue. It does not resume execution until all other ready threads are executed.

5.2 Benchmark Applications

The SPLASH suite consists of a mixture of complete applications and computational kernels [42]. The programs present a variety of scientific, engineering and graphics applications that have different communication requirements. Four programs are used in this study, barnes-hut, cholesky, mp3d and locus route. The programs used were chosen to represent a variety of important applications. Each of the benchmark applications used has some unique characteristics. The following subsections have a brief description of each of these applications.

5.2.1 Barnes –Hut

This application simulates the evaluation of a system of bodies in three dimensions under the influence of gravitational forces. Each body is modeled as point mass and extracts forces on all other bodies in the system. The simulation proceeds over a number of time steps, using the Barnes-Hut hierarchical N-body method. Each step computes the net force on every body and thereby updates each body's position and other attributes.

The Barnes-Hut algorithm is based on a hierarchical octree representation of space in three dimensions. The root of the tree represents a space cell containing all the bodies of the system. The tree is built by adding particles into the initially empty root cell and subdividing a cell into its eight children as soon as it contains more than a single body. The tree is traversed once per body to compute the net force acting on that body. If the center of mass of the cell is far enough away from the body, the entire subtree under that cell is approximated by a single particle at the center of mass of the cell. If the center of mass is not far enough away, the cell must be "opened" and each body is computed. Therefore, a body computes interactions directly with other bodies that are close to it. This algorithm reduces the complexity of force computation among N bodies from $O(N^2)$ to $O(N\log N)$.

The program executes each of the phases within a time step in parallel, but it does not explicitly exploit parallelism across phases or time-steps. The tree building and center of mass computation phases require both interprocessor communication and synchronization. Communication patterns are dependent on the particle distribution and are quite unstructured. Excellent data locality is achieved mainly due to temporal reuse of cached data. Because of the nature of the tree traversal, most of the interactions computed are between bodies on the same processors.

5.2.2 Cholesky Factorization

This program performs parallel Cholesky factorization of a positive sparse definite matrix. It factors a sparse matrix into the product of a lower triangular matrix and its transpose. Given a positive definite matrix A, the program finds a lower triangular matrix L, such that $A = LL^T$. Cholesky factorization typically proceeds in three steps. The first step is the ordering of the rows and columns of A to reduce the amount of fill in the factor L. The second step is the symbolic factorization, which determine the non-zero structure of the factor matrix. The third step is the numeric factorization, which determines the actual numeric values of the non-zero entries in L. This program does no reordering, and the input

matrix is a reordered matrix. The numeric factorization is the most time-consuming computation and is performed using a dynamic version of the supernodal fan-out method.

Several processors may modify a supernode before it is placed on the task queue. It is then read and used by one processor to modify other supernodes. After the processor completes all the modifications to other super nodes, it is no longer referenced by any processor. This application has large communication to computation ratio for comparable problem sizes.

5.2.3 Molecular Dynamics (MP3D)

MP3d solves a problem in rarefied fluid flow simulation. It models particle flow in extremely low-density medium. Under such conditions, the traditional fluid flow models that assumes continuous medium, such as Navier-Stokes, are not reliable. MP3D uses the Monte Carlo method as an alternative. It simulates the trajectories of a collection of representative molecules, subject to collisions with boundaries of the physical domain, with object under study, and with other molecules. In each time step, particles are moved according to their velocity vectors and collisions are modeled. After a steady state is reached, analysis of the trajectory data produced is used to estimate the flow field for the configuration under study. This method is compute intensive.

The molecules are statically scheduled on processors and are not related to their position in space, which changes with time. Data is actively being updated and references are relatively random depending on the location of the particle being moved. Access patterns to the space array therefore exhibit lower processor locality, which severely degrade parallel performance.

5.2.4 LocusRoute

LocusRoute is a commercial quality VLSI standard cell router. It evaluates the standard cell circuit placements by routing them efficiently and determining the area of the resulting layout. The program routes the wires through regions, which have few wires running through them. Routing a wire segment requires the evaluation of several routes then choosing the best path. A cost function is evaluated for each path based on the number of wires the segment will pass through. Several possible routes can be evaluated in parallel. This route evaluation involves reading and comparing linear sequences of vertical and horizontal array elements. This application has limited parallelism and high data access rate. One lock per task is used to guarantee mutual exclusion. It uses barrier synchronization to separate between iterations of wire routing, but no barriers are used within an iteration.

Application	Description	Main Characteristics
Barnes-Hut	Hierarchical N-body Simulator	Very low miss rate. Global synchronization
Cholesky	Cholesky factorization of space m	Moderate miss rate. long local synchronization
MP3D	Fluid flow simulation	Very low data reuse. High data access rate
Locus Route	VLSI standard cell router	High data reuse Low data miss rate.

Table 5.1 Application Characteristics

The characteristics of the application used are summarized in Table 5.1. The performance of multithreading is affected by the data access rate of the applications. In our system model, the threads

are switched on a remote memory accesses. Applications that have high data access rates are expected to benefit from multithreading with a larger number of threads. Applications with high data reuse can have a good performance with less number of threads. Barrier synchronization is also expected to affect the multithreading performance. Table 5.2 lists the input data sets used for the different application.

Application	Input Data Set
Barnes-Hut	8192 particles.
Cholesky	1806-by-1806 with 30,824 non-zeros (bcsstk14)
MP3D	10K molecules, 50 steps, test.geom
Locus Route	Primary1.grin, with 1266 wires and 481-by-481 Coast array

Table 5.2 Problem Sizes Used in Simulation

Chapter 6

MCOMA Performance Evaluation and Simulation Results

6.1 Introduction

In this chapter, the performance evaluation of the MCOMA architecture and the effectiveness of multithreading are presented. As was discussed in chapter 3, there are some advantages and disadvantages of combining COMA and multithreading. While multithreading hides the long remote data access time of the COMA, it may cause more data displacement and more remote accesses on a processing node. Our results, which are presented in the following sections, show that combining the new MCOMA architecture and multithreading improved the overall performance of the architecture.

The performance evaluation of the MCOMA architecture is done by using the simulator described in chapter 5. The simulated model assumes 32 processing nodes, each with a separate cache per thread and has no second level cache. The model assumes a multithreading support for up to 4 threads per processor. The processing nodes are connected by an 8x4-mesh interconnection. A common bus connects the group directory nodes. In the simulated model, threads are switched on a remote data access. The contention on the directory node bus is also simulated.

A subset of the SPLASH suite has been used to evaluate the MCOMA. The simulation results of the architecture are presented in the following sections. The multithreading effect on data locality in the attraction memories, the group nodes and the global bus traffic is also presented. It is assumed that threads that are allocated to a processor stay on this processor for its lifetime.

In this performance evaluation, unless otherwise specified, each processor is assumed to have 8Kbyte fully associative cache per thread. The cache block size is 32bytes. The attraction memory is assumed 2Mbytes per processor with a set assocciativity of eight. The attraction memory block size is 32 bytes.

6.2 Architecture Speedup

In this section, the speedup of the four different applications for the MCOMA architecture is presented. Each application has been run on various numbers of nodes between 1 and 32 nodes and for different number of threads 1, 2, and 4. The speedup of the application is defined as the speed up of a given number of processors relative to the fastest single processor, single-thread processing node. The fastest single processor is assumed to have a 100% attraction memory hit rate.

The performance of Barnes-Hut is shown in Figure 6.1. For a single thread, MCOMA speeds up to 23 on 32 processing nodes. The two-thread speedup increases to 38 and up to 52 for 4 threads. When the speedup is calculated relative to the single processor with four threads, the speedup of the architecture with 32 processor nodes each executing four threads is limited to 18. Since Barnes-Hut has many long synchronization stalls, it spends a long time on global communication during which time the threads are waiting for synchronization events and the processors are not utilized. This limits the benefits of multithreading, which is only useful during the computation phase of the application.

Cholesky speeds up is relatively better with multithreading on MCOMA. Figure 6.2 shows a steady speedup for the different number of threads. The speedup of the application was limited by the large number of coherence misses. As the number of processors increased, the coherence miss rate also increased. The performance of multithreading was affected by the synchronization waiting time. The BCSSTK14 is a relatively small matrix, which was used to limit the simulation time. Using a large number of threads limits the available concurrency on such a matrix, and threads tends to spend more

time waiting for synchronization events, which limits the overall system speedup. The use a of large matrix for this application usually reports better speedup results [50].

