A CONFLICT-FREE MEMORY DESIGN FOR MULTIPROCESSORS

By

Honda Shing

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Department of Computer Science

1992

t95-8722

ABSTRACT

A CONFLICT-FREE MEMORY DESIGN FOR MULTIPROCESSORS

By

Honda Shing

Multiprocessors have been widely used in achieving high performance computation. In a multiprocessor, applications are implemented with processes executing on multiple processors and cooperating with each other. It is hoped that, by using multiple processors, multiplied performance improvement can be obtained. Ideal speedup, however, can hardly be achieved due to several factors. In a shared-memory multiprocessor, memory conflicts and interconnection network contention are two of the most important performance degradation factors.

This dissertation proposes a conflict-free memory architecture, a new architecture based on block accesses, which solves the memory and network contention problems in shared-memory multiprocessors. With a synchronously coordinated memory access pattern and a contention-free interconnection network, memory conflicts and network contention can be eliminated. In addition, the tree saturation problem that results from the hot spot problem is solved. Possible extensions to the architecture and performance issues are also discussed in this dissertation.

An invalidation-based write-back cache coherence protocol is introduced for the conflict-free memory architecture. The new cache protocol preserves the advantages of both conventional snoopy protocols and directory-based protocols. With this cache coherence protocol, efficient synchronization operations can be implemented to support the weak memory consistency model. Furthermore, high level process synchronization mechanisms can be implemented with low overhead and low latency. The scalability of both the conflict-free memory architecture and the cache coherence protocol are demonstrated.

This dissertation also introduces the new parallel programming paradigm, *resource binding*, which can be efficiently implemented on the conflict-free memory architecture. With simple primitives, the programming paradigm handles shared data protection and process synchronization in a flexible and consistent way. In addition, the resource binding paradigm can be implemented in other shared-memory and distributed-memory systems, which allows programmers to write portable parallel programs that are easy to understand and to debug.

To my parents

ACKNOWLEDGEMENTS

I wish to thank my advisor, Dr. Lionel M. Ni, who has guided me throughout the academic and research years of my Ph.D. program. Without his patient instruction and imaginative enlightening, this dissertation would have been impossible.

I would also like to thank Dr. Anthony S. Wojcik for his valuable suggestions and comments on this dissertation and careful reviewing of the manuscript. I appreciate Dr. Richard Enbody, who has always been willing to answer my questions and to spend time on discussions. I am grateful to Dr. R. V. Ramamoorthy for his encouragement and support.

A person cannot accomplish anything without the help and encouragement of others. I would like to thank all the people who helped me during my stay at Michigan State University. In particular, I wish to acknowledge my friends Ten Hwan Tzen, Arun Nanda, and brothers and sisters in the Michigan State University Chinese Christian Fellowship.

Finally, I thank my parents for their continuous support, patience, and love.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Computer performance has dramatically progressed over the past decades, yet computer scientists have never stopped seeking new technologies for more performance improvement. Rapid IC technology improvements have increased both the number of components per chip and the circuit speed [1], which in turn have had a great impact on computer architecture designs. New technologies have not only enabled larger main memory sizes, but also allowed more powerful processors to be built. Processors have been designed with faster clock rates, more functional units, and higher degrees of pipelining. Furthermore, in order to compensate for the growing gap between processor speed and memory speed, caches have been widely used in improving memory access performance. Powerful processors with high bandwidth memory systems have become a common way of achieving high performance computation.

Besides powerful processors, multiprocessing is another approach to achieving high performance. Multiprocessors have been widely used in application areas which have high demand for computation performance. In a multiprocessor, applications are implemented with processes executing on multiple processors and cooperating with each other. It is hoped that, by using multiple processors, multiplied performance improvement can be obtained; however, ideal (or linear) speedup can hardly be achieved due to a number of performance degradation problems. The purpose of

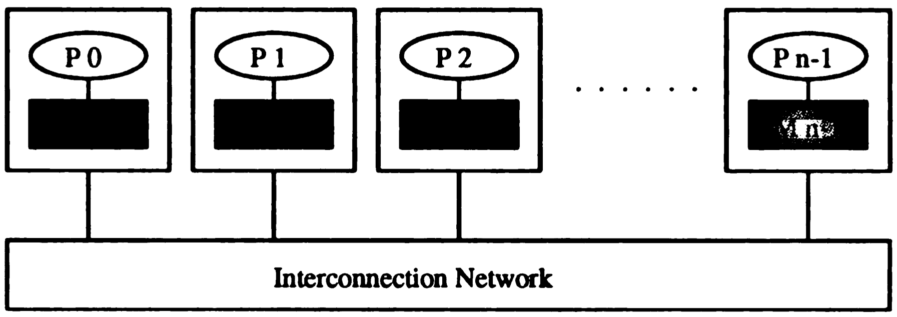


Figure 1.1. A typical distributed-memory multiprocessor.

this dissertation research is to improve multiprocessor performance by solving some of the most important performance degradation problems. This chapter first defines the basic categories of multiprocessors, namely, distributed-memory multiprocessors and shared-memory multiprocessors; states some performance degradation problems observed in these multiprocessors, and then gives an overview of the dissertation.

1.1 Distributed-Memory Multiprocessors

Multiprocessor architectures can basically be divided into two categories, namely, distributed-memory systems and shared-memory systems. On a distributed-memory multiprocessor, the memory system is distributed among processors. Since there is no common memory directly accessible by more than one processor, processors communicate with each other through message-passing. Figure 1.1 presents a typical distributed-memory multiprocessor. As can be seen, each processor has a local memory module physically associated with it. Messages between processors are transferred through the interconnection network. The Intel iPSC [2] and the nCUBE [3] are two well-known examples of distributed-memory multiprocessors.

Distributed-memory multiprocessors have the advantage of scalability. Large scale systems can be built by replicating processor nodes, including their local memories, and increasing the interconnection network bandwidth. The performance of a distributed-memory multiprocessor, however, is dependent on the interconnection network latency. Modern distributed-memory multiprocessors employ network protocols, such as *wormhole routing*, which allow the network latency to be independent of the distance between two processor nodes. This enables large scale distributed-memory systems to be constructed without imposing too much penalty on network latencies. Performance degradation, however, can still result from contention in the interconnection network.

The major disadvantage of distributed-memory multiprocessors is in its programming difficulty. In a distributed-memory multiprocessor, messages must be passed back and forth among large numbers of processors. This increases programming complexity and severely limits efficient use of processor resources. In order to solve the problem, some approaches support shared-memory environments on top of physically distributed memory architectures. In such a system, shared memory accesses are handled by underlying network controllers and network switches through message-passing. Examples include the BBN Butterfly [4, 5], the NYU Ultracomputer [6], and the IBM RP3 [7]. These architectures are referred to as "logically shared-memory multiprocessors", and together with "physically shared-memory multiprocessors", which are to be discussed in the following section, are the focus of this dissertation.

1.2 Shared-Memory Multiprocessors

In a shared-memory multiprocessor, processes interact with each other through shared physical address space. Figure 1.2 shows a typical shared-memory multiprocessor, where a number of memory modules are equally accessible from all processors. The processors access the shared memory modules through an interconnection network. Accessing shared memory can cause contention, both in memory modules and in the

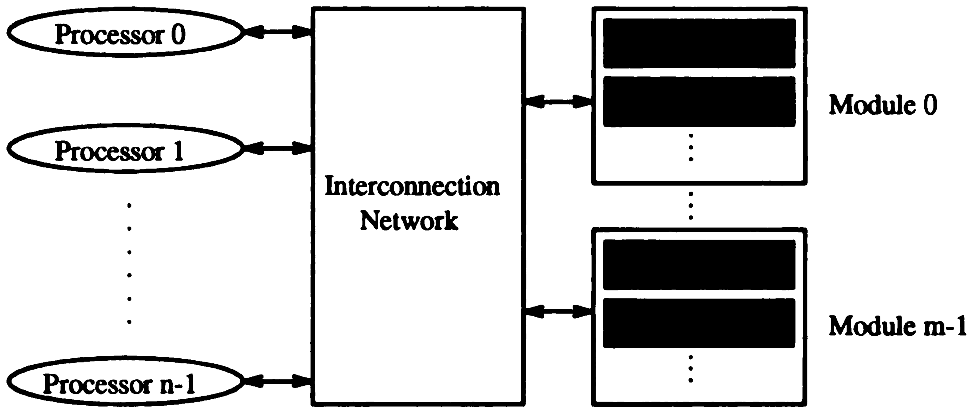


Figure 1.2. A typical shared-memory multiprocessor.

interconnection network, and decrease system performance. Additional overhead can also be introduced by network latency. Because of the comparatively long memory access latency in a multiprocessor, memory accesses are usually in blocks. Each memory block contains a number of memory words. In order to efficiently support block accesses, a memory module is usually composed of several memory banks. A memory word in this dissertation is defined as the data unit retrieved from or stored in a memory bank within one memory access, which can be one byte, two bytes, or other sizes.

In order to hide the long memory latency, there is usually a local cache installed with each processor. When accessing shared memory, processors usually talk to caches only, instead of directly accessing main memory modules. Blocks of data or instructions are transferred between caches and memory modules. In order to maintain cache coherence among processors, however, there may be some exceptions. While some multiprocessor designs force shared data to be non-cacheable and transferred in words instead of blocks, others employ the *write-through* protocol for cache coherence. Both approaches degrade system performance because of increased memory accesses and contention. Most recent approaches make everything cacheable and use the *write-back* cache protocol. In such systems, memory accesses are always in blocks.

In order to distribute workload and reduce memory conflicts, addresses may be arranged in an interleaved fashion. For a block access, however, if the address assignment is interleaved across memory modules, a processor needs to access multiple modules. Accesses to the memory modules have to go through different paths, which further increase the network traffic and latency. In practical multiprocessor designs, the address assignment within each memory module is sequential; while, in order to increase performance, each module has multiple banks organized in an interleaved fashion. Either way, however, there is still network and memory contention.

As mentioned, memory access contention and interconnection network latency are two factors that degrade shared-memory multiprocessor performance. In addition to those, synchronization among concurrent processes can further decrease the performance. Consider the commonly used synchronization mechanism, lock/unlock. Multiple processors may concurrently and repeatedly access the same lock variable and create intensive memory and interconnection network contention. This results in the "hot spot" problem described in [8]. The contention problem and the hot spot problem will be discussed further in Section 2.1.

This dissertation focuses on the design, implementation, and other issues related to the physically shared-memory multiprocessors described in this section as well as the logically shared-memory multiprocessors mentioned in the previous section. From this point, the term "shared-memory multiprocessors" will be used to denote both logically and physically shared-memory multiprocessors.

1.3 Dissertation Outline

This dissertation introduces the *Conflict-Free Memory* (CFM) architecture for multiprocessors [9]. Most memory system designs are based on word accesses. In contrast, the CFM architecture is based on block accesses. For multiprocessors within a certain scale range, the architecture eliminates memory and network contention as well as the hot spot problem without increasing latency and overhead. For large scale systems, the architecture improves system performance by reducing memory and network contention and increasing effective memory bandwidth and resource utilization. Moreover, it reduces the setup time and propagation delay caused by message routing in multiprocessors using MINs, such as the BBN Butterfly, the NYU Ultracomputer, and the IBM RP3. With an invalidation-based write-back cache protocol, the CFM architecture also efficiently supports high level process synchronization.

Chapter 2 addresses shared-memory design considerations for multiprocessors. It describes the contention problem found in shared-memory multiprocessors. A number of well-known multiprocessors are illustrated to demonstrate and evaluate several approaches to reducing or tolerating the contention problem. In addition, this chapter describes various memory consistency models that can be applied in multiprocessor designs.

In Chapter 3, the basic concept of the CFM architecture is introduced and a contention-free interconnection network that can be used in implementing the CFM architecture is presented. This chapter discusses various CFM configurations with different system parameters and their tradeoffs. In addition, it introduces a partially conflict-free extension to the basic CFM architecture for implementing larger scale multiprocessors. This chapter briefly analyzes the memory access efficiencies of partially conflict-free systems. A comparison between the efficiencies of conventional memory systems and partially conflict-free systems under different access rates and data localities is made. This chapter also discusses other performance issues, such as interconnection network overhead and memory access latency.

Chapter 4 presents a data inconsistency problem in the CFM architecture. It introduces an address tracking mechanism, which maintains data consistency among concurrent accesses to the same memory block. The mechanism is also used to implement atomic operations, which in turn support higher level process synchronization.

Chapter 5, after reviewing some existing cache protocols, introduces the CFM cache coherence protocol. The new cache protocol is an invalidation-based write-back protocol. Based on the CFM cache protocol, synchronization operations can be implemented and can support ***weak consistency.*** Simple lock/unlock as well as atomic multiple locks can be implemented using the ***busy-waiting*** scheme with low overhead. Furthermore, the CFM cache protocol can be recursively scaled for large scale CFM extensions. This chapter demonstrates a hierarchical extension to both the CFM architecture and the cache coherence protocol.

Parallel programming is an important issue in achieving high performance computation in a parallel processing environment. Chapter 6 presents the portable parallel programming paradigm, ***resource binding*** [10], which can be efficiently implemented on the CFM architecture. The programming paradigm supports a flexible environment for parallel programming in various computation models. Moreover, the paradigm manages both data sharing and process synchronization in a consistent manner.

Chapter 7 is a summary of this dissertation. Major contributions of the research presented is highlighted. This chapter also lists interesting topics for future work.

CHAPTER 2

SHARED-MEMORY DESIGN CONSIDERATIONS

Resource contention and memory consistency models are two important issues in designing a shared-memory multiprocessor. The contention problem not only occurs in shared memory modules, but also in interconnection networks. Serious performance degradation can happen when the contention rate increases. In order to reduce or tolerate the effect of contention, various approaches have been applied by some multiprocessors and are reviewed in this chapter. Different memory consistency models explore parallelism from different perspectives. They can be implemented with various hardware complexity. This chapter also discusses several memory consistency models used in multiprocessor designs.

2.1 The Contention Problem and Solutions

Shared-memory multiprocessors suffer memory conflicts when several processors access the same shared-memory module simultaneously. In a shared-memory multiprocessor, contention also occurs in the interconnection network. Examples include multiprocessors with *Multistage Interconnection Networks* (MIN) [4, 5, 11, 6, 7]. Both

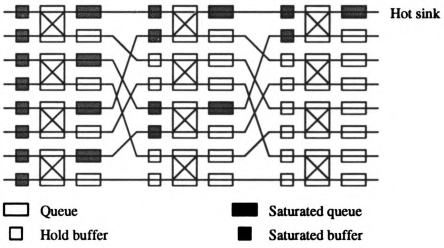


Figure 2.1. Tree saturation caused by a hot spot.

memory conflicts and interconnection network contention cause higher overhead and degrade system performance. Furthermore, the hot spot problem occurs when many processors try to access data in the same memory module or through the same network switch at a high rate [8]. This may be observed, for example, when several processors execute spin-lock on a single semaphore variable. In a MIN multiprocessor, the hot spot problem may result in the "tree saturation" effect, as shown in Figure 2.1. The highly contended hot sink can block its neighboring network switch buffers. The affected network switches can, in turn, block other network switches. Eventually, all memory accesses may incur considerable delays due to contention. This section presents several well-know multiprocessors and their approaches to solving the contention problem.

2.1.1 The NYU Ultracomputer and the IBM RP3

The prevention or reduction of memory and network contention becomes a very important issue in achieving high speedup in multiprocessors. Numerous approaches have been implemented or proposed in order to solve the contention problem. For example, the NYU Ultracomputer and the IBM RP3, MIN-based multiprocessors, employ a *combining* network to reduce memory and network contention [6, 7]. They implement atomic *fetch-and-add* operations that can be combined at any switch of the interconnection network. In case two fetch-and-add operations from two different processors accessing the same memory location arrive at a switch about the same time, they can be combined, and only one memory access need to be completed at the memory location. Combining, however, can be applied only among operations that access the same memory location. This restriction limits the usage of the combining technique, as there may be accesses to different locations in the same memory module or accesses to the same location but several clocks apart in their arrival times. In either case, network or memory contention remains.

2.1.2 The BBN Butterfly and Monarch

The Butterfly GP1000 and TC2000, built by BBN, are two other MIN-based multiprocessors [4, 5]. Both multiprocessors use circuit-switching in their interconnection networks. When a memory access request encounters memory or network contention, it is aborted and retried later rather than buffered at switches. This solves the tree saturation problem mentioned earlier. Since each memory access holds an entire path during its execution, the probability of network contention is increased. In order to reduce memory and network contention, the multiprocessors may provide alternative paths for memory accesses. Severe performance degradation due to contention, however, can still occur when shared memory access rate is high. Moreover, the circuit-switching network requires each conflicting memory access request to be retransmitted, which also increases the network overhead.

A variety of memory interleaving, skewing, and random mapping schemes have been developed to reduce contention in shared memory accesses. Some of these provide conflict-free access for a limited set of access patterns [12, 13], while others improve the average access performance [14, 15, 16]. The Monarch, a massively parallel processing computer being developed by BBN, is an example which applies random mapping on memory addresses to reduce memory and network contention [11]. The Monarch also employs read combining to reduce contention. In order to support the read combining, all memory accesses execute synchronously. Accessing requests are sent to memory banks in one cycle, and data returned in another. As a consequence, when a memory access is issued in a wrong cycle, a stall is required. The Monarch uses dual memory banks and double interconnection networks to increase memory bandwidth, which results in higher hardware cost.

2.1.3 The OMP multiprocessor

Another example of a shared-memory multiprocessor is OMP, a RISC-based multiprocessor using orthogonal-access memories and multiple spanning buses [17]. In an n-processor OMP, n^2 memory banks are arranged as an $n \times n$ 2-D mesh with the processors connected to the n diagonal locations. Memory access cycles are divided into two modes, row and column. All processors and memory banks are synchronized, so that they are all either in the row mode or in the column mode. The synchronous row and column access style prevents processors from interfering with each other, thus, no memory conflict may occur. The scheme, however, introduces long delays when a processor attempts a row or column access during a column or row mode, respectively. Furthermore, due to the long latency of an orthogonal memory access, each processor requires a local memory as well as a large register file and accesses the orthogonal memory in a DMA interrupt fashion. This causes the orthogonal memory to be more like a RAM-disk instead of a shared main memory. In order to support n processors, the multiprocessor requires n^2 memory banks, which is also considered to be too expensive.

2.1.4 The Cedar Project

Cedar is a research project on parallel processing at the University of Illinois. The project has great emphasis on parallel software development techniques and tools, and a goal of developing a multigigaflops machine supporting a wide application range [18]. Figure 2.2 shows the Cedar architecture. Processors are grouped into clusters. Each processor has its local memory and can access the local memories of other processors in the same cluster through a high-speed switching network. Furthermore, there are global memories shared by processors in all clusters. Each processor can directly access global memories for data that are not in local memory through the global memory network.

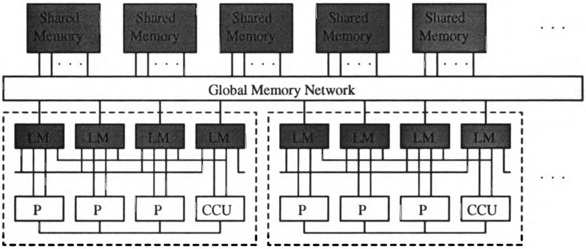


Figure 2.2. The Cedar architecture.

As can be seen, the Cedar architecture allows a memory to be local to each processor, some memories to be shared within each cluster, and other memories to be globally accessible. This reduces memory latency for accesses that can be served locally or within a cluster, which also eliminates the traffic that can otherwise be introduced to the global memory network. Conflicts in the global memory and contention in the global memory network can thus be reduced. The reduced memory latency and memory contention improve the overall performance, however, contention can still occur both within clusters and in the global memories and network.

2.2 Memory Consistency Models

In order to bridge the growing gap between processor cycle times and memory cycle times, caches have been widely used in current multiprocessors. In cache-based multiprocessors, a protocol is required to maintain data consistency among replicated copies of shared writable data. Furthermore, memory accesses may be buffered and pipelined to reduce the effect of long memory access latencies. Without proper programming practice, these architectural optimizations can cause memory accesses to be executed in an order different from what the programmer expects. The set of allowable memory access orderings forms the memory consistency model for an architecture [19]. This section describes several memory consistency models, which include *sequential consistency* [20], *processor consistency* [21], *weak consistency* [22], and *release consistency* [19]. Chapter 5 will further introduce a cache coherence protocol which implements the weak consistency model.

2.2.1 Sequential Consistency

Sequential consistency [20] requires the execution of a parallel program to appear as some interleaving of the execution of the parallel processes on a sequential machine. Before defining the sufficient conditions for sequential consistency, two definitions from Dubois *et al.* [22, 23] are presented below. In the following, P_i refers to processor i.

Definition 2.1 (Performing a Memory Request) *A* `load` *by* P_i *is considered performed with respect to* P_k *at a point in time when the issuing of a* `store` *to the* *same address by P_k cannot affect the value returned by the* `load`*. A* **`store`** *by P_i is considered performed with respect to P_k at a point in time when an issued* `load` *to the same address by P_k returns the value defined by this* **`store`** *(or a subsequent* **`store`** *to the same location). An access is performed when it is performed with respect to all processors.*

Definition 2.2 (Performing a `load` Globally) *A* `load` *is globally performed if it is performed and if the* **`store`** *that is the source of the returned value has been performed.*

In the following discussion, the phrase "previous accesses" is used to denote accesses in the program order that is before the current access. Condition 2.1 presents sufficient conditions of sequential consistency [19].

Condition 2.1 (Conditions for Sequential Consistency)

1. *before a* `load` *is allowed to perform with respect to any other processor, all previous* `load` *accesses must be globally performed and all previous* **`store`** *accesses must be performed, and*

2. *before a* **`store`** *is allowed to perform with respect to any other processor, all previous* `load` *accesses must be globally performed and all previous* **`store`** *accesses must be performed.*

Sequential consistency imposes stricter limitation than the other consistency models discussed later. This prohibits some hardware optimizations for performance improvements that are allowed by those models.

2.2.2 Processor Consistency

Processor consistency, introduced by Goodman [21], requires that **`store`**s issued from a processor must be observed in their issued order. It does not, however, require `stores` from two processors to be observed in the same order by themselves or other processors. Processor consistency relies on programmers to use explicit synchronization rather than the memory system to guarantee strict event ordering. Conditions to satisfy processor consistency is shown in Condition 2.2 [19].

Condition 2.2 (Conditions for Processor Consistency)

1. *before a* `load` *is allowed to perform with respect to any other processor, all previous* `load` *accesses must be performed, and*

2. *before a* `store` *is allowed to perform with respect to any other processor, all previous accesses (*`loads` *and* `stores`*) must be performed.*

Processor consistency explores additional parallelism and improves performance by allowing a `load` access to perform before previously issued `store` accesses have been performed.

2.2.3 Weak Consistency

Based on the assumption that critical memory accesses are regulated by programmers through synchronization points and critical sections, a weaker consistency model is defined. The weak consistency model requires that all synchronization accesses must be identified by programmers or compilers. With this reasonable programming requirement, weak consistency permits multiple memory accesses to be pipelined. The weak consistency model requires the following conditions to be satisfied [22, 19].

Condition 2.3 (Conditions for Weak Consistency)

1. *before an ordinary* `load` *or* `store` *access is allowed to perform with respect to any other processor, all previous synchronization accesses must be performed, and*

2. *before a synchronization access is allowed to perform with respect to any other processor, all previous ordinary* **`load`** *or* **`store`** *accesses must be performed, and*

3. *synchronization accesses are sequentially consistent with respect to one another.*

Weak consistency allows ordinary **`loads`** and **`stores`** within a critical section to be pipelined, which improves the performance of a multiprocessor.

2.2.4 Release Consistency

Release consistency further relaxes the ordering requirements of memory events by dividing synchronization accesses into **`acquire`** and **`release`** accesses [19]. An acquire synchronization access is performed to gain access to a set of shared locations. A release synchronization access grants this permission. An acquire is accomplished by reading a shared location until an appropriate value is read. Thus, an acquire is always associated with a read synchronization access. Similarly, a release is always associated with a write synchronization access. Condition 2.4 lists the conditions to satisfy release consistency.

Condition 2.4 (Conditions for Release Consistency)

1. *before an ordinary* **`load`** *or* **`store`** *access is allowed to perform with respect to any other processor, all previous acquire accesses must be performed, and*

2. *before a release access is allowed to perform with respect to any other processor, all previous ordinary* **`load`** *or* **`store`** *accesses must be performed, and*

3. *synchronization accesses are processor consistent with respect to one another.*

In comparison to weak consistency, release consistency allows additional overlapping among ordinary accesses and synchronization accesses. First, ordinary **`load`** and **`store`** accesses following a release access do not have to wait for the release to complete. Second, an acquire synchronization access needs not be delayed for previous ordinary `load` and `store` accesses to be performed. Furthermore, synchronization accesses are only required to be processor consistent rather than sequentially consistent. One disadvantage of the release consistency model is the higher hardware complexity it requires. As an example, counters are required to maintain statuses of outstanding ordinary accesses and synchronization accesses.

