Cache Coherence Protocols for Large-Scale Multiprocessors

by

David Lars Chaiken

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1990

Signature of Author ....
Department of Electrical Engineering and Computer Science
August 31, 1990

Certified by..
Anant Agarwal
Assistant Professor of Computer Science and Electrical Engineering
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

Cache Coherence Protocols for Large-Scale Multiprocessors

by

David Lars Chaiken

Submitted to the Department of Electrical Engineering and Computer Science
on August 31, 1990, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

Caches have the potential to provide multiprocessors with an automatic mechanism for reducing both network traffic and average memory access latency. However, cache-based systems must address the problem of cache coherence. This thesis presents the results of the search for a cache coherence solution for Alewife, a large-scale multiprocessor being built at MIT. The research focuses on coherence protocols that use a directory, a list of cached copies of data, to avoid the need for a system-wide broadcast mechanism. The structure and the implementation of a number of coherence schemes are evaluated with coupled and decoupled simulation techniques. In addition to comparing the protocols in terms of hardware overhead and performance, the thesis reports on the experience gained by implementing several different schemes in ASIM, the Alewife machine simulator. The protocol search reaches two major conclusions: First, by using system-level optimizations, it is possible to use caches to build large-scale shared-memory multiprocessors. Second, the Alewife machine should use the integrated systems approach — handling common cases in hardware and exceptional cases in software — to solve the cache coherence problem.

Keywords: cache coherence, multiprocessor system, directory, multiprocessor simulation, shared-memory multiprocessor, interconnection network

Thesis Supervisor: Anant Agarwal
Title: Assistant Professor of Computer Science and Electrical Engineering

Acknowledgments

I could not have completed this thesis without help from many people. It was during the spring before I entered MIT that Anant Agarwal, my thesis advisor, gave me an introductory tutorial on cache-coherent multiprocessors. At that time, I had no idea that the topic was going to occupy me for the following two years. Since then, Professor Agarwal has taught me how to reason about computer architectures, how to evaluate alternatives for designing multiprocessors, and how to communicate my ideas to other researchers. It has been a pleasure to be a member of Anant's research group.

Most of the ideas that this thesis evaluates were developed through a joint effort of the Alewife group at MIT. The other members of the group have been my teachers, conspirators, and friends. By advising me when I arrived at MIT, Gino Maa gave me the assistance that I needed to start doing the research that led to my thesis. Gino also wrote the network simulator that I used to analyze the cache coherence schemes. Not only did Beng-Hong Lim manage to tolerate sharing an office with me for two years (so far), but he also wrote portions of the SPARCLE processor simulator and run-time system. Mathews Cherian laid the foundation for my work by writing both the post-mortem scheduler with Kimming So at IBM and the cache and directory simulator. Kiyoshi Kurihara wrote the dynamic post-mortem scheduler and worked with me to understand the balance between software and hardware in shared-memory multiprocessors. By questioning and challenging my ideas, no matter how great or how small, Dan Nussbaum has increased my knowledge in many areas of computer science. David Kranz wrote the Mul-T compiler, which is the foundation of Alewife's software system. My discussions with David have given me a practical understanding of the power and the limitations of compilers and run-time systems. Kirk Johnson has helped me gain insight into the difficult job of programming parallel processing systems. Kirk was responsible for the results derived from the Speech application, which he wrote. John Kubiatowicz, who has led the hardware design effort for Alewife, worked with me to develop an accurate model of the cache controller. I have enjoyed long design sessions with Kubi, which typically resulted in frustration, understanding, and (I hope) an efficient design for Alewife's memory system implementation. I look forward to continuing to work as a member of the Alewife team.

The evaluation methodology used in this thesis required support from a wide range of sources outside of MIT. Harold Stone and Kimming So of IBM helped obtain the traces of the Weather, Simple, and FFT programs. Pat Teller and Allan Gottlieb of New York University arranged to send me the sources for these programs. Wolf-Dietrich Weber and Anoop Gupta provided the four VAX T-bit traces, which were generated using a system developed by Steve Goldschmidt at Stanford. The notation for the protocol state transition diagrams borrows from the doctoral thesis of James Archibald and from work done in the MIT Parallel Processing Group by Ingmar Vuong-Adlerberg. Digital Computer Corporation, Encore Computer Corporation, Harris Computer Systems, and SUN Microsystems contributed the machines that I have used to analyze the performance of large-scale multiprocessors. The research reported in this thesis is funded by DARPA contract # N00014-87-K-0825.

I would also like to thank Ema and Aba for encouraging and supporting my education. And finally, hugs to Ora for reminding me that there are many things that are more important than coherent caches.

Praised are You, the Eternal, our God, Ruler of the Universe, who has graced us with life, sustained us, and allowed us to reach this season.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A multiprocessor's shared-memory system provides a mechanism for programmers to partition programs among many processors, and allows the processors to communicate and to synchronize with each other. As processing speed increases relative to the latency of interprocessor communication, the latency and the bandwidth of the memory system limits the speed of computation. Small multiprocessors remove this mismatch between processor and memory speeds by equipping each processor with a fast, local memory, called a cache. By storing copies of frequently accessed data, a cache can satisfy a large fraction of its processor's memory requests, thereby reducing both the average memory latency and the processor's demand on the interprocessor communication network.

However, caches in a multiprocessing environment introduce the cache coherence problem. When multiple processors maintain locally cached copies of a unique shared-memory location, any local modification of the location can result in a globally inconsistent view of memory, violating the shared-memory abstraction. Cache-coherence protocols prevent this problem by maintaining a uniform state for each cached block of data. While it is possible to implement a cache coherence protocol with a multiprocessor-wide broadcast, such a mechanism negates the bandwidth reduction that makes caches attractive in the first place. Furthermore, in large-scale multiprocessors, broadcast mechanisms are either inefficient or prohibitively expensive to implement.

Is it possible to use caches to build an efficient shared-memory system for a large-scale multiprocessor? The answer to this question depends on finding a solution to the cache coherence problem that relies only on scalable hardware mechanisms.

1.1 Cache Coherence for the Alewife Machine

This thesis describes the search for a cache coherence protocol for Alewife, a large-scale multiprocessor being built at MIT. Not only does the Alewife project provide concrete motivation for solving the cache coherence problem, but it also establishes fundamental constraints on potential solutions to the problem. Since the Alewife machine is intended to be both scalable and easily programmable, its memory system must conform to both of these goals:

First, the memory system's cache coherence protocol must be scalable. That is, the physical resources required to implement the cache coherence scheme must be cost-effective, and independent of the number of processors in the system. An optimal solution would require only a small, constant amount of overhead per processing node. Given such a solution, the physical size of the memory system grows as $\Theta(N)$, where N is the number of processors in the machine. More realistically, since the size of a processor identifier in a system grows as $\Theta(\log N)$, it is reasonable to expect the cache coherence overhead to grow logarithmically with the number of processors. This design decision requires a cache coherence protocol that transmits all memory transactions over a mesh network without a broadcast mechanism.

Second, to allow Alewife to be both scalable and easily programmable, not only must the interprocessor communication system exploit locality to minimize the latency needed to service processor requests, but it must also provide mechanisms for automatic locality management. Caches allow the system to automatically move data to processors, thereby increasing the locality of access within processing nodes, in a manner that is completely transparent to the user. Furthermore, by distributing the shared-memory modules to the processing nodes and by using a packet-switched mesh network to interconnect the nodes, the memory system allows the software to take advantage of communication locality between processing elements.

The protocols considered for the Alewife machine solve the cache coherence problem in the absence of a broadcast mechanism. Each coherence scheme allocates a section of the multiprocessor's memory, called a directory, to store the locations of the cached copies of each data block. Instead of broadcasting the fact that a processor has modified a data location, the memory system sends an individual message to each cache that has a copy of the data. The protocol must also record the acknowledgment of each of these messages to ensure that the global view of memory is actually consistent. This message-based approach dramatically reduces the network bandwidth needed to enforce coherence.

In order to determine whether such protocols meet the requirements of Alewife, several different simulation techniques are used. First, a hybrid decoupled simulation methodology provides evidence that a cached-based shared-memory system is viable. Then, coupled simulations of the Alewife machine allow a complete analysis of both the implementation and the behavior of potential cache coherence schemes. In the end, the choice of a protocol is based on the complexity and the performance of each coherence scheme, as measured by memory overhead and processing speed for a range of benchmark applications.

1.2 Cache Coherence as a General Problem

Although the Alewife project motivates the search for scalable solutions to the cache coherence problem, both the results and the methodology of the search are applicable to the more general task of designing large-scale shared-memory multiprocessors. This thesis contributes some of the first results for directory-based coherence protocols that are implemented and evaluated on memory systems with interconnection networks other than buses. The detailed implementation of various coherence schemes helps to isolate the protocol features that strongly effect the performance of shared memory from the components that have only a weak effect.

The results of the evaluation of different cache coherence protocols emphasizes

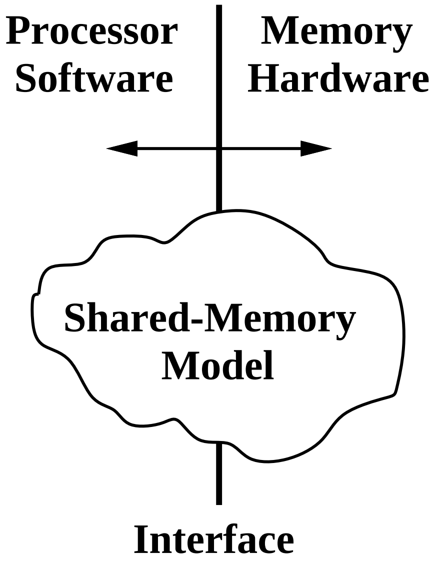


Figure 1-1: The implementation of a shared-memory model must be divided between the processor and the memory system.

the importance of the integrated systems approach. In the case of a shared-memory multiprocessor, the approach balances the size of a shared-memory system's hardware with the complexity of a multiprocessor's software by handling common events in hardware and exceptional situations in software. This philosophy helps to draw the line between the subset of the memory model that is implemented in hardware and the subset that is implemented in software. As depicted in Figure 1-1, the architectural problem of implementing a shared-memory system is solved by specifying the appropriate interface between the processor and the memory system.

The evaluation of coherence methods reveals a number of software optimizations that increase the performance of a cache-based memory system. Dividing the responsibility for implementing the memory model between hardware and software helps to mitigate the effects of widely-shared data on the performance of cache coherence schemes. In fact, the search for a coherence protocol culminates with the definition of the scalable LimitLESS directory protocol, which uses a combination of hardware and software methods to realize the performance of the best non-scalable directory protocol. Perhaps the largest contribution of the research lies in establishing the trade-offs between hardware and software that are necessary to achieve both scalability and high performance in a large shared-memory multiprocessor.

1.3 Organization of this Thesis

The rest of this thesis is organized as follows: Chapter 2 establishes the characteristics of the cache coherence problem by discussing previous attempts to solve the problem. Chapter 3 clarifies the constraints on coherence protocols for large-scale multiprocessors by describing the relevant features of the Alewife architecture. Chapter 3 also defines the LimitLESS protocol in the context of the Alewife machine. Chapter 4 describes the simulation techniques that are used to evaluate the protocols. Finally, Chapter 5 presents the results from the multiprocessor simulations and draws conclusions about the performance of large-scale shared-memory systems.

Chapter 2

Background

2.1 Why Study Caches?

Multiprocessor caches promise to yield performance gains similar to the gains from caches in uniprocessor applications, despite increased implementation complexity. It is the promise of a high-bandwidth, low-latency path to memory that makes multiprocessor cache systems a useful and interesting topic of research. To understand the benefits and problems of using caches in machines with many processors, it is necessary to understand the place of caches in single processor systems.

2.1.1 Single Processor Caches

A cache is a data storage repository that provides fast access to a subset of data from a larger, slower block of memory. Common examples of caches include memory buffers for disk accesses and translation lookaside buffers (TLB) for paging tables of virtual memory systems. In the context of a physical memory system, the term *cache* refers to a chunk of fast — low latency — memory (typically implemented as static random access memory, or SRAM) that provides a processor with a local subset of data from main memory (typically implemented as dynamic random access memory, or DRAM). The physical proximity of a cache to a processor permits a high-bandwidth data path between the two. Processor caches have been implemented in applications from personal computers to supercomputers.

When a processor requests data from its cache, the data may already be located in the cache (a *hit*), or the cache may have to fetch the data from main memory (a *miss*). Uniprocessor caches work due to two principles of memory access patterns: temporal locality and spatial locality. *Temporal locality* refers to the fact that if a processor accesses a unit of memory (called a *word*), then it is likely to access the same word in the near future. *Spatial locality* refers to the fact that if a processor accesses a word of memory, then it is likely to access a nearby word in the future. A processor's cache capitalizes on temporal locality by retaining copies of words of memory that the processor accesses, and on spatial locality by fetching a *block* (a number of consecutive memory words) at once.

The actual performance of a cache is measured by the *average memory access time*, T_a, a metric that encapsulates the effects of both locality and the physical parameters of the cache. $T_a = hT_h + mT_m$, where h is the hit rate in the cache, $m = 1 - h$ is the miss rate in the cache, T_h is the time required to service a hit in the cache, and T_m is the time required to service a miss in the cache. If the time to service a hit in the cache is the same as the processor cycle time, the average access time and miss access time can be normalized to the cycle time: $T_a = h + mT_m$, where T_a and T_m are given in terms of processor cycles. Perhaps a more intuitive metric of cache performance is the *processor utilization*, defined as the fraction of time that the processor is not waiting for memory and is therefore doing useful work. Processor utilization, U, is given by:

$$U = \frac{1}{1 + mT_m}$$

The cache design parameters that affect the hit rate and access times for a cache have been studied by simulating cache based memory systems on various memory access patterns. These access patterns may be generated by statistical methods or by traces of actual processor memory requests. The issues of cache implementation in uniprocessor applications have been exhaustively studied, and in 1982, Alan Jay Smith published a paper that is considered to be the authoritative work on the subject [41].

It is important to realize that given any caching scheme, it is always possible to write an application that "breaks" the scheme. Some programs simply do not exhibit the proper temporal or spatial locality to utilize a cache. However, the vast majority of uniprocessor programs profit from caches, and caching has proved to be a successful strategy for memory system design.

2.1.2 Multiprocessor Caches

The argument for using caches in the implementation of a multiprocessor's shared-memory system is even more compelling than the argument for single processor memory systems. In multiprocessors with shared memory, cache-based memory systems automatically move data where it is needed. When a processor attempts to read or write a unit of data, the memory system fetches the data from a remote memory module into a cache that is located in the same physical node as the processor. Subsequent load and store accesses to the same data are satisfied within the local processing node. After the working-set of each processor migrates into its cache, the memory system can perform a large percentage of processor requests without communicating over the multiprocessor's interconnection network.

Satisfying most memory requests in a cache increases the performance of the system in two ways: First, since typical cache access times are an order of magnitude lower than interprocessor communication times, the memory access latency incurred by each processor is lower than in a system that does not cache data. Second, when most requests are performed within processing nodes, the absolute amount of traffic that the network must transport is lower than in a system without caches. For caches to be effective in multiprocessors, parallel programs must display processor locality, in addition to the usual temporal and spatial locality. *Processor locality* is defined as the tendency of a processor to repeatedly access a block of data before the block must be relinquished upon a request (typically a write) from another processor [4].

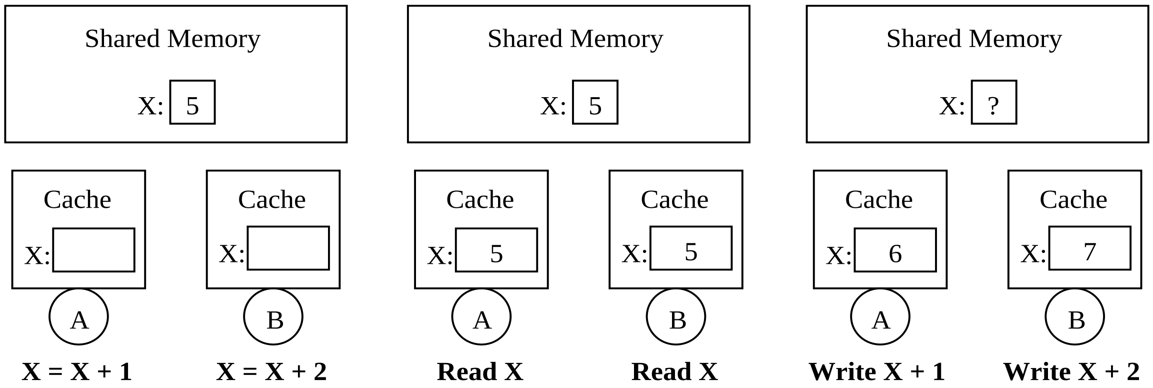


Figure 2-1: Illustration of the cache coherence problem.

2.2 The Cache Coherence Problem

The fact that more than one processor can cache a single block of data leads to the *cache coherence problem*, which is usually explained by an example such as the one that follows: Suppose that processor A and processor B both try to add the integers 1 and 2, respectively, to the value stored in memory location X, which contains the integer 5. (See Figure 2-1.) Both processors cache copies of X, and add their respective values to the cached value. But what should be the final value stored in memory location X?