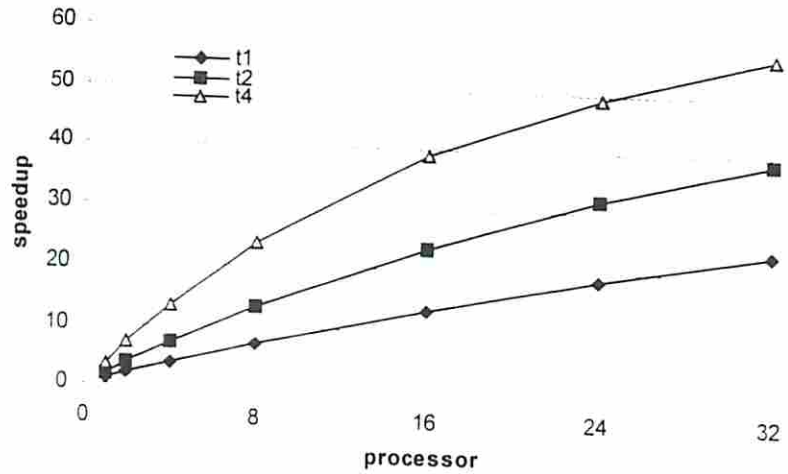


Figure 6.1 Barnes-Hut Speedup

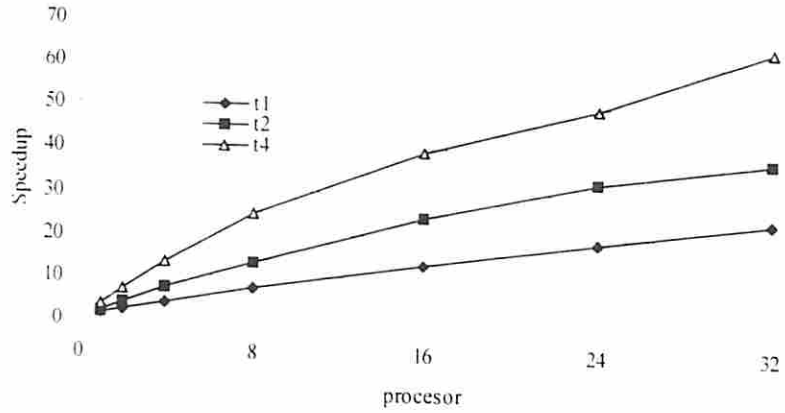


Figure 6.2 Cholesky Speedup

The MP3D speedup is shown in Figure 6.3. This application has a high data miss rate that limits the gain of this application on parallel processors. Since the molecules are assigned statically to threads, there is no attempt to assign nearby molecules to the same thread. The molecules are continuously

moving, thus the interaction between them is changing. The space cells are referenced in a relatively random manner depending on the location of the particle being moved. This causes poor data locality for this application. This application also suffers from a low data reuse rate. Data moved to local memory is used but would not be reference gain in the near future because of the continuous data and molecules movement. A remote memory access is costly in a COMA architecture since it involves a long data search, but since the application has poor locality, it does not make use of the data moved to its attraction memory. As multithreading is added, it hides some of these remote memory latencies, which improves the system performance, but it introduces more data conflicts in the attraction memory used by all threads.

The MP3D has a limited speedup for one thread. Similar limited performance has been reported on parallel machines due to the shared-memory workload [3,33]. The use of multithreading slightly improved the performance of the application. When multithreading is used, the number of data accesses per node increases. The speedup gain for 1, 2, and 4 threads is almost linear for up to 8 processors, where all the accesses were within the group nodes. As the number of nodes increased, more node groups were formed, and the remote data accesses must use the global bus. The high traffic on the search bus slowes the system speedup since the number of remote accesses for this application is high. This explains the sudden drop in system performance for more than eight nodes.

Figure 6.4 shows that LocusRoute has a good speed up on MCOMA relative to other reported architectures. LocusRoute has a high data access rate with high data reuse. The limited improvement in speedup with multithreading is due to the higher number of lock synchronizations. If a thread is switched out on a remote data miss while holding a lock, the local processor utilization might be improved but the other threads, if waiting on the same lock, are delayed until the lock is released. As the number of threads increased, threads are switched more often due to conflict misses in the attraction

memory. The threads holding locks are expected to resume execution sooner. This improves the total system performance with larger number of threads.

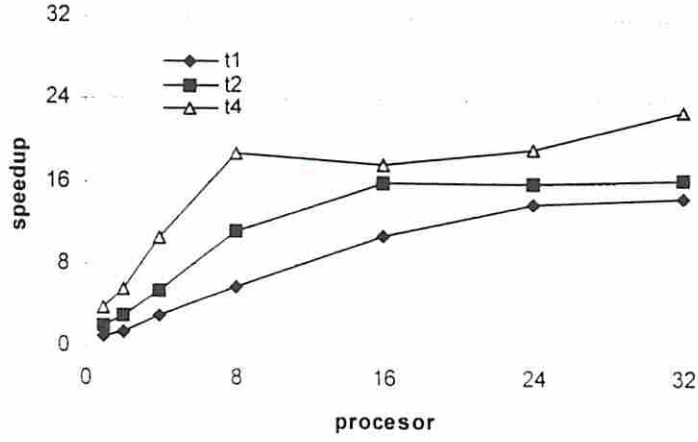


Figure 6.3 MP3D Speedup

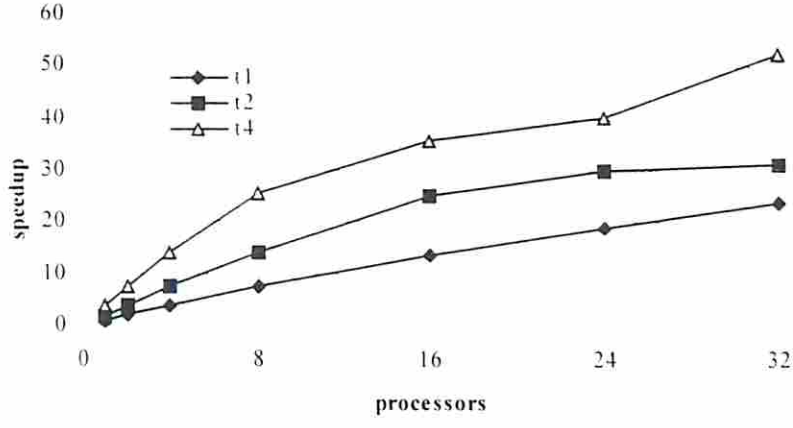


Figure 6.4 LocusRoute Speedup

The above results shows that the performance of the MCOMA architecture achieved a good speedup compared to other systems speedups reported in [3, 33]. It also shows that multithreading improved the overall system speedup for all applications even though these applications were not optimized to be

used on MCOMA. MP3D is the only application with limited speedup but this is consistent with its performance on other architectures where it tends to have the lowest speedup of all the applications due to its very low locality.

6.3 Cache and Attraction Memory Performance

The performance of MCOMA is greatly dependent on the processor cache and its attraction memory performance. It is also dependent on the group organization and the localities on the neighboring nodes. In this section, the impact of the capacity and coherence misses on the system performance is discussed. Different cache size variations as well as its organization and its effect on both the attraction memory and the group hit rate is presented. The impact of the variation of these parameters on the processor execution time is evaluated. Then, the impact of using multithreading on different variation of the application parameters of the architecture parameter is also presented.

6.3.1 Capacity and Coherence Miss Rates

Capacity miss rate of a node is defined as the fraction of cache data misses due to the insufficient space in the cache. When data is displaced from the cache due to limited cache space in COMA architecture, it is still available on the local attraction memory. Since the attraction memory is usually large enough to capture the working set of most applications, the data movement due to cache capacity can still be accessed locally from the attraction memory. The coherence miss is defined as the data miss in cache due to data invalidation. A future access to these missed data has to request the data from remote nodes. These types of data miss affects the performance of the COMA architecture, since the remote data access involves long data search on different processing nodes.

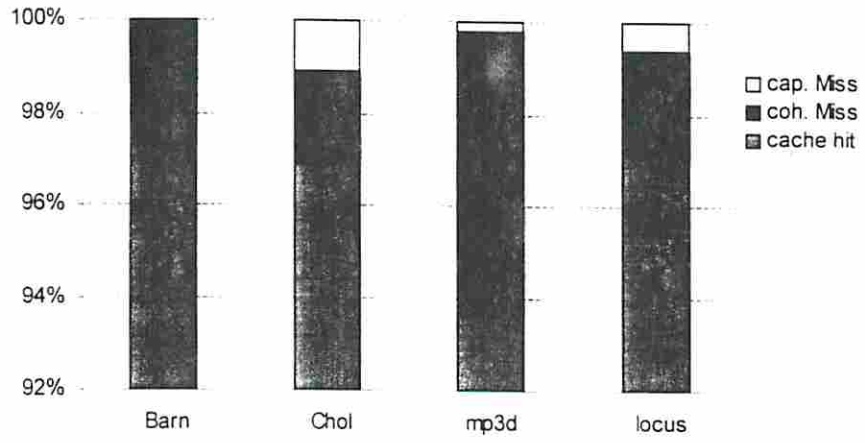


Figure 6.5 Capacity and Coherence Miss Rate for One Thread

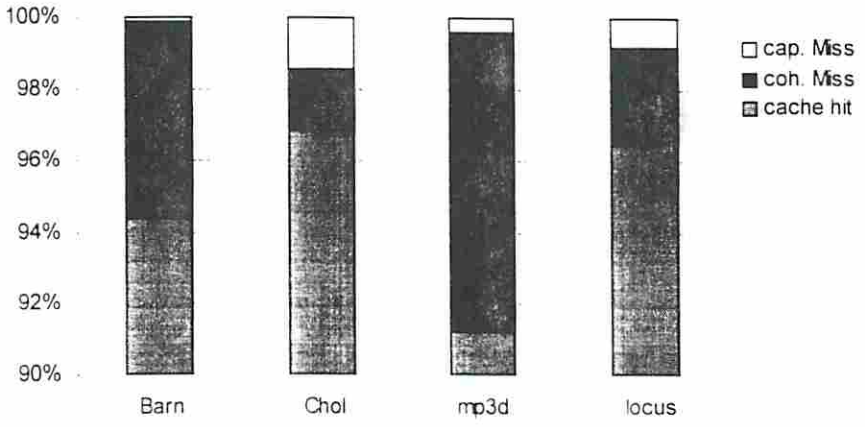


Figure 6.6 Capacity and Coherence Miss Rate for 2 Threads

Figure 6.5 shows the different cache miss rates for the four applications, for a single-thread processor. There is a large variation on the capacity and coherence misses for different applications. MP3D has a high coherence rate and measurable capacity miss rate, which explains the low application speedup due to remote accesses, as discussed in the pervious section. Barnes-Hut has a high hit rate in the simulated result despite of the dynamic change in the problem domain. The high miss rate is because of the hierarchical tree traversed which allows for interacting bodies to be assigned to the same node.