CHAPTER 3

CONFLICT-FREE MEMORY

3.1 The CFM Architecture

A memory module can be roughly characterized in three different dimensions, namely, *address space*, *word width*, and *number of memory banks*. Throughout this dissertation, the term, address space, refers to *physical address space*, which defines the total number of addressable elements in a memory module. It is limited by physical memory chips, address bus width, and interfacing schemes. A processor retrieves or stores a number of bits in one memory access. The number of bits processed in one access is defined as word width. In order to keep up with the fast speed of more and more powerful processors, multiple-bank memory modules have been developed and widely used. The number of memory banks defines the maximum number of memory addresses that can be accessed at one time, which along with other parameters specify the total bandwidth of a memory module. Due to conflicts in memory accesses, however, the effective memory bandwidth is usually lower. The rest of this section shows how the CFM architecture is designed to increase the effective memory bandwidth.

3.1.1 The AT–Space and Block Accesses

Each of the three dimensions mentioned above characterizes the function or improves the performance of a memory device. One may start to wonder: can a fourth dimension be defined, and will it bring any new advantages to memory performance? The answers to both of these questions are "yes". As in most physics subjects, the fourth dimension of a memory device can be defined as the "time dimension". With the introduction of this new degree of freedom, conflicts among shared memory accesses in multiprocessors can be eliminated.

A conventional interleaved memory module can be viewed as a function, M, mapping from its address space A to the range of data elements D. A read operation in the memory module can be depicted by the function $d = M(a \cdot b)$, where d is the data retrieved, and $a \cdot b$ denotes the memory address being accessed. The components a and b represent the address offset in a memory bank and the bank number, respectively. In case of two or more memory accesses to the same memory bank, there are memory conflicts, even if they are not to the same location.

As in a conventional interleaved memory module, an address in the CFM consists of its offset in a memory bank and the bank number. The bank number, however, is not part of the input to a memory access. Instead, it is defined by the time slot number in which the data is accessed. A time slot is usually the length of a CPU cycle. A constant number (usually the number of memory banks) of time slots compose a time period which characterizes the fourth dimension of the CFM architecture. The new memory function is now defined as a mapping from its address–time space AT to the range of data elements D. A read operation in this model can be described by the function $d = M(a \cdot t)$, where d is the content at the address offset a in the memory bank defined by the time slot t in which the data is accessed.

Figure 3.1 shows an AT–space with an address space distributed in four memory

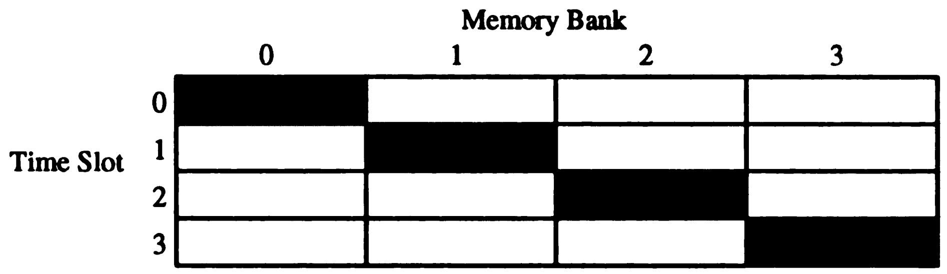


Figure 3.1. *AT*–space and the accessible subset.

banks and a time period of four slots. Each shaded area denotes the memory bank defined by each time slot in the period. As can be seen, only a subset of the *AT*–space can be accessed within the time period. The blank areas simply represent the idle time slots of the memory banks. These may not be attractive to a single processor system with only one path between the CPU and the memory device. To a multi-processor with shared memory, however, the *AT*–space model offers an efficient way of eliminating shared memory conflicts among the processors. This will be explained later.

In this dissertation, each set of memory locations with the same offset in all the memory banks of a memory module is defined as a block, whose size is determined by cache line size. (Relationship or tradeoffs among block size, number of memory banks, and memory word width are discussed in Section 3.1.4.) Memory accesses in the CFM architecture are in blocks, each of which involves retrieving or storing words with the same offset from all memory banks. To simplify the discussion here, let us assume that the memory bank cycle is the same as the CPU cycle. Suppose that there are b memory banks, each block access takes b time slots. In a real case, the memory cycle is usually longer than the CPU cycle. The CFM design in such a case is explained later in this section.

It is important to note that a block access can start at any time slot. There is no delay required before starting a block access. This is because that a block access does

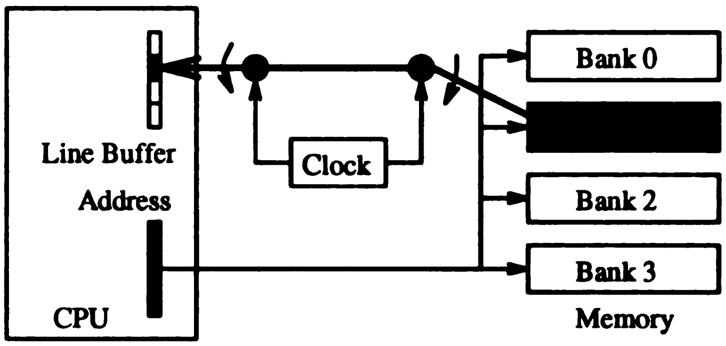


Figure 3.2. Mechanisms for non-stall block accesses.

not have to start at the first bank, instead, each access starts at the bank defined by the time slot in which the access request is received by the memory module. This avoids unnecessary stalls, which occur in the Monarch and the OMP when a memory access arrives at a memory bank in a wrong time phase. Figure 3.2 shows the mechanism which implements the block access style. The CPU and the memory banks are fully synchronized to ensure that each word of a block is transferred between its memory bank and the corresponding section of a CPU line buffer.

3.1.2 Designing the CFM with AT–Space Partitioning

In a shared-memory multiprocessor with a single memory module, the processors share a single address space. Assume the memory module is composed of a number of memory banks. Memory conflicts occur when two or more processors access the same memory bank at the same time. Most of the memory conflicts are unnecessary and undesirable, since they are caused by processors accessing different locations in the same memory bank. These memory conflicts need to be eliminated in order to increase effective memory bandwidth. Some researchers have attempted to reduce memory conflicts by memory skewing or random mapping. In contrast, the CFM scheme presented here is based on a deterministic and highly synchronized model of

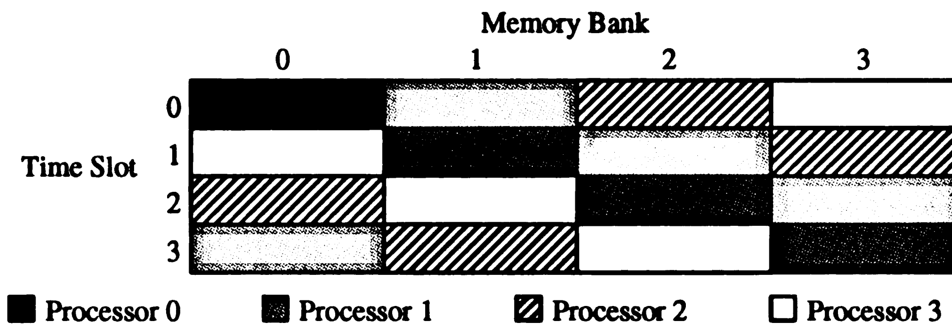


Figure 3.3. Mutually exclusive subsets in AT–space.

block accessing style.

With an appropriate partitioning of the AT–space, as shown in Figure 3.1, four mutually exclusive subsets of the space can be formed and assigned to four different processors in a multiprocessor. It is easy to show that, with such an assignment, the architecture guarantees conflict-free accesses to the shared memory. Figure 3.3 demonstrates a partitioning and an assignment of the AT–space. It is shown that, at time slot t, processor p can only access memory bank $((t+p) \bmod 4)$, for $0 \leq t, p \leq 3$. With the mapping of the processor number p and the time slot t to the accessible memory bank b, each processor occupies an independent subset of the AT–space for a block access, and thus, the memory architecture is guaranteed to be conflict-free.

The above AT–space partitioning and mapping scheme can be implemented with a simple synchronous switch box. Figure 3.4a shows such a switch box with four input/output ports on each side, connecting four processors and four memory banks. The switch box is similar to an ordinary *crossbar*, but much simpler as it requires neither address decoding nor setup delay for routing decisions. Its four routing states, shown in Figures 3.4b, c, d and e, are driven by the system clock. At time slot t, input port i is connected to output port $((t+i) \bmod 4)$, for all $0 \leq t, i \leq 3$. Every four CPU cycles, it completes a time period, which is totally deterministic. This mechanism implements the mutually exclusive partitioning and assignment of the

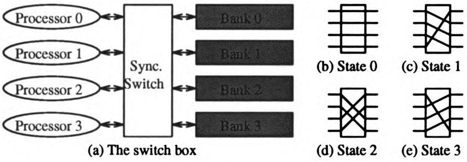


Figure 3.4. A 4 × 4 synchronous switch box.

AT–space shown in Figure 3.3, which guarantees conflict-free block accesses. For example, at time slot 0, block accesses issued by processor 0 and 1 start retrieving or storing data in bank 0 and 1, respectively, without contention. During the middle of the accesses, at slot 2, a write operation issued by processor 3 can start storing data in bank 1 without interfering with the two accesses issued earlier. Section 3.2 shows how the synchronous switch can be extended to support larger and more complicated interconnection networks for conflict-free memory accesses.

3.1.3 Constructing the CFM with Longer Memory Bank Cycle

The time required to complete a memory word access in a memory bank is defined as a *memory bank cycle*. In practical multiprocessors, a memory bank cycle is usually longer than a CPU cycle. Under such circumstances, designing the CFM architecture with the *AT*–space partitioning and assignment becomes infeasible. Since memory bank access cannot be finished within one time slot, the cost of a block access becomes too high. Consider a conventional memory module with multiple banks. The memory banks can be interleaved and accessed in a pipelined fashion. When an access is received by the memory module, the input address can be used by each of the memory banks. The memory banks start to process the request of continuous

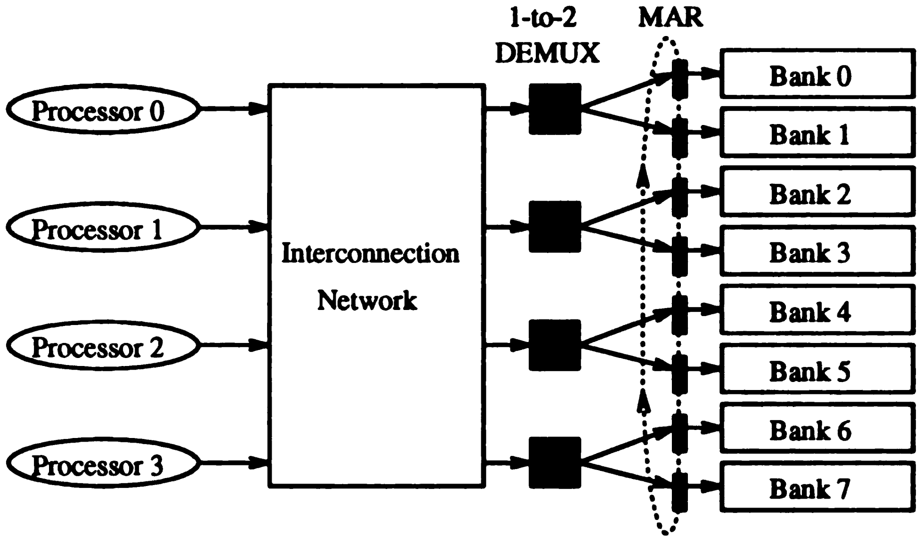


Figure 3.5. The CFM with memory bank cycle equal to 2 CPU cycles.

addresses at different times. Eventually, the data from or to different banks will be piped through the interconnection network. The same concept can be applied to the CFM architecture.

Figure 3.5 presents the organization of the CFM with memory bank cycle equal to two CPU cycles. For better readability, only address paths are shown in the figure. Data transfer can share a complicated bi-directional network with the address paths or be handled by a separate network. In order to reduce the speed gap between the processors and the memory banks, there are twice the number of memory banks as the number of processors. Besides the synchronous switch, there are a column of one-to-two demultiplexers between the switch and the memory banks. Like the switch, the demultiplexers are also driven by the system clock. The switch and the demultiplexers divide each time period into eight time slots for the four processors and the eight memory banks. The state transitions of the switch and the demultiplexers are arranged appropriately, so that, at time slot t, processor p is connected to memory bank $((t + 2p) \bmod 8)$, for all $0 \leq t \leq 7$ and $0 \leq p \leq 3$. Table 3.1 presents the address paths connected to the memory banks at different time slots. The data path connections are similar but shifted by one time slot.

Table 3.1. Address path connections.

	B0	B1	B2	B3	B4	B5	B6	B7
Slot 0	P0		P1		P2		P3	
Slot 1		P0		P1		P2		P3
Slot 2	P3		P0		P1		P2	
Slot 3		P3		P0		P1		P2
Slot 4	P2		P3		P0		P1	
Slot 5		P2		P3		P0		P1
Slot 6	P1		P2		P3		P0	
Slot 7		P1		P2		P3		P0

It can be seen that, at each time slot, an accessed memory bank is at least two banks away from any other accessed memory bank. The input address of each access can be shifted directly between the MARs (memory address registers) of the memory banks, instead of retransferred from the processors, as presented in Figure 3.5. These allow a memory access to be processed in a pipelined fashion, where a memory bank processing the second CPU cycle stage of an access can have its successor processing the first CPU cycle stage of the same access. The result is also transferred from or to the memory banks in a pipelined fashion which starts one time slot later than the access being issued. For example, if processor 0 issues a read operation at time slot 0, it will receive the retrieved data from memory banks 0 and 1 at slots 1 and 2, respectively. The timing diagram of this example is shown in Figure 3.6.

The concept can be extended to memory banks with longer cycles. Basically, for memory banks with an access cycle equal to c, the number of memory banks must be c times the number of processors. One-to-c demultiplexers are used with

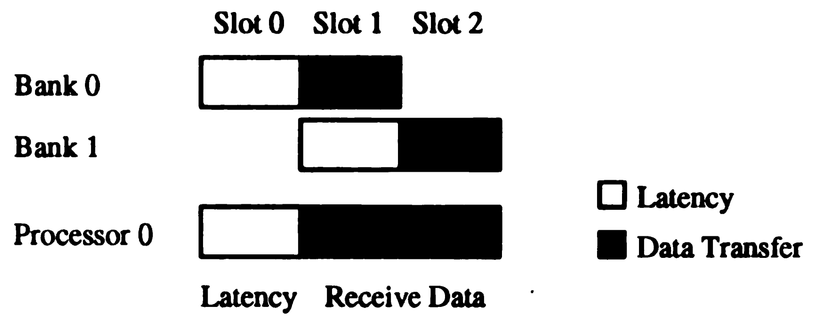


Figure 3.6. The timing diagram of a read operation.

the synchronous interconnection network to connect the processors and the memory banks. For the purpose of simplicity, in the discussion of the following sections, the memory bank cycle is assumed to be identical to the CPU cycle. The number of memory banks is also assumed to be equal to the number of processors. The concept explained here, however, can be extended to all the examples mentioned later for practical situations.

3.1.4 The CFM Configurations and Parameters

With respect to system parameters, the CFM architecture can be implemented in different configurations. These parameters include numbers of processors and memory banks, block size, memory word width, and memory bank cycle. Word width is defined as the number of bits retrieved or stored in one memory bank access. Table 3.2 shows the definition of a list of notations to be used in the following discussion.

Consider a system with n processors and b memory banks containing w-bit memory words. Since each block contains a memory word from each memory bank, the block (and cache line) size, ℓ, can be defined as follows.

$$\ell = bw$$

Table 3.2. Definition of notations.

Notation	Definition
n	Number of processors
b	Number of memory banks
m	Number of memory modules
ℓ	Block (and cache line) size (in bits)
w	Memory word width (in bits)
c	Memory bank cycle (in CPU cycles)
β	Block access time (in CPU cycles)

Assume each memory bank access takes c CPU cycles. In order to support conflict-free accesses, the number of memory banks needs to be c times the number of processors, as shown below.

$$b = cn$$

As a consequence, the number of processors that can be supported in the CFM system is

$$n = \frac{b}{c} = \frac{\ell}{cw}$$

It is shown that, with more memory banks and larger blocks, more processors can be supported for conflict-free accesses. Since the block size is defined to be identical to the cache line size, systems with larger cache lines can support more processors. Detailed study of the tradeoff between cache line size and system performance can be found in [24]. Some advanced processors have the tendency using large cache lines. For example, the cache line size in the Intel i860 processor is 32 bytes (256 bits), while the cache line size in the IBM RS/6000 is 128 bytes.

In order to retrieve or store a block of memory words, the memory banks cooperate in a pipelined fashion. The latency of a block access in the CFM architecture is the same as that of a conventional memory block access. The block access time, β, can be computed as follows.

$$\beta = b + c - 1$$

As can be seen, when the number of memory banks increases, the block size and the block access time also increase. The number of memory banks needs to be limited in order to maintain appropriate cache line size and efficient conflict-free memory access. This presents a tradeoff between the memory latency and the number of processors that can be supported in the conflict-free system.

Table 3.3. Trade-off in the CFM configurations. ($\ell = 256$, $c = 2$)

Memory banks	Word width	Memory latency	Processors
256	1	257	128
128	2	129	64
64	4	65	32
32	8	33	16
16	16	17	8
8	32	9	4

To get a clearer idea about the tradeoff among the CFM configurations, let us define the block size ℓ and the memory bank cycle c as constants. By decreasing the number of memory banks b and increasing the memory word width w, the memory latency for each block access can be reduced. The number of processors supported for conflict-free memory accesses, however, is reduced. Table 3.3 demonstrates the tradeoff observed in a system with cache line size ℓ equal to 256 bits and memory bank cycle c equal to two. It is shown that, with smaller memory word width, more processors can be supported for conflict-free accesses. This is adequate for the practical situation since, in larger scale systems, data paths tend to be narrower due to higher connectivity between processors and memory banks. For example, the Monarch with 64K processors uses one-bit serial data paths for shared memory accesses.

Although memory latency places a limitation on the number of processors that can be supported in the CFM configurations, there are ways to compensate the problem. First, cache line prefetching techniques implemented in some parallel compilers can be employed to reduce the effect of a long memory latency. The NYU Ultracomputer is an example of this approach [25]. Second, a partially conflict-free approach using the CFM concept can be used to support large scale multiprocessors without increasing memory latency. This approach groups memory banks into a number of memory modules with smaller blocks. Processors are also divided into several "contention sets". Each contention set is assigned a particular AT–space division for accessing the memory modules. A "conflict-free" cluster can be formed by selecting one processor from each of the contention sets. Processors in the cluster do not conflict with each other in accessing the memory modules as they are from different contention sets and use different AT–space divisions. With proper processor allocation and memory locality, this partially conflict-free approach can reduce memory and network contention to a minimum. The implementation issues of this approach is described in Section 3.2.2.

3.2 Contention-Free Interconnection Networks

The synchronous switch box shown in Figure 3.4 serves as the interconnect between processors and memory banks for synchronous block accesses. Like all crossbar switches, however, it has a limitation on the number of I/O ports that can be connected to other devices. Crossbar interconnections are used on supercomputers and multiprocessors that have a limited number of processors and memory banks. For a system with a large number of processors, a more sophisticated interconnection

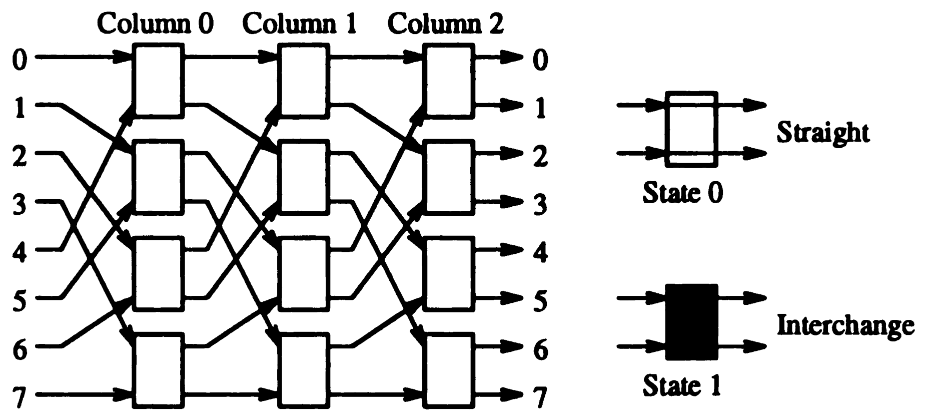


Figure 3.7. An 8 × 8 omega network.

scheme is needed. *Multistage Interconnection Network* (MIN) is one of the widely used interconnection schemes in large scale multiprocessors [4, 5, 7]. The network is built by connecting a number of crossbar switches in a number of layers. Figure 3.7 shows an *omega* network, which is an example of MIN, and the possible states of its switches. Through circuit switching, messages can be routed and transferred by the intermediate switches between processors and memory banks. Memory accesses implemented with message passing in a MIN introduce routing and message header overhead. Moreover, contention encountered in switches also degrades effective memory bandwidth. To solve these problems, this section presents a synchronous interconnection network based on the omega network topology, which eliminates all possible switch contention as well as message passing overhead.

3.2.1 Synchronous Omega Networks

Like building circuit-switching omega networks with crossbar switches for conventional multiprocessors, synchronous omega networks can be built with synchronous switches for supporting contention-free interconnections. For example, an 8 × 8 synchronous omega network can be built with 12 2×2 synchronous switches arranged in

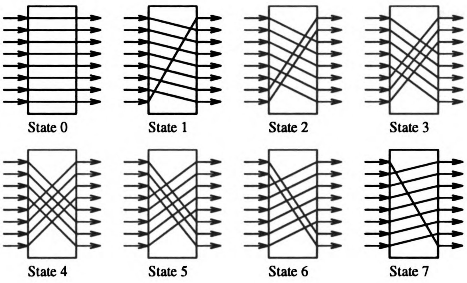


Figure 3.8. States of an 8 × 8 synchronous omega network.

3 columns, as shown in Figure 3.7. The goal of building such a synchronous omega network is to support block accesses just as an ordinary 8 × 8 synchronous switch does. The network should have a similar state transition pattern as a single switch, that is, at time slot t, input port p is mapped to output port $((t+p) \bmod 8)$, for all $0 \leq t, i \leq 7$. It has been shown by Lawrie [26] that such mappings can be done with no contention.

Figure 3.8 presents the states of the 8 × 8 synchronous omega network. The connection state of the synchronous omega network fully depends on the state of each synchronous switch. The switches have two connection states: "straight" and "interchange", denoted by state 0 and 1, respectively. Table 3.4 shows the states of the switches at each time slot. Note that, since all the switches are synchronous, correct connection states for all switches can be set simultaneously for each time slot. There is neither setup time nor propagation delay required for the switches on a path. This is unlike the situations in the BBN Butterfly [4, 5] and the RP3 [7], where setup time and propagation delay are needed for routing and flow control.

Table 3.4. States of switches in an 8 × 8 synchronous omega network.

Column	0				1				2			
Switch	0	1	2	3	0	1	2	3	0	1	2	3
Slot 0	0	0	0	0	0	0	0	0	0	0	0	0
Slot 1	0	0	0	1	0	0	1	1	1	1	1	1
Slot 2	0	0	1	1	1	1	1	1	0	0	0	0
Slot 3	0	1	1	1	1	1	0	0	1	1	1	1
Slot 4	1	1	1	1	0	0	0	0	0	0	0	0
Slot 5	1	1	1	0	0	0	1	1	1	1	1	1
Slot 6	1	1	0	0	1	1	1	1	0	0	0	0
Slot 7	1	0	0	0	1	1	0	0	1	1	1	1

In a circuit-switching omega network, a memory address is represented by a memory module number and an offset. The memory module number is used by each switch column as routing information and required to be included in the header of each memory access request. While in a synchronous omega network, a memory address is composed of an offset and a memory bank number. Since all the switch routings are driven by the system clock, only the offset is required in the header of a memory access request. Figure 3.9 shows a comparison between the message headers used by a circuit-switching omega network and a synchronous omega network. It is clear that the new scheme reduces communication overhead by decreasing the message size of each memory access request.

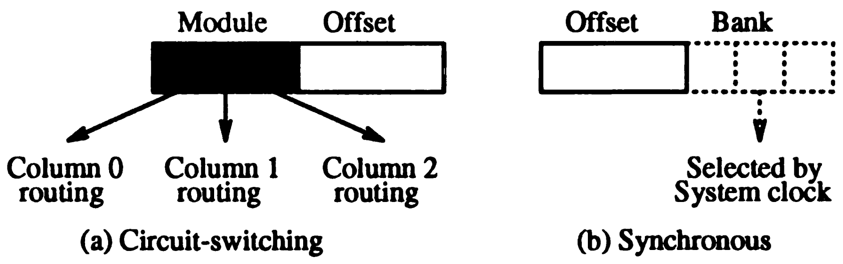


Figure 3.9. Message headers of memory access requests.