The answer to this question depends on the shared memory model that is specified for the multiprocessor containing A, B, and X. A programmer's intuition would say that one processor (either A or B) should see the original value (5), the other processor should see the intermediate value (6 or 7), and the final value should be the integer 8, since $5 + 1 + 2 = 5 + 2 + 1 = 8$. Leslie Lamport [33] captured this programmer's intuition when he wrote that a multiprocessor's memory should obey the condition of *sequential consistency*:

> "...the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."

To guarantee sequential consistency (or any other shared-memory model) in a cache-based memory system, all transactions between caches and shared-memory modules are conducted through a *cache coherence protocol*. A cache coherence protocol consists of the set of possible states in the caches, the states of the data in the main memory, and the messages that must be transported through the processor interconnect to keep memory consistent. In order to guarantee consistency, a coherence scheme must sometimes force caches to evict data. Processor locality is determined by the number of times that a processor accesses a cached block of data before the coherence protocol causes the block to be evicted. Dubois, Scheurich, and Briggs have described the various alternatives for maintaining cache coherence and have summarized the methods implemented to date [18, 19]. Archibald's doctoral dissertation [7] is a good reference on the implementation of a wide class of coherence protocols, although the paper by Yen, Yen, and Fu [45] is easier to find in the literature.

While sequential consistency presents a tractable memory model to programmers, it places strong constraints on the memory system. Enforcing sequential consistency requires caches to stall processor write requests to ensure that only one write request is pending at any given time. Otherwise, individual processors may observe write operations in different orders, leading to a violation of the memory model. Several memory system designers have identified less stringent memory models that allow write requests to be overlapped with other processor accesses to shared memory [1, 19, 21]. These *weakly-ordered* models permit the memory system to overlap certain memory transactions by forbidding certain kinds of data sharing semantics. For example, weakly-ordered systems do not guarantee the appropriate behavior in the scenario depicted by Figure 2-1, unless the two processors synchronize between accesses to location X. However, the ability to overlap a number of memory transactions allows a processor to tolerate the latency of access to shared memory. Weakly ordered systems also tend to improve performance by increasing processor locality.

The Alewife architecture takes a slightly different approach to the problem of shared-memory latency. As described in Chapter 3, Alewife's SPARCLE processor overlaps memory transactions by switching between threads of control — which are implemented as hardware contexts — when its cache needs to transmit a request to a remote shared-memory module. Thus, the memory system can overlap one memory transaction per context, while still enforcing a sequentially consistent memory model for each thread of control. Since this research focuses on finding a cache coherence protocol for Alewife, only implementations of sequentially consistent shared-memory models are evaluated. Nevertheless, all of the general classes of protocols that are considered can be implemented to enforce either sequential consistency or weak ordering. Although the main thrust of this research attempts to use caches to minimize the remote latency through the use of caches, it also investigates the relative benefits of weak ordering and multiple contexts.

2.2.1 Classes of Cache Coherence Protocols

The cache coherence problem has been solved adequately for multiprocessors with a small number of processors. Systems of up to about sixteen processors may be constructed by connecting each processor node (consisting of a processor, a cache, and a bus interface) and the memory modules (often distributed with the processor nodes) to a bus. Since each node transmits and receives all of its memory transactions via this common communication resource, any processor node can observe the memory transactions of all of the other nodes in the system. Protocols that take advantage of this technique of covert observation are called *snoopy* cache coherence protocols, because each processor node snoops on the bus transactions of the other nodes. Snoopy protocols have been extensively studied, and have been implemented in several systems [20, 22, 26, 27].

Unfortunately, the protocols used in small multiprocessing systems do not scale up to systems with large numbers of processors, due to physical constraints on the processor interconnect structure. Specifically, a bus simply does not have the bandwidth to support a large number of high-speed processors. Furthermore, transmission problems in a multidrop environment cause the latency of bus transactions to rise with the number of nodes connected to a bus. So, systems with large numbers of processing elements must use point-to-point interconnection networks. But replacing a bus with another type of interconnect removes the broadcast mechanism that is implicit in bus operation. While it is possible to implement a broadcast as a part of a multiprocessor network, such a mechanism incurs a long latency or large hardware cost due to the necessity for receiving acknowledgments from every node in the system. The lack of a broadcast mechanism renders snoopy protocols infeasible for large multiprocessor designs.

Some methods for solving the cache coherence problem in large multiprocessors bypass the problem entirely. For example, the Denelcor HEP [25] avoids the use of caches by hiding memory access latency with fine grain multitasking. However, this system requires an interconnection network with a very high bandwidth.

The NYU Ultracomputer [23] and the IBM RP3 [39] use caches, but avoid the coherence problem by not caching shared data. Instead, caches are only allowed to store copies of private data, shared data that is read-only, and instructions, while accesses to shared data bypass the cache. These systems often try to overlap the latency of write requests to shared data by allowing multiple outstanding writes. When necessary, the multiprocessor software must enforce consistency by issuing *fence* instructions, which stall a processor until all of its previous shared memory requests have been satisfied. While the experimentation with reduced cache use has led to some interesting results in the relationship between program execution and software-based coherence mechanisms [40], the efforts in shared memory research have shifted towards systems that have hardware mechanisms to enforce cache coherence.

This shift has occurred partially due to the fact that, in practice, shared variables must be statically identified to use this coherence scheme. Software systems that can not use compiler or programmer analysis to differentiate between private and shared variables are precluded from using this scheme. Nonetheless, the software-based coherence method is not rejected out-of-hand and is compared with the other protocols for large-scale machines. In later analysis, this coherence method is designated by the acronym OCPD, which stands for *only cache private data.* To facilitate the comparison of this method with hardware-based coherence scheme, sequential consistency is ensured by an implicit fence operation after every reference to a shared variable.

Hardware-based cache coherence protocols that do not use broadcasts store the locations of all cached copies of each block of shared data. This list of cached locations, whether organized as a table or a linked-list, is called a *directory*. There is a directory *entry* for each block of data containing a number of *pointers* to specify the locations of copies of the block. Each directory entry also contains a *dirty* bit to specify whether or not a unique cache has permission to write a given block of data.

2.3 Directory-Based Cache Coherence Protocols

The different flavors of directory protocols that have been devised previously fall under three primary categories: *full-map directories*, *limited directories,* and *chained directories*. Full-map directories [9] store enough state associated with each block in global memory so that every cache in the system can simultaneously store a copy of any block of data. That is, each directory entry contains N pointers, where N is the number of processors in the system. Such directories can be optimized to use a single bit pointer, and the directory can also be physically *distributed* along with main memory, to allow the directory to match the bandwidth of main memory. Due to bandwidth requirements, only *distributed directory schemes* are considered for Alewife. Limited directories [6] differ from full-map directories in that they have a fixed number of pointers per entry, regardless of the number of processors in the system. In order to avoid using a broadcast mechanism, limited directory schemes only permit a small, fixed number of copies of any given block to be cached simultaneously. Chained directories [24] emulate the full-map schemes by building each directory entry as a linked-list structure and by allocating one link to each cache in the list.

Full-Map Directories

Full-map protocols were proposed by Tang [42] and Censier and Feautrier [9]. The Censier and Feautrier protocol uses directory entries with one bit per processor and a dirty bit. Each bit represents the status of the block in the corresponding processor's cache (present or absent). If the dirty bit is set, then one and only one processor's bit is set, and that processor has permission to write into the block. A cache maintains two bits of state per block: One bit indicates whether or not a block is valid; the other bit indicates whether or not a valid block may be written. The cache coherence protocol must keep the state bits in the memory directory and those in the caches consistent.

Figure 2-2a illustrates three different states of a full-map directory. In the first state, *location* X is absent from all of the caches in the system. The second state results from three caches (*caches* A, B, and C) requesting copies of *location* X: Three pointers (processor bits) are set in the entry to indicate the caches that have copies of the block of data. In the first two states, the dirty bit — on the left of the directory entry — is set to clean (C), to indicate that no processor has permission to write to the block of data. The third state results from *cache* C requesting write permission for the block. In this final state, the dirty bit is set to dirty (D), and there is a single pointer to the block of data in *cache* C.

It is worth examining the transition from the second to third states in more detail. Once *processor* C issues the write to *cache* C, the following events transpire:

1. *Cache* C detects that the block containing *location* X is valid, but that the processor does not have permission to write to the block as indicated by the block's write-permission bit in the cache.

2. *Cache* C issues a write request to the memory module containing *location* X and stalls *processor* C.

3. The memory module issues invalidate requests to *caches* A and B.

4. *Cache* A and *cache* B receive the invalidate requests, set the appropriate bit to indicate that the block containing *location* X is invalid, and send acknowledgments back to the memory module.

5. The memory module receives the acknowledgments, sets the dirty bit, clears the pointers to *caches* A and B, and sends write permission to *cache* C.

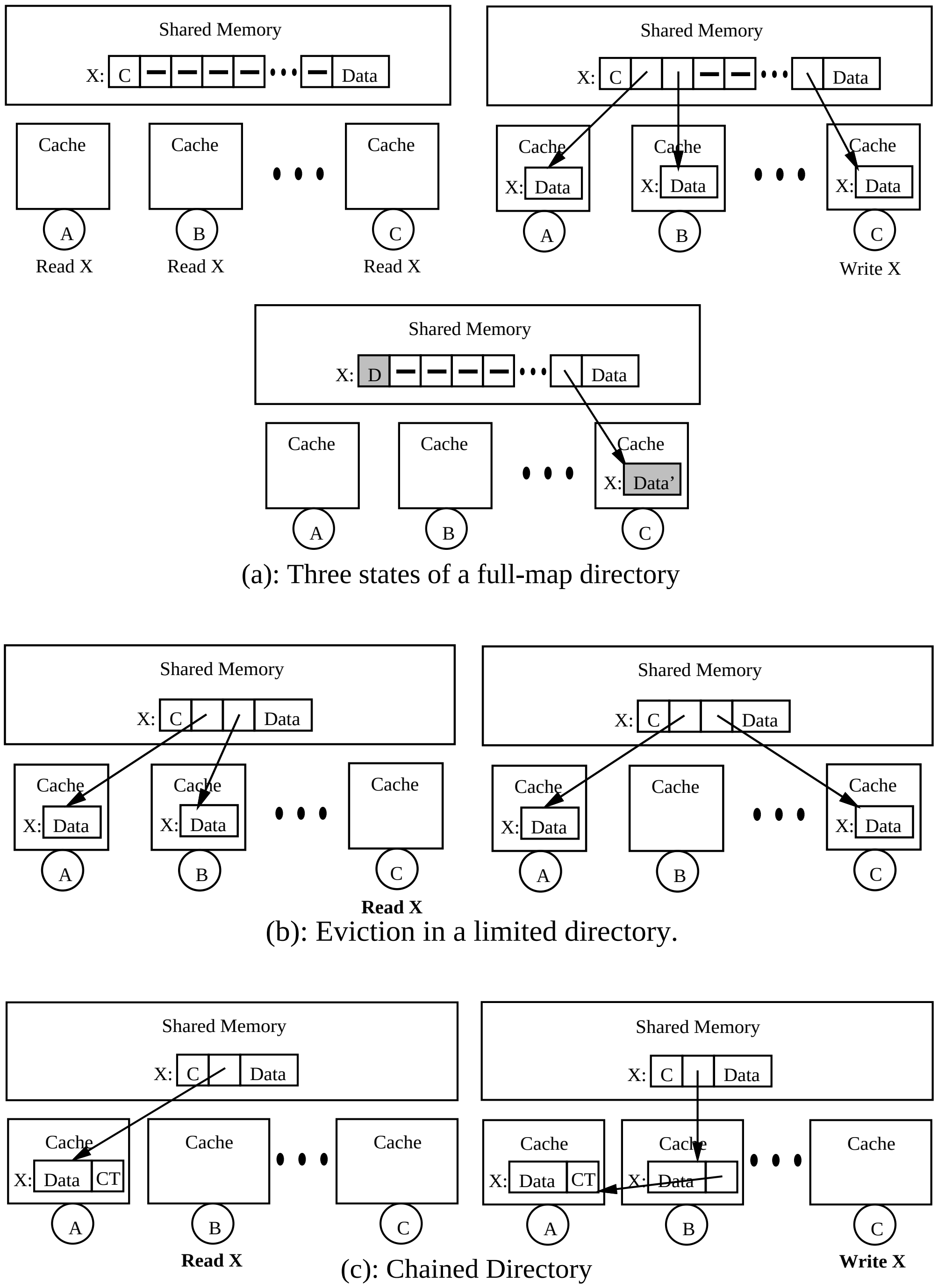


Figure 2-2: Three types of directory protocols.

6. *Cache C* receives the write permission message, updates the state in the cache, and reactivates *processor C*.

Note that the memory module waits to receive the acknowledgments before allowing *processor C* to complete its write transaction. By waiting for acknowledgments, the protocol guarantees that the memory system ensures sequential consistency.

The full-map protocol provides a useful upper bound for the performance of directory-based cache coherence. However, it is not scalable with respect to memory overhead: Assume that the amount of distributed shared memory increases linearly with the number of processors N. Because the size of the directory entry that is associated with each block of memory is proportional to the number of processors, the memory consumed by the directory is proportional to the size of memory ($\Theta(N)$), multiplied by the size of the directory entry ($\Theta(N)$). Thus, the total memory overhead scales as the square of the number of processors ($\Theta(N^2)$).

Limited Directories

Limited directory protocols are designed to solve the directory size problem [6]. By restricting the number of simultaneously cached copies of any particular block of data, it is possible to limit the growth of the directory to a constant factor.

A directory protocol can be classified as Dir_iX using the notation from [6]. The symbol i stands for the number of pointers, and X is NB for a scheme with *N*o *B*roadcast and B for one with *B*roadcast. A full-map scheme without broadcast is represented as Dir_NNB. A limited directory protocol that uses $i < N$ pointers is denoted Dir_iNB. The limited directory protocol is similar to the full-map directory, except in the case when more than i caches request read copies of a particular block of data.

Figure 2-2b shows the situation when three caches request read copies in a memory system with a Dir_2NB protocol. In this case, the 2-pointer directory may be viewed as a 2-way set-associative cache of pointers to shared copies. When *cache C* requests a copy of *location X*, the memory module must invalidate the copy in either *cache A* or *cache B*. This process of pointer replacement is sometimes called *eviction*. Since the directory acts as a set-associative cache, it must have a pointer replacement policy. The protocols evaluated in this study use a pseudorandom eviction policy that is easy to implement and requires no extra memory overhead. In Figure 2-2b, the pointer to *cache B* is replaced by the pointer to *cache C*.

Why might limited directories be successful? If a parallel program exhibits the worker-set property in the sense that in any given interval of time only a small subset of all the processors access a given memory word, then a limited directory is sufficient to capture this small "worker-set" of processors. It is important to recognize that in a properly written multiprocessor program, there can not be an overwhelming number of variables that are both widely-shared and frequently written. If a program utilizes such variables, then the amount of traffic needed to transmit the data associated with the variables would surely exhaust the bandwidth of the interconnection network, no matter what mechanism is used to ensure the shared-memory model.

Directory pointers in a Dir_iNB protocol encode binary processor identifiers, so each pointer requires $i \log_2 N$ bits of memory, where N is the number of processors in the system. Given the same assumptions as for the full-map protocol, the memory overhead of limited directory schemes grows as $\Theta(N \log N)$. These protocols are considered scalable with respect to memory overhead because the resources required to implement them grow approximately linearly with the number of processors in the system.

Dir_iB protocols allow more than i copies of each block of data to exist, but they resort to a broadcast mechanism when more than i cached copies of a block need to be invalidated. However, interconnection networks with point-to-point wires do not provide an efficient system-wide broadcast capability. In such networks, it is also difficult to determine the completion of a broadcast to ensure sequential consistency. While it is possible to limit some Dir_iB broadcasts to a subset of the system (see [6]), the evaluation of limited directories for Alewife is restricted to the Dir_iNB protocols.

Chained Directories

Chained directories, the third option for cache coherence schemes that do not utilize a broadcast mechanism, realize the scalability of limited directories without restricting the number of shared copies of data blocks [24]. This type of cache coherence scheme is called a *chained* scheme, because it keeps track of shared copies of data by maintaining a linked-list of directory pointers.

Two different chained directory schemes have been proposed. The simpler of the two schemes implements a singly-linked chain, which is best described by example (see Figure 2-2c.). Suppose there are no shared copies of *location X*. If *processor A* reads *location* X, the memory sends a copy to *cache A,* along with a chain termination (CT) pointer. The memory also keeps a pointer to *cache A.* Subsequently, when *processor B* reads location X, the memory sends a copy to *cache B,* along with the pointer to *cache A.* The memory then keeps a pointer to *cache B.* By repeating this step, all of the caches can store a copy of *location X.* If *processor C* writes to *location X,* it is necessary to send a data invalidation message down the chain. To ensure sequential consistency, *processor C* is denied write permission by the memory module until the processor with the chain termination pointer acknowledges the invalidation of the chain. Perhaps this scheme should be called a *gossip* protocol (as opposed to a *snoopy* protocol) because information is passed from individual to individual, rather than being spread by covert observation!