Cholesky and LocusRoute have a moderate coherence miss rate and relatively good cache hit rate, which is reflected in these applications' speedup.

As multithreading is added to the system, Figure 6.6 and Figure 6.7 show that the MP3D capacity miss rate is further increased due to the data conflict of multiple threads using the attraction memory. Although, different threads have separate caches, the data conflicts as well as invalidation in the local attraction memory cause more cache misses in the threads' caches. Increasing the number of threads caused LocusRoute to suffer from higher capacity misses due to its high data access rate.

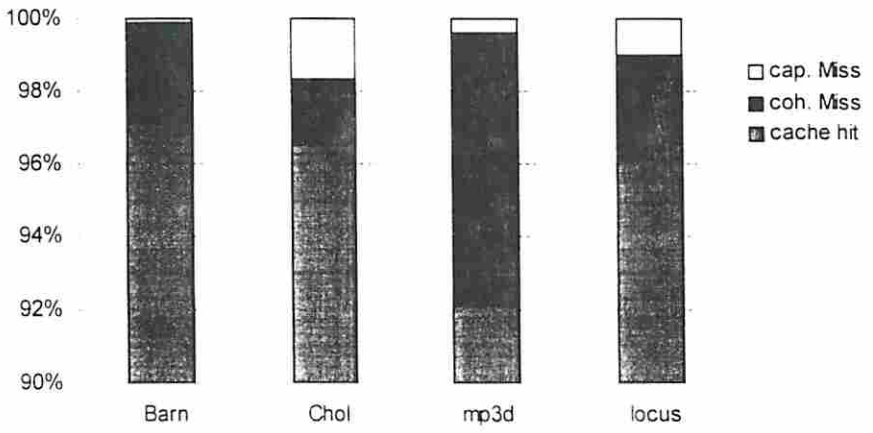


Figure 6.7 Capacity and Coherence Miss Rate for 4 Threads

6.3.2 Attraction Memory Hit Rate

Attraction memory in COMA works as a large secondary cache that accumulates data used by the local processor. Most of the data lost from local cache due to its limited size is kept in the local attraction memory unless it is invalidated by another processor or replaced by a new data item. Figure 6.8 shows that the attraction memory used in our simulation usually was sufficient to hold the working set of all

the applications except for MP3D, due to its poor data locality. Increasing the number of threads reduces the attraction memory hit rate, due to the data conflicts from different threads.

6.3.3 Group Hit Rate

The group-hit rate is defined as the percentage of remote data requests that is satisfied from the group nodes. The higher the group hit rate, the less request messages are sent over the search bus. This will depend mainly on the application data access pattern and the allocation of related threads on the same group nodes. In our simulation, there was no allocation scheme used to ensure any thread allocation. The threads were allocated mainly at random.

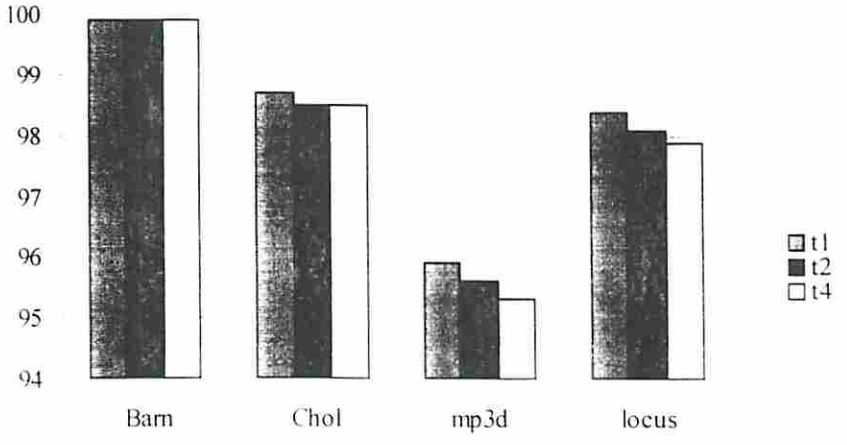


Figure 6. 8 AM Hit Rate For Various Number of Threads

The group hit rate varied for all applications as shown in Figure 6.9. For Barnes-Hut, the Group hit rate is high since the application organizes the bodies as a tree data structure. This allows for better data allocation on the neighboring processors. Consequently, the data is more likely to be found on the group

nodes. This lowers the remote access time for this application. As the simulation results show, this application has the lowest remote access time with respect to the total execution time.

For LocusRoute the group locality improved by adding multithreading. Locus uses geographic scheduling, where regions of the circuit are associated with a task queue and processes are assigned to regions as part of the initialization. As the number of processes increase, several processes are assigned to the same region. This improves the data locality of the program on the group nodes. The MP3D group hit rate is the lowest of the applications due to its poor data locality. Increasing the number of threads per processor causes the group hit rate to deteriorate more.

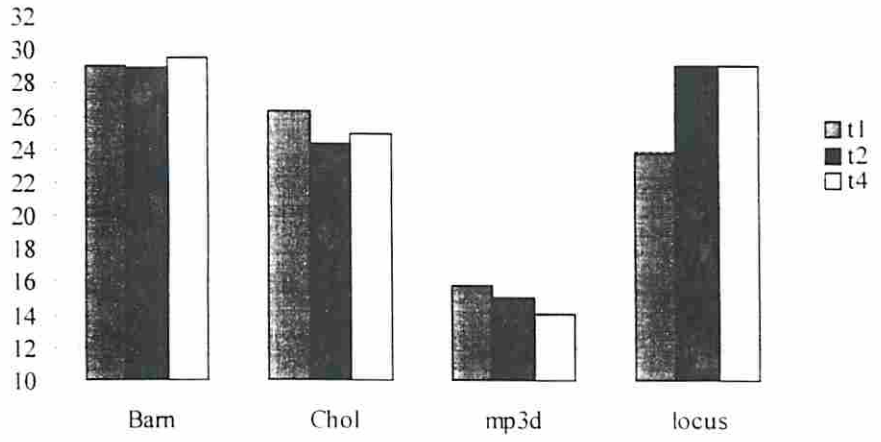


Figure 6.9 Group Hit Rates for Various Number of Threads

6.3.4 Block Size Effect

As the block size increases, more data is prefetched on a data miss. This improves the performance of applications that have good spatial locality. On the other hand, it increases the remote memory access time since the number of flits per message increases. It also reduces the cache hit rate due to false sharing of data. The number of blocks per cache is reduced, which causes more blocks to be replaced. The results reported so far used a block size of 32bytes and a cache size of 8Kbytes per thread.

The effect of the block size on the cache-hit rate for one, two, and four threads is shown in Figure 6.10, Figure 6.11 and Figure 6.12, respectively. Overall, the cache hit rates improved by increasing the cache block size. The same was true with multithreading, except for MP3D, where the cache hit rate decreased for 4 threads and 128-byte blocks size. This can be due to the effect of the false data sharing. This can also be observed from the decrease in the attraction memory hit rate for this application with the increase in the number of threads.