3.2.2 An Extension for Large Scale Multiprocessors

When the number of memory banks becomes larger, the block access style mentioned is no longer appropriate. Imagine a system with 64K processors and the same number of memory banks such as the Monarch [11], a block becomes too large to be efficient. By modifying the synchronous omega network mentioned above, however, memory banks can be grouped into a number of conflict-free memory modules with smaller blocks. Each memory module contains a continuous physical address space. Within each module, memory addresses are interleaved among memory banks and mutually exclusive AT–space division is maintained. This modified synchronous omega network supports *partially conflict-free memory* accesses.

In such a system, a memory address can be represented by a module number, an offset, and a bank number. Only the module number is used for routing, and only the module number and the offset are required in the message header of a memory access request. The bank number of a memory access is selected automatically by the system clock. Figures 3.10a and b show the memory addresses and message headers of a system with 4 two-bank memory modules and a system with 2 four-bank memory modules, respectively. The two-bank memory module system has two-word blocks, while the four-bank module system has four-word blocks.

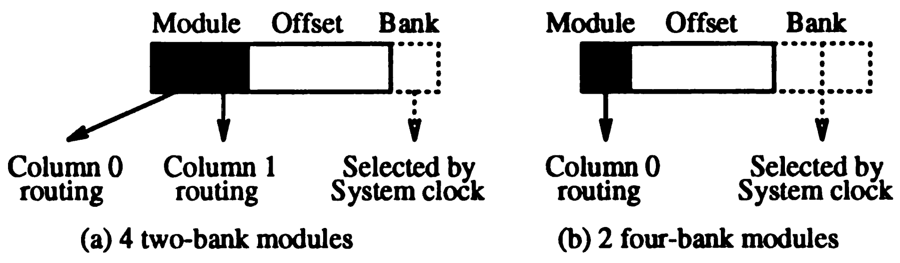


Figure 3.10. Message headers of partially synchronous omega networks.

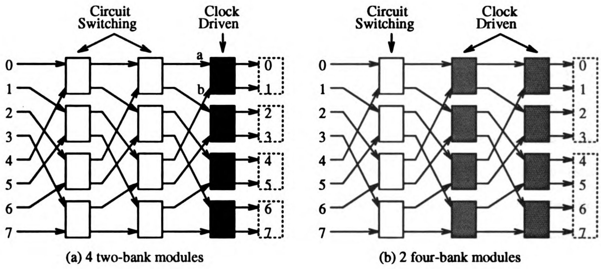


Figure 3.11. Partially synchronous omega networks.

To construct a system with 4 two-bank memory modules using 2×2 switches, an omega network can be configured so that the first two columns are implemented with crossbars and the third column implemented with clock-driven synchronous switches. Figure 3.11*a* presents such a partially synchronous omega network. The first and second columns of the network are routed by circuit-switching, while the last column is controlled by the system clock. As can be seen, requests being sent to ports *a* and *b* can access memory banks 0 and 1 synchronously without contention, thus, banks 0 and 1 form a conflict-free memory module. The same concept applies to memory banks 2 and 3, 4 and 5, and 6 and 7. Since processors 0, 2, 4 and 6 always access banks 0 and 1 through port *a* and access other memory modules also through particular ports, they form a *contention set.* Likewise, processors 1, 3, 5 and 7 form another contention set. A conflict-free cluster can be formed by selecting one processor from each of the contention sets. This clustering mechanism reduces the block size into two words and supports partially conflict-free memory accesses.

Figure 3.11*b* shows another partially synchronous omega network which has only its first column switches routed by circuit-switching. Both the second and the last columns are controlled by the system clock. The network groups the memory banks into two conflict-free modules. The first module includes memory banks 0, 1, 2 and 3. The second module contains banks 4, 5, 6 and 7. According to access paths to these conflict-free memory modules, processors are divided into four contention sets: (0, 4), (1, 5), (2, 6), and (3, 7). A conflict-free cluster can easily be formed by selecting one processor from each of the sets. The configuration reduces the block size into four words. It can be seen that, there is a tradeoff between the block size and the degree of conflict-free access. By adjusting the numbers of circuit-switching columns and clock-driven columns of the omega network, a multiprocessor can be configured with different block sizes. Table 3.5 illustrates possible configurations of a 64-bank multiprocessor implemented with 2 × 2 switches.

Table 3.5. Different configurations of a 64-bank multiprocessor.

Module	Bank	Block size	Circuit-switching	Clock-driven	Remark
1	64	64 words	0 column	6 columns	CFM
2	32	32 words	1 column	5 columns	
4	16	16 words	2 columns	4 columns	
8	8	8 words	3 columns	3 columns	
16	4	4 words	4 columns	2 columns	
32	2	2 words	5 columns	1 column	
64	1	1 word	6 columns	0 column	Conventional

It is worth mentioning that, in some large scale multiprocessors, the CFM architecture can even be applied to memory word access. In an extremely large scale multiprocessor, due to the high connectivity between processors and memory banks, wide data paths are unlikely to be implemented in the interconnection network. Usually, a wide memory word is divided into several sections and transferred in several CPU cycles. In such a system, a block can be defined as identical to a memory word, which is composed of data stored in a number of memory banks. The number of memory banks form a conflict-free memory module. For example, the Monarch uses bit-serial data paths in its interconnection network to transfer 64-bit memory words [11]. With the CFM architecture, each memory module can be composed of 64 memory banks, where each bank provides one-bit data, to support partially conflict-free memory word access with shorter latency than the Monarch.

3.3 Other CFM Extensions

In addition to the omega network scheme, there are other solutions to extending the CFM concept for large scale multiprocessing. The CFM architecture mentioned have assumed the same or proportional number of processors, memory banks and time slots. In the CFM architecture, however, the number of processors can be less, leaving free slots for other purposes such as DMA and remote memory accesses.

Figure 3.12 presents two CFM clusters connected to each other. Each CFM cluster has three processors and four memory banks supporting four time slots for conflict-free memory accesses. Since each cluster has one free time slot, the slot can be used to serve remote memory access requests from each other. For example, processor 0 in cluster A makes an access request to memory banks in cluster B. The request is sent to cluster B through a memory-mapped I/O port and the interconnection between the clusters. Upon receiving the request, cluster B serves as if it is an ordinary memory access from a local processor. The service does not introduce network and memory contention to cluster B, since it uses the free time slot. To processor 0, the remote memory access can be considered as just a "slower" regular memory access. Contention on the interconnection between clusters, however, is still possible.

The multiple-cluster connection scheme can be used to extend the CFM architecture for constructing multiprocessors with various scales, connectivity, and topologies.

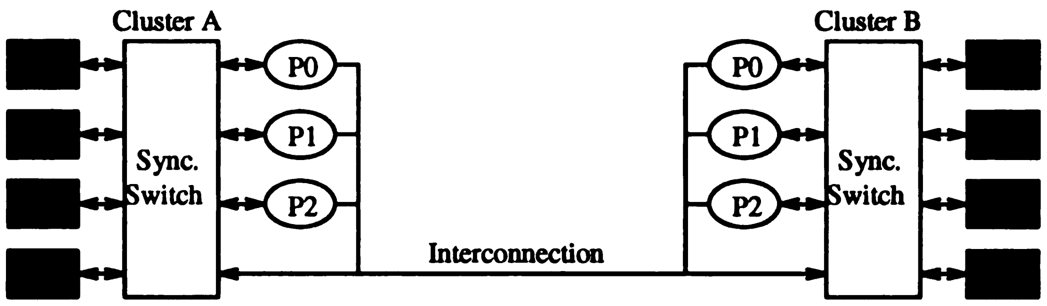


Figure 3.12. A system with two conflict-free clusters.

These include *hypercube*, *2-D mesh*, etc. Furthermore, A hierarchical extension approach will be presented in Chapter 5.4 after the introduction of a new cache coherence protocol for the CFM architecture.

3.4 The CFM Performance

Performance of a multiprocessor is influenced by numerous factors. These factors include the nature of applications, computation models, and hardware architectures. From the architecture point of view, speedup can be limited by effective memory bandwidth, interconnection network overhead, and other parameters. The CFM architecture improves system performance by increasing effective memory bandwidth as well as reducing interconnection network overhead. This section gives a brief discussion about the impact of the CFM architecture on the performance of a multiprocessor.

3.4.1 Efficiency of Conventional Memory Systems

Consider a conventional multiprocessor with n processors and m memory modules. Assume each processor uniformly generates memory accesses to the memory modules at a rate of r accesses per CPU cycle. All memory accesses are in blocks. Each memory module may consist of multiple banks to support pipelining for block accesses. Each block access takes β CPU cycles, where the value of β is dependent upon the block size and $nr\beta \leq m$. To a memory access, the probability that the target memory module is busy serving another access can roughly be computed as $P(r) = \frac{(n-1)r\beta}{m}$. Without considering interconnection network contention, a memory access request being sent to a memory module has the probability $P(r)$ of conflict with other memory accesses. The expected number of retries is given below.

$$\text{Expected number of retries} = \frac{1}{1-P(r)} - 1 = \frac{P(r)}{1-P(r)}$$

To simplify the model, assume each failed memory access consumes an average of $\frac{\beta}{2}$ CPU cycles before a possibly successful retry. The expected time, $M(r)$, it takes to complete a memory access can be approximated as follows:

$$M(r) = \beta + \frac{P(r)}{1-P(r)} \times \frac{\beta}{2} = \frac{2-P(r)}{2-2P(r)} \times \beta$$

The efficiency of memory accesses can be depicted by the following formula.

$$E(r) = \frac{\beta}{M(r)} = \frac{2-2P(r)}{2-P(r)} = \frac{2m-2(n-1)r\beta}{2m-(n-1)r\beta}$$

For example, suppose a multiprocessor has 8 processors and 8 memory modules. Each memory block contains 16 words. A block access takes 17 CPU cycles to complete. Figure 3.13 shows the efficiency of memory accesses under different memory access schemes and rates. In the case of conventional memory, as the access rate increases, the efficiency of memory accesses drops due to increased memory conflicts. Note that, because of interconnection network contention, the actual efficiency of the conventional memory is even lower than depicted in the figure. While in the CFM architecture, since both memory conflicts and network contention are eliminated, the efficiency of memory accesses can roughly be thought of as 100%. It is clear that, when memory access rate is expected to be high, the CFM architecture is preferable.

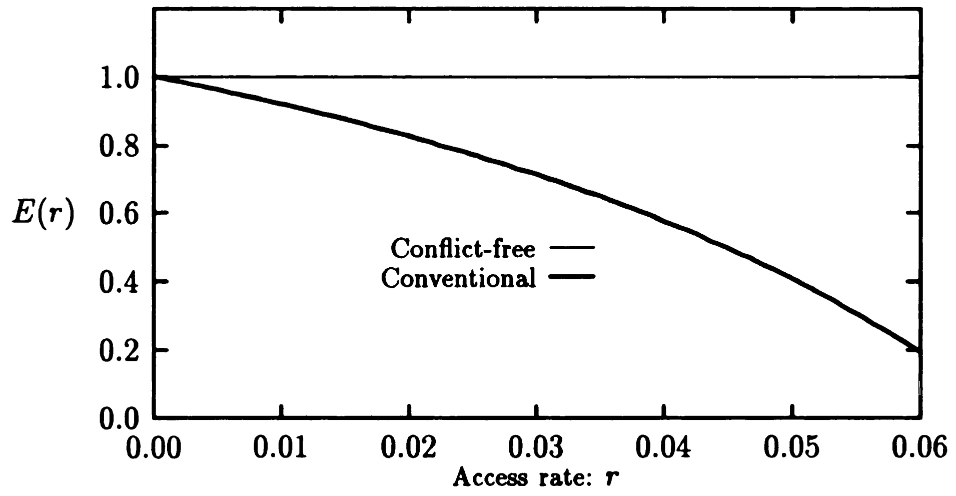


Figure 3.13. Memory access efficiency. (n = 8, m = 8, block size=16, β = 17)

3.4.2 Efficiency of the CFM Architecture

The CFM architecture improves memory access efficiency by eliminating memory access conflicts, both in interconnection networks and in memory modules. As just mentioned, in the fully conflict-free system, the efficiency of memory accesses is approximately 100%. Here, the efficiency of the partially conflict-free architecture mentioned in Section 3.2.2 is further studied. The memory access efficiency in a partially conflict-free system is dependent upon several factors, which include cluster size and data locality. Consider a partially conflict-free system with a cache line size equal to b words. Each conflict-free memory module is composed of b banks. Assume each memory bank access takes c CPU cycles. As described in Section 3.1.4, the block access latency can be computed as $\beta = b + c - 1$. In order to install n processors, cn memory banks are required to be arranged in $m = \frac{cn}{b}$ memory modules. (The fully conflict-free system mentioned in Section 3.1.4 has the characteristic of $\frac{cn}{b}$ equal to one.)

Each memory module can simultaneously support $\frac{b}{c}$ processors for conflict-free accesses, in other words, every $\frac{b}{c}$ processors accompanied by one memory module can form a conflict-free cluster. There are $\frac{n}{b/c} = \frac{cn}{b} = m$ clusters. Each time slot of a memory module is shared by accesses requiring that time slot from the local cluster and from the $(m-1)$ remote clusters. Local accesses do not conflict with each other, since they use different time slots. Assuming remote accesses to be uniformly distributed, each memory module serves $\frac{1}{m-1}$ of the remote accesses from each of the $(m - 1)$ remote clusters. The locality, λ, of an application can be defined as the proportion of accesses to a local cluster. The probability, P_1, that a time slot is used by a remote access is computed as follows:

$$P_1 = \frac{(m-1)(1-\lambda)r\beta}{m-1} = (1-\lambda)r\beta$$

In other words, a local access has the probability P_1 of being blocked by a remote access. The probability, P_2, that a remote access encounters a conflict is shown below.

$$P_2 = (1 - \frac{1-\lambda}{m-1})r\beta$$

In order to simplify the model for study purposes, the combined probability of $P_1(r, \lambda)$ and $P_2(r, \lambda)$ is used for the subsequent discussion. The combined probability, $P(r, \lambda)$, i

fre

17 (

CPU

of λ :

is computed as follows:

$$P(r,\lambda) = P_1(r,\lambda)\lambda + P_2(r,\lambda)(1-\lambda) = (\frac{-m\lambda^2 + 2\lambda + m - 2}{m-1})r\beta$$

Using the same assumptions as for the conventional memory case discussed earlier, the efficiency, $E(r,\lambda)$, is given below.

$$E(r,\lambda) = \frac{2 - 2P(r,\lambda)}{2 - P(r,\lambda)} = \frac{2(m-1) - 2(-m\lambda^2 + 2\lambda + m - 2)r\beta}{2(m-1) - (-m\lambda^2 + 2\lambda + m - 2)r\beta}$$

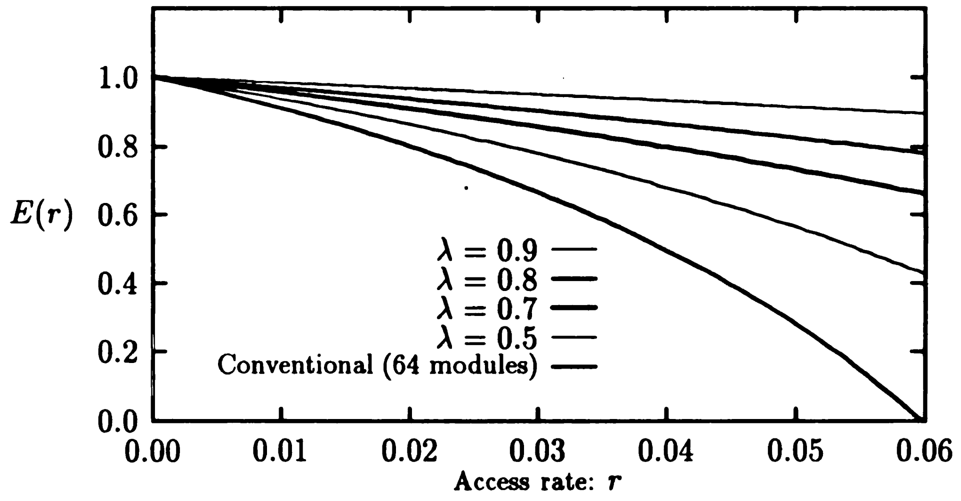


Figure 3.14. Memory access efficiency. ($n = 64$, $m = 8$, block size=16, $\beta = 17$)

Figure 3.14 shows the example of a multiprocessor with 64 processors, 8 conflict-free memory modules, and 16-word-wide block size. The block access latency is again 17 CPU cycles, with the assumption that a memory bank access cycle is equal to two CPU cycles. The efficiency of the partially conflict-free system under the situations of $\lambda = 0.9$, 0.7, 0.5, and 0.3; and the efficiency of the conventional system are shown.

For comparison purposes, the conventional system is assumed to have 64 memory modules, so that it requires the same connectivity in its interconnection network as the partially conflict-free system. As can be seen, although the efficiency of the partially conflict-free system drops when the locality decreases, it is still superior to that of the conventional system, especially in the cases of high access rates. Figure 3.15 presents another example, which is a multiprocessor with 128 processors and 16 conflict-free modules. Again, the partially conflict-free system shows its increased memory access efficiency in comparison to the conventional 128 processors, 128 modules system.

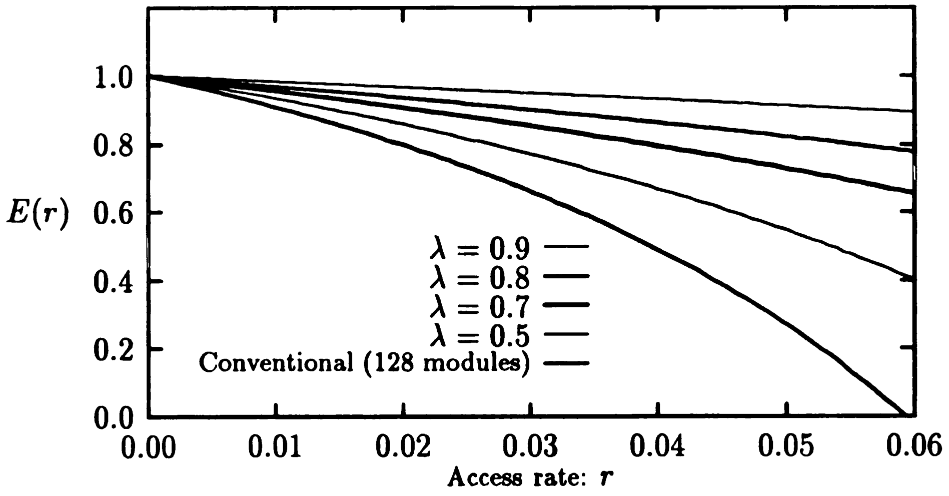


Figure 3.15. Memory access efficiency. (n = 128, m = 16, block size=16, β = 17)

3.4.3 Overhead of Interconnection Networks

As mentioned in Section 3.2, the interconnection between processors and memory banks of the CFM architecture can be implemented with one or more synchronous switches. The switches are driven by the system clock and have fully synchronous and deterministic state transition patterns. Data and address paths in the contention-free interconnection network can be established for each CPU cycle. Unlike conventional circuit-switching networks, there is no setup or routing delay in the contention-free interconnection network implemented with synchronous switches. This is extremely attractive for optical interconnection networks, since, without the clock-driven nature, they would require analog/digital conversion for obtaining routing information in each switch.

Moreover, since the mapping between processors and memory banks is specified at each time slot, there is no need for a memory access request to carry the memory bank number. This reduces the message size of an access request and improves the efficiency of data transfer. In order to support a shared memory space that is larger than 4 GB, the BBN TC2000 uses the 34-bit-wide *system physical address* in its interconnection network, which is different from the 32-bit-wide address in its Motorola MC88100 CPU [5]. A special address transformation strategy needs to be employed for the address conversion. While in the CFM architecture, a shared memory space larger than 4 GB can easily be handled without special hardware support or extra overhead.

In conventional multiprocessors, flow control and conflict resolution are two other issues that need to be considered. The NYU Ultracomputer and the IBM RP3 implements operation combination logic in the switches of its interconnection network to solve some of the same-address memory conflicts [6, 7]. While in the BBN Butterfly, alternative paths and random delayed retries are employed to solve memory conflicts. Furthermore, a time-out mechanism and "REJECT" signal are used in the BBN machines for reliable data transfer and flow control. In the CFM architecture, however, the arrangements for flow control and conflict resolution are not required, since there is no interconnection network or memory contention at all.

3.4.4 Latencies of Memory Accesses

In the CFM architecture, each access involves retrieving or storing an entire block from all the memory banks. Since a block access needs not be started at the first memory bank, unnecessary stalls can be avoided. Although processing an entire block for each memory access may cause higher latencies than a word access, based on the assumption of program locality, fetching a larger memory block reduces the number of accesses required, which compensates for the higher latencies. There are various techniques designed to reduce or tolerate memory latencies. Software controlled prefetching techniques hide large latencies by bringing data close to processors before it is actually needed [27, 28, 29, 30]. Relaxed memory consistency models also help reduce the effects of long memory latencies [22, 31, 19]. Multiple contexts allow a processor to hide memory latencies by switching from one context to another during a high-latency memory access. Furthermore, in large scale multiprocessors, as mentioned in Section 3.2.2, memory banks can be grouped into conflict-free modules in order to reduce the block size, thus also the latency of each block access.

CHAPTER 4

DATA CONSISTENCY AND ATOMIC OPERATIONS

Shared memory conflicts are one of the most important factors which degrade performance of multiprocessors. Memory conflicts occur when two or more processors access data in the same memory bank at the same time, even though they are at different locations of the memory bank. The CFM architecture eliminates any possible memory conflicts and increases effective memory bandwidth. There are certain memory conflicts, however, that need to be managed differently for data consistency and correct program execution. For example, when two processors write data to the same block at the same time, only one of the operations need to be executed and only one value should be stored in the block. In addition, it is important for a memory read operation to ensure that the block read is composed of the same version elements from the memory banks, even if there is a proceeding write operation from another processor. This chapter introduces an address tracking mechanism which can be applied in the CFM architecture for maintaining data consistency. Furthermore, atomic operations that support higher level synchronization can also be implemented using the address tracking mechanism.

4.1 Data Consistency

4.1.1 Inconsistency Problem Caused by the CFM

Consider the CFM architecture presented in Figure 3.4. The system eliminates all memory conflicts, even if the accesses are to the same block. Assume processor 0 and 1 both start writing to the same block at time slot 0. Processor 0 proceeds its write operation from memory bank 0 through 3. While processor 1, instead, proceeds its write operation from memory bank 1 through 3, then 0. It can be seen that, after processor 1 has stored data in bank 1 through 3, they will all be overwritten by processor 0, and processor 1 will overwrite the data in memory bank 0 at the last time slot. The final result turns out to be that bank 0 contains data from processor 1 and the others contain data from processor 0. Figure 4.1 demonstrates the disaster of inconsistency. The problem not only occurs in simultaneous write operations, but also in write operations that are issued at neighboring time slots. Likewise, a write operation may interfere with a read operation accessing the same block, and cause the read operation to return a partially old, partially new block to the requester.

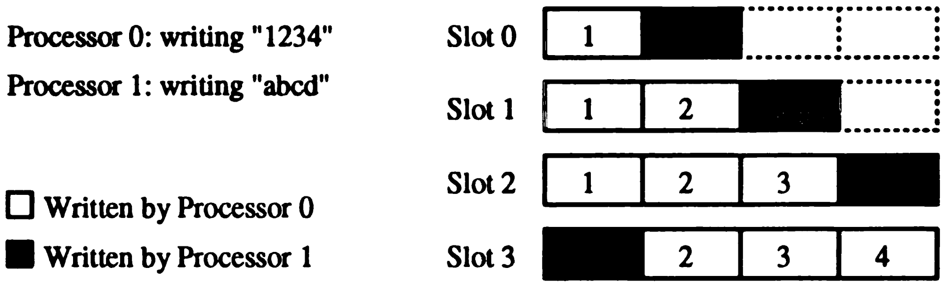


Figure 4.1. A data inconsistency caused by the CFM architecture.

These problem do not occur in all systems. Some cache coherence protocols, e.g., *write-back* protocol, have the nature that no multiple processors flush the same cache line, or one processor flushes the cache line and another loads it at the same time. Chapter 5 introduces one example of such cache coherence protocols. In multiprocessors using these protocols, the problem mentioned above can never happen, since it is handled by synchronization mechanisms related to cache coherence. For other systems, however, extra hardware control is needed in preventing the data inconsistency problem from happening.