Chained directory protocols are complicated by the possibility of cache block replacement. Suppose that *cache* A_1 through *cache* A_N all have copies of *location X,* and that *location* X and *location* Y map to the same (direct-mapped) cache line. If *processor* A_i reads *location* Y, it must first evict *location* X from its cache. In this situation, there are two possibilities: 1) Send a message down the chain to *cache* A_{i-1} with a pointer to *cache* A_{i+1} and splice A_i out of the chain, or 2) Invalidate *location* X in *cache* A_{i+1} through *cache* A_n.

Another solution to the replacement problem is to use a doubly-linked chain. This scheme maintains forward and backward chain pointers for each cached copy so that the protocol does not have to traverse the chain when there is a cache replacement.

The doubly-linked directory optimizes the replacement condition at the cost of a larger average message block size (due to the transmission of extra directory pointers), twice the pointer memory in the caches, and a more complex coherence protocol.

Although the chained protocols are more complex than the limited directory protocols, they are still scalable in terms of the amount of memory used for the directories. The pointer sizes grow as the logarithm of the number of processors, and the number of pointers per cache or memory block is independent of the number of processors.

2.4 Qualitative Evaluation of the Protocols

The abundance of protocols in the literature might lead to the belief that the cache coherence problem has already been solved. However, each of the protocols described in this chapter have defects in terms of scalability or performance. While the full-map directory protocol allows unlimited data sharing, its hardware overhead becomes costly for systems with hundreds of processors. Limited directory protocols solve this problem, but they depend on the assumption that worker-set sizes for *all* memory locations are small. The simulations described in Chapters 4 and 5 show that it is possible to optimize software to reduce the sizes of worker-sets, thereby making limited directory protocols a viable coherence method. However, the performance of a shared-memory system based on a limited directory protocol is extremely sensitive to the extent that multiprocessor software can manage data sharing.

Chained directory protocols suffer from a more subtle problem. Although chained directories perform only slightly worse than full-map directories in simulations of systems with up to 256 processors, the scalability of this class of protocols may be hampered by the structure of the directory. Full-map and limited directory protocols can service processor write requests by transmitting multiple invalidations through the interconnection network at once. The latency of a write request is determined by the rate at which invalidations can be formatted by a memory module, transmitted by the network, and acknowledged by the caches. In contrast, chained directory protocols must perform invalidations in the sequence determined by the linked-list representation of directory entries. Since chained schemes do not benefit from the parallelism inherent in a multiprocessor, the latency of a write request depends strongly on the number of caches in the system. As in the case of limited directories, if worker-set sizes are small, then chained directories should work well even for large systems. However, the linked-list structure may cause extreme sensitivity to the worker-set sizes of individual variables.

As defined, the directory schemes rely on the software to improve performance by reducing the size of shared variables' worker-sets, but none of the protocols allow the software to assume a subset of the memory system's functionality. This restriction is reasonable for systems built with contemporary hardware technology. However, as processor computation speed increases relative to interprocessor communication latency, implementing cache coherence functionality in software will become more and more attractive. The next chapter describes the Alewife architecture and its support for a protocol that allows a more integrated approach to the cache coherence problem.

Chapter 3

Implementing Cache Coherence in Alewife

Since the Alewife project serves as the platform for the analysis of cache coherence protocols, it is important to understand how the the memory system interacts with the rest of the Alewife architecture. This chapter begins by summarizing relevant features of the Alewife multiprocessor, and then describes the implementation of its cache-based memory system. The description emphasizes the reasoning behind design decisions that affect the complexity and the performance of the shared-memory system, rather than the minutia of the implementation of the coherence protocols. As part of the discussion of cache coherence implementation, the specification of the LimitLESS protocol serves as a case study of applying the integrated systems approach to shared memory systems.

3.1 Scalability and Programmability

The goal of the Alewife experiment is to demonstrate that a parallel computer system can be both scalable and easily programmable. For the purposes of this thesis, *scalability* implies that both the physical size of each processing node and the size of the wires that connect the processors are constant with respect to the total number of nodes in the system. This definition of scalability is the basis for the discussion in

Chapter 2 that analyzes coherence protocols in terms of their memory overhead.

3.1.1 The Alewife Processing Node

The Alewife multiprocessor, as depicted in Figure 3-1, is designed to be physically scalable. The system consists of a set of processing nodes that are connected by a mesh network. This type of network is scalable, because the interconnection wire lengths do not depend on the number of nodes in the system. Each node consists of a network router, a SPARCLE processor, a floating-point unit, a cache, a cache-memory controller, and a portion of globally-shared memory. The shared memory is distributed to the processing nodes so that the system does not suffer from the bandwidth bottleneck of a single, monolithic memory. Figure 3-1 shows that the directory used by the cache coherence protocol is also distributed to the processing nodes. The description of the structures within the directory is deferred until later in this chapter.

Dividing the shared memory between the processing elements, rather than allocating it to specialized memory modules as in dance-hall architectures [23], has several benefits: First, the multiprocessor contains only one kind of node. This homogeneity makes the design of the system simpler, because only one type of circuit board needs to be fabricated. Second, the processor can access its local block of shared-memory without using the interconnection network. By allocating private data structures in local memory, Alewife's software reduces the network bandwidth that it requires. Third, the cache-memory controller, implemented as a single VLSI chip, incorporates all of the mechanisms needed to enforce cache coherence for both the cache and the local portion of shared memory. As is shown later in this chapter, the system also profits from the fact that each controller can interact closely with the local SPARCLE processor.

Assuming a scalable cache coherence protocol, the system's processing node is designed to be physically scalable. However, the Alewife architecture is intended to be scalable in a more powerful sense. As the system grows in terms of size, it should also scale in terms of performance. Although recent studies have attempted to

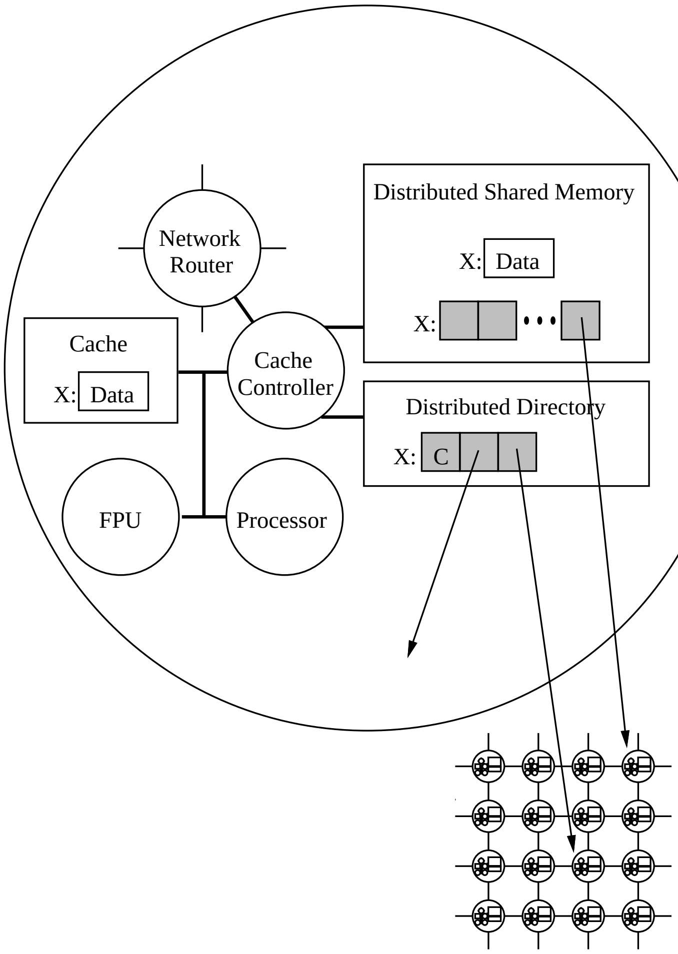


Figure 3-1: An Alewife processing element with a LimitLESS directory entry.

characterize scalable architectures [36], the concept of performance scalability does not have a widely-accepted definition. But from the point of view of the memory system, it is sufficient to recognize that the time needed for processors to communicate and synchronize with each other in a large system is the major obstacle to scalability. If this is the case, then a scalable system somehow must counteract the effects of interprocessor communication latency.

As described in [3], the Alewife architecture accomplishes the goal of scalability by exploiting communication locality:

> A program running on a parallel machine displays communication locality if the probability of communication with various nodes decreases with physical distance.

Unfortunately, it is either difficult or tedious for programmers to have to create communication locality in a system. To achieve both scalability and programmability, the Alewife system presents the programmer with the abstraction of a monolithic, sequentially-consistent memory, while managing locality automatically. Alewife uses a combination of hardware and software techniques to *avoid* communication latency by exploiting locality and to *tolerate* latency when it is unavoidable. The memory system plays a key role in both of these strategies.

3.1.2 Latency Avoidance

Processor caches provide the first line of defense against communication latency. A cache-based memory system automatically moves data into the processor nodes where it is needed. In doing so, the caches take advantage of the principles of temporal, spatial, and processor locality to increase the number of memory accesses that can be performed within a processing node, thereby enhancing the communication locality of multiprocessor programs.

However, the action of caches only enhances locality within processing nodes. The Alewife software implements mechanisms that increase the locality of communication between processing nodes. These software techniques partition multiprocess programs into fragments, called tasks, and attempt to schedule these task in a way that enhances communication locality. Such methods include a partitioning mechanism that balances the size of tasks with the overhead of task creation, communication, and synchronization; scheduling heuristics that assign related tasks to processors that are physically near to one another; and compiler schemes that allocate tasks and their data to maximize locality. These methods are crucial to the scalability of the system as a whole, but they are beyond the scope of this thesis. The evaluation of coherence protocols concentrates on the interplay between software and hardware that is required to enhance locality within processing nodes.

3.1.3 Latency Tolerance

When the system can not avoid interprocessor communication, Alewife attempts to tolerate the latency by switching between hardware contexts on the SPARCLE processor, an implementation of the APRIL processor architecture for multiprocessing [5]. Each of the four hardware contexts on SPARCLE can contain the state of an executable task. When one task must be stalled due to internode communication caused by accessing a variable in a remote portion of memory, the SPARCLE processor rapidly switches to another task's context. (A context switch takes 11 cycles in the current implementation of the processor.) Thus, the context-switch allows the processor to overlap communication latency with useful execution of another part of the program. A cache coherence scheme must assure that the memory requests from each context are satisfied efficiently and correctly. Section 3.3.2 elaborates on the protocol features that are required to support multiple contexts.

3.2 Implementing Directory Protocols

Due to the reasons explained in Chapter 2, all of the potential cache coherence schemes for Alewife are directory-based protocols. For purposes of comparison, other types of protocols have been analyzed during the search for a protocol for Alewife; however, the main thrust of the memory system experimentation involves the simulation and measurement of several different directory-based protocols, including full-map, limited, and chained varieties. This analysis helps determine the relationship between the implementation of a protocol's directory structure and the performance of a shared-memory system.

The most dramatic differences in performance between protocols is caused by the structure of the directory. For applications that use variables with small worker-sets, all of the protocols perform similarly. On the other hand, applications with variables that are shared by many processors exhibit behavior that correlates with the type of directory used by the protocol. Except in anomalous situations, the full-map directory performs better than any other directory-based protocol. This observation should not be surprising, since the full-map protocol is also not scalable in terms of memory overhead. By committing enough resources to cache coherence, it is possible to achieve good performance.

Simulations show that limited directory protocols can perform as well as full-mapped directory protocols, subject to optimization of the software running on a system [11]. Although this result testifies to the fact that scalable cache coherence is possible, limited directories are extremely sensitive to the worker-sets of a program's variables. Chapter 5 examines a case-study of a multiprocessor application that — when properly modified — runs approximately as fast with a limited directory as with a full-map directory. However, when one variable in the program is widely shared, limited directory protocols cause more than a 100% increase in time needed to finish executing the application. This sensitivity to worker-set sizes varies with the program running on the system; but in general, the more variables that are shared among many processors, the worse limited directories perform.

Since chained directory protocols maintain pointers in a linked-list structure, they avoid the problems of limited directories by allowing an unlimited number of cached copies of any given variable. This class of directory protocols performs slightly worse than the full-map protocol, due to the latency of write transactions to shared variables. However, the fact that write latency differentiates the chained and full-map protocols even in 64 processor systems should cause some trepidation. As system sizes grow, the effect of the linked-list structure becomes more pronounced.

3.2.1 The LimitLESS Directory Protocol

The LimitLESS directory protocol uses a different approach to solve the problem of finding a scalable cache coherence problem. As do limited directory protocols, the LimitLESS directory scheme capitalizes on the observation that only a few shared memory data types are widely shared among processors. Many shared data structures have a small *worker-set*, which is defined as the set of processors that concurrently read a memory location. The worker-set of a memory block corresponds to the number of active pointers it would have in a full-map directory entry. The observation that worker-sets are often small has led some memory-system designers to propose the use of a hardware cache of pointers to augment the limited-directory for a few widely-shared memory blocks [37]. However, when running properly optimized software, a directory entry overflow is an exceptional condition in the memory system. The LimitLESS protocol handles such "protocol exceptions" in software. This is the integrated systems approach — handling common cases in hardware and exceptional cases in software.

The LimitLESS scheme implements a small number of hardware pointers for each directory entry. If these pointers are not sufficient to store the locations of all of the cached copies of a given block of memory, then the memory module will interrupt the local processor. The processor will then emulate a full-map directory for the block of memory that caused the interrupt. The structure of the Alewife machine supports an efficient implementation of this memory system extension. Since each processing node in Alewife contains both a memory controller and a processor, it is a straightforward modification of the architecture to couple the responsibilities of these two functional units. This scheme is called LimitLESS, to indicate that it employs a *Limit*ed directory that is *L*ocally *E*xtended through *S*oftware *S*upport. Figure 3-1, an enlarged view of a node in the Alewife machine, depicts a set of directory pointers that correspond to shared data block X, copies of which exist in several caches. In the figure, the software has extended the directory pointer array (which is shaded) into local memory.

Since Alewife's SPARCLE processor is designed with a fast trap mechanism, the overhead of the LimitLESS interrupt is not prohibitive. The emulation of a full-map directory in software prevents the LimitLESS protocol from exhibiting the sensitivity to software optimization that is exhibited by limited directory schemes. But given current technology, the delay needed to emulate a full-map directory completely in software is significant. Consequently, the LimitLESS protocol supports small worker-sets of processors in its limited directory entries, implemented in hardware.

3.2.2 A Simple Model of the LimitLESS Protocol

Before discussing the details of the new coherence scheme, it is instructive to examine a simple model of the relationship between the performance of a full-map directory and the LimitLESS directory scheme. Let T_h be the average remote memory access latency for a full-map directory protocol. T_h encapsulates factors such as the delay in the cache and memory controllers, invalidation latencies, and network latency. Given the hardware protocol latency, T_h, it is possible to estimate the average remote memory access latency for the LimitLESS protocol with the formula: $T_h + mT_s$, where T_s (the software latency) is the average delay for the full-map directory emulation interrupt, and m is the fraction of memory accesses that overflow the small set of pointers implemented in hardware.

For example, simulations of a Weather Forecasting program running on 64 node Alewife memory system (see Chapter 5) indicate that $T_h \approx 35$ cycles. If $T_s = 100$ cycles, then remote accesses with the LimitLESS scheme will be 10% slower (on average) than with the full-map protocol when $m \approx 3\%$. Since the Weather program is, in fact, optimized such that 97% of accesses to remote data locations "hit" in the limited directory, the full-map emulation will cause a 10% delay in servicing requests for data.

LimitLESS directories are scalable, because their memory overhead grows as $\Theta(N \log N)$, and their performance approaches that of a full-map directory as system size increases. Although in a 64 processor machine, T_h and T_s are comparable, in much larger systems the internode communication latency will be much larger than the processors' interrupt handling latency ($T_h \gg T_s$). Furthermore, improving processor technology will make T_s even less significant. In such systems, the LimitLESS protocol will perform about as well as the full-map protocol, even if $m = 1$. This approximation indicates that if both processor speeds and multiprocessor sizes increase, handling cache coherence completely in software will become a viable option. In fact, the LimitLESS protocol is the first step on the migration path towards interrupt-driven cache coherence. Other systems [14] have also experimented with handling cache misses entirely in software.

3.2.3 Background: Implementing a Full-Map Directory

Since the LimitLESS coherence scheme is a hybrid of the full-map and limited directory protocols, this new cache coherence scheme should be studied in the context of its predecessors. In the case of a full-map directory, one pointer for every cache in the multiprocessor is stored, along with the state of the associated memory block, in a single directory entry. The directory entry, illustrated in Figure 3-2, is physically located in the same node as the associated data. Since there is a one-to-one mapping between the caches and the pointers, the full-map protocol optimizes the size of the pointer array by storing just one bit per cache. A pointer-bit indicates whether or not the corresponding cache has a copy of the data. The implementation of the protocol allows a memory block to be listed in one of four states, which are listed in Table 3.1. These states are mirrored by the state of the block in the caches, also listed in Table 3.1. It is the responsibility of the protocol to keep the states of the memory and cache blocks coherent. For example, a block in the Read-Only state may be shared by a number of caches (as indicated by the pointer array). Each of these cached copies are marked with the Read-Only cache state to indicate that the local processor may only read the data in the block.