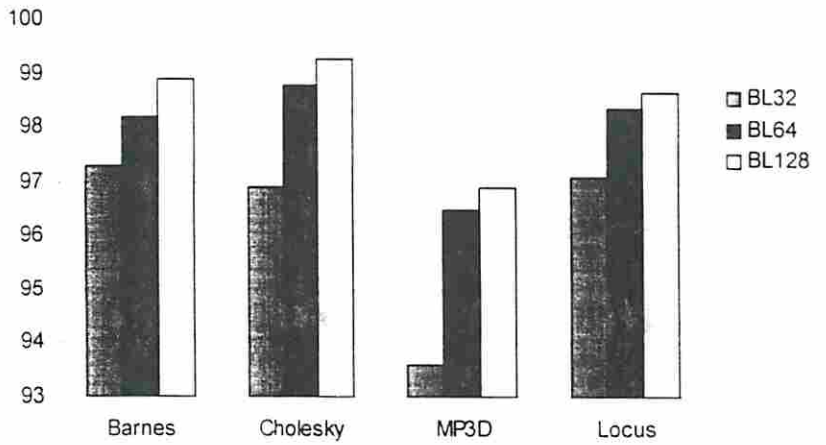


Figure 6. 10 Cache Hit Rates for one Thread and Various Block Sizes

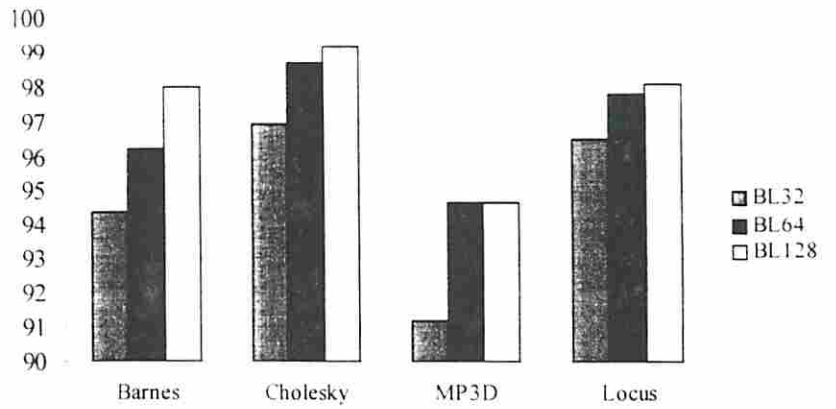


Figure 6. 11 Cache Hit Rates for Two Threads and Various Block Sizes

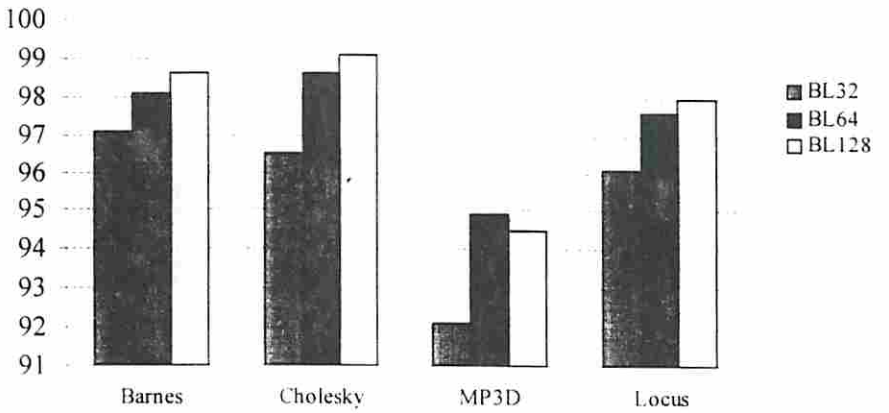


Figure 6.12 Cache Hit Rates for Four Threads and Various Block Sizes

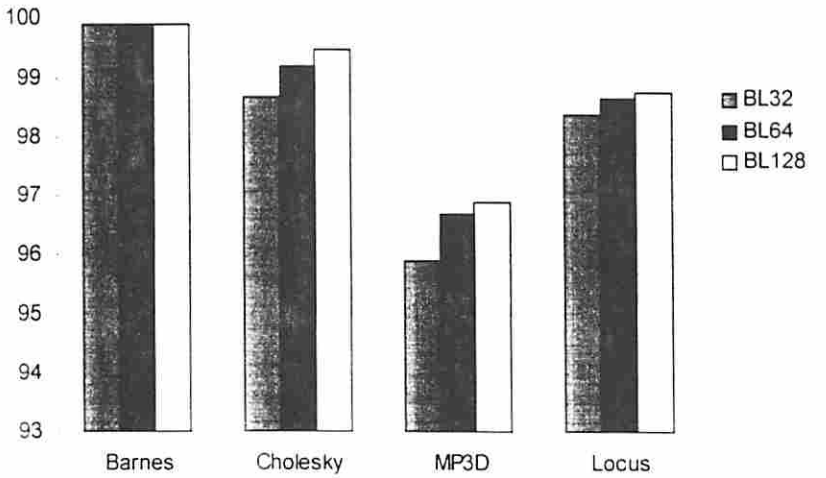


Figure 6.13 AM Hit Rates for one Thread and Various Block Sizes

Figure 6.13, Figure 6.14 and Figure 6.15 show the attraction memory hit rate for different block sizes and different number of threads. Checking the attraction memory hit rate for the different applications, the hit rate improved as the locality improved with the increase of block size. Figure 6.16, Figure 6.17 and Figure 6.18 show the group hit rate for different block sizes and different numbers of threads. The group hit rate is reduced when the block length is increased because the nodes have a smaller number of

lines per attraction memory which is mainly used by the local working sets. The attraction memory needs to replace most of the unused local data, hence, it tends to lose data that might be used by the neighboring nodes. The effect of the variable block length on the execution time was not very significant in our architecture since the improvement of the local node hit rate, in both the cache and the attraction memory, is affected by the decrease in the group hit rate. This causes longer remote access latencies.

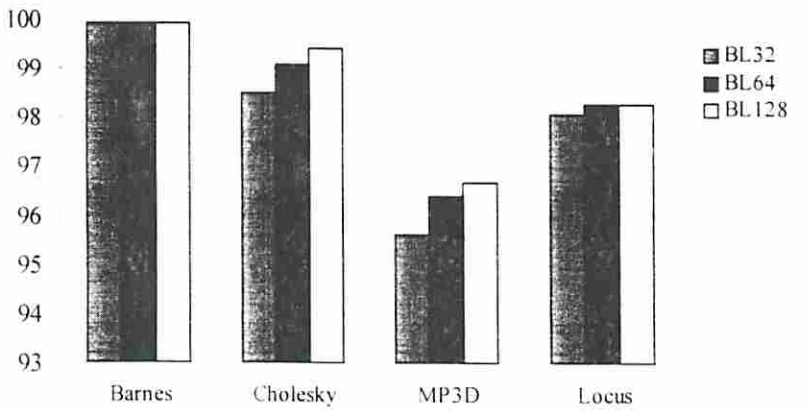


Figure 6.14 AM Hit Rates for Two Threads and Various Block Sizes

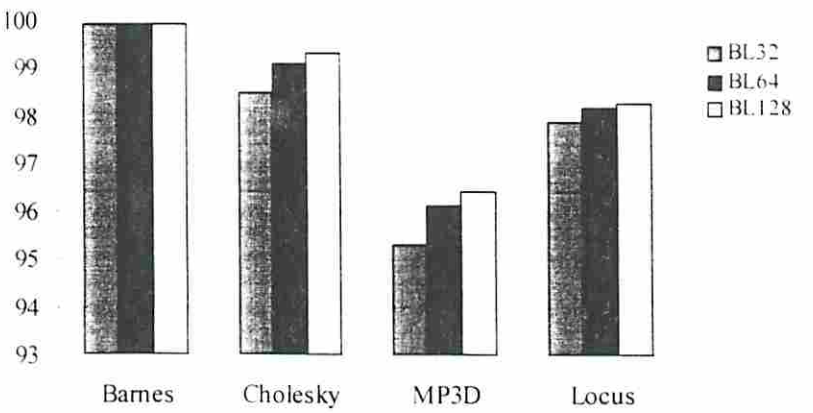


Figure 6.15 AM Hit Rates for Four Threads and Varios Block Sizes

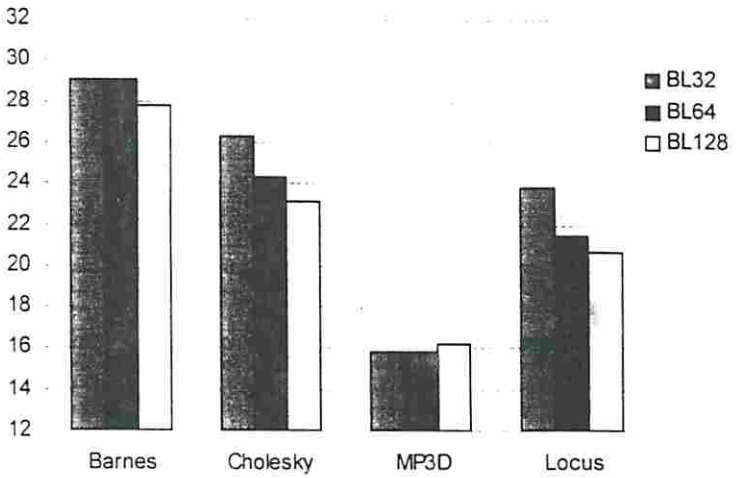


Figure 6.16 Group Hit Rates for one Thread and Various Block Sizes

6.3.5 Cache Size Variation

To improve the cache hit rate, capacity and coherence misses should be reduced. Since the coherence misses are mainly due to invalidation, it is dependent on the application and the scheduling of threads. To reduce the capacity misses, the cache size can be increased. This improves the performance of the applications that have this type of miss as the dominant miss.