4.1.2 Ensuring Data Consistency with Address Tracking

This section presents an address tracking technique, which solves the data inconsistency problem by introducing access controls on read and write operations. For a same-address write conflict, the goal is to ensure that exactly one of the competing write operations completes. All other write operations abort their executions before overwriting the data written by the completing write operation. This is accomplished by defining a priority among these competing write operations. If the competing write operations are issued at different time slots, only the latest issued write operation completes. If the write operations are issued at the same time slot, only the first one modifies bank 0 completes its access. All other competing write operations simply abort as they will be overwritten.

For a same-address read-write conflict, the read operation is restarted from the memory bank in which a conflict is detected, so that the retrieved data is from the same block version. In other words, a write operation has a higher priority than a read operation. This is important for efficiently supporting higher level synchronization paradigms, which is discussed later.

Figure 4.2 shows the hardware construction used to implement the tracking mechanism. Associating with each memory bank, there is an *Address Tracking Table* (ATT) implemented with an $(m-1) \times a$ associative memory, where m is the number of memory banks, a is the width of an address offset. Each ATT operates as a queue which

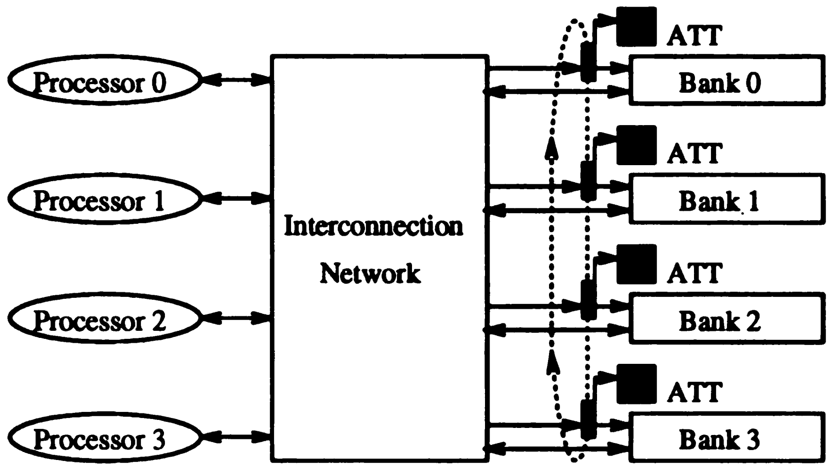


Figure 4.2. The address tracking hardware configuration.

shifts its entries from its head toward its end by one location at each time slot. Upon receiving a write request, if a memory bank is the first bank accessed by the operation, it inserts the requesting address offset at the head of its corresponding ATT after updating the content at the address offset. If it is not the first memory bank in a write operation, a "blank" address is inserted into the queue. It takes m time slots for an entry to be shifted through and removed from an ATT queue. The non-blank entries in an ATT represent some incomplete accesses currently proceeding in other memory banks.

Before a write operation can proceed in each memory bank, its accessing address offset is compared with a proper subset of the entries in the corresponding ATT to detect any proceeding same-address write operation. In case of a detected same-address operation, the write operation aborts, since it will eventually be overwritten by the detected write operation anyway. The subset of an ATT to be compared with the address offset of a write request is defined in such a way that all except one competing write operations detect others. In this case, exactly one of the competing write can complete, thus, data consistency is maintained.

Assume a write operation has updated n memory banks and is proceeding with its

qu

A

up

In

is

e

a

c

d

u

I

b

r

F

slot 0.

Since

its $(n+1)$-th update in a memory bank. The first n entries in the corresponding ATT queue represent the activity of the memory bank within the most recent n time slots. A same-address entry detected in the first $n-1$ entries denotes a write operation updating the same block which was issued later than the current write operation. In this case, the current write operation can simply be aborted, since the block it is updating will be overwritten. While a same-address entry detected in the n-th entry denotes a write operation updating the same block issued at the same time as the current write operation. In order to guarantee that exactly one of the write operations completes, a certain priority needs to be maintained. This is achieved by defining the comparing set as the first n entries before the current write operation updates memory bank 0, and the first $n-1$ entries after it updates memory bank 0. In this case, whichever simultaneous same-address write operation accesses memory bank 0 first will have the highest priority. The algorithm of a write operation in each memory bank is shown below.

```
write(offset, data)
    {
    n = number of updated memory banks;
    if(n==0)
        insert offset into ATT;
    if(bank 0 has been updated)
        comparing_set = first n-1 ATT entries;
    else
        comparing_set = first n ATT entries;
    if(conflict in comparing_set)
        abort;
    store data at offset;
    }
```

Figure 4.3 shows an example of the access control mechanism mentioned. At time slot 0, processor 1 issues write operation a in memory bank 1, as shown in Figure 4.3a. Since bank 1 is the first bank in the write operation, the offset is inserted into ATT 1.

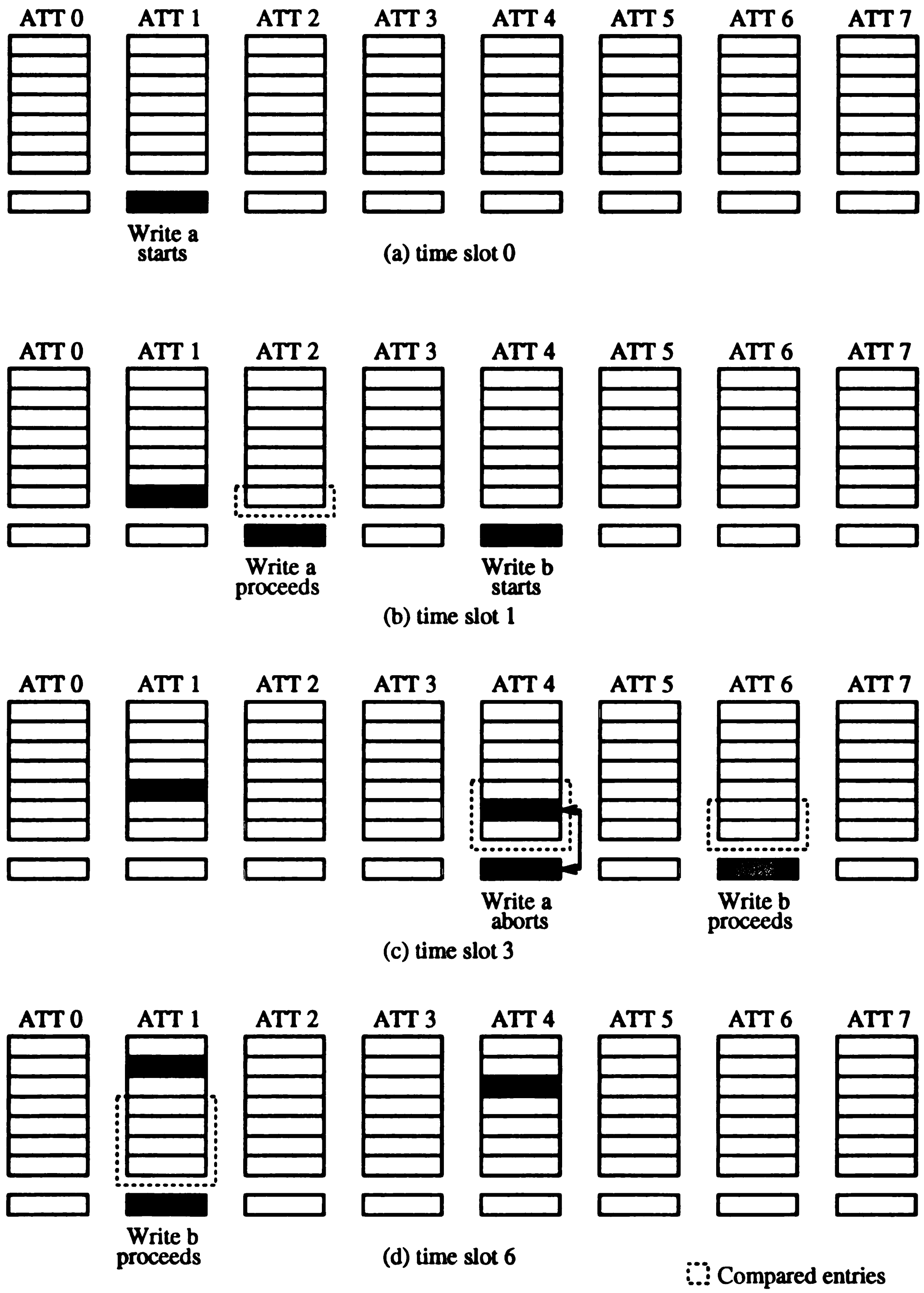


Figure 4.3. Write access control with address tracking.

At time 1, the offset in ATT 1 is shifted by one location, and processor 3 issues write operation *b* in bank 4, as presented in Figure 4.3*b*. Assume operations *a* and *b* access the same block. At time slot 3, operation *a* is aborted by memory bank 4 due to the detected same-address write operation *b*, which is issued later than operation *a*. This is shown in Figure 4.3*c*. Figure 4.3*d* presents the status of the ATTs at time slot 6. Operation *b* is updating data in memory bank 1. The operation is not aborted since the offset *a* entry has already been shifted off the comparing set of operation *b*. Operation *b* will complete if there is no other same-address write operation issued later conflicting with it.

Figure 4.4 shows how this access control mechanism solves the simultaneous same-address write problem. Operations *c* and *d* are simultaneous same-address write accesses issued in memory banks 1 and 5, respectively. When operation *c* tries to make its fifth update in memory bank 5, its offset is first compared with the first four entries of ATT 5. It is aborted because of the detection of access *d*. At the same time slot, operation *d* is trying to make its fifth update in memory bank 1. Since it has already updated memory bank 0, its offset is only compared with the first three entries of ATT 1. Since no same-address write access is detected within the subset, access *d* can proceed.

The access control of read operations is much simpler than that of write operations, since read operations do not interfere with each other. To prevent a read operation from being interfered by possible same-address write operations, the accessing address of the read operation needs to be compared with all the entries in the ATT of each memory bank. Whenever a same-address write operation is detected, the read operation simply restarts its access cycle from the current memory bank. This ensures the entire block read is from the same version. Figure 4.5 shows an example of a restarted read operation. Read operation *e* starts in memory bank 1, and restarts in bank 3 due to the detection of write operation *f*. A read operation may

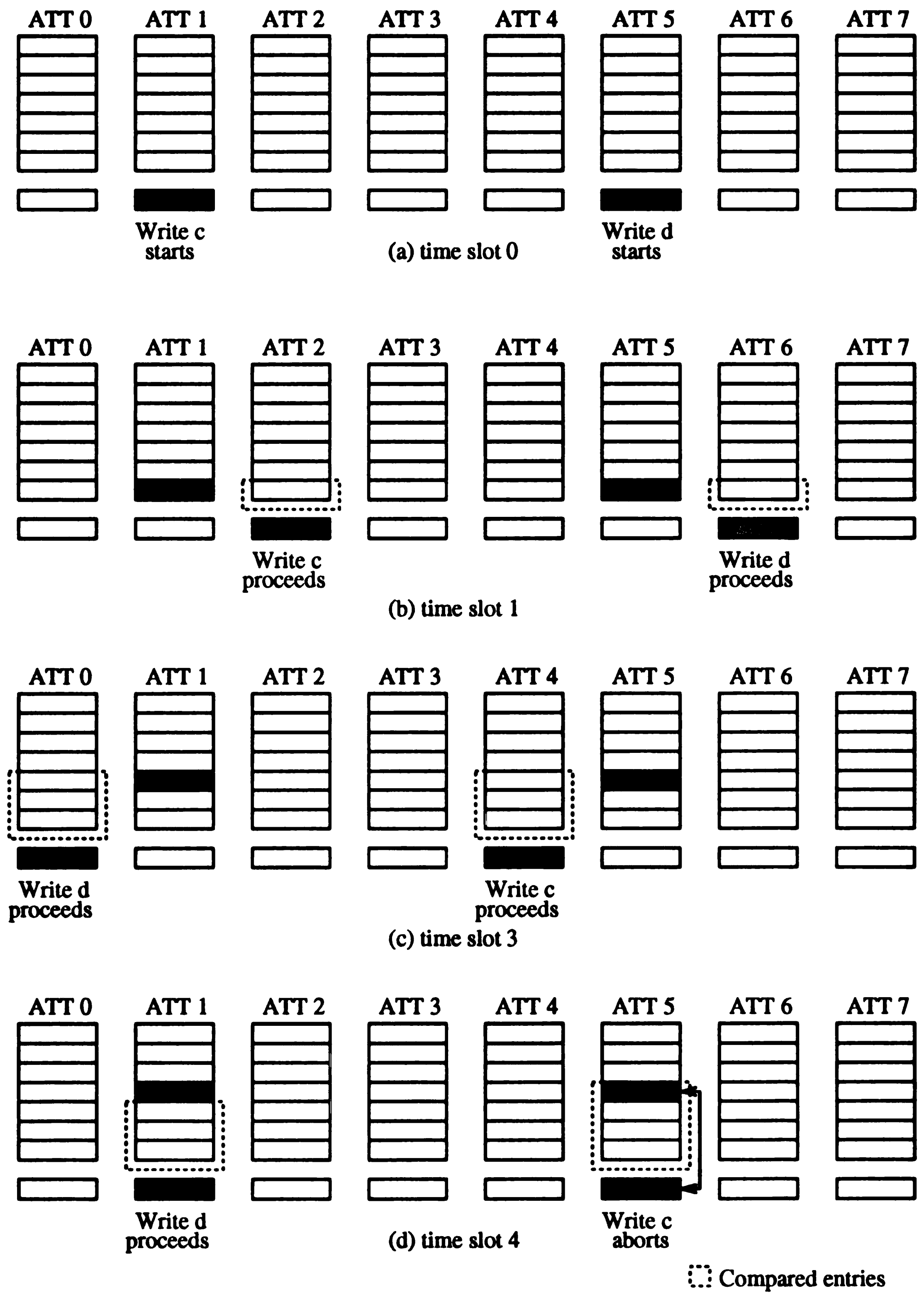


Figure 4.4. Solving simultaneous same-address write problem.

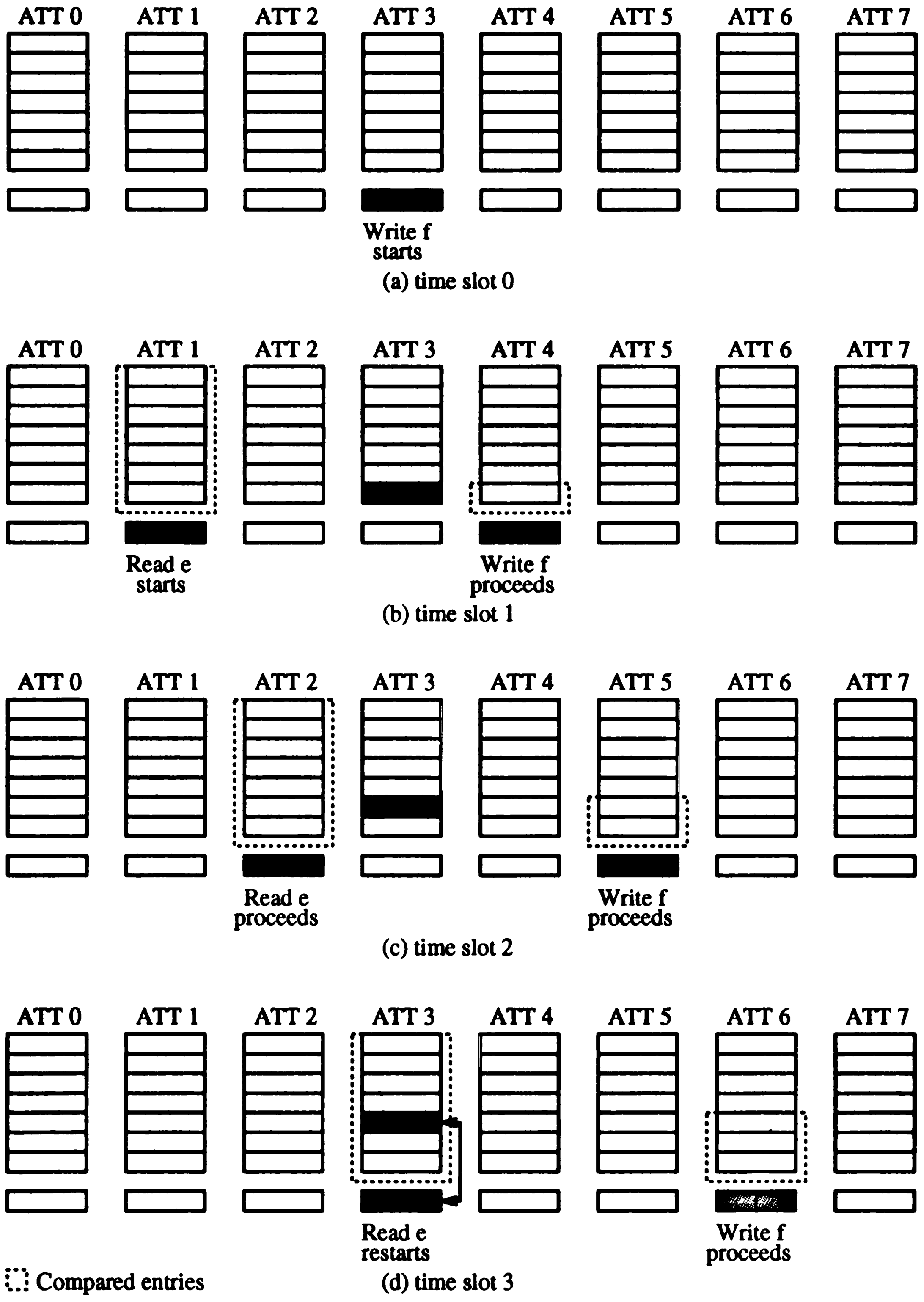


Figure 4.5. Read access control with address tracking.

be restarted several times due to multiple same-address write operations detected in different memory banks.

The access control mechanism does not introduce additional overhead to memory access time, since the ATTs are implemented with associative memories, and comparison between address offsets can be done concurrently with address decoding. The mechanism of address tracking can be extended to support atomic operations, which is described below.

4.2 Implementing Atomic Operations with Address Tracking

In a multiprocessor, high performance computation is achieved by incorporating concurrent processes running on multiple processors. At some points, these processes may need to be synchronized with each other for events like shared variable accesses and critical section executions. Most hardware architectures support primitives for the purpose of process synchronization. Based on these primitives, a number of higher level synchronization paradigms, e.g., locking semaphores and monitors, can be implemented. The synchronization primitives used in multiprocessors are basically some type of atomic operations such as "fetch-and-add" in the NYU Ultracomputer and the IBM RP3 and "load-then-store" in BBN Butterfly. Like other architectures, the CFM architecture also supports atomic memory operations that can be used for higher level process synchronization. This section presents how atomic operations can be implemented using the address tracking technique mentioned above.

4.2.1 The Atomic Swap Operation

The atomic swap operation exchanges the contents of a processor register with the contents of a block. It is composed of a read phase and a write phase executing a read operation and a write operation, respectively. The operations execute sequentially and atomically on the same block. In order to implement the atomic swap operation, the priority of competing write operations defined earlier have to be changed. Among competing same-address write operations, the first issued write operation, instead of the latest issued one, is now defined to have the highest priority. This means that a write operation detects competing operations issued earlier, instead of later, than itself by comparing a different subset of ATTs.

Furthermore, the simple access control mentioned in Section 4.1 requires a write operation to abort when detecting other competing operations. Here, a more complicated rule is needed. When a simple write operation detects the write of a swap operation, the simple write operation restarts instead of aborts. When the write of a swap operation detects another write operation, no matter it is a simple write or the write of a swap operation, the entire swap operation restarts. Assuming a and b are two swap operations accessing the same block, consider the following situations.

- The read operation of a detects the write operation of b: Operation a restarts.
- The write operation of a detects the write operation of b: Operation a restarts.
- No same-address write operation detected: Operation a and b finish in a certain order without extra delay.

It can be seen that each of the situations preserves the atomicity of both operations a and b. Though there can be an overlap in their executions, the result is identical to one of their possible sequential execution orders. Figure 4.6a, b, and c present the situations listed above. Figure 4.6d demonstrates that a simple write operation

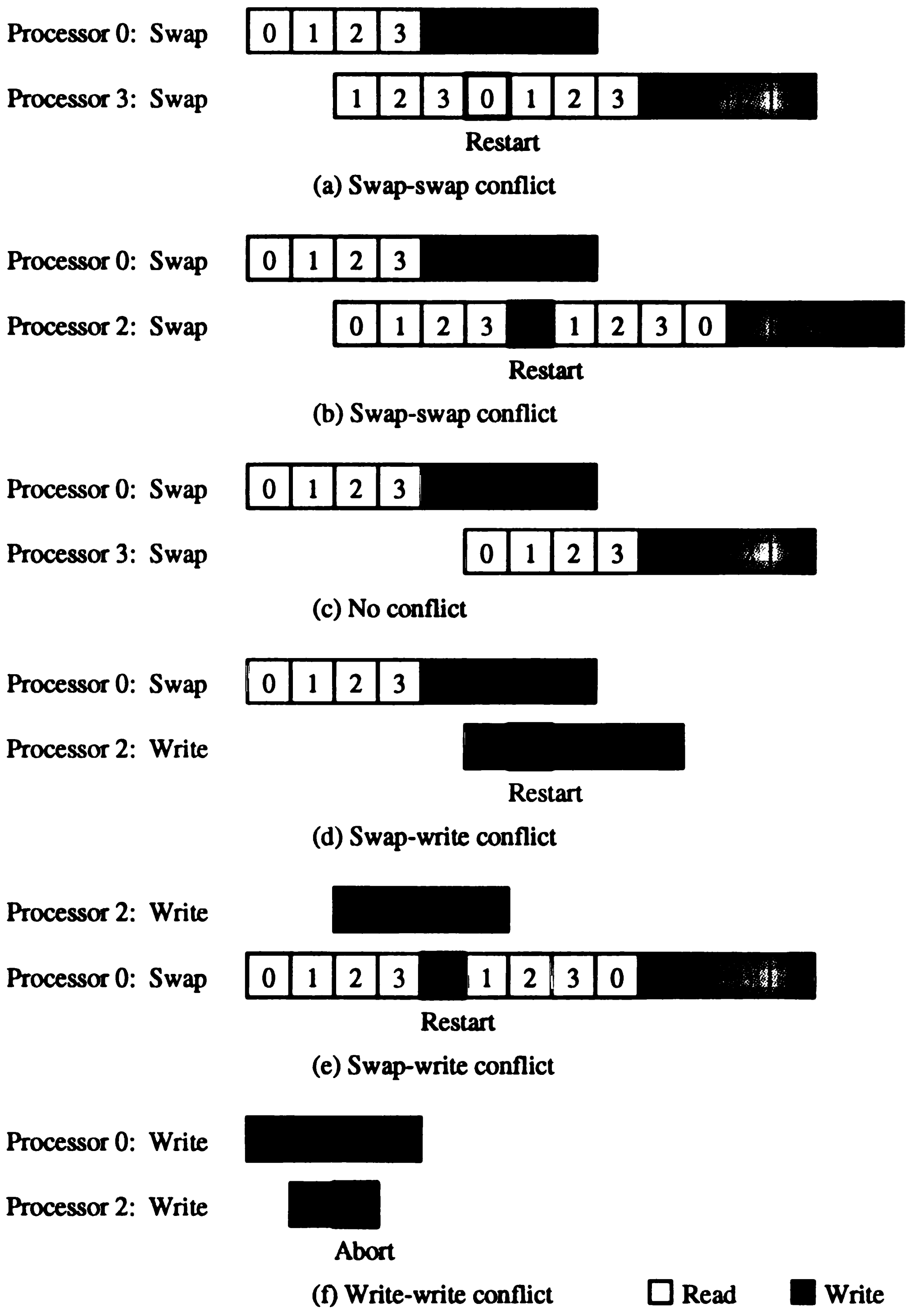


Figure 4.6. Interaction among swap operations and write operations.

restarts after detecting the write of a swap operation. A swap operation also restarts if it detects a simple write operation, as shown in Figure 4.6e. Figure 4.6f illustrates an example of write-write conflict. A simple write operation aborts after detecting another simple write operation.

Likewise, other atomic operations such as "read-modify-write" can also be implemented. When the read access of an atomic operation is proceeding, the processor that issued the atomic operation can modify the retrieved data for the write access in a pipelined fashion. The read and write accesses of the atomic operation can proceed continuously without extra delay.

4.2.2 Implementing Lock/Unlock with Swap

The atomic swap operation can be used to implement process synchronization paradigms such as *lock/unlock*. In traditional systems, lock/unlock can be implemented with two protocols, namely, ***busy-waiting*** and ***passive-wakeup***. In the first protocol, a process waiting for a lock repeatedly tests the lock until it successfully obtains the lock. This scheme is suitable for finer grain parallel computation because of its low latency, but creates heavy traffic in the interconnection network and high contention in the shared memory, which degrades overall system performance. Moreover, the hot spot problem may occur due to a number of processors requesting a lock. Alternatively, the second protocol forces the process waiting for a lock to sleep until the process holding the lock wakes it up when unlocking the lock. This scheme does not create heavy load in the interconnection network and the shared memory; however, it has higher latency and is unsuitable for fine grain parallel computation.