Before any processor modifies a block in an Invalid or Read-Only cache state, it first requests permission from the memory module that contains the data. At this point, the memory controller sends invalidations to each of the cached copies.

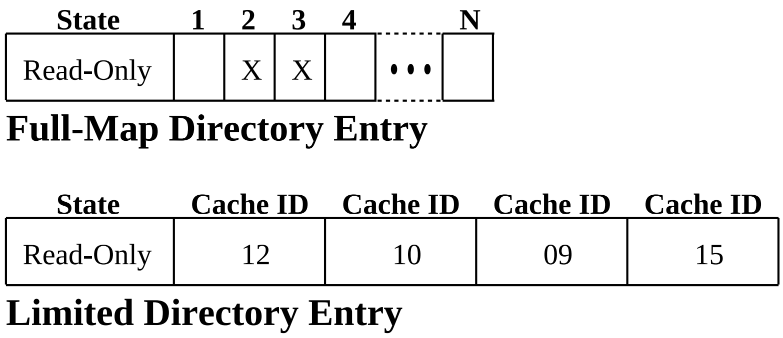


Figure 3-2: Full-map and limited directory entries. The full-map pointer array is optimized as a bit-vector. The limited directory entry has four pointers.

Component	Name	Meaning
Memory	Read-Only	Caches have read-only copies of the data.
	Read-Write	One cache has a read-write copy of the data.
	Read-Transaction	Holding read request, update is in progress.
	Write-Transaction	Holding write request, invalidation is in progress.
Cache	Invalid	Cache block may not be read or written.
	Read-Only	Cache block may be read, but not written.
	Read-Write	Cache block may be read or written.

Table 3.1: Directory states.

The caches then invalidate the copy (change the block's state from Read-Only to Invalid), and send an acknowledgment message back to the memory. The memory controller uses the Write-Transaction state to indicate that a memory location is awaiting acknowledgments, and sets a pointer to designate the cache that initiated the request. If the memory controller has a mechanism for counting the number of invalidations sent and the number of acknowledgments received, then the invalidations and acknowledgments may travel through the system's interconnection network in parallel. By allowing only one outstanding transaction per memory module, it is possible to avoid the extra memory overhead needed to store the Write-Transaction state and the acknowledgment counter for each directory entry. However, such a modification reduces the overall bandwidth of the memory system and exacerbates the effect of hot-spots, which may be caused by accesses to different variables that happen to be allocated in the same memory module. When the memory controller receives the appropriate number of acknowledgments, it changes the state of the block to Read-Write and sends a write permission message to the cache that originated the transaction. In a sense, the cache "owns" the block until another cache requests access to the data.

Changing a block from the Read-Write to the Read-Only state involves an analogous (but slightly simpler) transaction. When a cache requests to read a block that is currently in the Read-Write state, the memory controller sends an update request to the cache that owns the data. The memory controller marks a block that is waiting for data with the Read-Transaction state. As in the Write-Transaction state, a pointer is set to indicate the cache that initiated the read transaction. When a cache receives an update request, it invalidates its copy of the data, and replies to the memory controller with an update message that contains the modified data, so that the original read request can be satisfied.

The protocol might be modified so that a cache changes a block from the Read-Write to the Read-Only (instead of the Invalid) state upon receiving an update request. Such a modification assumes that data that is written by one processor and then read by another will be read by both processors after the write. While this modification optimizes the protocol for frequently-written and widely-shared variables, it increases the latency of access to migratory or producer-consumer data types. The dilemma of choosing between these two types of data raises an important question: Should a cache coherence protocol optimize for frequently-written and widely-shared data? This type of data requires excessive bandwidth from a multiprocessor's interconnection network, whether or not the system employs caches to reduce the *average* memory access latency. Since the problems of frequently-written and widely-shared data are independent of the coherence scheme, it seems futile to try to optimize accesses to this type of data, when the accesses to other data types can be expedited. This decision forces the onus of eliminating the troublesome data type on the multiprocessor software designer. However, the decision seems reasonable in light of the physical limitations of communication networks.

The basic protocol that is described above is somewhat complicated by the associativity of cache lines. In a cache, more than one memory block can map to a single block of storage, called a line. Depending on the memory access pattern of its processor, a cache may need to replace a block of memory with another block of memory that must be stored in the same cache line. While the protocol must account for the effect of replacements to ensure a coherent model of shared memory, in systems with large caches, replacements are rare events except in pathological memory access patterns. Several options for handling the replacement problem in various coherence protocols have been explored. Simulations show that the differences between these options do not contribute significantly to the bottom-line performance of the coherence schemes, so the final choice of replacement handling for the Alewife machine should optimize for the simplicity of the protocol (in terms of cache states, memory states, and the number of messages).

3.2.4 Specification of the LimitLESS Scheme

The model in Section 3.2.2 assumes that the hardware latency (T_h) is approximately equal for the full-map and the LimitLESS directories, because the LimitLESS protocol has the same state transition diagram as the full-map protocol. The memory controller side of this protocol is illustrated in Figure 3-3, which contains the memory states listed in Table 3.1. Both the full-map and the LimitLESS protocols enforce coherence by transmitting messages (listed in Table 3.3) between the cache/memory controllers. Every message contains the address of a memory block, to indicate which directory entry should be used when processing the message. Table 3.3 also indicates whether a message contains the data associated with a memory block.

The state transition diagram in Figure 3-3 specifies the states, the composition of the pointer set (P), and the transitions between the states. This diagram specifies a simplified version of the protocol implemented in ASIM, the Alewife system simulator, which is described in Chapter 4. For the purposes of describing the implementation of directory protocols, Figure 3-3 includes only the core of the full-map and LimitLESS protocols. Unessential optimizations and other types of coherence schemes have been omitted to emphasize the important features of the coherence schemes. See Appendix B for a complete definition of the protocols implemented in ASIM.

Each transition in the diagram is labeled with a number that refers to its specifica-

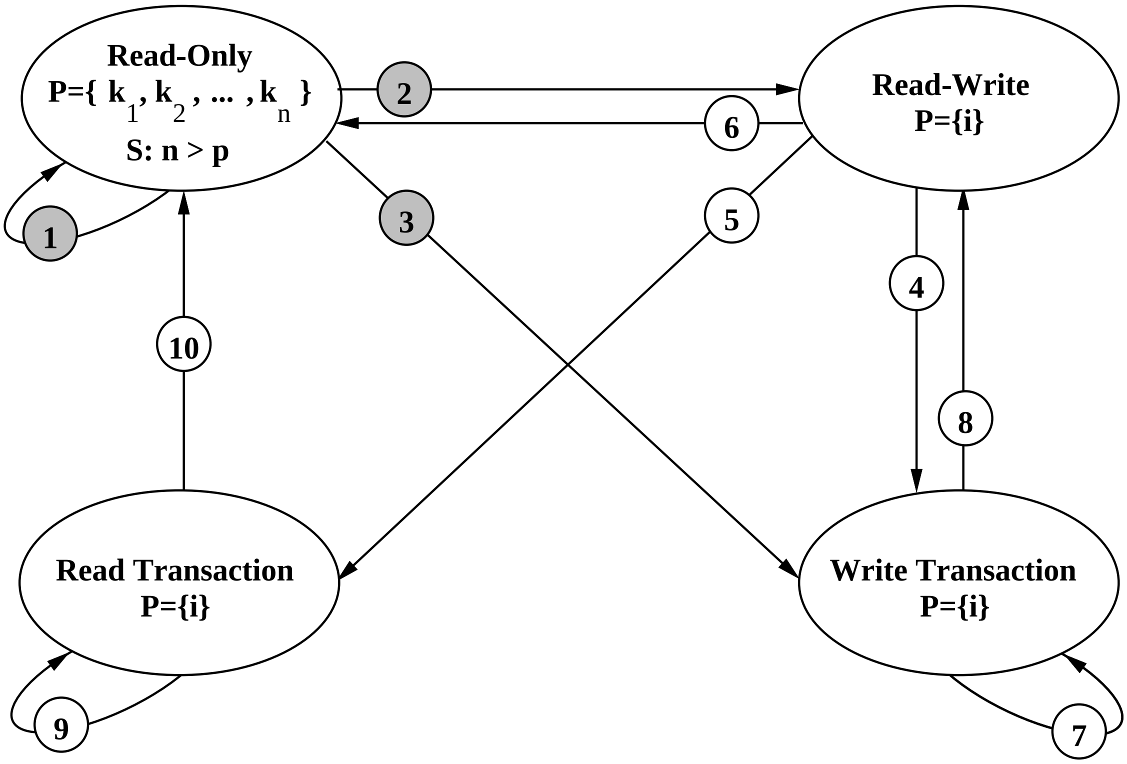


Figure 3-3: Directory state transition diagram for the full-map and LimitLESS coherence schemes.

#	Input Message	Precondition	Directory Entry Change	Output Message(s)
1	$i \rightarrow$ RREQ	—	$P = P \cup \{i\}$	RDATA $\rightarrow i$
2	$i \rightarrow$ WREQ	$P = \{i\}$	—	WDATA $\rightarrow i$
	$i \rightarrow$ WREQ	$P = \{\}$	$P = \{i\}$	WDATA $\rightarrow i$
3	$i \rightarrow$ WREQ	$P = \{k_1, \ldots, k_n\} \wedge i \notin P$	$P = \{i\}$, AckCtr $= n$	$\forall k_j$ INV $\rightarrow k_j$
	$i \rightarrow$ WREQ	$P = \{k_1, \ldots, k_n\} \wedge i \in P$	$P = \{i\}$, AckCtr $= n - 1$	$\forall k_j \neq i$ INV $\rightarrow k_j$
4	$j \rightarrow$ WREQ	$P = \{i\}$	$P = \{j\}$	INV $\rightarrow i$
5	$j \rightarrow$ RREQ	$P = \{i\}$	$P = \{j\}$	INV $\rightarrow i$
6	$i \rightarrow$ REPM	$P = \{i\}$	$P = \{\}$	—
7	$j \rightarrow$ RREQ	—	—	BUSY $\rightarrow j$
	$j \rightarrow$ WREQ	—	—	BUSY $\rightarrow j$
	$j \rightarrow$ ACKC	AckCtr $\neq 1$	AckCtr $=$ AckCtr $- 1$	—
	$j \rightarrow$ REPM	—	—	—
8	$j \rightarrow$ ACKC	AckCtr $= 1$, $P = \{i\}$	—	WDATA $\rightarrow i$
	$j \rightarrow$ UPDATE	$P = \{i\}$	—	WDATA $\rightarrow i$
9	$j \rightarrow$ RREQ	—	—	BUSY $\rightarrow j$
	$j \rightarrow$ WREQ	—	—	BUSY $\rightarrow j$
	$j \rightarrow$ REPM	—	—	—
10	$j \rightarrow$ UPDATE	$P = \{i\}$	—	RDATA $\rightarrow i$

Table 3.2: Annotation of the state transition diagram.

tion in Table 3.2. This table annotates the transitions with the following information: 1. The *input message* from a cache that initiates the transaction and the identifier of the cache that sends it. 2. A *precondition* (if any) for executing the transition. 3. Any *directory entry change* that the transition may require. 4. The *output message* or messages that are sent in response to the input message. Note that certain transitions require the use of an acknowledgment counter (*AckCtr*), which is used to ensure that cached copies are invalidated before allowing a write transaction to be completed.

For example, Transition 2 from the Read-Only state to the Read-Write state is taken when cache i requests write permission (WREQ) and the pointer set is empty or contains just cache i ($P = \{\}$ or $P = \{i\}$). In this case, the pointer set is modified to contain i (if necessary) and the memory controller issues a message containing the data of the block to be written (WDATA).

Following the notation in [6], both full-map and LimitLESS are members of the $Dir_N NB$ class of cache coherence protocols. Therefore, from the point of view of the protocol specification, the LimitLESS scheme does not differ substantially from the full-map protocol. In fact, the LimitLESS protocol is also specified by Figure 3-3.

Type	Symbol	Name	Data?
Cache to Memory	RREQ	Read Request	
	WREQ	Write Request	
	REPM	Replace Modified	$\surd$
	UPDATE	Update	$\surd$
	ACKC	Invalidate Acknowledge	
Memory to Cache	RDATA	Read Data	$\surd$
	WDATA	Write Data	$\surd$
	INV	Invalidate	
	BUSY	Busy Signal	

Table 3.3: Cache coherence protocol messages.

The extra notation on the Read-Only ellipse ($S : n > p$) indicates that the state is handled in software when the size of the pointer set (n) exceeds the size of the limited directory entry (p). In this situation, the transitions with the shaded labels (1, 2, and 3) are executed by the interrupt handler on the processor that is local to the overflowing directory. When the protocol changes from a software-handled state to a hardware-handled state, the processor must modify the directory state so that the memory controller can resume responsibility for the protocol transitions.

While the low-level implementation of the LimitLESS directory scheme is beyond the scope of this thesis, the hardware mechanisms that are required to implement the protocol are as follows:

1. A fast interrupt mechanism: A processor must be able to interrupt application code and switch to LimitLESS protocol code rapidly. This ability makes the overhead of emulating a full-map directory (T_s) small, and thus makes the LimitLESS scheme competitive with schemes that are implemented completely in hardware. The initial implementation of SPARCLE will be able to switch to LimitLESS code in five to ten cycles.

2. Processor to network interface: In order to emulate the protocol functions normally performed by the cache/memory controller, the processor must be able to send and to receive messages from the interconnection network.

3. Extra directory state: Each directory entry must hold the extra state necessary to indicate whether the processor is holding overflow pointers.

In Alewife, none of these mechanisms exist exclusively to support the LimitLESS protocol. The SPARCLE processor uses the same mechanism to execute an interrupt quickly as it uses to provide a fast context-switch. The processor to network interface is implemented through the interprocessor interrupt (IPI) mechanism, which is designed to increase Alewife's I/O performance and to provide a generic message-passing primitive. A small extension to the extra directory state required for the LimitLESS protocol allows diverse coherence and synchronization data types to be constructed in software. See [12, 31] for details of the implementation of these mechanisms.

3.2.5 Estimating LimitLESS Directory Performance

For the purpose of evaluating the potential benefits of the LimitLESS coherence scheme, an approximation of the protocol was implemented in ASIM. The technique assumes that the overhead of the LimitLESS full-map emulation interrupt is approximately the same for all memory requests that overflow a directory entry's pointer array. This is the T_s parameter described in Section 3.2.2. During the simulations, ASIM simulates an ordinary full-map protocol. When the simulator encounters a pointer array overflow, it stalls both the memory controller and the processor that would handle the LimitLESS interrupt for T_s cycles. While this evaluation technique only approximates the actual behavior of the fully-operational LimitLESS scheme, it is a reasonable method for determining whether to expend the greater effort needed to implement the complete protocol.

Initial estimates of the performance of the LimitLESS protocol are encouraging. For applications that perform as well with a limited directory as with a full-map directory, the LimitLESS directory causes little degradation in performance. When limited directories perform significantly worse than a full-map directory, the LimitLESS scheme tends to perform about as well as full-map, depending on the number of widely-shared variables. If a program has just one or two widely-shared variables, a LimitLESS protocol avoids hot-spot contention that tends to destroy the performance of limited directories. Chapter 5 gives a case-study of such an application. On the other hand, the performance of the LimitLESS protocol degrades when a program utilizes variables that are both widely-shared and frequently written. But as discussed in previous sections, these types of variables tend to exhaust the bandwidth of the interconnection network, no matter what coherence scheme is used by the memory system.

In general, preliminary simulation results indicate that the LimitLESS scheme approaches the performance of a full-mapped directory protocol with the memory efficiency of a limited directory protocol. The success of this new coherence protocol emphasizes two key principles: First, the integrated systems approach can successfully be applied to the design of a shared-memory system. Second, the implementation of a protocol's directory structure correlates closely with the performance of the memory system as a whole.

3.3 Implementation Issues in Alewife

The state transition diagram in Figure 3-3 specifies a basic cache coherence protocol that may be implemented on any multiprocessor with distributed memory. However, there are some features of the Alewife architecture that allow optimizations of the protocols or that require additional support from the protocols. This section specifies protocol functionality that supports two special features of the Alewife machine; namely, the distribution of shared memory to processing elements and the fast context switch capability.

3.3.1 Alewife's Processor-Controller Interface

When a processor needs to perform a load or store access to shared memory, the controller responds with one of the three signals listed in Table 3.4. The READY response indicates that the access is complete. In the case of a load, the data is available in the cache. In the case of a store, the coherence protocol has obtained permission to write the data in the cache. The SWITCH response indicates that a transaction with a remote node is necessary to service the processor's request. Normally, this condition will cause the processor to switch contexts.