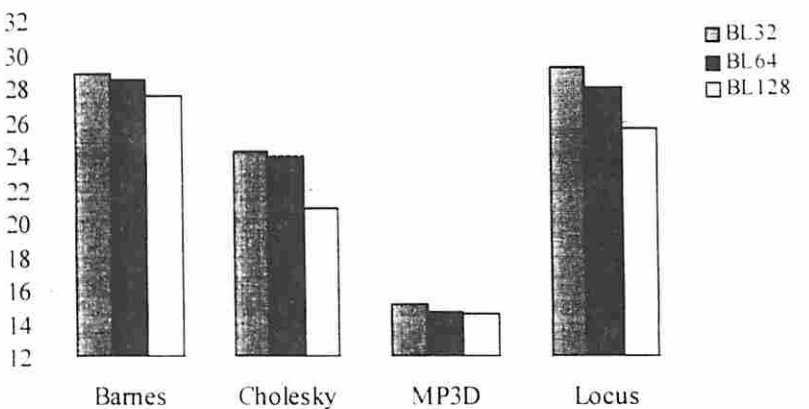


Figure 6.17 Group Hit Rates for Two Threads and Various Block Sizes

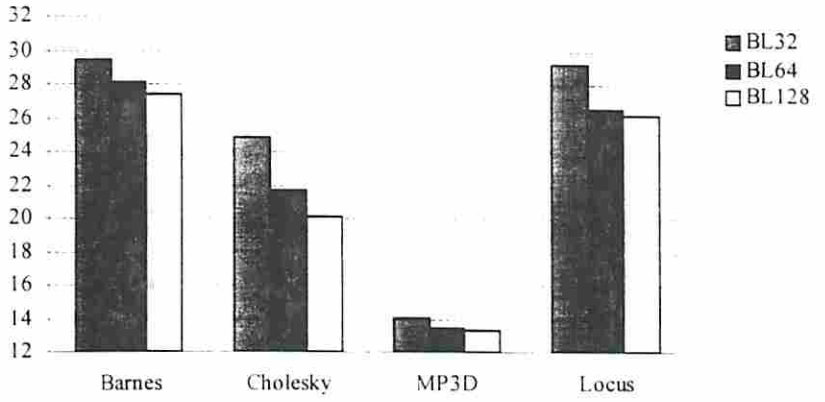


Figure 6.18 Group Hit Rates for Four Threads and Varios Block Sizes

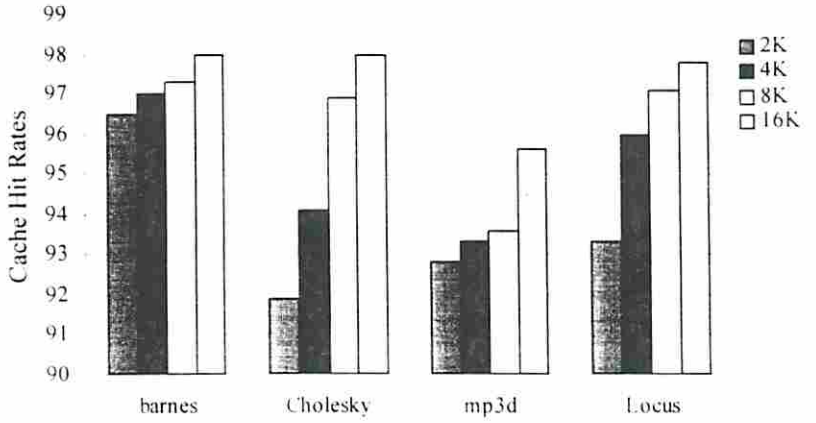


Figure 6.19 Cache Hit Rate For One Threads and Various Cache Sizes

Figure 6.19, Figure 6.20 and Figure 6.21 show the cache hit rate for different thread numbers and various cache sizes. It is obvious that some applications benefit more from the large cache size, especially those that have a high reuse of data such as LocusRoute and MP3d where capacity misses are the high. The attraction memory hit rate was not noticeably affected by the variation in cache size since the data in the attraction memory stayed the same unless invalidated or replaced. Therefore, the cache size variation did not affect the attraction memory. The application execution time was affected by the cache size change since more data accesses are satisfied from the cache. Figure 6.22, Figure 6.23 and

Figure 6.24 show the effect of different cache sizes on the applications execution time for different number of threads.