The lock/unlock paradigm can be implemented in the CFM architecture using the busy-waiting scheme without creating network and memory contention.

```
lock(int &s)
    {
        while(swap(1, s))
            while(s);
        }

    unlock(int &s)
        {
        s = 0;
        }
```

Assume the function, **swap**, atomically stores its first argument to the location specified by its second argument and returns the old contents of the second argument. Since the write operation and the swap operation have higher priority than the read operation, a processor repeatedly checking a lock does not delay the swap operation issued by the process holding the lock.

As can be seen, the atomic operations introduced operate on an entire block instead of a single bit or word, which introduces longer latency. The latency, however, is much less than that of passive-wakeup protocol. More importantly, this implementation eliminates the high network traffic and overhead problem caused by conventional busy-waiting protocol. The hot spot problem can never occur in the interconnection network or the shared memory. Furthermore, atomic operations on blocks can be used to implement multiple locks at the same time. For example, some related lock variables can be allocated to different bits or bytes in the same block. A processor can then acquire either all the locks or none. This is a powerful support for higher level synchronization paradigms such as *resource binding*, which will be presented in Chapter 6.

CHAPTER 5

CACHE COHERENCE AND SYNCHRONIZATION

As the gap between processor speed and memory speed grows, memory access latency becomes a very important factor in designing high performance architectures. In a shared-memory multiprocessor, due to the increased physical dimensions, network overhead, and hardware complexity, memory access latency is expected to be high. In current multiprocessors, memory access latency can be tens of CPU cycles [32]. In the future, access latency can be expected to exceed 100 cycles [33]. Large memory latency can quickly offset any performance gains expected from the use of parallelism. The cache technique allows data to be cached and significantly reduces the memory latency observed by processors. On a sequential machine, cache design is straightforward. While in a shared-memory multiprocessor, a memory location can have multiple copies stored in multiple caches, so that cache coherence becomes a very important issue. This chapter first reviews previous cache coherence schemes, then presents the new cache coherence protocol designed for the CFM architecture, and also discusses the synchronization supports and scalability issues.

5.1 Cache Coherence Protocol Review

A system of caches is said to be *coherent* if all copies of a memory location in multiple caches remain consistent when the contents of that memory location are modified [34]. A *cache coherence protocol* is the mechanism by which the coherence of the caches is maintained. Cache coherence protocols can basically be divided into two categories, namely, *snoopy cache protocols* and *directory-based cache protocols.*

5.1.1 Snoopy Cache Protocols

Snoopy cache protocols are so called because each cache in the system must listen to all coherence transactions to determine when consistency-related actions should take place for shared data. This requires an interconnection network with broadcast capability, usually a bus. Each cache stores the states of cache lines in its cache directory. The state transitions of each cache line are determined by the access requests from the local processor and the activities detected on the bus. The states associated with a cache line basically include *invalid*, *valid*, and *dirty*. The invalid state denotes that a cache line does not contain cached data block, while the valid state represents a cache line with a cached data block. The cached block may be shared and have multiple copies in other caches. When the content of a cache line is altered by write accesses from its local processor, the cache line is in the dirty state. The dirty state is exclusive, which means no more than one cache can have a dirty copy of a data block.

There are two types of schemes for maintaining cache coherence when one processor writes to a cached block that exists in other caches, namely, *invalidation* schemes and *write update* schemes. The invalidation schemes require a write to a cache line to invalidate copies of the cached data in other caches [35, 36]. If writes to cached data always cause invalidations, heavy traffic and high overhead can be created on the interconnection network with even a few participating processors [37]. The *write once* protocol developed by Goodman reduces the invalidation rate [35]. This scheme requires only the first write to a cached block to update main memory, which is then detected by other caches containing copies of the cached block as a cue to invalidate their copies. The write update schemes require all copies of a cached data block to be updated upon a write to one of the copies. The unit of data transfer and update in the schemes is a word instead of an entire cached block. The performance of both types of schemes is sensitive to the data sharing pattern.

The major disadvantage of snoopy cache protocols is in its scalability. Since the protocols rely on a broadcast interconnection network, the number of processors that can be supported is limited by the network bandwidth. Snoopy cache protocols are usually implemented in bus-based shared-memory multiprocessor systems, which can connect probably no more than 20 processors. Some shared-memory multiprocessors with large numbers of processors, such as BBN GP1000 [4], do not provide caches. Others, such as the RP3 [7], provide caches that must be kept coherent by software.

5.1.2 Directory-Based Cache Protocols

Directory-based cache protocols are more suitable for large-scale cache-coherent multiprocessors, as they do not rely on interconnection networks with broadcast capability. The major advantage directory schemes have over snoopy cache protocols is that the locations of the caches that have a copy of a shared data item are known [38]. This means that a broadcast is not required to find all the copies. Upon write invalidation or write update, individual messages can be sent to the caches containing copies of the modified data block. Since these are point-to-point messages, they can be easily sent over any arbitrary interconnection network, instead of just a bus. Freedom from a broadcast requirement contributes to the high scalability of directory-based cache protocols. There are various directory-based protocols being developed or proposed, each of which has a different memory overhead and hardware complexity.

Tang's scheme [39] requires a dirty bit maintained by each cache line as well as a central directory kept in memory. The central directory contains a copy of all the tags and dirty bits in each cache. It is checked on a read miss to see if the requested block is dirty in another cache. If so, consistency is maintained by flushing the dirty copy back to memory before data is supplied. In this case, the state of the cache line is set to valid in the central directory. The central directory is also updated to indicate that the requesting cache now has a valid copy of the data. On a write miss, if the intended data block has a dirty copy in another cache, it is first copied back to memory from the remote cache before data is supplied. If it has clean copies in other caches, they are first invalidated. The central directory is updated to indicate that the requesting cache now has a dirty copy of the data. On a write hit, the cache's dirty bit is checked. If it is dirty already, no memory access is required. If the block is clean, the central directory is informed to invalidate all other copies of the block.

Censier and Feautrier [40] proposed a similar cache coherence protocol. The protocol requires a dirty bit and a number of valid bits equal to the number of caches to be associated with each memory block. This allows direct accesses to the central directory using the address supplied to the main memory. Each valid bit of a memory block indicates whether a copy of the block exists in the corresponding cache. As can be seen, this cache coherence protocol has high storage overhead in maintaining the central directory. When the number of processors increases, the valid bit vector size associated with each memory block as well as the total storage requirement also increase.

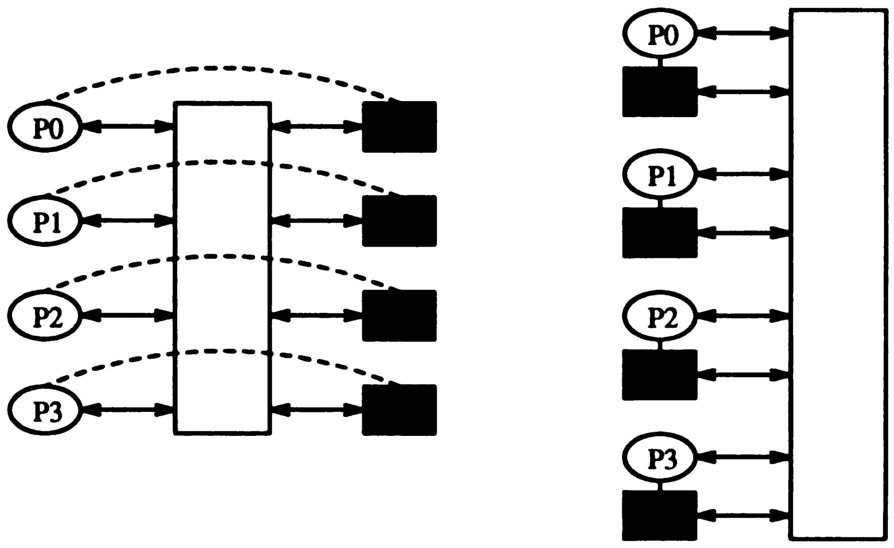
There are various other schemes proposed with reduced storage overhead. Some use a limited number of pointers associated with each memory block to indicate where possible cached copies of the memory block may reside [38, 41]. Others maintain links among cached memory block copies, so that the pointers associated with each memory block can be further reduced [42, 43]. The following sections present the CFM cache coherence protocol. The protocol preserves the low storage overhead of snoopy cache protocols, while offering the high scalability of directory-based protocols. The CFM cache coherence protocol is an invalidation-based protocol with write-back. Efficient synchronization supports can also be implemented with the protocol.

5.2 The CFM Cache Protocol

The discussion in the previous section shows that snoopy cache protocols have the advantages of simplicity, ease of implementation, and superior performance. Their major disadvantage is the strict limitation of scalability. While directory-based cache protocols scale well to larger configurations, they also require more complex hardware and memory overhead. Without broadcast capability, directory-based protocols suffer long communication latency caused by point-to-point invalidation messages and acknowledgements. This section describes how the nature of the CFM architecture allows the CFM cache protocol to accommodate the advantages of both snoopy protocols and directory-based protocols. First, a slight modification to the CFM architecture is presented. The invalidation-based write-back protocol used in the architecture is then demonstrated. In order to clearly explain how cache consistency is maintained with the scheme, each of the primitive operations used in the protocol is also defined in this section. Furthermore, this section gives a brief description of the access control among concurrent operations.

5.2.1 Hardware Configuration

As described in previous chapters, the CFM architecture relies on synchronous interconnection networks for data transfer in a pipelined fashion, which, unlike the bus, do not support hardware broadcast. The CFM interconnection, however, has the

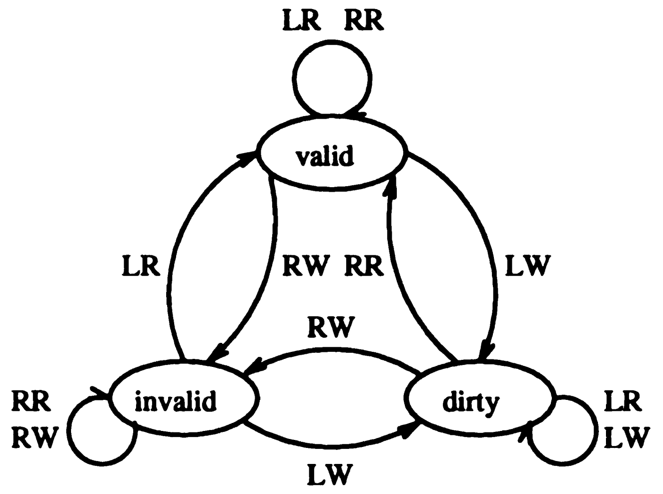


(a) Wrap-around control connection (b) Processor-memory coupling

Figure 5.1. Wrap-around control connection and processor-memory coupling.

characteristic that all memory banks are visited in each memory access. This characteristic allows the cache coherence information of each memory access to be broadcast among memory banks. Cache line state transitions similar to those of snoopy protocols can be determined in each memory bank by the broadcast information. In order for the cache line state information to be usable by processors in maintaining cache coherence, it is desirable for the processors to share cache state information with the memory banks. This is achieved by implementing an additional control connection associating each processor with a memory bank, as shown in Figure 5.1a.

The CFM architecture can be redrawn as Figure 5.1b, which shows each processor associated with a memory bank. This is unlike a message-passing distributed-memory system where each processor has a local memory and remote accesses are accomplished through message-passing. In the modified CFM architecture, processors access memory banks only through the interconnection network in a synchronous and pipelined fashion. There is no direct data transfer between a processor and its associated memory bank. The control connection between the processor and memory bank in an



LR	Local Read Request	RR	Remote Read Request
LW	Local Write Request	RW	Remote Write Request

Figure 5.2. Invalidation-based write-back protocol.

associated pair simply represents the sharing of their cache directory, which contains coherence information. Each shared directory entry contains a *state* field and a *tag*. All the CFM caches are assumed to be direct-mapped throughout this dissertation, although other approaches can also be used.

5.2.2 Invalidation-Based Write-Back Protocol

The CFM cache protocol is an invalidation-based protocol with write-back policy. Each cache line can be in one of three states: invalid, valid, and dirty, as mentioned in Section 5.1.1. A valid data block can be shared and can reside in many caches; however, a dirty block cannot exist in more than one cache. This means that the dirty state is exclusive, for there can be at most one dirty copy of a data block in all the caches of a CFM system. Figure 5.2 shows the state transitions of an invalidation-based write-back protocol.

An invalid cache line becomes valid when a data block is retrieved from memory upon a read request issued by the local processor. As long as the cache line remains valid, subsequent local read requests can be served by the local cache without imposing any memory accesses. A processor issuing a write request must first obtain the exclusive ownership of the target data block. This is accomplished by invalidating all remote copies of the data block and changing the local cache copy of the data block to be in the dirty state. All subsequent write requests issued by the local processor only update the local dirty copy of the data block. The dirty copy of the data block is written back to memory when it is replaced or when an access to the data block is requested by a remote processor. A remote read request causes the dirty copy to become valid after the updated data is written back to memory, while a remote write request causes it to become invalid. Write-back cache protocols perform better than write-through protocols in the sense that it does not require all updates of a data block to be written back to memory. As a consequence, memory and interconnection network overhead can be reduced.

Three primitive operations, *read*, *read-invalidate*, and *write-back*, are used to implement the CFM protocol. Read operations retrieve data from memory to caches. Read-invalidate operations are similar to read operations, however, they also obtain exclusive ownership of data blocks by invalidating remote cache copies of the data blocks, if any. Write-back operations flush updated data blocks from caches back to memory. Table 5.1 shows cache hits and misses under different circumstances and their corresponding actions to be taken. On a read miss, a memory read operation is issued. In case the requested data block has a dirty copy in a remote cache, this read operation will trigger the corresponding remote processor to write-back the data block before the local processor can retrieve the block. On a write hit, if the cache line is already dirty, no memory access is required. In case the cache line is valid, a read-invalidate operation is issued to obtain the exclusive ownership of the data block. On a write miss, likewise, a read-invalidate operation is issued. If the data block is dirty and owned by a remote cache, this operation will trigger the corresponding remote processor to write-back the data block.

Table 5.1. Cache hits, misses, and corresponding actions.

Events	Local	Remote	Final	Actions
Read hit	v	v or i	v	no memory access
	d	i	d	no memory access
Read miss	i	v or i	v	read
		d	v	read (trigger remote write-back)
Write hit	v	v or i	d	read-invalidate
	d	i	d	no memory access
Write miss	i	v or i	d	read-invalidate
		d	d	read-invalidate (trigger remote write-back)

5.2.3 Primitive Operations

As mentioned above, the CFM cache protocol is implemented with three primitive operations, read, read-invalidate, and write-back. All the primitive operations follow the block access mechanism described in Section 3.1.1. Since all memory banks are visited within each operation, the cache directory of the processor coupled with each memory bank can be checked to determine actions to be taken. Invalidations imposed by the read-invalidate operation can also be completed in remote caches when each of the memory banks is visited. Unlike snoopy protocols, the state checking and transition do not rely on a broadcast network. The latency of invalidating remote caches is much lower than that of other directory-based protocols, since they are achieved synchronously in a pipelined fashion. With autonomous access control among the operations, which will be discussed later, no acknowledgement message is required for invalidations. This is unlike other directory-based protocols, such as the DASH protocol [32, 44], where high network overhead is introduced due to point-to-point invalidation messages and required acknowledgements.

Read operations, when visiting each memory bank, check the cache directory of the corresponding processor cache. Upon detecting a dirty copy of the requested data block, a read operation triggers a write-back from the remote cache containing the dirty copy. During the remote write-back, the read operation can keep retrying until successful retrieval of the block from memory. This does not impose extra overhead on the interconnection network, since there is no contention problem in the CFM architecture. In a system with software prefetching [27, 28, 29, 30], however, it may be desirable to delay the retry for other possible memory accesses. Read operations do not alter cache line states in remote caches.

Read-invalidate operations are required for obtaining exclusive ownership of data blocks. When visiting the memory banks, a read-invalidate operation invalidates any valid copies of the requested data block. If a dirty copy of the data block is detected in a remote cache, the data block is currently owned by that cache. Under such circumstances, like read operations, the read-invalidate operation requests the remote cache to write-back the block and release the ownership. The operation retries, with or without delay, until it successfully obtains the ownership.

Write-back operations are issued either when requested by remote processors or when a dirty cache line is replaced. Since only the exclusive owner of a data block can issue a write-back operation, no other cache can contain any copy of the data block before and during the write-back operation. Consequently, no state checking and transition need to be applied in other caches. After the write-back operation completes, the local cache line is set to the valid state, which may later be changed to invalid if the write-back was requested by a read-invalidate from a remote processor.

5.2.4 Autonomous Access Control

Race conditions similar to the problem mentioned in Section 4.1.1 may occur when there are multiple memory operations proceeding concurrently. To primitive operations, unlike the situation stated in Section 4.1.1, the major interference resides in read-invalidate operations rather than write-back operations. Since there can be at most one dirty copy of a data block, which is exclusively owned by a cache, there cannot be more than one processor executing write-back on the same data block. No interference between two write-back operations can ever happen. Read-invalidate operations, however, may conflict with each other when competing for the exclusive ownership of a data block. Furthermore, a read-invalidate on a data block may interfere with a write-back operation updating the same block. In this case, the read-invalidate operation must abort and retry later.

Table 5.2. Access control among primitive operations.

	Read	Read-invalidate	Write-back
Read	—	retry later	retry later
Read-invalidate	—	retry later	retry later
Write-back	—	—	—

Table 5.2 demonstrates the action to be taken when each operation in the first column detects each operation in the top row when accessing the same data block. It can be seen that write-back has the highest priority, read-invalidate the next, and read the lowest. When a read operation detects a read-invalidate, it may retry after a delay. But if the read operation detects a write-back, it may retry immediately, as the data block may become available right after the write-back completes. For a read-invalidate operation, the same actions apply.

The problem left is how a primitive operation detects other concurrent operations accessing the same data block. Obviously, this can be achieved by the address tracking technique introduced in Section 4.1.2. Each read-invalidate or write-back operation inserts the address offset of the intended data block into an ATT when visiting its first memory bank. Read and read-invalidate operations detect only other read-invalidate and write-back operations in ATTs, while write-back operations do not detect any other operation. When multiple read-invalidate operations are competing for the exclusive ownership of a single data block, the address tracking scheme guarantees exactly one of them obtaining the ownership.

While the address tracking scheme implements the memory access control presented in Table 5.2, it is not the only solution to race conditions. With the processor-memory coupling in the CFM architecture, an ongoing primitive operation as well as its issued time slot can be recorded in the issuing processor. This information can be checked by operations from other processors to detect interfering conditions. With a proper priority definition similar to that of the address tracking scheme, correct memory access control can be autonomously coordinated among primitive operations. The hardware cost of this scheme is reduced in comparison to that of the address tracking scheme, while it offers the same access control.

Figure 5.3 demonstrates possible interactions between a write-back operation and a read-invalidate operation. At time slot 0, processor 0 issues a write-back operation to update a memory block. Processor 2 issues a read-invalidate at the same time slot trying to obtain the ownership of the same memory block. Both the operations proceed at time slot 1 without any conflict. At time slot 2, processor 2 detects the same-address write-back operation issued by processor 0. As a consequence, processor 2 has to abort the current execution and to retry later. Processor 0 finishes its write-back operation at time slot 3. Its cache copy of the memory block changes from the dirty state to the valid state. Later, when processor 2 finally completes its

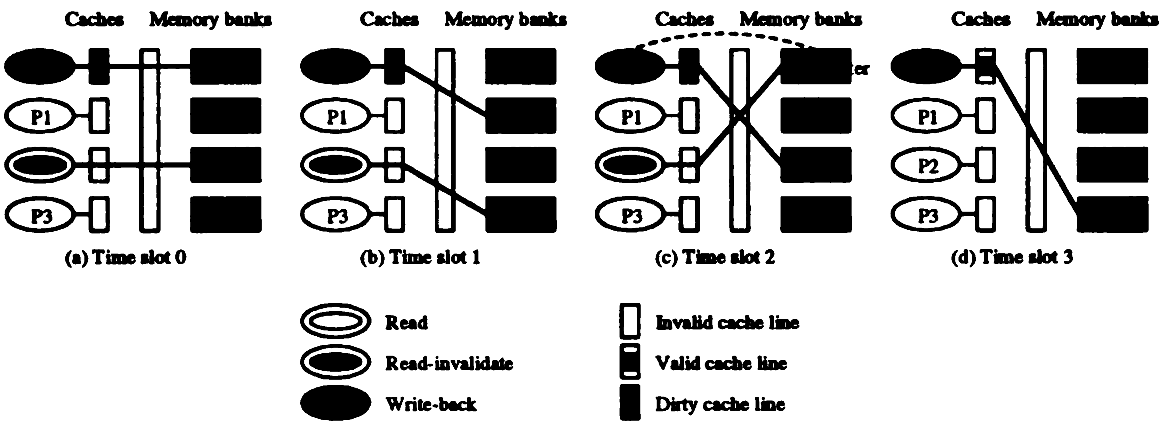


Figure 5.3. Access control between a write-back and a read-invalidate.

read-invalidate operation, the cache copy of processor 0 is invalidated, which is not shown in the figure. Interactions between same-address read-invalidate operations are similar to that of a write-back and a read-invalidate operations. Priority, however, needs to be properly defined so that exactly one among several concurrent read-invalidate operations obtains the ownership of the requested memory block.

5.3 Synchronization Supports

The CFM architecture supports higher level synchronization with atomic operations such as swap and "read-modify-write". Section 4.2.1 demonstrates an implementation of the atomic swap operation using the address tracking technique. While in this section, an efficient implementation of atomic "read-modify-write" and other synchronization operations using the CFM cache protocol is introduced. The synchronization operations can be used to implement higher level synchronization mechanisms such as atomic multiple lock/unlock.

5.3.1 Synchronization Operations

The implementation of atomic synchronization operations using the CFM cache protocol is quite straightforward. As mentioned, before a processor can update a data block, it must first obtain the exclusive ownership of that block. This is achieved by issuing a read-invalidate operation, which also retrieves the data block from memory if it is not yet in the local cache. By modifying the data block and flushing it back to memory with a write-back operation, an atomic read-modify-write is completed. Remotely triggered write-back of this data block is disabled during the modification phase to prevent premature write-back. The read-modify-write operation is atomic, since no other processor can read or update the data block during the period that it is exclusively owned by the local processor. Atomic operations such as swap, test-and-set, and fetch-and-add are special cases of the atomic read-modify-write operation.

Weak consistency [22] can be implemented with the atomic synchronization operations mentioned above. The weak consistency model requires the following conditions to be satisfied.

1. All previously issued synchronization operations must be performed before a synchronization operation is allowed to perform.

2. All previously issued ordinary read or write operations must be performed before a synchronization operation is allowed to perform.

3. All previously issued synchronization operations must be performed before an ordinary read or write operation is allowed to perform.

Conditions 1 and 2 are satisfied by allowing a synchronization operation to execute only after all previous read and read-invalidate operations are performed and all previous local cache accesses are completed. The synchronization operation does not have to wait for local dirty cache lines to be written-back. This is because previous write operations are considered performed once the issuing processor has obtained the ownerships of the targeting blocks and completed modifications on their local cache copies. Condition 3 can be satisfied by enforcing the read-invalidate and modification phases of a synchronization operation to be blocking. The CFM cache coherence protocol, not only efficiently supports synchronization operations and weak consistency, but requires much less states and simpler hardware than other schemes that support synchronization with cache protocols, such as [44].

5.3.2 Simple Lock/Unlock

Lock/unlock can be efficiently implemented with the atomic synchronization operations. As mentioned in Section 4.2.2, because of the conflict-free nature of the CFM architecture, lock/unlock can be implemented using the busy-waiting scheme without creating overhead to other memory accesses. This is unlike many cache protocols in other architectures, where both acquiring and releasing a lock can create high network overhead due to cache invalidations. In order to reduce the overhead, some protocols implement the passive-wakeup scheme by maintaining a queue of lock requests. This requires higher hardware complexity and may increase lock/unlock latency.

Consider implementing lock/unlock with the same busy-waiting-on-read scheme as described in section 4.2.2 based on the CFM cache protocol. When a highly contended lock is released, the dirty block containing the lock is written back to memory. The waiting processors repeatedly reading the block can complete retrieving the new value concurrently without interfering with each other. After observing the new value of the lock, the processors start to compete for the exclusive ownership of the block with read-invalidate operations. Only one among them succeeds. The succeeding processor modifies the lock value and writes the block back to memory. The rest of the waiting processors then obtain the block sequentially and resume read-looping after checking the lock. This does not impose extra delay to the lock transfer, since the new lock holder can proceed without waiting for the waiting processors to become stable again. The entire lock transfer takes approximately the time required to complete three memory accesses: write-back by the original lock holder, read by the new lock holder, and read-invalidate by the new lock holder. The write-back of the block issued by the new lock holder can be overlapped with the critical section execution.