Name	Controller Response
READY	Access complete at the end of the current cycle
SWITCH	Context switch
WAIT	Repeat the same access on the next cycle

Table 3.4: Controller Response Types

The Alewife controller uses the WAIT response, which is analogous to a uniprocessor memory wait state, to optimize requests from a processor to its local memory. Since each ALEWIFE node contains a cache memory, a main memory, and a cache controller (see Figure 3-1), it is possible that the controller may need to send a protocol message to itself. For example, the directory associated with a controller may have to invalidate a block in the controller's cache. From the standpoint of the hardware, it is wasteful (and perhaps impossible) to use the standard mechanism for transmitting a message through the network when the message is directed from a node to itself.

When a processor needs to access a location in its local portion of shared memory that is not present in its cache, the controller can sometimes satisfy the request in a shorter amount of time than if the controller had to transmit a request through the network. In this case, the controller causes the processor to WAIT until the data can be read from local memory into the cache. While this mechanism is not an essential part of the protocol, it demonstrates how a coherence protocol may take advantage of its target architecture.

3.3.2 Support for Multiple Contexts

Alewife's memory system must provide for the context switching processor in two ways: First, it is possible for a context to be switched (but not unloaded) when it makes an unsuccessful data access, but to become active again before its initial request is satisfied. In the worst case, all of the contexts executing on a processor may be waiting for outstanding memory transactions. If these contexts repeatedly transmit their data requests, they can overload the network with redundant traffic. Thus, the protocol can allow only one outstanding request for any memory transaction.

Second, the protocol must ensure *forward progress* in all accesses to memory.

Given an incorrect coherence protocol, the interference of cache accesses by different contexts can create cyclic *thrashing* situations, which prevent both contexts from ever receiving data. While such situations may be rare, they are as fatal as any other infinite loop.

In this section, scenarios involving processor data requests are illustrated to clarify the need for various protocol features. Each figure consists of a number pairs of "frames." A frame on the left-hand side of a figure is the cause of a protocol transition, and a frame on the right is the effect of the same transition. For example, in Figure 3-4, Frame 1 depicts a write access from *Context 1* of processor *Processor A*. Frame 2 shows the effect of the action in Frame 1: the cache controller sends a WREQ message to a shared-memory module, and instructs the processor to SWITCH contexts. All of the transitions in the figure are derived from the protocol state transition diagrams that are presented in Appendix B. The frames in each scenario are numbered in chronological order.

Network Wait States

In a machine with processors that do not have a context-switching capability, the memory system must sometimes stall processors that require access to remote data. The network wait cache states generalize on the stall mechanism to solve the problem of redundant data requests, described above. Table 3.5 lists the additional states, which augment the states in Table 3.1.

The function of the network wait states is best illustrated by an example. Assume that *Context 1* and *Context 2* on *Processor A* are both accessing *X*, a location in memory. In Frame 1 of Figure 3-4, *Context 1* attempts to modify the location. Since the cache block containing *X* is in the Read Only state, *Context 1* can not write to the location. So in Frame 2, the controller switches the context on *Processor A* and sends a write request over the network. At this point, the state of the cache block is changed to Read Only Network Wait to indicate that the data in the cached location is valid (although it may not be written) and that there is currently an outstanding request for the cache line. Because the block is in the Read Only Network Wait state,

Name	Meaning
Invalid Network Wait	Invalid, network request is pending for the block.
Read Only Network Wait	Read Only, network request is pending for the block.
Read/Write Network Wait	Read/Write, network request is pending for the block.

Table 3.5: Cache network wait states.

after *Context 2* attempts to read X in the Frame 3, its request is satisfied in Frame 4.

This reflects an important optimization of the network wait states. It would be possible to consolidate the three different network wait states (in Table 3.5) into one network wait state. However, a unique network wait state would disallow accesses to a cache block while network requests associated with the block were pending. Furthermore, when a cache receives a BUSY signal from a memory module, it can restore the cache line to its state before the busied request was sent. With a unique network wait state, a BUSY signal would force the cache line to become invalid. Thus, separating the various network wait states improves the average latency of memory accesses when different contexts share the same cache.

Frames 5 and 6 of Figure 3-4 illustrate the other important property of the network wait states. If *Context 1* becomes active before its data request is completed, its subsequent requests for the same data location will not create additional requests, which could clog the network. Note that in Frame 2 (after the initial access), the controller transmits a WREQ message; but in Frame 6 (after a subsequent request), the controller does not transmit any messages.

Frames 7 through 10 of Figure 3-5 show the normal completion of the transaction: In Frame 7, the response from the remote memory module arrives and causes the cache block state to be changed to Read-Write in Frame 8. Finally, *Context 1* requests the data in Frame 9 and receives a READY signal in Frame 10.

Types of Thrashing

Ensuring forward progress (or preventing thrashing cycles) is a more subtle problem than preventing multiple outstanding data requests. There are two forms of thrashing, both of which are caused by the fact that a context may not be active when its data

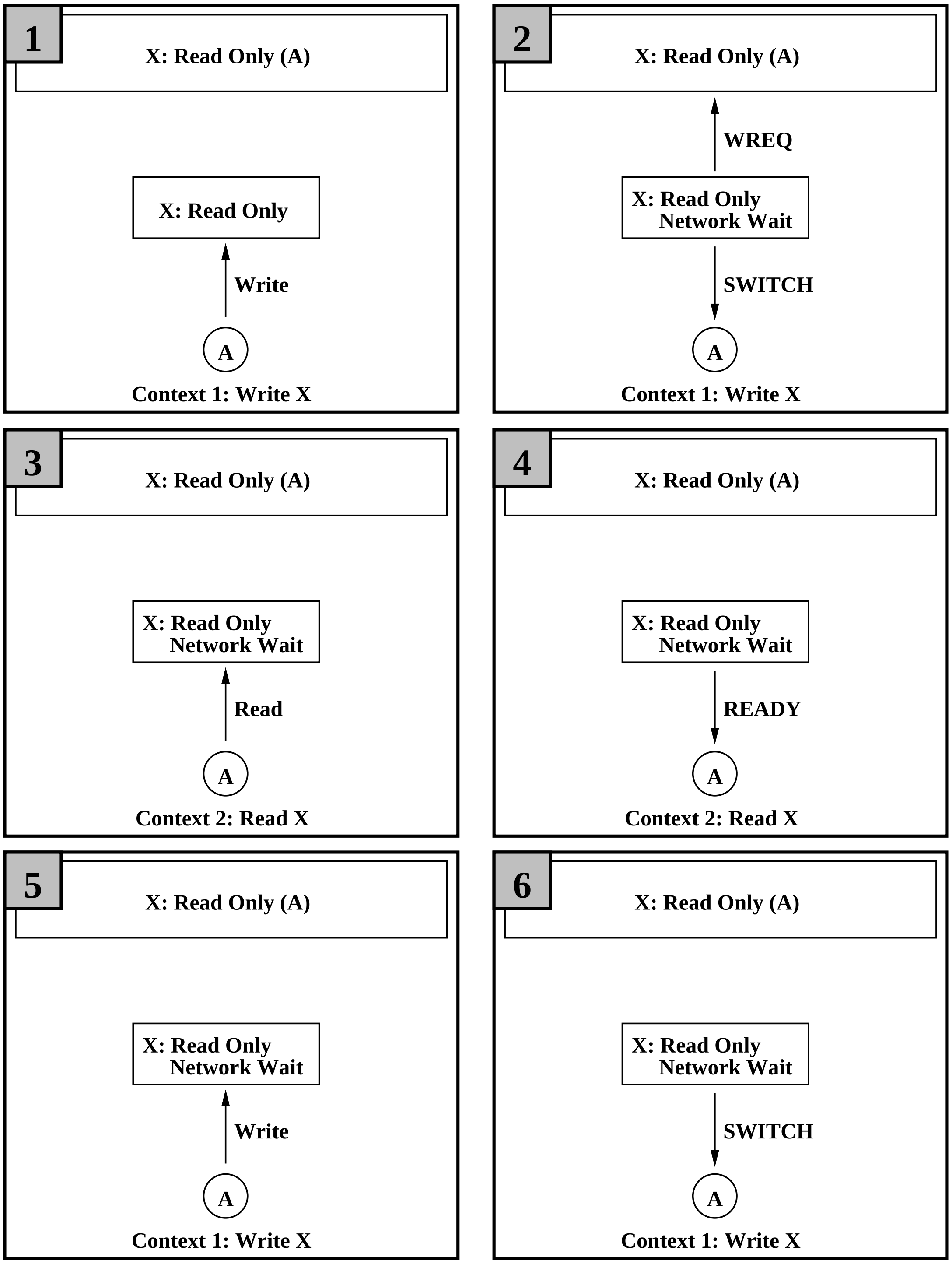


Figure 3-4: Scenario involving a network wait state. (Frames 1 through 6.)

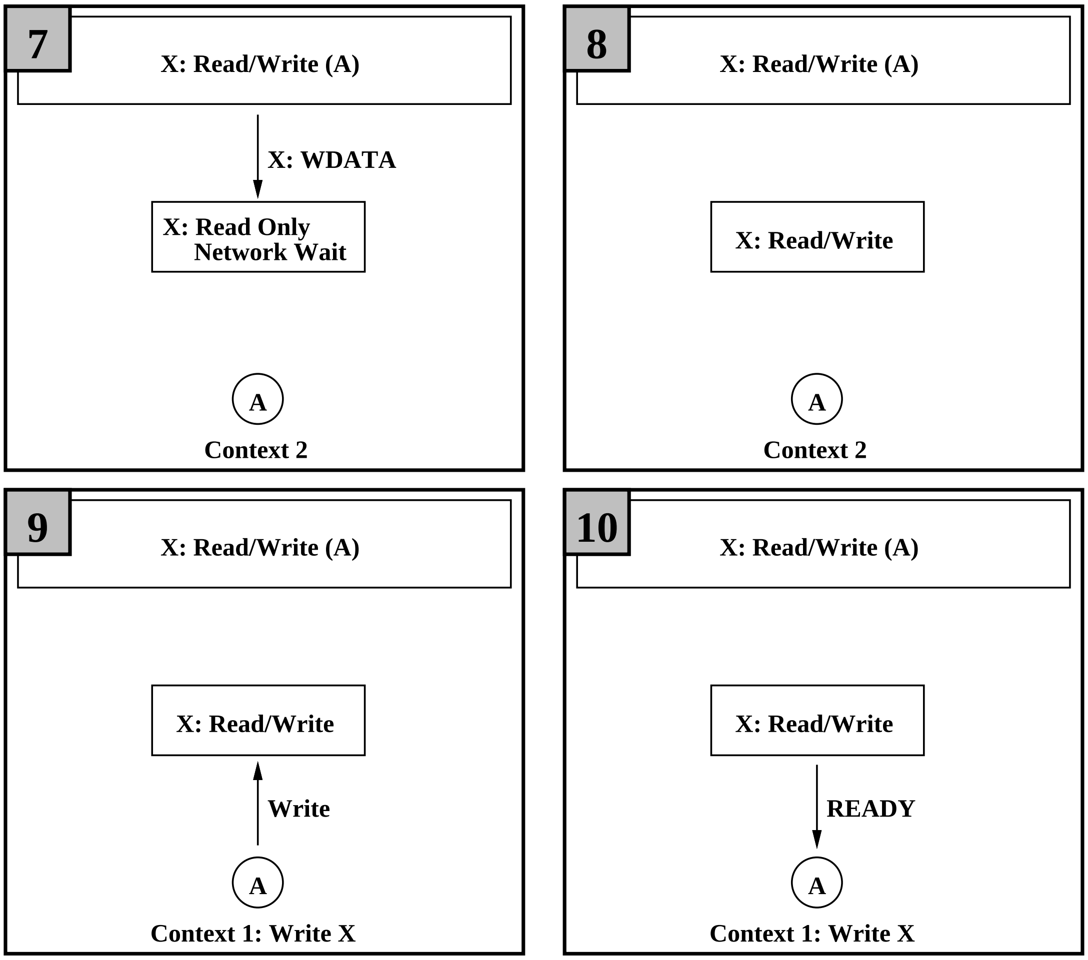


Figure 3-5: Scenario involving a network wait state. (Frames 7 through 10.)

request is satisfied. *Replacement thrashing* occurs when two contexts on the *same* processor attempt to access two data blocks that map to the same cache line, and *invalidation thrashing* occurs when contexts on two *different* processors attempt to access the same data block.

Figures 3-6 and 3-7 show an example of replacement thrashing. *Context 1* on *Processor A* attempts to write to *Location X* in memory, while *Context 3* on the same processor attempts to write to *Location Y*. Unfortunately, both *X* and *Y* map to the same cache line, so the two memory locations cannot be stored in the cache simultaneously. In Frame 1 of Figure 3-6, *Context 1* encounters a normal write miss in the cache. The controller sends out a WREQ message, changes the state of the cache block to Read/Write Network Wait, and context switches the processor in Frame 2. Frames 3 and 4 show the arrival of the data for *Location X* at *Processor A* when *Context 1* is not active. In Frame 5, *Context 3* becomes active and finds that its data is not in the cache, and initiates a request to write *Location Y* in Frame 6. Frames 7 and 8 are symmetric to Frames 3 and 4: Data arrives for *Context 3*, but the context is not active at the time. The loop is completed in Frames 9 and 10 when *Context 1* repeats the cache miss that initiated the cycle. In this scenario, *Context 2* and *Context 4* may make forward progress, but it is possible that these two contexts are also trapped in a replacement thrashing loop. If this is the case, then the whole system will eventually come to a grinding halt while it waits forever for the contexts on *Processor A* to terminate.

Figures 3-8 and 3-9 illustrate a sequence of accesses that causes invalidation thrashing. *Context 1* on *Processor A* and *Context 1* on *Processor B* are both attempting to write to *X*, a location in shared memory. The first two frames of Figure 3-8 show the initial request from *Processor A* and the resulting WREQ message that is transmitted over the network. In Frames 3 and 4, the data corresponding to the request from *Context 1* arrives at *Processor A*, but *Context 1* is not active at the time. While *Context 1* on *Processor A* is inactive, *Context 1* on *Processor B* also makes a write request for *X* in Frames 5 and 6. In Frame 7, the invalidation message (INV) caused by the write request from *Processor B* arrives at *Processor A* before *Context*

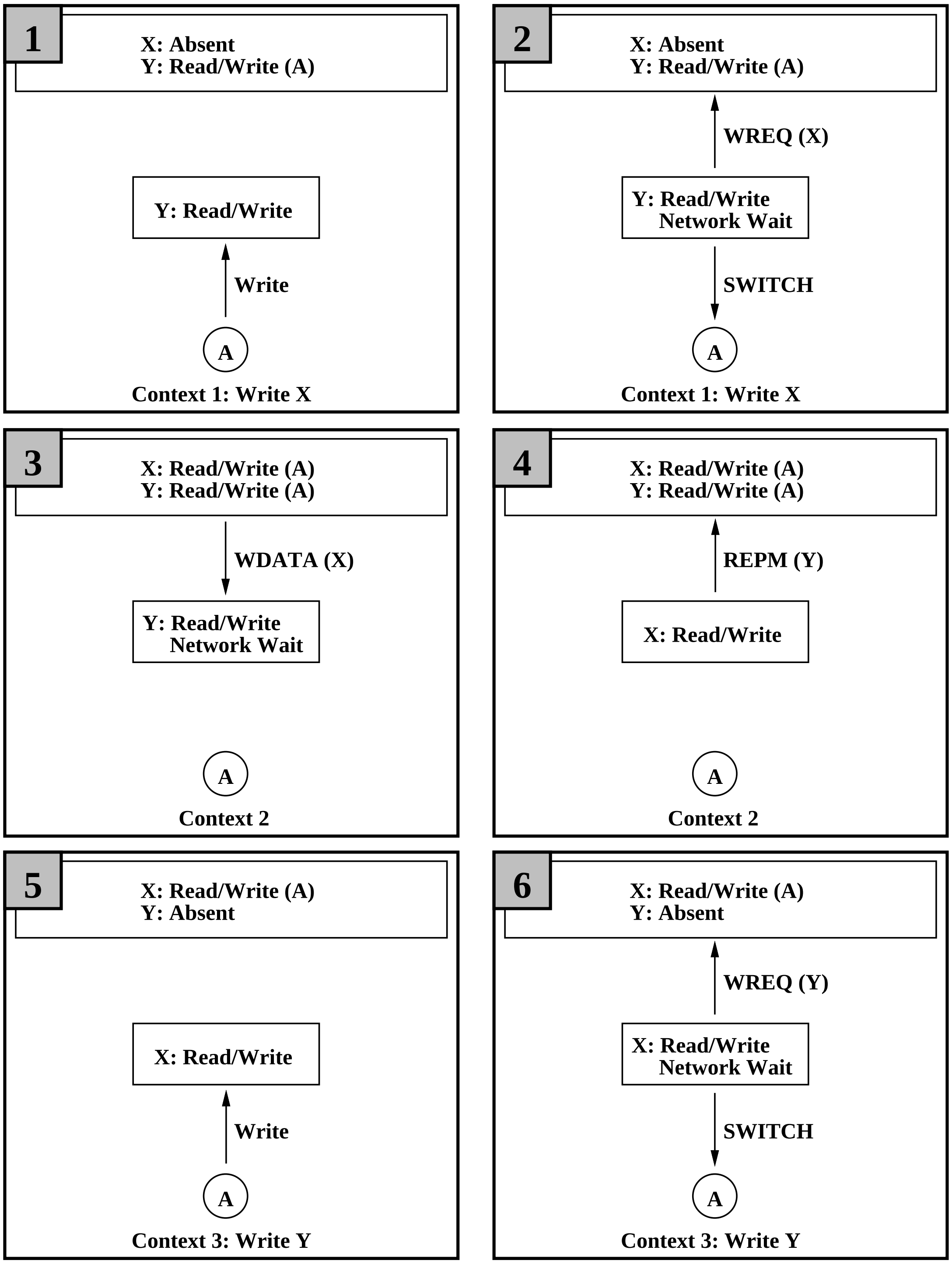


Figure 3-6: Replacement thrashing scenario. (Frames 1 through 6.)