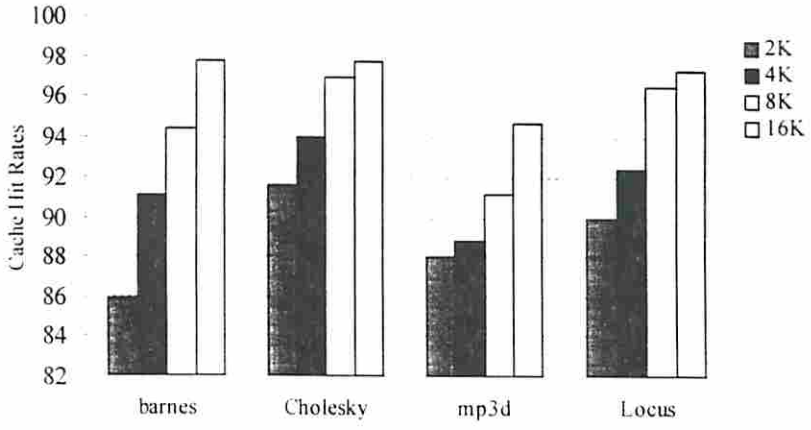


Figure 6.20 Cache Hit Rates For Two Threads and Various Cache Sizes

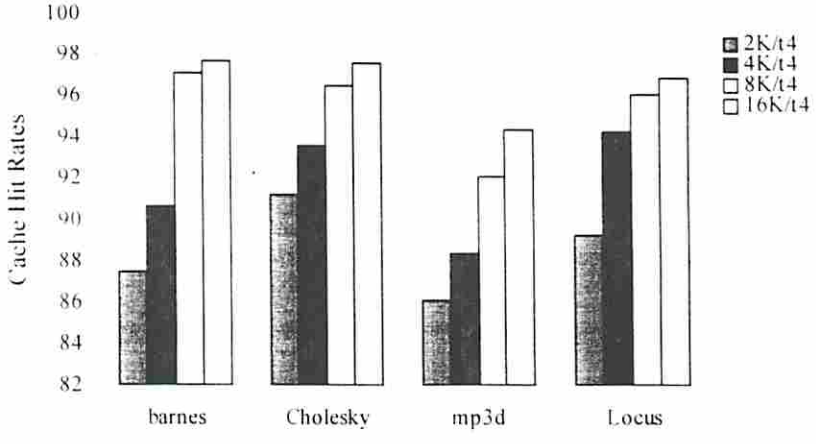


Figure 6.21 Cache Hit Rates For Four Threads and Various Cache Sizes

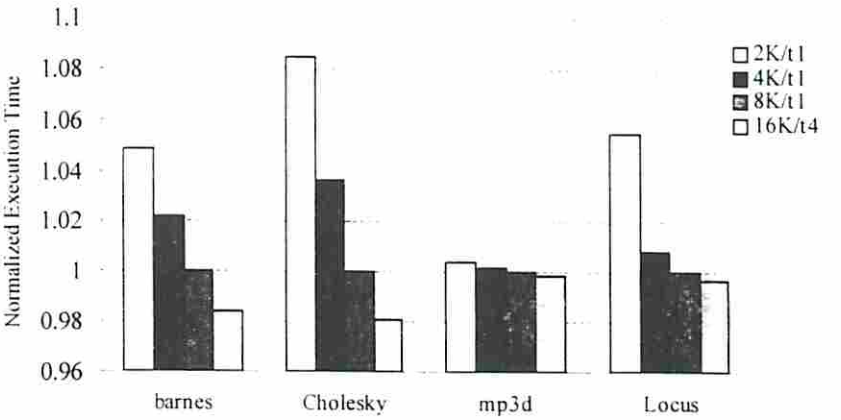


Figure 6.22 Normalized Execution Time for one Thread and Various Cache Sizes

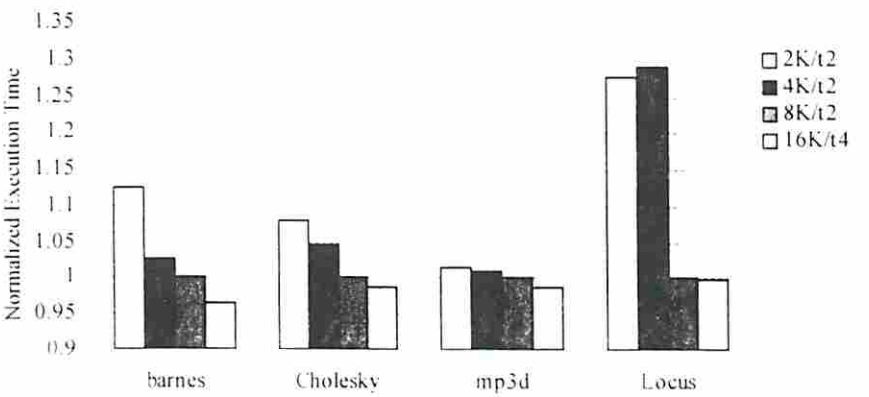


Figure 6.23 Normalized Execution Time for Two Threads and Verious Cache Sizes

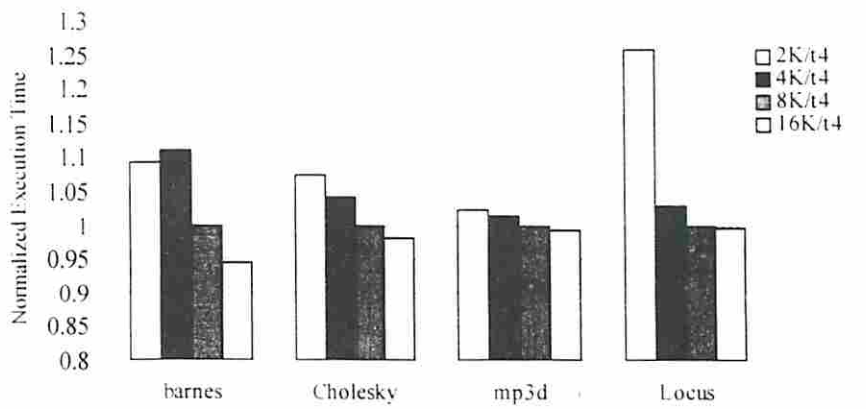


Figure 6.24 Normalized Execution Time for Four Threads and Various Cache Sizes

6.3.6 Processor Speed Effect

With the advancement in technology, processor speed is improving at a much faster rate than the improvement in both memory and the interconnection network. Processor speed is doubling every eighteen months (according to Moore's Law) and the memory access time of DRAM is advancing very slowly. This gap in performance between the processor and the rest of the architecture limits the benefits of the improvement in the processor area. As the processor speed increases, more instructions are executed, which increases the rate of data access requests, which increases the contention on the memory as well as the interconnection network. This effect will be more visible as multithreading is added to the system and more requests are sent to memory and over the network.

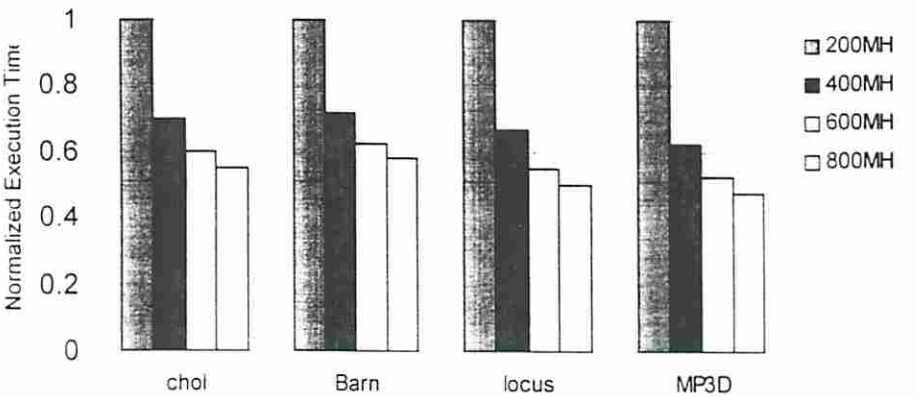


Figure 6.25 Normalized Execution Time for One Thread for Different Processor Speed

To study the effect of improving the processor speed on the overall system performance, the execution time of different application was measured assuming different processor clock speed with the memory access time and the interconnection time delay remaining the same. Figure 6.25 shows the execution time of different benchmark applications on MCOMA for 200, 400, 600 and 800 MHz processor clock rates. The execution time is normalized to the execution time of the 200 MHz processor clock rate. As the processor speed doubled from 200 to 400 MHz, a considerable improvement in the execution time

was noticeable for all applications. This rate of improvement did not hold steady as the processor speed improved. The limited performance was due to the effect of the memory and interconnection network delays.

Figure 6.26 and Figure 6.27 show the execution time for different applications for two and four threads respectively. The execution time is normalized to the execution time of the same number of threads for a processor speed 200. From these two figures, it is clear that the relative performance of MCOMA with multithreading would gain from the processor speed increase. This improvement is leveled due to the magnified effect of the interconnection as well as the increased data conflicts in the attraction memory, which further increase the contention on the search bus.

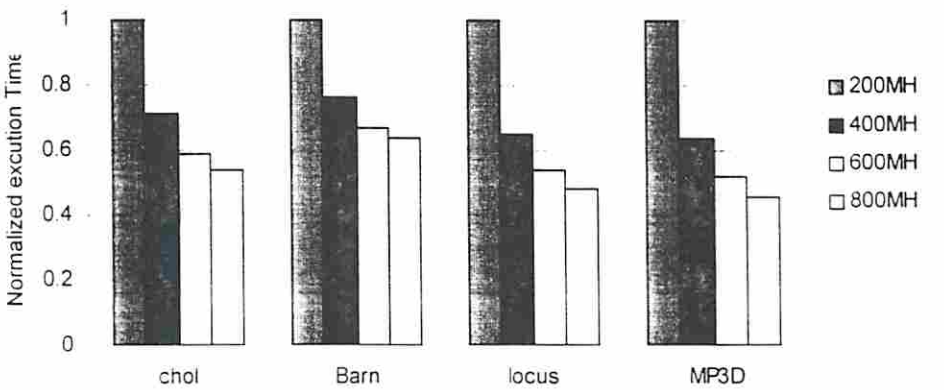


Figure 6.26 Normalized Excution Time for Two Threads for Different Processor Speed

6.3.7 Directory Bus Contention

The search interconnection in MCOMA deals with data request messages, which are short messages. The group directory satisfies part of the total data requests and only those requests that can not be

satisfied locally from the group are passed to the search interconnection. When data is to be replaced from one node, this cache protocol tries to place the data in one of the group node to minimize the search time and to reduce the traffic on the search bus. This also reduces the data transfer time between nodes and reduces the network and the search bus traffic.

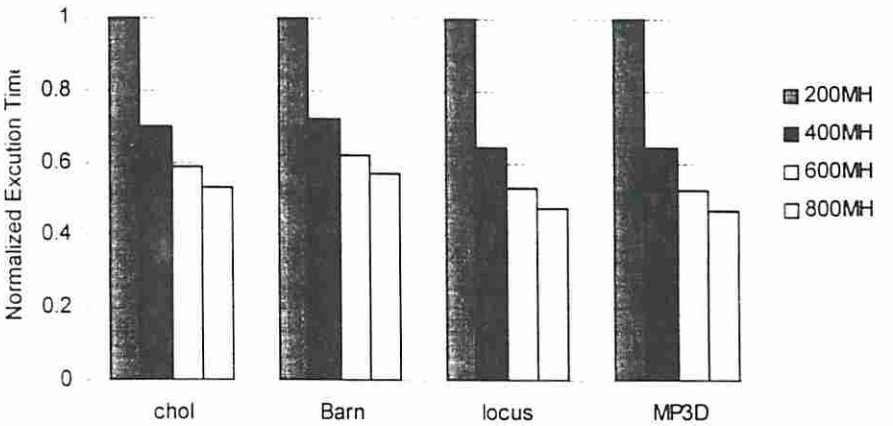


Figure 6.27 Normalized Excution Time for Four Threads for Different Processor Speed

For those applications that have, high node hit rates and a high group hit rate, bus traffic is kept very low. The bus utilization is defined as the percentage of the time that the bus was busy as a percentage of the total execution time. The bus utilization for different applications is shown in Figure 6.28. For Barnes-Hut, the application has the highest attraction memory hit rate and group hit rate. This kept the bus utilization to the minimum. MP3D has the worst data locality in all applications. This reflects on the bus utilization that reached around 30%. Adding multithreading increased the bus utilization for all applications except for MP3D, it slightly improves due to locating the data on neighboring nodes. For larger problem size, the bus utilization is expected to be higher and as discussed before, a wider and faster bus should be considered. Better data and thread allocation would also improve the bus performance. Using logically related threads on the same node and the neighboring nodes would improve the search bus utilization, which, in turn, improves the overall system performance. As

reported by other research groups [3,32], using a modified version of MP3D that alter the random scheduling of molecules on the processing nodes noticeably improved the application performance. This is expected to be true for the MCOMA where improving the group locality improves the system overall performance and reduces the traffic on the search bus.