Figures 5.4*a*–*p* demonstrates an example of a lock transfer. Assume processor 0 is the original lock holder. Processors 1 and 3 continuously read their local cache copies of the lock variable while waiting for the lock. When processor 0 releases the lock, it first issues a read-invalidate operation, as shown in Figures 5.4*a*–*d*. This read-invalidate operation invalidates processors 1 and 3's local lock copies at time slots 1 and 3, respectively. As a consequence, both processors 1 and 3 encounter read-misses and issue read operations at time slot 2 and 0, respectively, as shown in Figures 5.4*c* and *e*. The read operations issued by processors 1 and 3 are aborted at time slots 3 and 1, respectively, after detecting the dirty copy of the lock variable in the cache of processor 0. After processor 0 completes the read-invalidate operation and resets the lock variable, it issues a write-back operation to update the main memory and release the lock, as shown in Figures 5.4*e*–*h*.

Processor 1 retries its read operation at time slot 0, as presented in Figures 5.4*i*–*l*. Processor 3 also retries its read operation at time slot 2, as shown in Figures 5.4*k*–*n*. Both the read operations retrieve a *free* lock value from memory. Processor 1, however, obtains the value first and issues a read-invalidate operation requesting the exclusive ownership of the lock variable, which is presented in Figure 5.4*m*–*p*. Although, processor 3 also issues a read-invalidate, as shown in Figure 5.4*n* and *o*. This operation is aborted when detecting the read-invalidate issued by processor 1. In Figure 5.4*p*, processor 1 obtains the ownership of the lock variable. It can then assign the value *locked* to the variable and issue a write-back operation to flush the

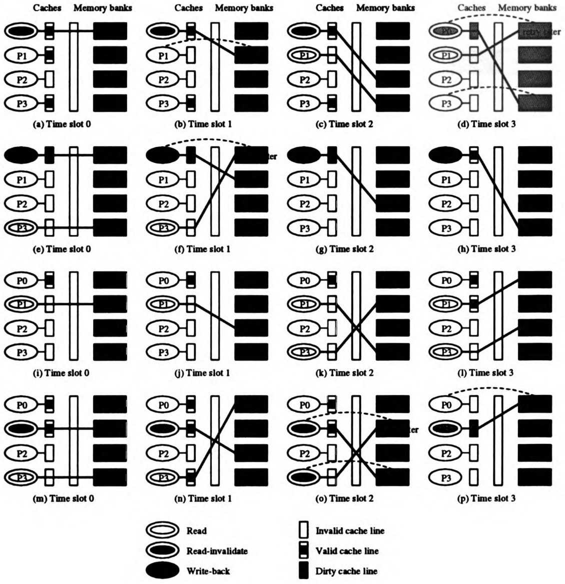


Figure 5.4. Lock transfer.

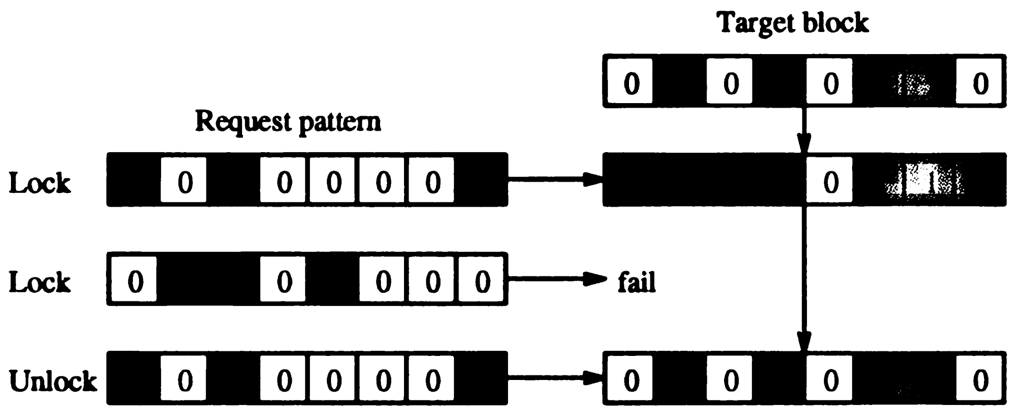


Figure 5.5. Atomic multiple lock/unlock.

new lock value. Processor 1 becomes the new lock holder. Processor 3 retries the read-invalidate operation and detects a the value *locked.* After writing back the value, it goes back to the read-looping status accessing its local lock variable copy.

5.3.3 Atomic Multiple Lock/Unlock

Atomic multiple lock/unlock support is very useful in many applications. It reduces the latency of repeatedly invoking several simple locks and eliminates some possible dead-lock situations. Chapter 6 introduces the *resource binding* parallel programming paradigm, which is an application of the efficient multiple lock support. Figure 5.5 demonstrates an example of an atomic multiple lock/unlock. Initially, the target block contains the bit map pattern **01010110** with "1" denoting *locked* and "0" denoting *free.* The first lock executes successfully with the request pattern **10100001**, which also sets the new value, **11110111**, to the target block. The second lock fails because some of its requested bit positions have been locked already. The unlock request release the bits locked by the first lock request.

Like the atomic operations mentioned in Section 4.2, a synchronization operation supported by the CFM cache protocol accesses an entire data block rather than a single word. This is a useful feature for implementing efficient atomic multiple lock/unlock. In order to implement multiple lock/unlock, a multiple test-and-set operation is defined. This operation atomically sets a subset of a target memory block with a bit map pattern if there is no common "1" at the corresponding bit positions of the target block and the pattern. A logic value is returned to indicate the action taken.

The operation first obtains the exclusive ownership of the target memory block and retrieves it into a cache line with a read-invalidate operation. The cache line and a bit map pattern is then compared. If there exists any "1" in the logic "and" of them, the ownership of the target block is released with a write-back operation. The multiple test-and-set operation returns a "true" value to indicate that the pattern cannot be set because of conflicts to the target block. If the logic "and" of the cache line and the pattern is all zero, the cache line is updated with the logic "or" of them and written back to memory. A "false" value is returned to indicate that the pattern has been successfully set to the target block. The atomicity of this operation is ensured by the read-invalidate operation. Remotely triggered write-back is disabled during the comparison and update to prevent premature write-back.

With the multiple test-and-set operation, atomic multiple lock/unlock can be efficiently implemented with the busy-waiting scheme, which is similar to the simple lock/unlock implementation. The latency of a multiple lock transfer is also similar to that of a simple lock.

```
multiple_lock(BLOCK &s, BIT_MAP p)
    {
    while(multiple_test_and_set(s, p))
        while(s&p);
    }

multiple_unlock(BLOCK &s, BIT_MAP p)
    {
        s = s&(~p);
        }
```

5.4 Scalability of the CFM Cache Protocol

The CFM cache protocol is superior to other directory-based protocols in the sense that it is scalable in a consistent and recursive fashion. This section introduces a hierarchical extension to the CFM architecture, which has multiple levels of caches. Accesses to all levels of caches as well as to memory banks are conflict-free. Based on this hierarchical CFM extension, a recursively-defined write-back cache coherence protocol is designed.

5.4.1 A Hierarchical CFM Architecture

Section 3.1 describes how the CFM architecture can be implemented by integrating processors and memory banks with a synchronous interconnection network. Section 5.2 adds to the CFM architecture control connections through processor-memory coupling for cache coherence control. This architecture can be defined as a conflict-free processor-memory cluster. A larger scale system can be implemented by integrating several conflict-free clusters as well as some global memory banks using a synchronous interconnection network similar to the simple CFM architecture. The memory banks within a cluster can be viewed as a second-level cache local to that cluster. The second-level caches together with the first-level caches and the global memory banks form a three-level memory hierarchy of the two-level hierarchical CFM architecture. The same concept can be applied recursively for implementing larger hierarchical CFM architectures with more processors and cache levels.

Figure 5.6 shows a two-level CFM architecture with a three-level memory hierarchy. Since the memory banks in a cluster form a second-level cache, for clearer

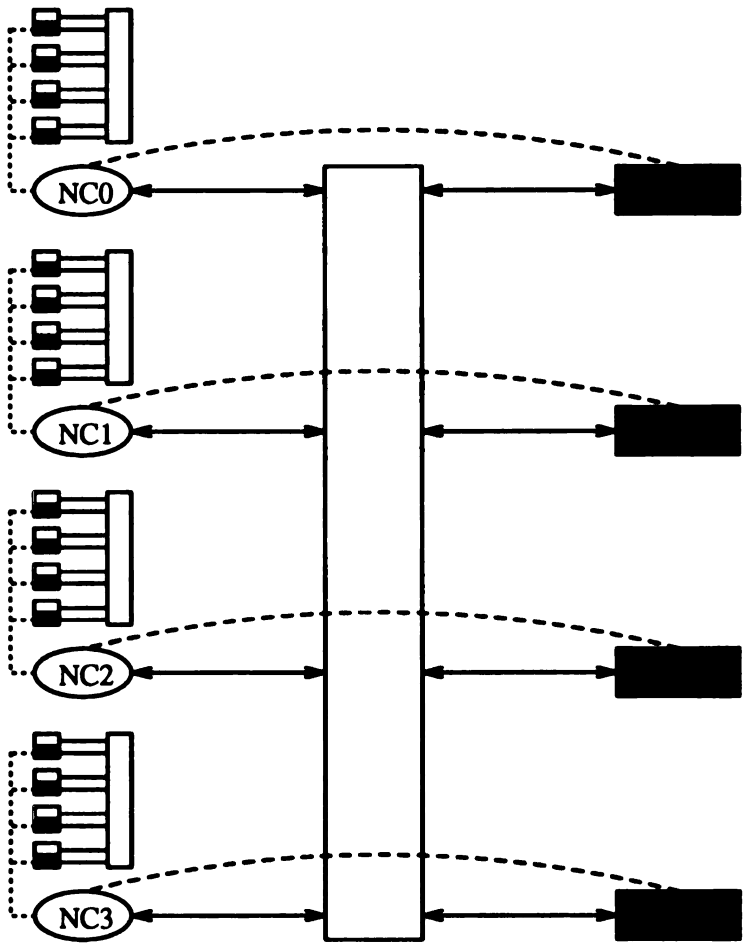


Figure 5.6. A hierarchical CFM architecture.

explanation, they are now called ***second-level cache banks*** or simply ***cache banks***. Each block from the cache banks is treated as a second-level cache line, which consists of words with the same address offset in all the cache banks of the cluster. The second-level cache of each cluster has a corresponding cache directory. As in a first-level cache directory, a second-level cache directory entry is composed of a state field and a tag for maintaining cache coherence. Each second-level cache line can also be in one of the three states: invalid, valid, and dirty.

Associated with each conflict-free cluster, there is a network controller. All cache misses in the second-level caches are handled by network controllers. Network controllers operate as pseudo processors accessing the global memory banks through the global synchronous interconnection network, which is similar to that of the simple CFM architecture. A network controller is directly connected to all the cache banks in its associated cluster. It fetches and flushes second-level cache lines from and to the global memory banks by using free time slots of the cache banks or by stealing time slots from the processors in the cluster. The network controller also maintains the second-level cache directory in its associated cluster. Like a processor in the simple CFM architecture, each network controller is coupled with a global memory bank by sharing its corresponding second-level cache directory, as represented by the dash lines in Figure 5.6.

5.4.2 Scalable Write-Back Cache Protocol

The CFM cache coherence protocol can be recursively defined and applied to multi-level CFM architectures. In order to clearly explain the scalability of the protocol, however, the two-level CFM architecture presented in Figure 5.6 is used as an example. Within each conflict-free cluster, first-level cache coherence is basically maintained with the same CFM cache protocol described in Section 5.2. The CFM cache protocol for a conflict-free cluster maintains consistency among first-level caches and its associated second-level cache rather than memory banks. Unlike a memory block, each second-level cache line has a directory entry corresponding to it. A primitive operation causing a state transition of a first-level cache line may also change the state of the corresponding second-level cache line. The coherence among second-level caches is also maintained with the same CFM cache protocol using the three primitive operations: read, read-invalidate, and write-back.

The state of a first-level cache line is closely related to the state of its corresponding second-level cache line. Table 5.3 shows the possible state combinations of a first-level cache line and its corresponding second-level cache line. A first-level cache line can be valid only if the corresponding second-level cache line is valid or dirty. A valid first-level cache line with a dirty second-level cache line denotes a data block exclusively owned by a network controller and shared among processors in its associated cluster. This can occur after a write-back from an owner processor to a second-level cache and before a write-back from the second-level cache to the global memory banks. A first-level cache line can be dirty only if its corresponding second-level cache line is dirty. This is because a network controller must first obtain the exclusive ownership of a data block before a processor in its associated cluster can become the exclusive owner of the block.

Table 5.3. States of corresponding cache lines.

First-level cache line	Second-level cache line
invalid	invalid, valid, dirty
valid	valid, dirty
dirty	dirty

On a first-level read miss, a read operation is issued to retrieve data from a second-level cache. If the requested data is not in the second-level cache, a second-level read miss occurs. In this case, the associated network controller issues a second-level read operation to retrieve the data from the global memory banks. If the data block has a dirty copy in a remote processor of another cluster, the second-level read operation first triggers the remote network controller to write-back the block. The remote network cluster in turn triggers the remote processor to write-back the block. After the first-level and second-level write-back from the remote processor and network controller, respectively, the data block can be retrieved by the local network controller to the local second-level cache. The second-level cache line containing the retrieved data becomes valid. The first-level read operation is then completed by retrieving data from the second-level cache line.

On a write hit, if the first-level cache line is already dirty, no memory access is required. If it is valid and the corresponding second-level cache line is dirty, then a read-invalidate operation is issued in the local cluster, just as in the case of the simple CFM architecture. If both the first-level and second-level cache lines are valid, the read-invalidate operation issued by the processor causes the local network controller to issue a second-level read-invalidate operation. This second-level read-invalidate operation causes all remote network controllers to invalidate the data block copies in their associated clusters. As in the read operation case, if the intended data block has a remote dirty copy, a remote write-back is first triggered before the invalidation.

5.4.3 Other Issues of the Scalable Cache Protocol

The CFM cache coherence protocol can be applied recursively to hierarchical CFM architectures with more levels of caches. The memory access latency of the worst cache miss situation increases logarithmically with the total number of processors, thus making the scalability of the CFM architecture and cache protocol very attractive. The hierarchical extension approach supports conflict-free accesses in each level of a CFM system. Contention, however, can still occur in a network controller when there are multiple requests from different processors or from its higher-level network controller. Each network controller must maintain a queue and serve the requests based on a properly defined priority such that no dead-lock situation can occur. For example, write-back needs to be served first if it is not disabled within a synchronization operation. Also, invalidation requests from the higher-level network controller has higher priority than read-invalidate requests from lower-level network controllers or processors in the associated cluster. This is to ensure that only one exclusive ownership of a data block is granted at any time. Table 5.4 shows a possible priority specification.

Table 5.4. Event priority in a network controller.

Priority	Requests
1	write-back
2	invalidation from the higher-level network controller
3	read-invalidate operation from the associated cluster
4	read

The contention in a network controller can be reduced by having its higher level cache or memory banks assigning it more than one free AT–space partition. With this approach, although the number of network controllers and clusters that can be connected is reduced, each network controller can serve more primitive operations concurrently.

Another problem of hierarchical CFM architectures is the interference between a network controller and its associated cluster, since both of them need to check the shared cache directory. One solution to the problem is to duplicate the shared cache directory. The same approach has been used in snoopy cache protocols for reducing processor-cache interference when snooping bus activities. Another solution to the problem is to assign a network controller a free AT–space partition in its associated cluster. In this case, the network controller is in the same position as other processors in the cluster, only it has the special purpose of handling data transfer between the cluster and higher level caches or memory banks and maintaining coherence among different clusters. There are more issues to be investigated concerning the hierarchical extension of the CFM architecture, which are interesting topics for future research.

5.4.4 Performance of Hierarchical CFM Architectures

As mentioned earlier, the worst-case cache miss penalty of a hierarchical CFM architecture increases logarithmically with the total number of processors. Since there is no simulation result available at this time, the following discussion will be based on comparisons of hierarchical CFM architectures and two other architecture designs, which are the DASH multiprocessor [32] and the KSR1 multiprocessor.

The DASH multiprocessor is a scalable shared-memory multiprocessor being developed at Stanford University's Computer Systems Laboratory. The DASH architecture consists of a set of clusters connected by a general interconnection network. Each cluster consists of a small number of high-performance processors and a portion of the shared memory connected by a bus. The architecture employs a directory-based cache coherence protocol, which is an invalidation-based ownership protocol similar to the CFM protocol described in Section 5.2.2. Inter-cluster memory accesses and invalidations are implemented by point-to-point message-passing. This not only supports high scalability of the architecture, but also enables the protocol to be independent from the interconnection network used. This, however, increases memory access latency. Furthermore, every invalidation message has to be acknowledged, which introduces more traffic on the interconnection network and, thus, higher probability of contention.

Table 5.5 presents a comparison of memory read latency on a two-level CFM architecture and a DASH multiprocessor [45]. Both systems consist of 16 processors organized in four clusters and use 16-byte cache lines. Assuming the CFM architecture has a memory bank cycle equal to two CPU cycles, a local cluster read operation (first-level read miss) takes nine CPU cycles to complete. A read access retrieving data from the global memory takes in average 27 CPU cycles. This includes activating the associated network controller to retrieve the requested data from the global memory banks and loading the data to the processor cache. As can be seen, the CFM architecture has shorter read latency than that of the DASH multiprocessor. Since there can be contention within a cluster and in the interconnection network of the DASH multiprocessor, its actual latency can be longer than the data presented in the table.

Table 5.5. Read latency of CFM and DASH.

Read Accesses	CFM	DASH
Retrieve from local cluster	9 cycles	29 cycles
Retrieve from global memory (remote cluster)	27 cycles	100 cycles
Retrieve from dirty remote	63 cycles	130 cycles

The KSR1 multiprocessor, developed by Kendall Square Research, is a highly parallel computer system designed to be scalable to thousands of processors. In the KSR1 multiprocessor, processors are arranged in a two-level hierarchy of unidirectional rings. Each processor has a local memory. The local memories of all the processors establish the KSR1 multiprocessor's ALLCACHE memory organization. Consistency is maintained by a mechanism similar to that of a virtual memory system.

Table 5.6 shows a comparison of memory read latency on a two-level CFM architecture and a KSR1 multiprocessor. Both systems consist of 1024 processors organized in 32 clusters (rings). The cache line size of both systems is 128 bytes. The CFM architecture has memory bank cycle equal to two CPU cycles. Since each processor in the KSR1 system has a local memory, for fair comparison, we assume each processor in the CFM architecture also has a local memory. First-level memory accesses transfer data between these local memories, instead of processor caches, and shared memory banks in the local cluster. Based on this assumption, intra-cluster accesses in the CFM architecture can be treated as local ring accesses in the KSR1 multiprocessor, and global memory accesses in the CFM architecture can be treated as global ring accesses in the KSR1 multiprocessor.

Table 5.6. Read latency of CFM and KSR1.

Read Accesses	CFM	KSR1
Retrieve from local cluster	65 cycles	175 cycles
Retrieve from global memory (remote cluster)	195 cycles	600 cycles

CHAPTER 6

THE RESOURCE BINDING PARALLEL PROGRAMMING PARADIGM

This chapter introduces the *resource binding* parallel programming paradigm, which can be efficiently implemented on the CFM architecture with the atomic multiple lock/unlock support. The paradigm is also portable to other parallel architectures. Writing parallel programs is much harder than writing sequential programs. Since inappropriate programming models in parallel systems may seriously decrease performance, ensuring appropriate programming models becomes an important issue. Furthermore, an efficient parallel program developed on a particular parallel system may not perform well on another. This raises the portability problem of parallel programming. There is a trade-off between performance and portability in various parallel architectures. To support parallel programming, several programming paradigms have been widely used or proposed for different parallel architectures. Some of these are for shared-memory systems, some are for distributed-memory systems, while others claim to be architecture independent. Here, we would like to define some conditions of a "good" parallel programming paradigm.

- Efficiency: The overhead should be low. The paradigm should be able to highly utilize resources offered by underlying hardware architectures.
- Flexibility: It should fit the needs of various applications and computation models.
- Portability: Programs developed using the paradigm should be easily portable to different parallel architectures, without losing much performance.
- Reliability: It should be easy to support mechanisms for preventing and detecting abnormal situations like deadlocks.
- Simplicity: The paradigm should be simple. It should enable a programmer to write parallel programs easily. The parallel programs designed using this paradigm should be easy to debug.

The *resource binding* mechanism presented in this chapter employs two simple primitives, `bind` and `unbind`, and supports a flexible environment for parallel programming in various computation models. Though the initial research of the scheme was done on shared-memory environments including the CFM architecture, it can be installed on message-passing distributed-memory systems, and still maintain good performance and high efficiency. Furthermore, mechanisms for detecting deadlock can be easily built into the resource binding paradigm, through which a more reliable environment can be supported.

6.1 Parallel Programming Paradigm Review

6.1.1 Locking Semaphores and Monitors

Semaphores are commonly used for synchronizing concurrently executing processes on shared-memory models. They are not only used on parallel computers, but have also been used on sequential machines for years. However, more considerations need to be taken into account when implementing semaphores on parallel computers. The scheme can be used for dependency control between concurrent processes, protection of shared resources, and constructions of many higher level computation models. A locking semaphore is one of the simplest semaphores. An example of using a locking semaphore for shared variable protection might look like:

```
int sh;          /* shared variable */
semaphore s_sh; /* semaphore for sh */
lock(s_sh);      /* lock the semaphore */
sh = sh+1;
unlock(s_sh);    /* release the lock */
```

The scheme is very simple and straightforward. However, programs using semaphores can be difficult to write, to debug and to read. As can be seen in the above example, the association between the shared variable **sh** and the semaphore **s_sh** is artificially enforced by the programmer. Physically, they have no relationship with each other. The programmer has the responsibility of relating them correctly. Imagine a parallel program with various shared resources and synchronization patterns. Handling such a program using semaphores may be extremely difficult. Another problem of the scheme is its restricted flexibility in handling large shared resources. Suppose a single semaphore is used for the protection of a large shared data structure, *i.e.*,, only one process can access the data structure at any time, the entire execution may become sequentialized. On the other hand, it is too expensive to keep one semaphore for each element in the large data structure. Finding the appropriate granularity can be difficult. We will show how easily resource binding handles this problem in the following sections.

Monitors are also used on shared-memory models for process synchronization. Unlike semaphores, a monitor encapsulates the state of a shared resource with the operations that manage it [46]. Programs using monitors are relatively easier to understand [47]. However, monitors, like semaphores, have the problem of finding the appropriate granularity, which reduces its flexibility in handling large shared resources. Because of its higher overhead, it may not be suitable for the scheme to handle very small granularity. Since parallel accesses to shared data protected by a monitor are completely inhibited, the program may be sequentialized severely.

6.1.2 Message Passing

Both semaphores and monitors are based on shared-memory models. For distributed-memory models, message passing is the most commonly used programming paradigm. In a message passing system, sending and receiving messages are the major operations representing interactions between concurrent processes. The operations may be either blocking or non-blocking. Process synchronization and interprocess communication can be achieved by using these operations. However, as in the case of semaphores, programmers are responsible for relating all the sending and receiving operations. For a typical parallel program using the message passing paradigm, the operations can be scattered throughout the entire program. As in programs using semaphores, message passing programs may be very difficult to write, to debug, and to understand.

6.1.3 Linda

So far, all the paradigms that have been discussed are dedicated either to shared-memory models or to distributed-memory models. Portability between different models is not supported in these mechanisms. Now, a discussion of a higher level parallel programming paradigm, *Linda*, which supports both shared-memory and distributed-memory models [48] follows. Instead of communicating with messages or through shared memory, Linda processes communicate with each other through a shared data space called *tuple space*. Tuple space acts like an associative memory or a mail box.

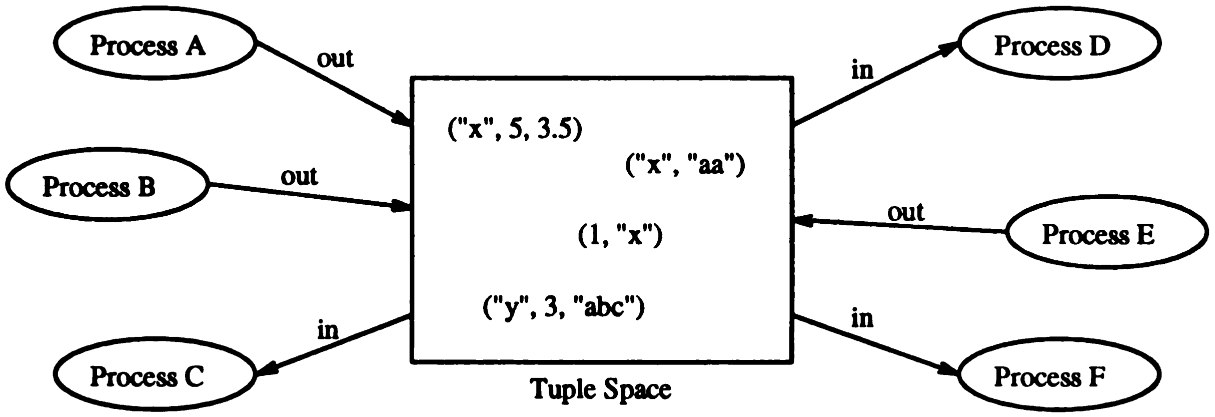


Figure 6.1. Linda processes and tuple space.