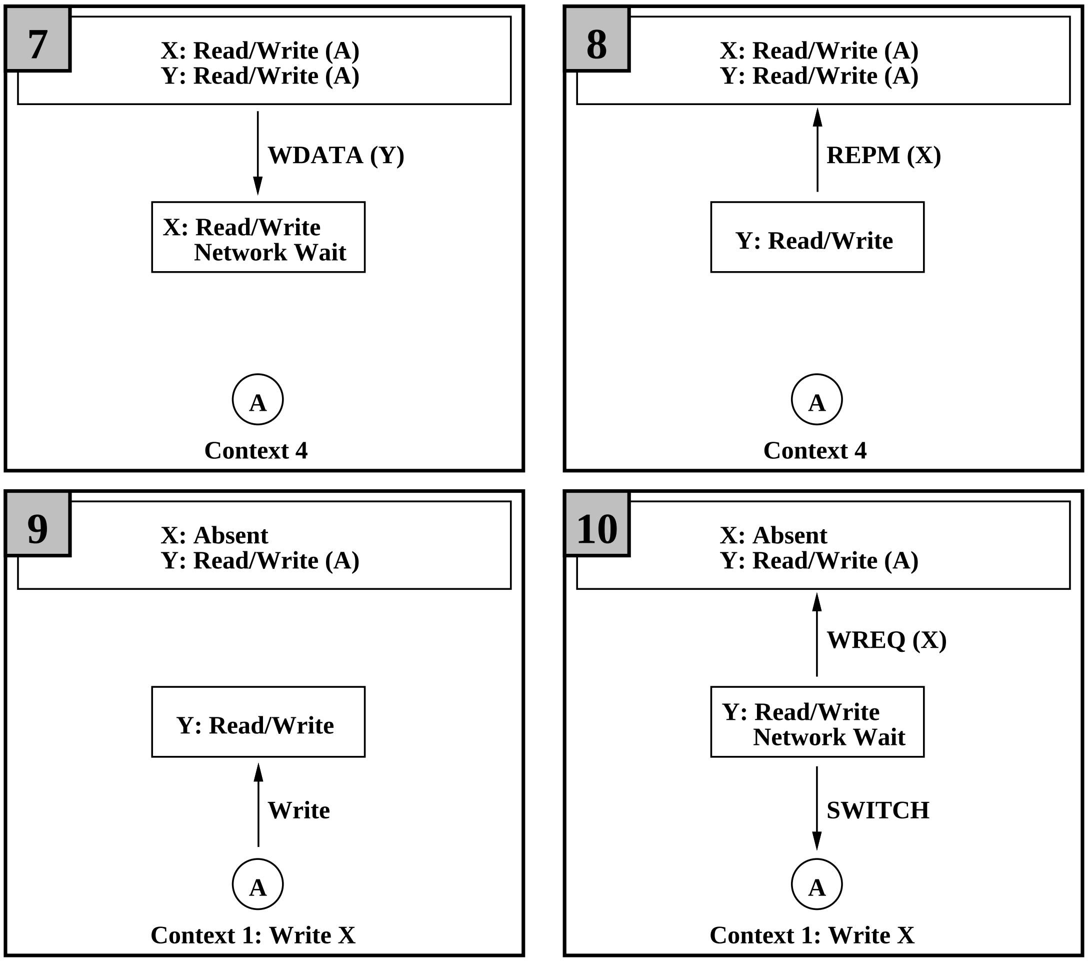


Figure 3-7: Replacement thrashing scenario. (Frames 7 through 10.)

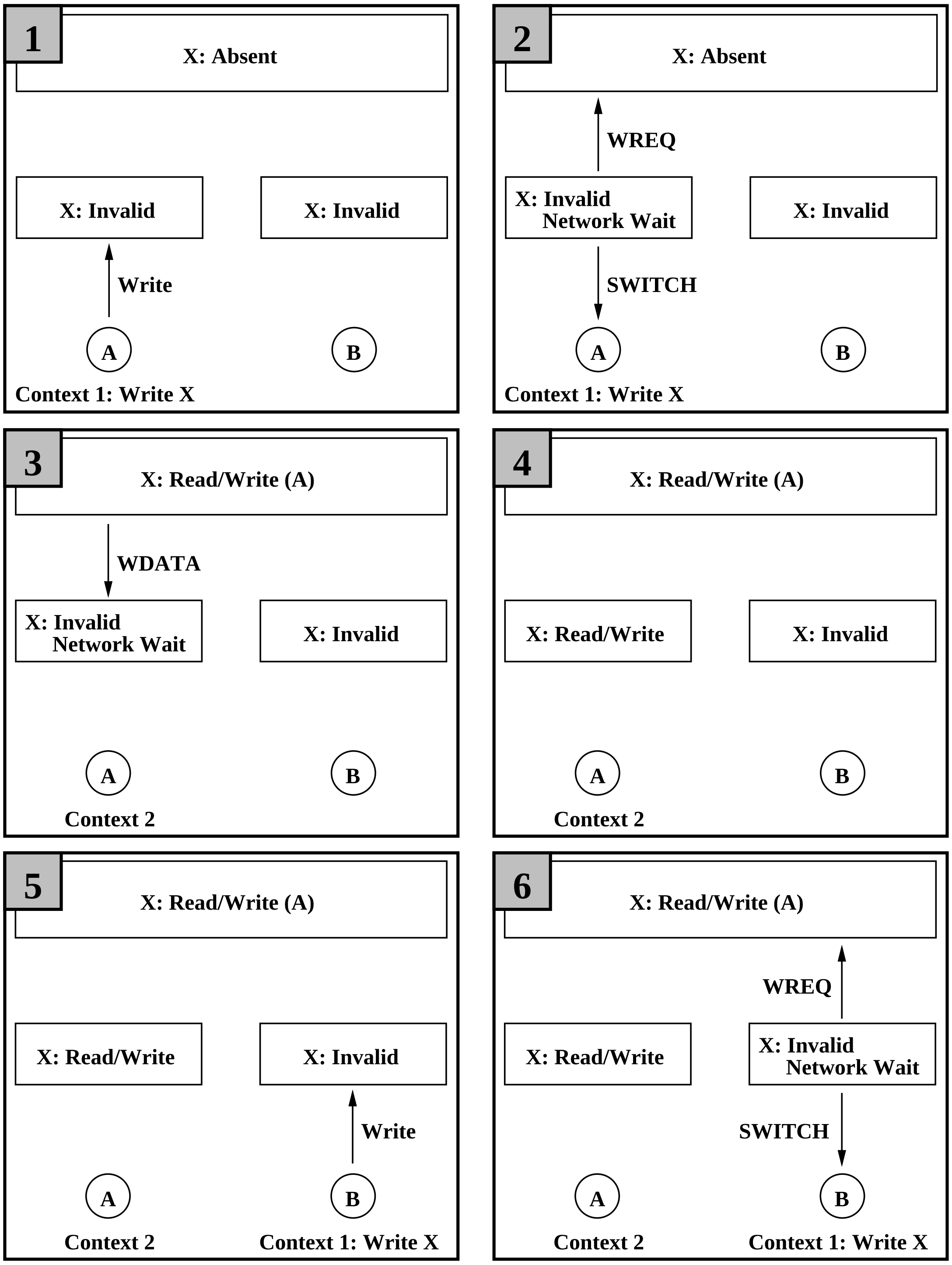


Figure 3-8: Invalidation thrashing scenario. (Frames 1 through 6.)

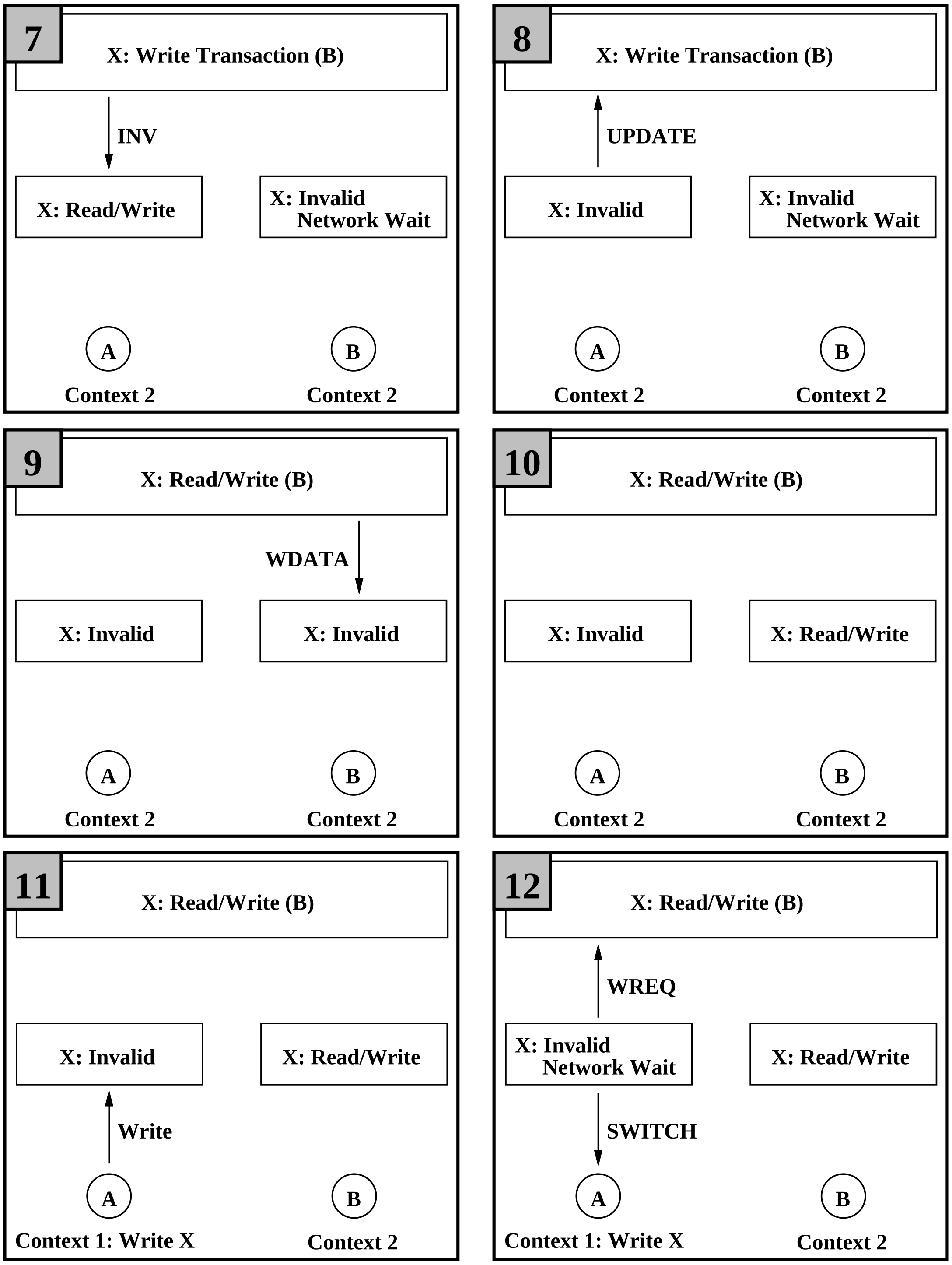


Figure 3-9: Invalidation thrashing scenario. (Frames 7 through 12.)

1 gets the chance to complete its write transaction. The response to the invalidation message, which is an UPDATE message, is transmitted in Frame 8. Frames 9 and 10 are symmetric to Frames 3 and 4, because *Processor B* receives the data corresponding to a data request while *Context 1* is still inactive. The vicious cycle is completed in Frames 11 and 12 when *Context 1* finally becomes active and finds that its data is not in the cache. *Context 1* then transmits a request that will eventually cause the data to be invalidated from *Processor B*.

As this scenario is repeated, X will ping-pong from one cache to another, and neither the context on *Processor A* nor the context on *Processor B* will ever actually access X. In this situation, all of the other contexts in the system will eventually have to wait for the contexts involved in the invalidation thrashing. From the user's point of view, the system is "hung," due to a problem with the cache coherence protocol. If invalidation thrashing is caused by contexts on different processors, then what does context switching have to do with this type of thrashing? The key events in the invalidation thrashing scenario are shown in Frames 3 and 9. When each context's data arrives at its cache, the context is not active. If there were only one context per processor, the timing could be arranged so that a process waiting for data would always be able to access data in its cache at least once before the data could be invalidated. Thrashing problems arise when it is not possible to ensure that a context can always access data before it is invalidated.

The Window of Vulnerability

The thrashing problems described in the previous section are caused by a *window of vulnerability* between the time that data arrives in the cache and the time that the processor accesses the data. During this period of time, it is possible for the data to be evicted from the cache, thereby preventing forward progress. On a processor without multiple contexts, the vulnerability period is typically only about one cycle long, so it does not provide a significant opportunity for thrashing scenarios to develop. However, the window is much wider for systems with multiple contexts per processor.

Experience with ASIM shows that while individual data accesses do not have a high probability of thrashing, programs that run for long periods of time generally encounter thrashing situations. In fact, all of the predicted thrashing scenarios have been observed during simulations of the Alewife system. Isolating and eliminating thrashing problems typically requires the cooperation of run-time system and memory system designers and demands intricate reasoning about the interactions between different features of the Alewife architecture.

The Thrash Wait Method

An economical solution to the forward progress problem has been devised. There are two components to this solution: First, the controller detects thrashing using an algorithm described below. Second, once thrashing is detected, the controller ensures forward progress by preventing the processor from switching contexts until the currently loaded context completes at least one successful memory access. Given the processor/controller interface specified in Table 3.4, it is easy to stop context switching by using the WAIT response type to stall the processor. Thus, this method solves the forward progress problem by temporarily eliminating the problem's source — context switching.

The algorithm that detects thrashing uses the cache block network wait states, one extra variable per context (called *tried once*), and one extra variable per controller (called *thrash wait*). The network wait states are the states specified in Table 3.5. Each context's tried once variable records whether or not the context has caused the controller to transmit a request for a data block. A controller's thrash wait variable indicates if some context is being held due to thrashing. When using the thrash wait method, the pseudocode[1] for the module that services processor to cache requests is as follows:

[1]The pseudocode notation is borrowed from [17].

```
Do_Processor_Request(Address, Context)
 1  if (data is ready for Address) ⇒ cache hit
 2    then clear tried_once[Context]
 3         clear thrash_wait
 4         return READY
 5  if (data is in a network wait state) ⇒ still waiting for transaction
 6    then if (thrash_wait is set)
 7           then return WAIT
 8           else return SWITCH
 9  if (tried_once[Context] is set) ⇒ detected thrashing!
10    then send RREQ or WREQ
11         set thrash_wait
12         return WAIT
13  ⇒ normal cache miss
14  send RREQ or WREQ
15  set tried_once[Context]
16  return SWITCH
```

The key to this algorithm is the conditional in line 9. If the data in the cache is not ready, the block is not in a network wait state, and the tried once variable is set for a context, then there are only two possibilities for the status of the data: 1) The requested data arrived at the cache, but was subsequently replaced by data for another context. That is, a thrashing situation exists. 2) The context's previous data request received a BUSY signal. Conversely, if a replacement or invalidation thrashing situation exists, or if a data request receives a BUSY signal, then the conditional in line 9 will be true, and the thrash wait variable will be set. Thus, the above algorithm will always detect a thrashing condition, when one exists. Furthermore, since simulations show that BUSY signals are rarely transmitted, it is reasonable to ignore the false thrashing detections that are caused by BUSY signals.

The thrash wait algorithm does not improve the performance of a system that suffers from excessive invalidation or replacement thrashing. The method merely ensures forward progress when thrashing exists. In a sense, it does not matter whether or not the algorithm is optimal in terms of memory latency: If thrashing has a dominant effect on a cache-based memory system, then the memory system is extremely slow, regardless of the method that it uses for ensuring forward progress.