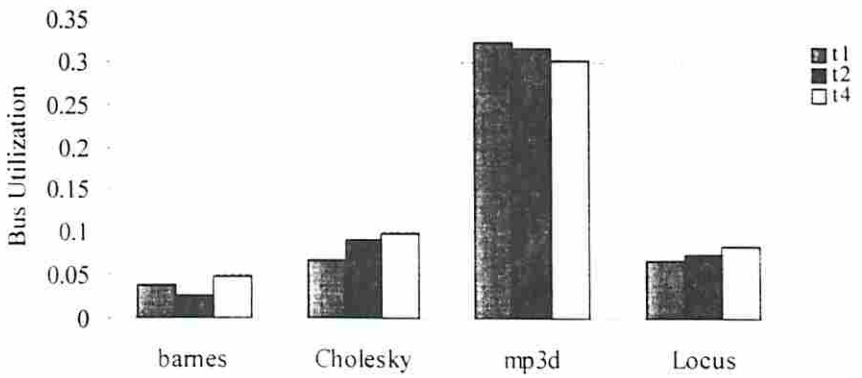


Figure 6.28 Bus Utilization for Various Threads

6.4 Summary

The simulation results of MCOMA have been very promising. The system achieved a good overall speedup on all four applications, especially the ones with good data locality. For MP3D, which has limited locality, the system showed an improvement in the speedup especially with the use of multithreading. The performance of the system has improved with the use of multithreading where all applications showed improvement in execution time. The performance of MCOMA is affected by many factors such as the thread cache size, the block size, and the interconnection network. The effect of all these has been evaluated. Increasing the cache size improved the system performance. The system captured the data locality of the applications. The directory interconnection affected the performance of MP3D only where the search bus contention approached 30%.

The results of single thread are consistent with those obtained by other research groups. The use of multithreading significantly improved the performance of MCOMA. The system speedup for MCOMA outperformed the reported speedup of other systems for the same applications under similar assumptions [3,33,46]. Results reported for MP3D use an unmodified version of the application (without code restructuring).

Both Alewife and the DDM reported results [3,35] using block multithreading as a latency tolerance mechanism. The DDM assumed the use of a multithreaded processor that switched on every memory access without performance penalty. Other reported results [7] used fixed network latency. These results also assumed an ideal machine that has zero latency and no contention on accesses to shared memory. However, the MCOMA simulator considered the shared memory latency as well as the interconnection latency including the search interconnection contention. All latencies assumed in the simulation reflect actual existing technology. Thread switching was supported in hardware and the context switching time was simulated. It was assumed that threads switch on remote access only. Switching threads on different bases could further improve the MCOMA performance.

Chapter 7

Conclusion and Future Direction

7.1 Conclusion

The main goal of this research was to develop a new multithreaded COMA architecture and evaluate the performance of this architecture and the effect of multithreading on improving the system overall performance. COMA attracts the local working data set to the local memory, which results in fewer remote memory accesses on a data miss, but COMA suffers from long data search time. By switching to another thread while one thread is waiting on a remote access, the performance of COMA was noticeably improved.

The new MCOMA architecture has been presented and simulated. Different variations of the main design were considered. The system overall performance was tested by a subset of the applications from the SPLASH benchmark suite and the results compared favorably to MCOMA. The model of MCOMA assumes a multithreaded processor that supports fast context switching. It has a separate cache per thread so threads do not lose locality while waiting on a remote reference. The architecture supports up to 4 threads at a time. In general, the architecture can be implemented for different interconnection networks. The simulated architecture assumes a mesh interconnection to connect the processing nodes. This interconnection carries request messages to directory nodes and data reply messages to the requesting node. The directory of a group of nodes has its dedicated fast interconnection that carries request messages between directories. In the simulated MCOMA, a wide bus was used. The results show that the architecture speedup was higher than those reported by other research groups under the

same assumptions. It also showed that the performance of all the applications improved by using multithreading on the MCOMA.

Using a separate cache per thread improved the multithreading performance. Since threads did not lose locality while waiting on a remote data access, the negative effect of multithreading on the cache was almost eliminated. The data conflict occurred mainly in the attraction memory that was reflected, with a small degree, on the cache-hit rate. Since any displacement of data from the attraction memory was moved to the closest node with available space in its attraction memory, the data can be reused with smaller delay penalty, if needed. Varying the cache size showed some effect on the applications execution time. Execution time of all applications increases with the cache size reduction.

7.2 Future Directions

A multithreaded COMA architecture offers a great potential for performance improvement over other existing shared memory multiprocessor systems. Since multithreading can hide the COMA long latency and COMA can improve thread localities, they both allow limited multithreading to be more effective. As parallel processors grow in size, interconnection network latencies grow longer. Moving the working set for a processor to the local or the surrounding nodes will improve overall system performances.

This dissertation research leads to many other areas of future work, which include the area of hardware and software development for MCOMA to study and enhance the system performance.

7.2.1 Architecture Variations

In this research, one variation of the MCOMA architecture has been simulated to evaluate the expected performance of MCOMA. Many hardware areas need to be further investigated.

Different Architecture Models:

Looking at the general architecture for the MCOMA, many other architectures that use different efficient interconnections can be developed. Different system configuration would yield a completely different multithreaded COMA architectures depending on the type of interconnection used and the way the processing nodes are grouped. Some of these configurations would be interesting to investigate. The results presented in this research showed that the search interconnection has a considerable effect on the overall system performance. The design of an efficient system is an overall design problem, which the interconnection networks, is an important part of it. In current MCOMA simulation, the contention of the interconnection network was not simulated. The effect of this contention becomes more important as the machine size increases and as the processor speed doubles in the near future. Efficient search interconnection also has the potential of improving the system performance. As processor speed improves with new technology, the interconnection latency will be the dominating factor in determining the performance of large machines. Faster processors tend to generate more data requests. Unless those data requests are satisfied locally or from the neighboring nodes, the interconnection will suffer from high contention that will deteriorate the system performance. More aggressive search interconnection should be considered. Optical interconnection can be a good candidate for the search interconnection.

Variation of Multithreading:

The presented system supports block multithreading. The processor was modified to have support for up to four threads. The number of threads supported per processor is limited by the chip area since it involves the duplication of the thread cache and the set of registers used per thread. In the simulated system, the performance of the used applications continued to improve by increasing the number of threads. As the number of threads needed to hide remote access latencies increases, a careful tradeoff between adding extra hardware or the use of thread scheduler; such as the one presented in chapter 4;

should be considered. Different thread scheduling techniques will affect the system performance since the number of threads sharing the attraction memory will increase which will increase the conflict misses in the attraction memory. If more than one thread are allowed to share a cache section, this will again increase the conflict misses in the cache and threads will start losing locality while waiting on events. A more careful study to reach the optimum number of threads per processor and the best scheduling technique will improve the system performance.

Other multithreading techniques can also be investigated on the MCOMA, such as the conditions on which threads are switched and the consideration of having the processor to switch after every cycle. Each of these modification would require the pipeline to be modified to allow instructions to carry its register set identification number down the pipeline as proposed in [7]. Threads can then be switched on cache miss not only on attraction memory miss since the switching penalty would be minimized. This modification will further improve the processor utilization. Another interesting variation of MCOMA that would have a good potential for improvement is the use of simultaneous multithreaded processors, which would allow multiple threads to run at the same time. Simultaneous multithreading has proven to be successful in other architectures [1,3], combining it with the data migration technique of COMA will be interesting to investigate.

7.2.2 Software Development

An important area for future work is the software development for the MCOMA. One of the main areas that should be addressed is the compiler support for this system. From the simulation results, it was clear that MCOMA could achieve a good performance with the use of existing applications without major modifications. All the programs used were parallel programs for shared memory multiprocessors, which adapted to MCOMA architecture without any change. This performance can be greatly improved by exploiting data localities in the applications and improve thread scheduling. In the current

simulation, the allocation of threads was done at random. Allocating related threads to the same node and neighboring nodes will improve the locality of each node and the group locality. Improving the thread allocation on the proposed architecture should be considered.

Many research groups have already developed compiler techniques that can benefit the MCOMA. The SUIF compiler at Stanford University, tried to reduce the cache misses by making the data accessed by each processor contiguous in the shared memory space. It also ensured that processors re-use the same data as much as possible [18]. These techniques, if applied to MCOMA can greatly improve the system performance since remote access latencies in COMA is the major limiting factor on the overall system performance. MCOMA would benefit from having the data used by processors available locally, as much as possible, or on the neighboring group nodes.

It is also useful to include other promising technology developed for existing multithreading systems such as Tera to further develop MCOMA [1,2,3]. Many recent researches have developed techniques to expose fine-grain parallelism and enhance locality. Partitioning the application programs at compile time would improve program localities and reduce the communication overhead. Multithreading is a valuable means to hide these latencies but more should be done to fully exploit the parallelism in the applications and use it to improve the multithreading performance.

Bibliography

[1] G. Alverson, P. Briggs, S. Coatney, S. Kahan and R. Korry. *Tera Hardware-Software Cooperation*. Proceeding of Supercomputing, 1997, San Jose, California.

[2] G. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porerfield and B. Smith. *The Tera Computer System*. International Conference on Supercomputing, 1990, p.p. 1-6.

[3] Anant Agarwal et al. *The MIT Alewife Machine: Architecture and Performance*. In Proceeding of the 22nd International Symposium on Computer Architectures, 1995, p.p. 2-13.

[4] Anant Agarwal. *Performance Tradeoffs in Multithreaded Processors*. IEEE Transaction on Parallel and Distributed Systems. Vol. 3, No. 5. September 1992.

[5] A. Agrawal, R. Sioni, J. Hennessy and M. Horowitz. *An Evaluation of Directory Schemes for Cache Coherence*. In the Proceeding of the 15th International Symposium on Computer Architectures, May 1988, p.p104-114.