A tuple in the tuple space is an ordered collection of data items, which is identified by its key rather than its address. The scheme uses only four simple primitives for handling concurrent processes:

- `out` places a tuple in tuple space
- `in` matches a tuple and removes it from tuple space
- `rd` matches a tuple and returns a copy of it
- `eval` creates an active tuple (a process)

Figure 6.1 shows the relationship between the tuple space and Linda processes. A shared-memory model can be simulated by treating the tuple space as a shared memory space with variable names as keys of the tuples. Each tuple can be accessed by matching its key with the associated variable name of that tuple. Protection of shared variables is not a problem since before a tuple can be accessed, it needs to be matched with the `in` operation and removed from the tuple space. Thus no other process can access it before the process holding the tuple puts it back in the tuple space with the `out` operation. Message passing models can also be supported by Linda. With a process doing an `out` operation and another process doing an `in` operation, the sending-receiving pair of a message passing system can be simulated.

One of the most important features of Linda is the decoupling between the sender and receiver of a message in both time and space. This feature makes Linda programs easy to write, to debug and to understand. It is also shown that Linda is flexible enough to handle many different computation models. However, Linda suffers from its high overhead due to searches required when matching a tuple in the tuple space. This searching requirement results from the support of decoupling between senders and receivers. Linda does not require programmers to provide knowledge about the connection between senders and receivers, but, without this knowledge, searching becomes unavoidable when matching tuples. Although there are methods to reduce the requirement of searching [49], the overhead is still high, since its complexity is some order of the tuple space size, which can be very large.

Another problem with Linda might be its difficulty in efficiently detecting deadlocks. Because of the decoupling between senders and receivers, there is no way to identify by which processes a process is blocked. Since Linda matches a tuple by its key rather then its address, there is no way to predict whether or not a tuple with a certain key value will ever be put into the tuple space. As a consequence, it is difficult to define whether or not a process waiting for a tuple with the key value is deadlocked.

6.2 The Concept of Resource Binding

Resource binding is an efficient paradigm which supports an architecture independent environment for parallel programming. Though it offers users an interface based on shared-memory programming models, it can be implemented on distributed-memory machines with high efficiency. With two fundamental operations, `bind` and `unbind`, the mechanism helps in designing parallel programs that are easy to write and to debug. Like monitors, the resource binding paradigm encapsulates the state of a shared resource with the operations that manage it. Unlike monitors, the paradigm offers much higher flexibility in managing large shared data structures by dividing a shared data structure into shared data regions. Moreover, the paradigm handles process synchronization in the same manner as managing shared data structures. With the fundamental operations, process dependency and synchronization can be defined in a simple and consistent way. In order to support a reliable environment, deadlock detection algorithms can be easily built into the paradigm. This section first discusses various shared resources in parallel systems, then presents fundamental operations as well as the basic concept of the paradigm by using shared data structures as an example of shared resources.

6.2.1 Managing Shared Resources in Parallel Processing Systems

In a parallel processing environment, high performance computations are achieved by incorporating concurrent processes running on multiple physical processors. As mentioned, there are many causes which degrade overall performance of a parallel system. Contention for shared resources is one of the most important causes which tends to serialize concurrent processes. Efficient management of shared resources plays an important role in a high performance parallel system.

In shared-memory programming models, concurrent processes communicate with each other through shared variables (or data structures). In many situations, when more than one process attempts to write at the same shared memory location, exclusive accesses need to be enforced for data consistency. While in other situations, some processes may read the same shared variable, in which case, no exclusive control needs to be invoked. Frequently, the cases are not determined in advance. When using locking semaphores, programmers have to take the pessimistic approach and enforce exclusive control upon the shared data structure in any case. In contrast, resource binding preserves high parallelism by supporting the *multiple-read/single-write* style of accessing shared data structures, which will be described shortly.

Processes can also be treated as a type of shared resources, and be managed in the same way as shared data structures. Often, in a parallel processing environment, execution of a process depends on the statuses of other processes. That means a process is permitted to execute only when the processes on which it depends come to a certain combination of states. Barrier is a typical example where processes are allowed to continue only when all other processes have reached the barrier point. The resource binding paradigm defines process dependency by having each process bind the processes it depends on with a permission level. A process can execute only when its permission level is reached. Section 6.4 explains the process binding scheme in detail.

There are other types of shared resources like files that can be managed using the paradigm. File locking has long been used in controlling concurrent accesses to files shared by multiple processes. Resource binding supports the same accessibility as file locking by allowing multiple-read/single-write. Moreover, it enables us to handle files in secondary storages just like data structures in core. However, that topic is beyond the scope of this dissertation.

6.2.2 The Fundamental Operations—Bind and Unbind

There are two fundamental operations in the resource binding mechanism: `bind` and `unbind`. Together, they handle process synchronization and protection of shared resources in a flexible and consistent manner. Only shared data binding is used in this section for illustration. As to the details of data binding and process binding, they are discussed further in the following sections. The syntax of the fundamental operations is listed below:

```
b = bind(target, access, sync, level);

unbind(b);
```

In the case of shared data binding, the first parameter, *target*, of the `bind` operation denotes the target shared data region. The region can be as large as the entire shared data structure or as small as a single element of the data structure. Just like accessing a file, a programmer need to open it before any data in the file can be accessed. When accessing a shared data region, the scheme forces programmers to bind it explicitly with the `bind` operation. Of course, its overhead is much lower than opening a file. Unlike the case of opening a file where the whole file is treated as a single object, a subset region of the data structure to be bound can be specified. Figure 6.2 shows some possible shared data regions.

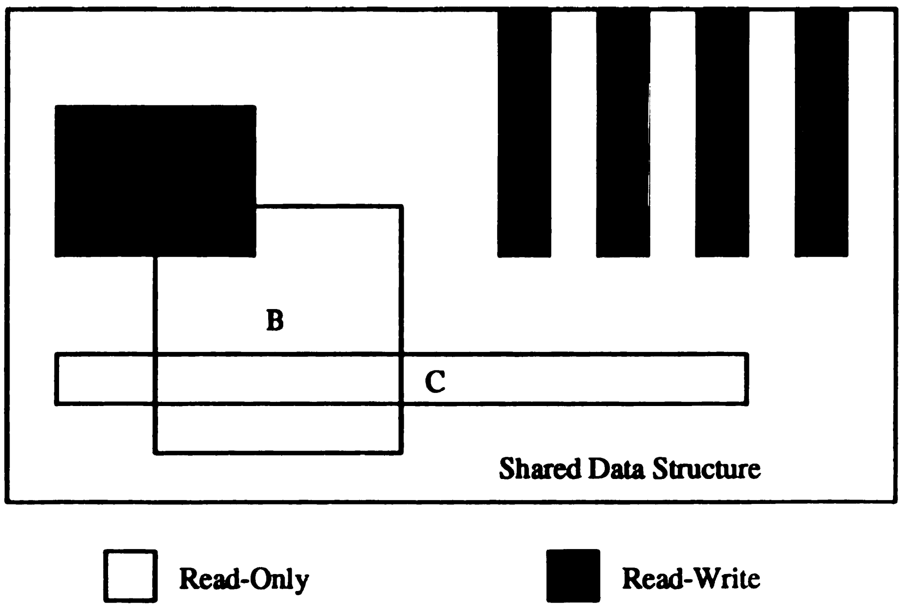


Figure 6.2. Shared data regions.

When binding a shared resource with the **bind** operation, a programmer can specify the access type intended to the target. The access type can be **ro**, **rw** or **ex**, denoting read-only, read-write or execution, respectively. The execution access type, which is used for defining dependency between concurrent processes, is described in Section 6.4. The read-only and read-write access types are mainly used for managing shared data structures. A process can bind a shared data region whenever there is no *conflicting region* currently bound by other processes. Two shared data regions are considered to be conflicting if they are bound by different processes, have intersections, and have at least one in **rw** access type. For example, in Figure 6.2, region A and region B are conflicting regions, while region B and region C are not conflicting regions. This means that **rw** is an exclusive access type, and a shared region being bound in **rw** by a process cannot overlap regions bound by any other processes. However, a **ro** region can overlap any number of **ro** regions. The classification of access types enables multiple-read/single-write, which preserves high parallelism when executing parallel programs.

Whenever a **bind** operation is performed successfully, a *binding descriptor*, **b**, is returned, which can be used in a subsequent **unbind** operation to release the shared resource. If a **bind** operation cannot be executed successfully, one of two procedures may be taken, depending on what synchronization status is specified by the parameter *sync*. The value of *sync* can be either **blocking** or **non-blocking**. A program performing a blocking **bind** operation is blocked until successful completion of the operation, upon which a binding descriptor will be returned. While a program executing a non-blocking **bind** operation will be returned an error code immediately upon unsuccessful operation due to conflicting regions.

The parameter *level* is only used with **ex** access type to support various synchronization patterns among processes, and is mentioned in Section 6.4. An **unbind** operation releases the shared resource identified by the binding descriptor **b** obtained from a previous **bind** operation. The following code, which atomically increases the shared variable **sh** by one, is an example of using **bind** and **unbind** operations.

```
b = bind(sh, rw, blocking, );
sh = sh+1;
unbind(b);
```

Note that one data binding is sufficient for handling a flexible size of shared data region. This is unlike locking semaphores, where each fixed component of a shared data region needs to be associated with a semaphore variable, and their relationship needs to be enforced manually. With just the two fundamental operations **bind** and **unbind**, the resource binding paradigm provides an easier way of writing parallel programs. Moreover, programs designed with this scheme are easier to understand and to debug. Less effort in implementing parallel programs and improved portability among various parallel architectures reduce software development cost for high performance computation on parallel computers.

6.3 Managing Shared Data Structures with Data Binding

The data binding scheme supports high flexibility in managing large shared data structures. As mentioned in the previous section, a target of the **bind** operation can be as large as an entire shared data structure or as small as a single element in the data structure. The flexibility is illustrated by a 2-dimensional array of C-language structure variables, though the scheme is not limited to working with C-language. The following example demonstrates several ways of binding shared data regions which are subsets of the 2-dimensional array.

```
struct
    {
```

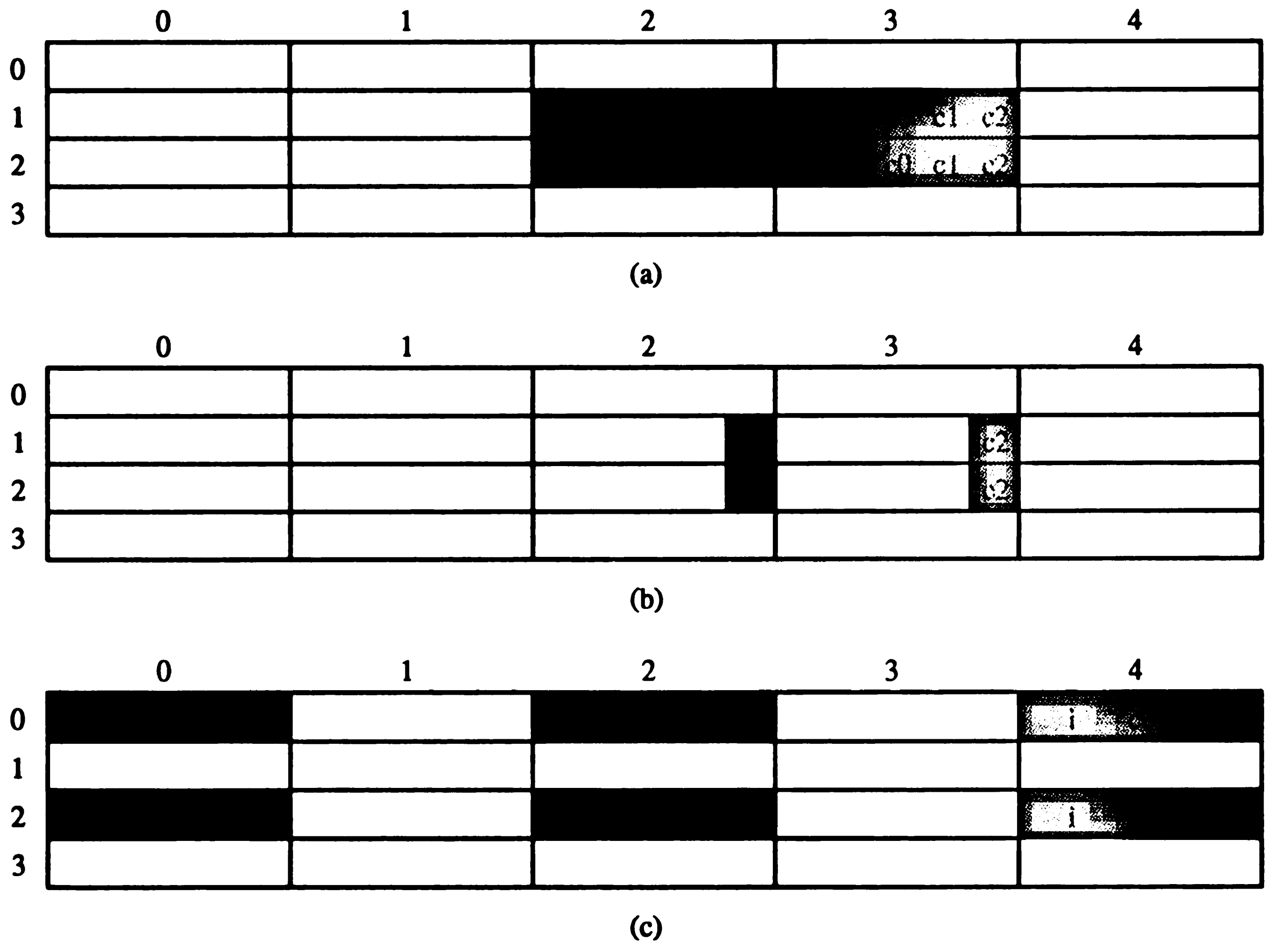


Figure 6.3. Shared data regions in a structure type array.

```
        int i;
        char c[3];
        }
        sh[4][5];

    b = bind(sh[1:2][2:3], rw, blocking, );
```

The item m:n in each dimension of the array represents a range of the index in that dimension. Figure 6.3a shows the array and the bound region. Note that each element of the array is a structure type variable. A more complicated shared data region can be specified as follows, and is shown in Figure 6.3b.

```
    b = bind(sh[1:2][2:3].c[2], rw, blocking, );
```

Furthermore, an optional step argument can be added to an index range item as in the following example. The corresponding shared data region can be found in Figure 6.3c.

```
    b = bind(sh[0:3:2][0:4:2], rw, non-blocking, );
```

The scheme can be further extended to allow triangular and other more complicated regions; however, these extensions are not covered in this dissertation. We have shown that the data binding paradigm provides a very flexible way of managing shared data structures. This not only reduces synchronization primitives to be handled for large data structures, but also preserves parallelism when executing parallel programs. Some illustrative examples are given below.

6.3.1 The Dining Philosophers Problem

The dining philosophers problem is a famous and frequently used benchmark for evaluating concurrent programming paradigms. Before we demonstrate the data binding approach and compare it with other approaches in solving this problem, we would like to give a brief description about the problem. A round table is set with some plates. There is a single chopstick between two neighboring plates. Philosophers think, eat, and repeat the cycle. Before a philosopher can eat, he/she needs to pick up the two chopsticks to the right and the left of his/her plate. There is a potential deadlock in this problem. If the table is full and all philosophers pick up their right chopsticks at the same time, no left chopsticks will be available, and all philosophers will be waiting for others. To prevent deadlock, different ways have been used by various programming paradigms. The goal is to prevent deadlocks and to preserve parallelism.

Figures 6.4 and 6.5 shows the Linda approach and data binding approach, respectively, of solving the dining philosophers problem. Linda prevents deadlocking by allowing only one less than the total number of philosophers into the dining room at any time [49]. This is controlled by the type of tuples called "room ticket", whose number is the total number of philosophers minus one. In contrast, the data binding approach allows several data items to be bound atomically. There is actually no potential deadlock in this approach. The algorithm first defines ranges and steps of data items in the "chopstick" array to be bound by each philosopher process. The actual data binding takes only a single `bind` operation instead of several `in` or `lock` operations. Programmers are not responsible for special arrangements like "room ticket" to prevent deadlocks in this problem.

6.3.2 Accessing Highly Overlapped Data Regions

The flexibility and power of data binding can be demonstrated with a more complicated case. Consider the 2-dimensional array shared by concurrent processes in Figure 6.6. The shaded areas denote data regions which are highly overlapped. Each region is accessed by one process at any one time. Assume all regions are accessed in read-write mode, which means that overlapping regions are conflicting regions and cannot be accessed simultaneously. However, in practical cases, read-only and read-

Linda approach:

```
    philosopher(int i)
        {
        while(1)
            {
            think();
            in("room ticket");
            in("chopstick", i);
            in("chopstick", (i+1)%Num);
            eat();
            out("chopstick", i);
            out("chopstick", (i+1)%Num);
            out("room ticket");
            }
        }

    initialize()
        {
        int i;
        for(i=0; i<Num; i++)
            {
            out("chopstick", i);
            eval(philosopher(i));
            if(i<(Num-1)) out("room ticket");
            }
        }
```

Figure 6.4. Solving dining philosophers problem with Linda.

Data binding approach:

```
shared int chopstick[Num];

philosopher(int i)
    {
    int j=(i+1)%Num, b;
    if(j==0) { i=0; j=Num-1; }
    while(1)
        {
        think();
        b = bind(chopstick[i:j:j-i], rw, blocking,);
        eat();
        unbind(b);
        }
    }
```

Figure 6.5. Solving dining philosophers problem with data binding.

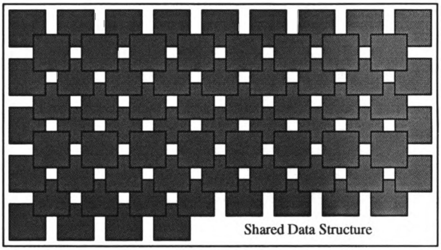


Figure 6.6. Highly overlapped data regions.

write accesses may be mixed arbitrarily, and the multiple-read/single-write accessing style can be used to maintain high parallelism.

Figure 6.7 presents algorithms using locking semaphores and data binding in accessing the highly overlapped data regions. As can be seen, the locking semaphore algorithm involves multiple locking operations for processing a single shared data region. It is the programmer's responsibility to arrange the locking operations for preventing deadlocks. In contrast, data binding protects a shared region with one atomic `bind` operation. Programmers need not worry about potential deadlocks. (Although, in some other applications where multiple data regions may be bound by a single process at one time and potential deadlocks exist, the paradigm has taken most responsibility of deadlock prevention away from programmers.)

Consider the case of using Linda for the problem. Linda suffers the same problem as that of locking semaphores, in that it needs to pass multiple tuples to the tuple space for each shared data region. Moreover, if the array is large, the tuple space may be filled with a large number of tuples, which can increase the overhead in searching tuples, and consequently reduce performance. In contrast, the overhead for data binding is independent of data size, and is only related to the number of processes.

6.4 Synchronizing Processes with Process Binding

One of the important features of resource binding is that it handles processes in the same way as ordinary shared data structures. The same flexibility we have shown for data binding applies to the management of concurrent processes. This section shows how processes can be treated as shared variables by introducing an abstract data type. With each abstract variable of this type representing a process, dependency between concurrent processes can be defined by the `bind` operation on the abstract

Locking semaphore approach:

```
shared int a[1000][1000];
shared semaphore s[1000][1000];

access_region(int r1, int c1, int r2, int c2)
    {
    int i, j;
    for(i=r1; i<=r2; i++)
        {
        for(j=c1; j<=c2; j++)
            lock(s[i][j]);
        }
    compute(r1, c1, r2, c2);
    for(i=r1; i<=r2; i++)
        {
        for(j=c1; j<=c2; j++)
            unlock(s[i][j]);
        }
    }
```

Data binding approach:

```
shared int a[1000][1000];

access_region(int r1, int c1, int r2, int c2, int access)
    {
    int b;
    b = bind(a[r1:r2][c1:c2], access, blocking,);
    compute(r1, c1, r2, c2);
    unbind(b);
    }
```

Figure 6.7. Accessing shared data regions with locking semaphores and data binding.

variables. The paradigm can be applied to various process synchronization patterns. This is demonstrated with two typical examples, barrier and pipelining.

6.4.1 An Abstract Data Type for Concurrent Processes

In order to flexibly manage concurrent processes, we introduce the abstract data type, PROC. Consider a program executed by concurrent processes. Each process has its own execution state, flow of control, and interface with the outside world. Processes communicate with each other through shared memory or message passing. Synchronization between processes are specified by their dependency relationship. Each process has an associated PROC type variable, which is owned by the process and can be accessed by other processes. To a particular process, all other processes are merely a number of PROC type variables. The definition of the data type, PROC, and the procedure of forking processes are shown below.

```
typedef struct
    {
    int pid[IDX];
    int lv1, lv2;
    ......                  /* internal control information */
    }
    PROC;

shared PROC p[100];

main()
    {
    PROC *pp, *bfork();
    if((pp=bfork(p[0:31]))==0)
        parent_proc();
    else
        child_proc(pp);
    }
```

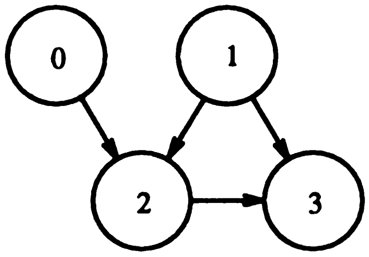
The function **bfork**, with one parameter, creates a number of processes at one time. The parameter, which is similar to the **target** parameter of the **bind** operation, specifies a set of **PROC** variables that will be associated with the child processes to be created. The number of processes created by a **bfork** function depends on the number of **PROC** variables specified in the set. For example, the above program creates 32 processes. The **bfork** function returns the parent process a zero and each child process a pointer to its associated **PROC** variable. In a **PROC** variable, the attribute **pid** is a series of pseudo process identification numbers of its corresponding process, and has nothing to do with the actual process id assigned by the operating system. The attribute **pid** is the index numbers corresponding to the process when the process is created by the **bfork** function. For example, the processes created by the above program will be assigned $0 \ldots 31$ in their corresponding **pid[0]** attributes. For a 2-dimensional array of **PROC** variables, both their **pid[0]** and **pid[1]** may be assigned values when specified in a **bfork** function call. The strategy allows computation patterns such as 2-D meshes to be programmed more conveniently.

The attributes **lv1** and **lv2** denote the range of permission levels specified by its owner process. When a process is created, its associated **PROC** variable is automatically bound by the process with range of permission levels initialized to empty. A **PROC** variable is unbound when its corresponding process terminates. There are other internal attributes in the **PROC** data type, which include information needed for detecting deadlocks.

6.4.2 Defining Process Dependency with Process Binding

Process dependency can be defined by using the **bind** operation with **ex** access type on **PROC** variables. The following statement is an example of this type of **bind** operation, which is invoked by a process on some other processes with a request level of 10.

```
    bind(p[10:20:2], ex, blocking, 10);
```



```
process_2()
    {
    bind(p[0:1], ex, blocking, 1);
    ......
    }

process_3()
    {
    bind(p[1:2], ex, blocking, 1);
    ......
    }
```

Figure 6.8. Process dependency and process binding.

When a process invokes a **bind** operation on some other processes with a certain request level, the operation succeeds only when the request level is within the range of permission levels specified by the processes to be bound. A process can modify its range of permission levels by invoking a **bind** operation on its associated **PROC** variables. In this case, the parameter **sync** of the **bind** operation is of no concern, since the operation always succeeds and can never be blocked. The following statement modifies the range of permission levels to $[100, 200]$, assuming that pp points to the associated **PROC** variable of the process.