As a final note, the thrash wait method for preventing thrashing assumes that it is always correct to prevent the processor from switching contexts. This assumption has curious ramifications when examined in conjunction with the LimitLESS coherence protocol. Since the LimitLESS protocol is actually an extension of the memory system, to avoid protocol deadlocks, it is sometimes necessary to interrupt a processor in *thrash wait* mode for the purpose of executing a LimitLESS protocol interrupt. In this situation, replacement thrashing could result if the LimitLESS interrupt executes in the same cache as the user code. To solve this problem, it is possible to mandate that LimitLESS interrupts run without using the cache; however, Alewife will probably be designed with a higher-performance solution to this problem.

Evaluation of Support for Multiple Contexts

The special features in the cache coherence protocol that are used to support multiple contexts are the results of about $1\frac{1}{2}$ years of simulation experience. Although the programs that currently run on ASIM all run to completion and therefore have no cyclic thrashing conditions, this experience is not a valid proof that the cache coherence protocol actually ensures forward progress. Before binding a protocol into hardware, it would be profitable to to prove that a proposed protocol obeys the properties of correctness and liveness.

Correctness refers to the fact that a protocol actually provides the shared memory model (*e.g.* sequential consistency) specified for the multiprocessor system. *Liveness* refers to the fact that the protocol will guarantee forward progress in memory accesses. Recent attempts to prove protocol correctness using the I/O Automata Model [2, 34] or home-grown models [7, 16, 43] show that the subject is a difficult one. This difficulty may indicate that correctness proofs for coherence protocols are truly complex in nature, or that a better abstraction is needed between memory model specifications and shared memory implementations.

A graph-based verification method has proven itself useful for detecting problems in finite state systems, including hardware controllers [15], sequential circuits [8], and cache coherence protocols [10]. While the verification method is not a formal proof technique, it does use an automatic verification algorithm to prove properties (stated in temporal logic) about finite state systems. Such an automatic verification tool may prove to be very useful for reasoning about the Alewife memory system, once the protocol specification has stabilized.

3.4 Second-Order Considerations

In addition to the protocol features that have a primary impact on the performance of a cache coherence scheme, there are a number of secondary implementation details that also contribute to the speed of the memory system. Examples of such details include several protocol messages that are not essential for ensuring a memory model, as well as the method used by a memory controller to count invalidation acknowledgment (ACKC) messages from caches. While these features may be interesting from the point of view of protocol design, they have only a small (but not insignificant) effect on the system as a whole.

3.4.1 Protocol Messages

The messages that are used by the hardware coherence protocols to keep the cache and the memory states consistent are listed in Table 3.3. The *Data?* column indicates the four messages that contain the data of the shared memory block. Table 3.6 lists three optional messages that are not essential to ensure cache coherence. Although the messages have mnemonic names, it is worth explaining the meaning of each message: The RREQ message is sent when a processor requests to read a block of data that is not contained in its cache. The RDATA message is the response to RREQ, and contains the data needed by the processor. The WREQ and WDATA messages are the request/response pair for processor write requests. Since more than one memory word is stored in a cache line, the WDATA message contains a copy of the data in the memory module.

The MREQ and MODG messages are used to service processor write requests when the cache contains a Read-Only copy of the data to be written. In this case,

Type	Symbol	Name	Data?
Cache to Memory	MREQ	Modify Request	
	REPU	Replace Unmodified	
Memory to Cache	MODG	Modify Granted	

Table 3.6: Optional protocol messages.

the processor does not need a copy of the data, so the MODG message does not contain the block of data. MREQ and MODG are really just an optimization of the WREQ and WDATA message combination for the limited directory protocol. This message pair is not essential to a protocol, because it is always *correct* to send a WDATA instead of a MODG message.

It is not obvious that the extra complications needed to implement the MODG message are justified by its performance benefits. The modify request and grant message pair optimizes for data locations that are read by a processor and then immediately written. This is especially important during cold-start periods when an application's working set does not reside in its cache. However, it is not possible to implement the MODG message in a chained protocol, because the memory module can not prevent a cache from receiving an invalidation message between the time that it sends a MREQ message and the time that it receives a MODG message. Full-map and limited protocols do not have this problem, because the directory is stored in the same node as the associated memory block. However, if the protocol needs to send an invalidation message to a cache before completing the write transaction, it is necessary for the directory to store a bit of state that indicates whether the initial request was a WREQ or a MREQ. Due to the complications caused by these messages, they are not included in the transition state diagrams, even though they are implemented in ASIM.

The INV and ACKC message combination is used to purge Read-Only copies of cached data. In the limited directory scheme, when a Read-Only memory block receives a WREQ message, the memory controller sends one INV message to each cache with a pointer in the directory. When a cache receives the INV message, it invalidates the appropriate cache line (if the cache tag matches the message's address), and responds with an ACKC. In the full-map and limited protocols, the controller may send one INV message on each cycle, so several INV messages with the same address may be working their way through the network at the same time. To keep track of the number of these parallel invalidations, the controller maintains an *acknowledgment counter*. The controller increments the counter when it transmits an INV message and decrements the counter when it receives an ACKC message. Thus, the counter remembers the total number of ACKC messages that it expects to receive. The counter waits for the acknowledgment counter to reach zero before responding to the initial WREQ message to ensure sequential consistency. To limit the amount of state that must be stored during a write transaction, the controller responds with a BUSY signal to any RREQ or WREQ messages to the memory block while invalidations are in progress for a memory block. If a controller accepts a RREQ or WREQ message, then the protocol guarantees to eventually satisfy the request. However, if a cache receives a BUSY signal, then it must retry the request.

Although the chained directory protocol does not perform invalidations due to a WREQ message in parallel, it also uses the acknowledgment counter. The controller increments the counter for each INV message and decrements it for every ACKC message. So, even if the linked-list has been fragmented by cache replacements (see below), the protocol can ensure sequential consistency by guaranteeing that no read only cached copies of a block exist when the block is written.

The acknowledgment counter is also used in the case of a limited directory eviction. When a directory entry does not have enough pointers to satisfy a RREQ message, it needs to replace one of the occupied pointers with a pointer to the requesting cache. Instead of locking the memory location while the eviction invalidation takes place, the protocol increments the acknowledgment counter. At first, this use of the acknowledgment counter may seem to be merely an optimization over locking the memory location. However, using the counter to keep track of evictions guarantees that a write transaction will never receive a BUSY signal due to a read transaction. This guarantee is necessary to ensure forward progress for locations that incur severe read-write traffic.

The INV and UPDATE messages are used to return modified data to the memory module. If a controller receives a RREQ message for a data block in the Read-Write state, the controller sends an INV message to the cache that currently has permission to write the data block. When this cache receives the INV message, it responds with an UPDATE message containing the modified data, rather than with an ACKC message, because the cached block is in the Read-Write or the Read-Write Network Wait state. At the same time, the cache invalidates the line that contains the data.

Since multiple addresses map to each cache line, a cache sometimes needs to *replace* one cached block of data with another. If a replaced block is in the Read-Write state, then the REPM message is used to send the *m*odified data back to memory. Otherwise, the data is *u*nmodified, and the REPU message is used to notify the directory about the replacement. In the case of the chained protocol, the REPU message contains the directory pointer that was associated with the replaced data, so that the directory can adjust the chain in one of three possible ways:

1. If the replaced copy of data was the first block in the chain, then the pointer in the REPU message points to the new beginning of the chain.

2. If the replaced copy of data was the last block in the chain, then no invalidation messages need to be sent through the network.

3. If the replaced copy of data was in the middle of the chain, then the directory sends an invalidation to the tail of the chain that began with the replaced data.

This procedure correctly ensures sequential consistency, due to two features of the chained protocol. First, the protocol correctly handles dangling chain pointers that are created by replacement in the singly-linked chain. If the address contained in an INV message does not match the tag in the cache, then the invalidation is acknowledged without invalidating the currently cached data. Second, the directory can monitor the number of breaks in the chain by incrementing the acknowledgment counter for every INV message that is sent and by decrementing the counter for every ACKC message that is received. As in the limited directory implementation, waiting for the acknowledgment counter to be decremented to zero before satisfying any write requests maintains sequential consistency.

Although the REPU message is central to enforcing cache coherence in the chained scheme, it is optional in the limited and full-map protocols. If a cache replaces a Read-Only copy of data but does not notify the directory, then it may receive a spurious INV message for the block at some point in the future. However, (as in the chained protocol) the address in the INV will not match the tag in the cache, so the spurious invalidation is acknowledged without invalidating the currently cached data. On the other side of the memory system, if a directory receives a RREQ message from a cache that already has a pointer, then it responds with a RDATA message. So, the REPU message may save an INV message, or it may create unnecessary network traffic. In order to examine the effects of the REPU message, ASIM has been instrumented with an option that determines whether or not the current coherence protocol uses the message.

3.4.2 Counting Acknowledgments

The current ASIM protocols implement one acknowledgment counter for each block in memory. This implementation allows one write transaction to be in progress for each memory block per controller. However, the independence of transactions on different memory blocks comes at the cost of the additional memory space (equal to one pointer per directory entry). It is also possible to implement one acknowledgment counter per controller. Because a controller-level counter keeps track of the sum of the outstanding acknowledgments for every block stored in the controller's memory, such an acknowledgment counter has to be substantially larger than each of the independent memory counters. Nevertheless, implementing the acknowledgment counter in the controller saves the memory otherwise taken by acknowledgment counters in each directory entry.

The savings in the directory size is offset by losing the independence of transactions on different memory locations. That is, during a write transaction, it is necessary to BUSY both read and write transactions to any address serviced by the controller for the following reasons: When a write transaction requires invalidations to be completed, there may already be other invalidations (due to evictions or chain fragmentation) en route in the network. So in order to complete the write transaction when the counter is finally decremented to zero, the controller must remember the identifier of the cache that initiated the transaction. To limit the amount of transaction state that must be stored in the controller, *it is necessary to limit the number of simultaneous write transactions to only one per controller.*

Furthermore, to ensure the eventual completion of a write transaction, it is necessary to guarantee that the acknowledgment counter will eventually be decremented to zero. Allowing read transactions to proceed during a suspended write transaction may violate this guarantee, due to the possibility of an eviction cycle in a limited directory protocol or a replacement cycle in a chained directory protocol. Thus, a controller may not service read transactions while a write transaction is outstanding. The trade-off between directory size and simultaneous transactions per controller is evaluated by implementing both counter schemes in ASIM.

3.4.3 Evaluation of Secondary Protocol Features

None of the protocol features discussed in this section exhibit more than a ten percent variation in execution time on ASIM. This behavior is expected, because the unessential components of protocols tend to interact with relatively infrequent events, such as cache line replacement or cold-start data accesses. Such low performance returns suggest that issues of complexity and cost can be used to decide whether or not to implement unessential protocol messages. Certain protocol messages may be rejected out-of-hand. For example, the replace unmodified (REPU) message sometimes degrades performance due to an increase in network traffic. Thus, it is not worth the extra complexity necessary to implement this message.

On the other hand, the modify request/grant (MREQ/MODG) message pair can increase performance by over five percent. While this performance gain does not justify the extra directory state needed to store the modify request during invalidations, it does imply that a simplified version of the feature would be appropriate. For example, a memory controller could respond to a MREQ with a MODG only when no invalidations need to be sent. This simplification would eliminate most of the extra cost of the modify grant optimization, while retaining the benefits of reduced latency for simple memory transactions that consist of a read request followed by a write request.

The acknowledgment counter implementation is driven by directory implementation considerations. Due to the current implementation of Alewife's directory structure, there will be enough space in each directory entry to implement one counter per memory block. However, this decision is more an artifact of the design process than a result of carefully considering simulation data. By analyzing simulations of many applications, it might be possible to reach a more scientific conclusion about the trade-off between implementing an acknowledgment counter versus an extra directory pointer. However, the marginal differences between the two possible designs do not justify an extensive investigation.

Chapter 4

Evaluation Methodology

The methodology for evaluating cache coherence protocols centers around two related means of analysis. A decoupled simulation technique, which incorporates trace-driven simulations and analytical modeling, permits the study of a wide range of applications, protocols, and target architectures, but suffers from inaccuracy due to the lack of feedback between system components. Coupled simulation techniques offer more accurate analysis methods, but require more time to implement and to take measurements. The trade-off between these two different simulation techniques suggests the following sequence of analysis: First, take measurements using decoupled simulation to establish the gross merits and problems of each of the protocols. Then, validate and augment the first round of decoupled simulations with a second round of coupled simulations.

4.1 Overview

Figure 4-1 shows the entire set of simulation systems used to analyze the protocols described in this thesis. The raw input to the evaluation consists of multiprocessor programs, written in the C, FORTRAN, and Mul-T programming languages. The figure depicts each step in the simulation process, from the raw application code to the statistics that are used to evaluate the multiprocessor performance.

The left side of Figure 4-1 illustrates the decoupled simulation methodology. De-

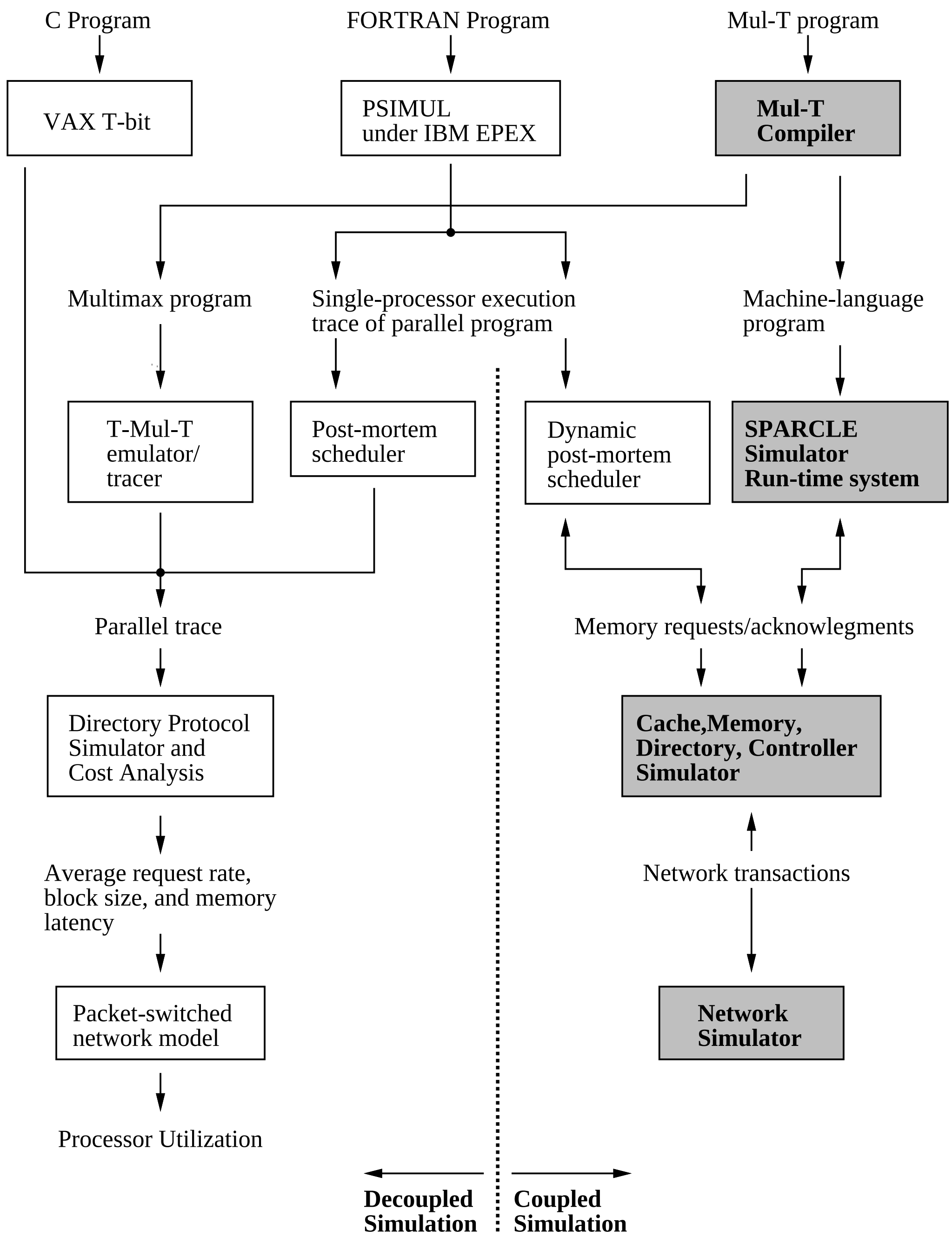


Figure 4-1: Simulation environments used to evaluate the performance of cache coherence protocols. The shaded rectangles represent ASIM.

coupled simulation allows information to flow in only one direction, from the original program to the statistical output. Multiprocessor address traces generated using three tracing methods from IBM, Stanford, and MIT, are run on a memory system simulator that counts the occurrences of different types of protocol transactions. Each of these transaction types is assigned a cost in order to produce the average processor request rate, the average network message block size, and the average memory latency per transaction. From these parameters, a packet-switched, pipelined, multistage interconnection network model calculates the average processor utilization, which measures the contribution of the memory system to the time needed to run a program on the system.