[6] A. Agrawal, B-H. Lim, D. Kranz and Kubiatowiz. *APRIL: Processor Architecture for Multiprocessing*. In the Proceeding of the 17th International Symposium on Computer Architectures, May 1990, p.p104-114.

[7] Bob Boothe and Abhiram Ranade. *Improved Multithreading Techniques for Hiding Communication Latency in Multiprocessors*. In Proceeding of the 18th International Symposium on Computer Architectures, 1992, p.p. 214-223.

[8] Henry Burkhardt et al. *Overview of the KSR 1 Computer System*. Technical Report KSR-TR-9202001, Kendall Square Research, February 1992.

[9] J. Carter, R. Kurasmkote, C. Kuo. Reducing *Consistency Traffic and Cache Misses in the Avalanche Multiprocessor*. Technical Report UUCS-95-91-023, University of Utah, 1995.

[10] D. Chaiken, C. Fields, K. Kurihara and A. Agrawal. *Directory-Based Cache Coherence in Large-Scale Multiprocessors*. Computer, Vol. 23, No. 6, June 1990, P.p.49-58.

[11] D. Chaiken, J. Kubiatowicz and A. Agrawal. *LimitLESS Directories: A Scaleable Cache Coherence Scheme*. In the proceeding of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems, April 1991, P.p. 224-234.

[12] F. Chong, B. Lim, R. Bianchini and J. Kubiatowicz. *Application Performance on the MIT Alewife Machine*. IEEE Computer, December 1996, P.p. 57-64.

[13] M. Dubois and C. Scheurich. *Memory Access Dependencies in Shared Memory Multiprocessors*. IEEE Transaction. On Software Engineering, June 1990.

[14] Thorsten Von Eicken, David E. Culler, Seth Copen Golgstein and Klaus Erik Schauser. *Active Messages: A Mechanism for Integrated Communication and Computation*. In Proceeding of the 19th International Symposium on Computer Architectures, 1992.

[15] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta and J. Hennessy. *Memory Consistency and Event Ordering in Scaleable Shared-Memory Multiprocessors.* In Proceeding of the 17th International Symposium on Computer Architectures, 1990.

[16] A. Gupta, W. D. Weber. *Cache Invalidation Patterns in Shared-Memory Multiprocessors.* IEEE Transaction on Computers, July 1992.

[17] E. Hagersten, A. Landin and S. Haridi. *DDM- A Cache-Only Memory Architecture.* Computer, Sept. 1992, p.p. 44-54.

[18] M.W. Hall, J.M. Anderson, S.P. Amarasinghe, B.R. Murphy, S. Liao, E. Bugnion, M. Lam. *Maximizing Multiprocessor Performance with the SUIF Compiler.* IEEE Computer, December 1996 (special issue on shared-memory multiprocessors), P.p. 84-89.

[19] C. Holt, M. Heinrich. J. Singh, E. Rothberg, and J. Hennesy. *The Effects of Latency, Occupancy, and Bandwidth in Distributed Shared Memory Multiprocessors.* Stanford University Technical Report No. CSL-TR-95-660. January 1995.

[20] H. Hum, O. Maquelin, K. Theobald, X. Tian, G. R. Gao, and L. Hendren. *A study of the EARTH-MANNA Multithreaded System.* International Journal on Parallel Programming, 24(4): 319-347, August 1996.

[21] R. Iannucci, G. R. Gao, R. H. Halstead, Jr., B. Smith. *Multithreaded Computer Architecture: A Summary of the State of The Art.* Kluwer Academic Publishers, 1994.

[22] Raj Jain. *The Art of Computer Systems Performance Analysis-Techniques for Experimental Design, Measurements, Simulation and Modeling.* John Wiley & Son, Inc., 1991.

[23] Truman Joe. *COMA-F: A Non-Hierarchical Cache Only Memory Architecture.* Ph.D. Thesis, Stanford University, EE Dept. March 1995.

[24] Truman Joe, John Hennessy. *Evaluating the Memory Overhead Required for COMA Architectures.* Proceeding of the 21st Annual International Symposium on Computer Architecture, April 1994.

[25] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins & R. G. Sheldon. *Implementing A Cache Consistency Protocol.* In the Proceeding of International Conference on Parallel Processing, 1985.

[26] J. Kong, G. lee. *Relaxing the Inclusion Property in Cache Only Memory Architecture.* In the Proceeding of Euro-Par'96.

[27] J. Kusin and D. Ofelt et al. *The Stanford FLASH Multiprocessor.* In Proceeding of the 21st International Symposium on Computer Architectures, 1994, p.p. 302-313.

[28] J. Kubiatowicz, D. Chaiken, and A. Agarwal. *Closing the Window of Vulnerability in Multiphase Memory Transactions.* In the 5th International Conference on Architectural Support for Programming Languages and Operating Systems, 1992, p.p. 274-284.

[29] L. Lampart. *How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Program.* IEEE Transaction on Computers, Sept. 1979.

[30] G. Lee, B. Quattlebaum, S. Cho, and L. Kinney. *Global bus Design of a Bus-Based COMA Multiprocessor DICE*. In Proceedings of International Conference on Computer Design, 1996.

[31] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta and J. Hennessy. *The Directory-Based Cache Coherence Protocol for the Dash Multiprocessor*. In the Proceeding of the 17th International Symposium on Computer Architectures, May 1990.

[32] D. Lenoski, J. Laudon, K. Gharachorloo, W. D. Weber, A. Gupta, J. Hennessy. M. Horowitz, and M. Lam. *The Stanford DASH Multiprocessor*. IEEE Computer, 25(3): 63-79, March 1992.

[33] H. Muller, P. W. Stallard and D. H. Warren. *Hiding Miss Latencies with Multithreading on the Data Diffusion Machine*. In the proceeding of International Conference on Parallel Processing, 1995. Pp. I-175-185.

[34] H. Muller, P. W. Stallard and D. H. Warren. *Implementing the Data Diffusion Machine Using Crossbar Routers*. IPPS 1996. Pp. 152-158.

[35] H. Muller, P. W. Stallard and D. H. Warren. *The Data Diffusion Machine with Scalable Point-to-Point Network*. University of Bristol Technical Report CSTR-93-17, October 1993.

[36] *MIPS R4000 User's Manual*. MIPS Computer Systems, Sunnyvale, California, 1991.

[37] R. Nikhil, G. Papadopoulos and Arvind. **T: A Multithreaded Massively Parallel Architecture*. In Proceeding of the 19th International Symposium on Computer Architectures, 1992, p.p. 156-167.

[38] R. Saavedra, D. E. Culler and T. Von Eicken. *Analysis of Multithreaded Architectures for Parallel Computing*. 2nd Annual ACM Symposium on Parallel Algorithms and Architectures, 1990, p.p. 169-178.

[39] R. Saavedra, R. Gaines and M. Carlton. *Micro Benchmark Analysis of the KSR1*. Supercomputing 1993, p.p. 202-212.

[40] A. Saulsbury, T. Wilkinson, J. Carter, A. Landin. *An Argument for Simple COMA*. First IEEE Symposium on High Performance Computer Architecture, January 1995, p.p. 276-285.

[41] J. Singh, T. Joe, A. Goupta and J. Hennesy. *An Empirical Comparison of the Kendall Square research KSR-1 and Stanford DASH Multiprocessors*. In the Proceeding of Supercomputing 1993, p.p. 214-223.

[42] J. Singh, W. D. Weber, and A. Gupta. *SPLASH: Stanford Parallel Applications for Shared Memory*. Technical Report CSL-TR-91-469, Stanford University, April 1991.

[43] Klaus Erik Schauser, David E. Culler and Thorsten Von Eicken. *Compiler-Controlled Multithreading for Lenient Languages*. In the Proceeding of FPCA' 91.

[44] P. Stenstrom. *A survey of Cache Coherence Schemes for Multiprocessors*. Computer, Vol. 23, No. 6, June 1990, P.p.12-24.

[45] P. Stenstrom, T. Joe and A. Gupta. *Comparative Performance Evaluation of Cache-Coherent NUMA and COMA Architectures*. In the Proceeding of the 19th International Symposium on Computer Architectures, 1992, p.p. 80-91.

[46] R. Thekkath. *Design and Performance of Multithreaded Architecture.* Ph.D. Thesis, University of Washington, 1995.

[47] R. Thekkath, S.J. Eggers. *Impact of Sharing-Based Thread Placement on Multithreaded Architectures.* In the Proceeding of the 21st International Symposium on Computer Architectures, 1994.

[48] J. E. Veenstra and R.J. Fowler. *MINT: A Front End Efficient Simulation of Shared Memory Multiprocessor.* In MASCOTS 1994, January 1994.

[49] A. W. Wilson Jr. *Hierarchical Cache Bus Architecture for Shared Memory Multiprocessors.* In the Proceeding of the 14th International Symposium on Computer Architectures, 1987.

[50] X. Zhang and Y. Yan. *Latency analysis of CC-NUMA and CC-COMA Rings.* In the Proceeding of International Conference on Parallel Processing, 1994. P.p. I 174-181.