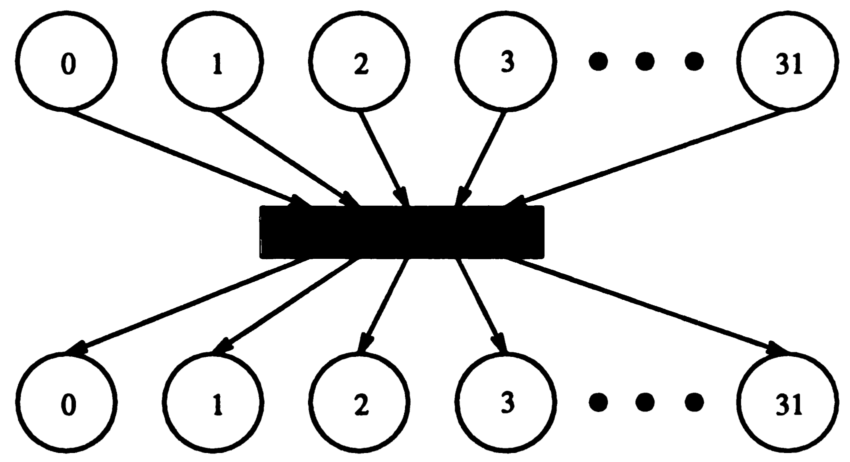
```
bind(*pp, ex, , 100:200);
```

Figure 6.8 shows a dependency graph of four concurrent processes as well as a part of its corresponding process binding program. The figure states that process 2 can execute only after processes 0 and 1 finish their executions, and process 3 can execute only after processes 1 and 2 terminate. As can be seen, process 2 makes a request of level 1 to processes 0 and 1. It will be blocked until both processes 0 and 1 terminate and thus unbind their associated `PROC` variables, or modify their ranges of permission levels to include level 1. The idea can be applied to much more complicated cases. Permission levels are significant, since they can be used to represent current states of concurrent processes. Execution of a process may not always depend on the termination of other processes, instead, they may depend on some state changes of other processes. The condition for a process to execute may be a complicated combination of the states of other processes. This is presently demonstrated.

6.4.3 Illustrative Examples—Barrier and Pipelining

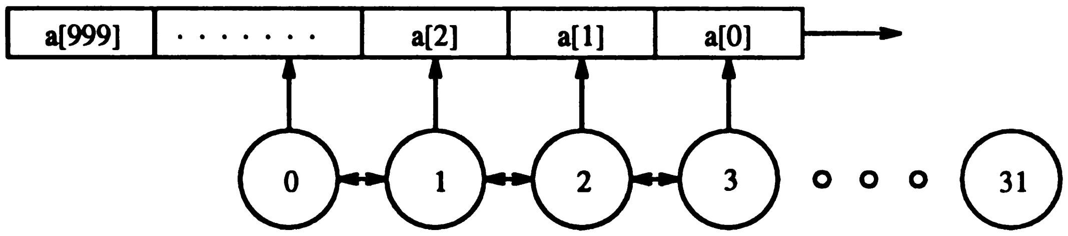
Barrier is a frequently used process synchronization pattern where processes are allowed to continue execution only when all other processes have reached the barrier operation. Figure 6.9 presents a dependency graph of several processes synchronized with a barrier, and shows its simple implementation using process binding. The parameter pp is a pointer to the associated `PROC` variable of each process. The first `bind` operation sets the permission level to 1, denoting that the current process has reached the barrier. The second `bind` operation makes a request of level 1 to all processes. It will be blocked until all processes have set their permission level to 1.

Another interesting example of using process binding is the representation of the pipelining parallel computation style. The concept is to divide a task into a sequence of stages. Input data items are piped into the sequence and processed by each stage. Each stage of the task works on a different data item at any one time. An input data item has to go through all stages to complete its computation. Figure 6.10 shows a



```
process(PROC *pp)
    {
    ......
    bind(*pp, ex, , 1);
    bind(p[0:31], ex, blocking, 1);
    ......
    }
```

Figure 6.9. Barrier and process binding.



```
shared PROC p[32];
shared int a[1000];

main()
    {
    PROC *pp;
    if((pp=bfork(p[0:31]))!=0) stage(pp);
    ......
    }

stage(PROC *pp)
    {
    int i, pid=pp->pid[0];
    for(i=0; i<1000; i++)
        {
        if(pid!=0) bind(p[pid-1], ex, blocking, i);
        compute(a[i]);
        bind(*pp, ex, , 0:i);
        }
    }
```

Figure 6.10. Pipelining processes.

list of pipelining processes working on the input array **a**. Each element of the array has to be processed by all processes of the list in sequence. Each process works on one element at a time, and no more than one process works on the same element at the same time. For example, when process i works on the array element `a[5]`, process $i+1$ and $i+2$ will be working on `a[4]` and `a[3]`, respectively.

A process binding program which implements the algorithm is also listed in Figure 6.10. The main program forks 32 child processes. Each of the child processes is assigned the address of its associated `PROC` variable, pp, and starts executing the function `stage`. The pseudo process id of each process is retrieved from the attribute `pid[0]` of its `PROC` variable. Before a process can compute an array element `a[i]`, the process has to make sure that the element has already been computed by the previous process. This is achieved by binding the previous process with the request level `i` in blocking mode. After computing an array element, the process modifies its permission status to include level `i`, which will allow the successive process to compute the element `a[i]`. The same idea can be applied to more complicated cases, such as 2-dimensional pipelining.

6.5 Implementation Issues

The resource binding scheme introduced is a portable parallel programming paradigm that can be implemented in various parallel architectures. As stated earlier, an efficient parallel program developed on a particular parallel system may not perform well on another. Appropriate implementations of the scheme on different architecture play an important role in achieving high performance. This section discusses the implementations of the resource binding paradigm in both shared-memory and distributed-memory multiprocessors as well as the CFM architecture.

6.5.1 Shared-Memory Multiprocessors and the CFM Architecture

In a shared-memory multiprocessor, binding requests from concurrent processes, if not conflicting with other active binds, are stored in an "active binding list", as shown in Figure 6.11. Each binding request contains the address pointers, indices, lengths, step widths, and other information of the target as well as the requested access type/level and synchronization status. A binding request is verified with the active binding list to detect any possible conflict before the bind can be granted and recorded in the list. In order to detect a possible conflict, both the target specification and the access type/level of the request are compared with each entry in the active binding list. Upon detecting a conflict and the synchronization status requested to be blocking, the request is placed in a "request queue" associated with the conflicting active bind.

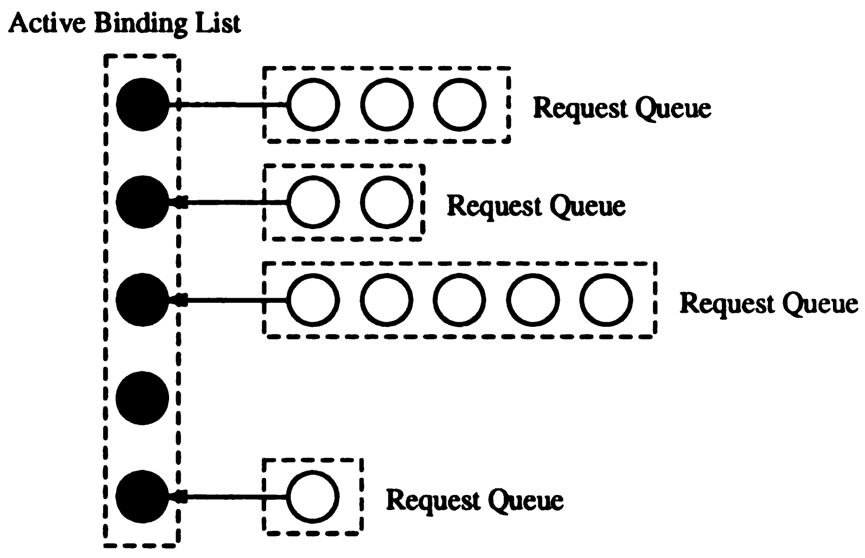


Figure 6.11. Resource binding implementation on a shared-memory multiprocessor.

When an unbinding request is received, the corresponding bind is removed from the active binding list. The first binding request in the associated request queue, if any, is signaled to retry. The binding request is again compared with the active binds in the active list. If no conflict is found, the requested bind is granted and the request is moved into the active list. If the request detects another conflict, however, it is moved from the original request list to the request list associated with the new conflicting active bind. The same procedure repeats for each request on the original queue.

In order to reduce the overhead of comparing data binding requests, active binds can be maintained hierarchically instead of in a single list. The active binding hierarchy is arranged according to the logic structure of the target data structure. This relaxes the requirement of comparing a data binding request with all active binds. Further information about the implementation considerations is yet to be presented in [50]. Using atomic multiple locks is another way of improving resource binding performance. The resource binding paradigm can be efficiently implemented in the CFM architecture, since it supports efficient atomic multiple lock. Information such as granularity of the binding targets can be collected during program compilation. For those data structures with fine granularity, the request comparison scheme described above can be used. While for data structures with larger granularity, they can be divided into components, with each component controlled by a lock. The sizes of the components are decided by the granularity information collected. A binding target can consist of multiple components and can be bound by applying an atomic multiple lock to the components.

6.5.2 Distributed-Memory Multiprocessors

A shared-memory programming environment can be established in a distributed-memory multiprocessor with the resource binding paradigm. Building a shared-memory environment on top of a distributed-memory system involves data transfer among memory modules in response to memory accesses. The primitives, `bind` and `unbind`, offer useful hints for when and where to move data. Each binding request is carried out by sending a request message to the server processor of the target data structures or processes. A daemon process on the server processor verifies the request and, if no conflict is detected, returns to the requesting process either an acknowledgement when it is a process binding, or the target data region when it is a data binding. An unbinding request on a `ro` type region sends a message to the server processor of the data region and asks the server processor to release the bind. An unbinding request on a `rw` type region, however, also sends the data region itself back to the server processor, which in turn updates the original copy of the region before releasing it.

As can be seen, data consistency is maintained by the resource binding paradigm through message-passing among the processors. Like a cache coherence protocol, the paradigm can be implemented to support various memory consistency models. Because of the nature of the resource binding paradigm, the release consistency model, which explores higher parallelism by allowing more buffering and pipelining than other consistency models [19], is the most suitable one to be implemented.

CHAPTER 7

CONCLUSION AND FUTURE WORK

This dissertation presents the basic concept, extension, coherence issues, and application of the CFM architecture. The final chapter summarizes the work discussed, highlights its major contributions, and outlines interesting topics for future work.

7.1 Summary

Chapter 1 reviews the basic concepts of multiprocessing, illustrates a typical distributed-memory multiprocessor and a typical shared-memory multiprocessor, and identifies problems that limit achievable speedup on multiprocessors. Among the problems, memory conflicts and interconnection network contention on shared-memory multiprocessors are of special concern to this dissertation. Chapter 2 discusses two of the important memory design considerations: contention and consistency. Illustrations of several well-known multiprocessors serve to describe the contention problem and the various approaches to its solution. The multiprocessors discussed include the NYU Ultracomputer, IBM RP3, BBN Butterfly and Monarch, OMP multiprocessor, and Cedar project. The chapter also defines various memory consistency models, including sequential consistency, processor consistency, weak consistency, and release consistency.

Chapter 3 describes the basic concept of the CFM architecture. The CFM architecture presented is based on block accesses. The architecture improves effective memory bandwidth by eliminating shared memory conflicts as well as interconnection network contention. With the introduction of the AT–space, the scheme extends to conventional memory architectures a new degree of freedom in the time dimension. In the CFM architecture, memory accesses are fully synchronized among processors. The mutually exclusive partitioning of the AT–space guarantees memory accesses from different processors to be conflict-free. The synchronous omega network introduced can be used to support contention-free interconnection. The new interconnection concept also reduces the setup time and the propagation delay observed in conventional MIN architectures.

For systems with a large number of processors, in order to reduce the block size, memory banks can be grouped into a number of conflict-free memory modules with smaller blocks. Section 3.2.2 shows one possible implementation, which constructs partially conflict-free memory architectures by using partially synchronous omega networks. It also presents the performance analysis of such architectures with different design parameters, which shows that partially conflict-free systems achieve higher efficiency than conventional systems under different data localities, especially in cases of high access rates.

The CFM architecture eliminates memory access conflicts; however, it also introduces race conditions when two or more processors access the same memory block concurrently. In order to solve the problem, Chapter 4 introduces the address tracking mechanism which records memory access activities in address tracking tables associated with memory banks. A memory access detecting other same-address accesses takes appropriate actions to prevent inconsistency problems. The mechanism can be used to implement the atomic swap operation, which in turn supports higher level process synchronization. Lock/unlock can be implemented using the busy-waiting scheme without creating overhead to other memory accesses.

Chapter 5 reviews some existing snoopy and directory-based cache coherence protocols and introduces the CFM cache coherence protocol. This invalidation-based write-back protocol preserves the low storage overhead of snoopy cache protocols, while it offers the high scalability of directory-based protocols. Synchronization operations are efficiently implemented with its three primitive operations: read, read-invalidate, and write. Weak consistency can be supported with the CFM cache protocol in the CFM architecture. Chapter 5 also introduces a hierarchical extension to the CFM architecture and shows that the CFM cache coherence protocol can be recursively defined for large scale hierarchical extensions.

Chapter 6 presents the technique of resource binding, which is an architecture independent paradigm for parallel programming. It handles problems such as shared data protection and process synchronization in a very flexible and consistent way on both shared-memory and distributed-memory machines. The resource binding paradigm can be efficiently implemented on the CFM architecture by way of its atomic multiple lock support. The scheme outperforms locking semaphores and monitors in the sense that it allows a multiple-read/single-write accessing style and offers a flexible way of specifying shared data regions. As a consequence, higher parallelism can be preserved. The paradigm allows programmers to write portable parallel programs that are easy to understand and to debug, yet its overhead is much lower than that of Linda.

The introduction of the abstract data type, "virtual processor", helps maintain states of processes in the same way as ordinary shared variables. Different features, such as exclusive accesses to shared data structures and dependency among concurrent processes, can all be managed using the two simple fundamental operations, bind and unbind. We show that typical process synchronization patterns such as barrier and pipelining can be represented by using the bind operation in a very simple way. In fact, all regular synchronization patterns can be represented directly using the process binding scheme.

7.2 Future Work

The partially conflict-free memory architecture introduced in Section 3.2.2 and the hierarchical CFM extension presented in Section 5.4 are two possible approaches to constructing large scale systems. In such systems, processor allocation is a very important issue. One of our future research topics is to design efficient processor allocation schemes that will reduce memory, network, or network controller contention in these systems.

Section 3.4 gives simple memory efficiency analyses of conventional systems and partially conflict-free systems. More accurate results may be obtained by also considering the effect of network contention. The analysis of the hierarchical CFM architecture is another topic for future research. In order to verify the analysis models, simulations under different application characteristics are needed for these architectures. With the knowledge that will be collected from the analyses and simulations, connection schemes among conflict-free clusters may be further improved. In addition, the optimal memory and network bandwidth assigned to a network controller in a hierarchical CFM system needs to be studied further.

It may be helpful to implement a "building block" for constructing large scale CFM architectures. A building block can be a board composed of multiple processors/ports and a conflict-free memory module with a number of memory banks. It would be more convenient if large scale multiprocessors could be implemented by integrating smaller building blocks such as four-bank CFM boards or eight-bank CFM boards. Methods of designing and integrating building blocks are both interesting future research topics.

The discussions in this dissertation assume that each processor in the CFM architecture or a conflict-free cluster is assigned a time slot for accessing shared memory. When a processor is not accessing memory, its time slot is wasted. One way to utilize this valuable resource is to assign a time slot to more than one processor. Although, processors sharing the same time slot can conflict with each other when accessing shared memory concurrently, the memory and network utilizations are further improved. This is especially attractive to computation-intensive applications. The optimal configurations and performance issues for different applications need to be studied further.

In addition to directly supporting parallel programming, the resource binding paradigm can be applied in many other ways. Extensions can be made to traditional high level programming languages such as FORTRAN and C by using the paradigm to support architecture independent parallel programming. For example, a parallel FORTRAN language can be implemented with a preprocessor based on resource binding instead of a direct compiler. In this case, the language can be portable to distributed-memory machines and thus provide a shared-memory environment. A high level parallel programming environment can be developed on top of the resource binding paradigm, which is independent of underlying hardware architectures. Furthermore, the paradigm can be used as the kernel of a parallel operating system to support shared-memory machines, distributed-memory machines, or even heterogeneous systems.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] J. L. Hennessy and N. P. Jouppi, "Computer technology and architecture: An evolving interaction," *IEEE Computer*, vol. 24, pp. 18 – 29, Sept. 1991.

[2] K. Hwang, "Advanced parallel processing with supercomputer architectures," in *Proceedings of the IEEE*, pp. 1348 – 1379, Oct. 1987.

[3] J. P. Hayes, T. N. Mudge, Q. F. Stout, S. Colley, and J. Palmer, "Architecture of a hypercube supercomputer," in *Proceedings of the 1986 International Conference on Parallel Processing*, pp. 653 – 660, 1986.

[4] BBN Advanced Computers Inc., Cambridge, Massachusetts, *Inside the GP1000*, 1989.

[5] BBN Advanced Computers Inc., Cambridge, Massachusetts, *Inside the TC2000 Computer*, 1990.

[6] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer — designing an mimd shared-memory parallel computer," in *Proceedings of the 9th AnnualInternational Symposium on Computer Architecture*, pp. 27 – 42, 1982.

[7] G. F. Pfister *et al.*, "An introduction to the IBM research parallel processor prototype (RP3)," in *Experimental Parallel Computing Architectures* (J. J. Dongarra, ed.), pp. 123 – 140, Elsevier Science Publishers B.V., Amsterdam, 1987.

[8] G. Pfister and A. Norton, "'Hot spot' contention and combining in multistage interconnect networks," *IEEE Transactions on Computers*, vol. C-34, pp. 943 – 948, Oct. 1985.

[9] H. Shing and L. M. Ni, "A conflict-free memory design for multiprocessors," in *Proceedings of Supercomputing '91 Conference*, pp. 46 – 55, Nov. 1991.

[10] H. Shing and L. M. Ni, "Resource binding—a universal approach to parallel programming," in *Proceedings of Supercomputing '90 Conference*, pp. 574 – 583, Nov. 1990.

[11] R. D. Rettberg, W. R. Crowther, P. P. Carvey, and R. S. Tomlinson, "The Monarch parallel processor hardware design," *IEEE Computer*, vol. 23, pp. 18 – 30, Apr. 1990.

[12] D. H. Lawrie and C. R. Vora, "The prime memory system for array access," *IEEE Transactions on Computers*, vol. C-31, pp. 435 – 442, May 1982.

[13] A. Norton and E. Melton, "A class of Boolean linear transformations for conflict-free power-of-two stride access," in *Proceedings of the 1987 International Conference on Parallel Processing*, pp. 247 – 254, 1987.

[14] D. T. Harper III and J. R. Jump, "Vector access performance in parallel memories using a skewed storage scheme," *IEEE Transactions on Computers*, vol. C-36, pp. 1440 – 1449, Dec. 1987.

[15] S. Weiss, "An aperiodic storage scheme to reduce memory conflicts in vector processors," in *Proceedings of the International Symposium on Computer Architecture*, pp. 380 – 385, 1989.

[16] R. Raghavan and J. P. Hayes, "On randomly interleaved memories," in *Proceedings of the Supercomputing '90 Conference*, 1990.

[17] K. Hwang *et al.*, "OMP: A RISC-based multiprocessor using orthogonal-access memories and multiple spanning buses," in *Proceedings of the ACM International Conference On Supercomputing*, (Amsterdam, The Netherlands), Association for Computing Machinery, June 1990.

[18] D. Gajski *et al.*, "Cedar construction of a large scale multiprocessor," Tech. Rep. UIUCDCS-R-83-1123, University of Illinois, Department of Computer Science, Feb. 1983.

[19] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory consistency and event ordering in scalable shared-memory multiprocessors," in *Proceedings of the 17th AnnualInternational Symposium on Computer Architecture*, pp. 15 – 16, May 1990.

[20] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Transactions on Computers*, vol. C-28, pp. 241 – 248, Sept. 1979.

[21] J. R. Goodman, "Cache consistency and sequential consistency," Tech. Rep. 61, SCI Committee, Mar. 1989.

[22] M. Dubois, C. Scheurich, and F. Briggs, "Memory access buffering in multiprocessors," in *Proceedings of the 13th AnnualInternational Symposium on Computer Architecture*, pp. 434–442, June 1986.

[23] C. Scheurich and M. Dubois, "Correct memory operation of cache-based multiprocessors," in *Proceedings of the 14th AnnualInternational Symposium on Computer Architecture*, pp. 234–243, June 1987.

[24] A. Smith, "Line (block) size choice for CPU cache memories," *IEEE Transactions on Computers*, vol. C-36, pp. 1063 - 1075, Sept. 1987.

[25] R. Kenner, S. Dickey, and P. J. Teller, "The design of processing elements on a multiprocessor system with a high-bandwidth, high-latency interconnection network," in *Proceedings of the 22nd Annual Hawaii International Conference on System Science*, pp. 319 - 328, Jan. 1989.

[26] D. H. Lawrie, "Access and alignment of data in an array processor," *IEEE Transactions on Computers*, vol. C-24, pp. 1145 - 1155, Dec. 1975.

[27] E. Gornish, E. Granston, and A. Veidenbaum, "Compiler-directed data prefetching in multiprocessors with memory hierarchies," in *International Conference on Supercomputing*, pp. 354 - 368, 1990.

[28] R. L. Lee, *The Effectiveness of Caches and Data Prefetch Buffers in Large-Scale Shared Memory Multiprocessors.* PhD thesis, University of Illinois at Urbana–Champaign, May 1987.

[29] T. Mowry and A. Gupta, "Tolerating latency through software-controlled prefetching in shared-memory multiprocessors," *Journal of Parallel and Distributed Computing*, June 1991.

[30] A. K. Porterfield, *Software Methods for Improvement of Cache Performance on Supercomputer Applications.* PhD thesis, Rice University, May 1989.

[31] S. Adve and M. Hill, "Weak ordering — a new definition," in *Proceedings of the 17th AnnualInternational Symposium on Computer Architecture*, pp. 2 - 14, May 1990.

[32] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, "The directory-based cache coherence protocol for the DASH multiprocessor," in *Proceedings of the 17th AnnualInternational Symposium on Computer Architecture*, pp. 148 – 159, May 1990.

[33] T. N. Mudge *et al.*, "The design of a microsupercomputer," *IEEE Computer*, vol. 24, pp. 57 – 64, Jan. 1991.

[34] J. Archibald and J.-L. Baer, "Cache coherence protocols: Evaluation using a multiprocessor simulation model," *ACM Transactions on Computer Systems*, pp. 273 – 298, Nov. 1986.

[35] J. R. Goodman, "Using cache memory to reduce processor-memory traffic," in *Proceedings of the 10th AnnualInternational Symposium on Computer Architecture*, pp. 124 – 131, June 1983.

[36] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G. Sheldon, "Implementing a cache consistency protocol," in *Proceedings of the 12th AnnualInternational Symposium on Computer Architecture*, pp. 276 – 283, June 1985.

[37] W. C. Yen, D. W. L. Yen, and K.-S. Fu, "Data coherence problems in multicache system," *IEEE Transactions on Computers*, vol. C-34, pp. 56 – 65, Jan. 1985.

[38] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz, "An evaluation of directory schemes for cache coherence," in *Proceedings of the 15th AnnualInternational Symposium on Computer Architecture*, pp. 280 – 289, June 1988.

[39] C. K. Tang, "Cache design in the tightly coupled multiprocessor system," in *AFIPS Conference Proceedings*, (National Computer Conference, NY, NY), pp. 749 – 753, June 1976.

[40] L. M. Censier and P. Feautrier, "A new solution to coherence problems in multicache systems," *IEEE Transactions on Computers*, vol. C-27, pp. 1112 – 1118, Dec. 1978.

[41] B. W. O'Krafka and A. R. Newton, "An empirical evaluation of two memory-efficient directory methods," in *Proceedings of the 17th AnnualInternational Symposium on Computer Architecture*, pp. 138 – 147, May 1990.

[42] D. James, A. T. Laundrie, S. Gjessing, and G. S. Sohi, "Distributed-directory scheme: Scalable coherent interface," *IEEE Computer*, vol. 23, pp. 74 – 77, June 1990.

[43] M. Thapar and B. Delagi, "Distributed-directory scheme: Stanford distributed-directory protocol," *IEEE Computer*, vol. 23, pp. 78 – 80, June 1990.

[44] J. Lee and U. Ramachandran, "Synchronization with multiprocessor caches," in *Proceedings of the 17th AnnualInternational Symposium on Computer Architecture*, pp. 27 – 37, May 1990.

[45] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam, "The stanford Dash multiprocessor," *IEEE Computer*, vol. 25, pp. 63 – 79, Mar. 1992.

[46] C. A. R. Hoare, "Monitors: An operating system structuring concept," *Communications of the ACM*, vol. 17, pp. 549 – 458, Oct. 1974.

[47] J. Boyle *et al.*, *Portable Programs for Parallel Processors.* Holt, Rinehart and Winston Inc., 1987.

[48] N. Carriers and D. Gelernter, "Linda in context," *Communications of the ACM*, vol. 32, pp. 444 – 458, Apr. 1989.

[49] I. Wm Leler, Cogent Research, "Linda meets unix," *IEEE Computer*, vol. 23, pp. 43 – 54, Feb. 1990.

[50] H. Shing and L. M. Ni, "The implementation of resource binding on shared-memory multiprocessors," Tech. Rep. (in preparation), Department of Computer Science, Michigan State University, 1991.