While it is possible to avoid the network model calculations by using a purely trace-driven decoupled simulation technique, such a methodology does not result in a true representation of multiprocessor performance. Consider a system that attaches a network simulator to the back-end of a trace-driven memory system simulator. Without the feedback between the network and the trace generation system, varying memory access delays cause a skew between the sense of time as determined by the execution of each processor's thread of control. In such a system, each simulated component operates without synchronizing correctly with any other component. Thus, a naive network simulator coupled with a trace-driven directory simulator will produce an incorrect multiprocessor execution.

Furthermore, the skew in a purely trace-driven system causes problems with running a simulation to completion. For some applications, the difference between the execution of different processors can grow to more than 10% of the entire duration of a trace. Not only does this skew cause the results of the simulation to be suspect, but it also generates huge queues within an event-driven network simulator. Such queues quickly thrash the virtual memory system of the machine running the trace-driven simulations. Thus, the hybrid decoupled simulation technique must be used to avoid the problems with a purely trace-driven methodology.

The coupled simulation technique, shown on the right side of Figure 4-1, models a shared-memory multiprocessor even more accurately. Except for during the program compilation stage, this process allows bidirectional interfaces between all of the components of the simulation engine. The shaded boxes represent the modules that comprise ASIM, the Alewife system simulator; namely, the SPARCLE processor and run-time system, the cache/memory controller, and the processor interconnection network. The interfaces between these modules approximate the actual communication boundaries between the hardware components of the Alewife machine. The memory-system and network modules of ASIM also accept input from a dynamic version of a trace-generation method, called post-mortem scheduling. This method uses a trace with statically encoded synchronization information to simulate the interaction between a processor and its memory system.

4.2 Decoupled Simulation

A hybrid of trace-driven simulation and analytical methods helps evaluate the performance of cache coherence schemes for a variety of multiprocessor applications. There are three primary phases of the decoupled simulation method: First, instrumented multiprocessing systems or simulators generate traces of parallel programs. Second, a program that simulates a multiprocessor's memory system processes the parallel traces. Third, a model of an interconnection network refines raw statistics from the memory-system simulation.

Recall from Section 2.1.1 that processor utilization (and therefore system speed-up) is impacted by the frequency of memory references and the latency of the memory system. If the probability of a memory request during any cycle is m, and the average latency of the round-trip through the memory-network system is T, the processor utilization U is given by:

$$U = \frac{1}{1 + mT}$$

The latency of a round-trip through the network depends on several factors, including the network topology and speed, the number of processors in the system, the frequency and size of the messages, and the memory latency. The cache coherence protocol determines the request rate, message size, and memory latency. Detailed models of

Source	Language	Processors	Application	Length
VAX T-bit	C	16	P-Thor	7.09
			MP3D	7.38
			LocusRoute	7.05
			SA-TSP	7.11
Post-mortem Scheduler	FORTRAN	64	FFT	7.44
			Weather	31.76
			Simple	27.03
T-Mul-T	Mul-T	64	Speech	11.77

Table 4.1: Summary of Trace Statistics: Length values are in millions of references to memory.

cache coherence protocols and interconnection networks must be used to calculate processor utilization from these parameters. Note that the network model that is used in conjunction with the decoupled evaluation does not account for the context-switching capability of the SPARCLE processor. The hybrid decoupled methodology is used to differentiate between coherence protocols, without complicating the issue with mechanisms that are used to tolerate memory latency, such as multiple contexts or weak ordering.

4.2.1 Getting Multiprocessor Address Trace Data

The address traces represent a wide range of parallel algorithms written in three different programming languages. The programs traced at Stanford were written in 'C', those from IBM were written in FORTRAN, and those produced at MIT were written in Mul-T (a variant of Multilisp). The implementation for the trace collector is different for each of these programming systems; each tracing system can theoretically obtain address traces for an arbitrary number of processors, enabling a study of the behavior of cache-coherent machines much larger than any built to date. Table 4.1 summarizes general characteristics of the traces.

The SA-TSP, MP3D, P-Thor, and LocusRoute traces were gathered by using the Trap-Bit method, configured with 16 processors. SA-TSP uses simulated annealing to solve the traveling salesman problem. MP3D is a 3-D particle simulator for rarified flow. P-Thor is a parallel logic simulator, and LocusRoute is a global router for VLSI standard cells. A detailed description of the applications can be found in [44].

Trap-bit (T-bit) tracing for multiprocessors is an extension of single processor trap-bit tracing. In the single processor implementation, the processor traps after each instruction if the trap bit is set, allowing the interpretation of the trapped instruction and emission of the corresponding memory addresses. Multiprocessor T-bit tracing extends this method by scheduling a new process on every trapped instruction. Once a process undergoes a trap, the trace mechanism performs several tasks: it records the corresponding memory addresses, saves the processor state of the trapped process, and schedules another process from its list of processes, typically in a round-robin fashion.

The Weather, Simple, and FFT traces were generated with the post-mortem scheduling method, developed at IBM [13]. The Weather application partitions the atmosphere around the globe into a three dimensional grid and uses finite-difference methods to solve a set of partial differential equations describing the state of the system. Simple models the behavior of fluids and employs finite difference methods to solve equations describing hydrodynamic behavior. FFT is a radix-2 Fast Fourier Transform.

Post-mortem scheduling is a technique that generates a parallel trace from a uniprocessor execution trace of a parallel application. The uniprocessor trace is a task trace with embedded synchronization information that can be scheduled, after execution (*post-mortem*), into a parallel trace that obeys the synchronization constraints. This type of trace generation uses only one processor to produce the trace and to perform the post-mortem scheduling. So, the number of processes is limited only by the application's synchronization constraints and by the number of parallel tasks in the single processor trace.

The Speech trace was generated by a compiler-aided tracing scheme. The application comprises the lexical decoding stage of a phonetically-based spoken language understanding system developed by the MIT Spoken Language Systems Group. The Speech application uses a dictionary of about 300 words represented by a 3500 node directed graph. The input to the lexical decoder is another directed graph representing possible sequences of phonemes in the utterance to be recognized. The application uses a modified Viterbi search algorithm to find the best match between paths through the two graphs.

In a compiler-based tracing scheme, code inserted into the instruction stream of a program at compile time records the addresses of memory references as a side-effect of normal execution. The compiler-aided scheme used to trace the Speech application is called T-Mul-T. T-Mul-T is a modification of the Mul-T programming environment that can be used to generate memory address traces for programs running on an arbitrary number of processors. Instructions are not currently traced in T-Mul-T, so it is necessary to assume that all instructions hit in the cache, and for the purpose of processor utilization computation, an instruction reference is associated with each data reference. This assumption is made only for the Speech application, because the other traces include instructions.

The trace gathering techniques also differ in their treatment of private data locations, which must be identified for the scheme that only caches private data. The private references are identified statically (at compile time) in the FORTRAN traces and are identified dynamically by post-processing the other traces. Since static methods must be more conservative than dynamic methods when partitioning private and shared data, the performance that decoupled simulations predict for the private data caching scheme on the C and Mul-T applications is slightly optimistic. In practice, the implementation of schemes that cache only private data is made difficult by the non-trivial problem of static data partitioning.

The address traces from each of these generation techniques are refined into a canonical format. Each entry in the trace specifies the processor number, the type of processor memory access, and the associated physical address.

4.2.2 Simulating a Cache Coherence Strategy

For each address reference in a trace, the directory simulator determines the effects on the state of the corresponding block in the cache and the directory. This state consists of the cache tags and directory pointers that are used to maintain cache coherence. In the simulation, there is no feedback from the network to the cache or memory modules; all side effects from each memory transaction (entry in the trace) are assumed to be stored simultaneously. While this simulation strategy does not accurately model the state of the memory system on a transaction-by-transaction basis, it does produce accurate counts of each type of protocol transaction over the length of a trace. Such a simulation also corresponds to a correct multiprocessor execution of the parallel program, because the order of the memory accesses in the traces is maintained throughout the simulation.

A memory transaction consists of a processor-to-memory reference and its effect on the state of the memory system. Any transaction that causes a message to be sent out over the network contributes to the three parameters that determine the contention in the memory system: average request rate, average message size, and average memory latency. Table 4.2 lists all of these types of transactions and their contributions to the effective latency suffered by the processor. The *Msgs* column gives the total number of messages needed to process the transaction. Invalidation and acknowledgment messages do not incur the full memory delay, so the *Mem. Lat.* column indicates the effective total memory latency for each transaction. The *Words* column gives the sum of the number of 32-bit words transmitted for all of the transaction messages (assuming 16-byte cache blocks).

Given a trace and a particular cache coherence protocol, the directory simulator determines the percentage of each transaction type in the trace. This percentage, multiplied by the cost of the transaction, gives the contribution of the transaction to each of the three parameters: average request rate is derived from the number of messages over the length of the trace; average memory latency is derived from the memory delay per transaction; and average message size is determined from the total number of words transmitted over the number of messages transmitted. Another set of transaction types may be used to determine the performance of schemes that do not cache shared variables. The transactions and costs used for schemes that only cache private data are also listed in Table 4.2.

In addition to the cache coherence strategy, there are other parameters that affect

Coherence Protocol	Transaction Type	Msgs	Mem. Lat.	Words
Directory	instruction miss	2	6	6
	read miss, block dirty in another cache	4	12	12
	read miss, block clean in another cache	2	6	6
	read miss, block not in any cache	2	6	6
	write miss, block dirty in another cache	4	12	12
	write miss, block clean in another cache	4	7	8
	write miss, block not in any other cache	2	6	6
	write hit, block clean in another cache	4	7	4
	replaces to dirty shared objects	1	3	5.5
	invalidations due to too few pointers	2	1	2
	invalidations caused by a write	2	1	2
Only Cache Private Data (OCPD)	instruction misses	2	6	6
	shared references	2	6	3
	private read misses	2	6	6
	private write misses	2	6	6

Table 4.2: Transaction Types and Costs.

the performance of the memory system. These parameters are listed in Table 4.3 along with their default values. Since the decoupled analysis was performed in the initial stages of development of the Alewife system, the method uses a model of an Omega network, which is described in Section 4.2.3. The Alewife system actually connects its processing nodes through a network with a mesh topology; however, the difference between Omega and mesh networks does not change the conclusions from the decoupled simulation technique.

4.2.3 The Interconnection Network Model

The cache coherence schemes that are considered for Alewife transmit messages over an interconnection network to maintain cache coherence. In order to analyze such a message-based memory system, the decoupled simulation technique uses a packet-switched, buffered, multistage interconnection network that belongs to the general class of Omega networks. The network switches are pipelined so a message header can leave a switch even while the rest of the message is still being serviced. A protocol message travels through n network switch stages to the destination node and takes M cycles for the memory access. The network is buffered and guarantees sequenced

Type of Parameter	Name	Default Value
Cache/Directory	cache size	256 Kbytes
	cache block size	16 bytes
	cache associativity	direct mapped
	cache update policy	write back
	directory pointer replace policy	random
Network	topology	Omega
	network message header size	16 bits
	network switch size	4×4
	network channel width	16 bits
	processor cycle time	$2 \times$ network switch cycle time
	memory address size	32 bits
	base memory access time	$6 \times$ network switch cycle time

Table 4.3: Simulation parameter defaults for the cache, directory, and network.

delivery of messages between any two nodes on the network.

Computation of the processor utilization is based on the analysis method used by Patel [38]. The network model yields the average latency T of a protocol message through the network with n stages, $k \times k$ size switches, and average memory delay M. The processor utilization U is derived from a set of three equations:

$$\begin{aligned} U &= \frac{1}{1+mT} \\ \rho &= UmB \\ T &= n + B + M - 1 + \left(\frac{\rho B \left(1 - \frac{1}{k}\right)}{2(1-\rho)} \right) n \end{aligned}$$

where m is the probability a message is generated on a given processor cycle, with corresponding network latency T. The channel utilization (ρ) is the product of the effective network request rate (Um) and the average message size B. The latency equation uses the packet-switched network model by Kruskal and Snir [30]. The first term in the equation ($n+B+M-1$) gives the latency through an unloaded network, and the second term gives the increase in latency due to network contention, which is the product of the contention delay through one switch and the number of stages. The above equations are solved to get a closed form solution for U:

$$U = \frac{1}{1 + \frac{m}{2}\left[n + M + 2B - 1 - \frac{1}{m} + \sqrt{(n + M - 1 + \frac{1}{m})^2 + 2nB^2\left(1 - \frac{1}{k}\right)}\right]}$$

Table 4.3 shows the default network parameters used in the decoupled analysis. While this evaluation technique was used to derive results for packet-switched multistage network, it is possible to derive results for other types of networks by varying the network model used in the final stage of the analysis. The ability to use the results from one set of directory simulations to derive statistics for a range of network or bus types displays the power of this modeling method.

4.2.4 A Sample Computation of Processor Utilization

The following example illustrates the process of deriving the processor utilization for a specific application and cache-coherence scheme. Several steps are used to determine the processor utilization for the Weather application, a full-map directory scheme, and the default simulation parameters:

1. Run the post-mortem scheduler on the annotated uniprocessor Weather application trace. The output of the scheduler is a trace file that is approximately 160Mbytes long, but can be compressed to about 75Mbytes.

2. Run the directory simulator on the scheduled Weather trace with cache size 256K bytes, cache block size 16 bytes, 64 processors, and a full-map directory protocol.

3. Count the number of each transaction type that is listed in Table 4.2. For example, the number of instruction misses was 22817.

4. Multiply the number of each transaction type by the costs given in Table 4.2: The 22817 instruction misses generated 45634 network messages that consisted of 136,902 bytes of data. These network messages incurred 136,902 cycles of memory latency.

5. Sum the costs for all of the transactions in the trace, and average over the total number of memory references in the trace. The average request rate into the network for the trace was 0.097 messages per processor cycle. The average message size was 2.74 32-bit words, and the average latency at memory was

2.95 cycles. Halve the request rate to 0.0485 to simulate a network cycle that is twice as fast as the processor cycle and double the message size to 5.48 to simulate a network with 16-bit data paths.

6. Substitute the values calculated in the previous step into the network model equations given in Section 4.2.3. The processor utilization for $n = 3$, $k = 4$, $m = 0.0485$, $B = 5.48$, and $M = 2.95$ is $U = 0.63$.

4.2.5 Sources of Error in Decoupled Simulations

Although the decoupled simulation technique allows the evaluations of a range of multiprocessor applications and cache coherence schemes, it suffers from the assumption that all state changes caused by a memory transaction are stored simultaneously. In combination with a network model, a decoupled evaluation methodology gives a good approximation of a multiprocessor's performance, averaged over the entire execution of an application. However, a model of average performance — as opposed to a cycle-by-cycle simulation — neglects both the latency sequential protocol operations and contention due to hot-spot contention.

For example, the linked-list representation of a chained directory entry causes such a protocol to invalidate cached copies in sequence, while limited and full-map directories can execute invalidations in parallel. The network model does not account for the difference between sequential and parallel invalidation latencies, because the traffic from all protocol messages is averaged over the entire duration of a trace. Thus, while the decoupled evaluation accounts for the bandwidth required by invalidations, it does not properly model the latency caused by them.

Furthermore, the decoupled method does not properly model hot-spot access to a memory module. Not only does the network model average network traffic over the entire duration of a program, but it also averages the traffic over all of the nodes in the multiprocessor. A hot-spot occurs when many processors simultaneously access data in one memory module. Such a situation generally causes long latencies, due to the contention in the interconnection and network and competition for the limited processing power of the single memory module. In Section 4.4.2, the Weather application is used to illustrate a hot-spot that is ignored by the decoupled analysis method.

4.3 Coupled Simulation

Coupled simulations solve the problems inherent in the decoupled methodology by modeling systems with feedback between all of the components in a multiprocessor. A dynamic version of the post-mortem scheduler directly addresses the feedback problem by coupling the trace generation module with the memory system simulator [32]. Complete system simulations such as ASIM go one step further: By modeling an entire multiprocessor, it is possible to investigate a wide range of topics, including all of the facets involved in programming and designing a parallel system. However, building, running, and administering a complete system simulation requires a much higher time investment than decoupled techniques.

4.3.1 Alewife System Simulator

ASIM, the Alewife System Simulator, is used to evaluate methods for designing the hardware and software components of a large-scale multiprocessor. Figure 4-1 shows the interfaces between the modules of ASIM. The simulator includes the Mul-T compiler, the Alewife run-time system, the SPARCLE simulator, the cache/memory controller simulator, and the network simulator. Viewed as a whole, ASIM allows a program to be compiled, linked, and run on an implementation of the Alewife architecture. Table 4.4 shows the specifications of the system modeled for studying coherence protocols.

When the system is configured with 64 processors and full statistics-gathering capability, it runs about 300,000 times slower than a hardware implementation would run. Nevertheless, since ASIM can generate measurements for a range of different hardware configurations, it is a powerful tool for analyzing trade-offs in the design of Alewife.

Type of Parameter	Name	Default Value
Processor	cycle time	2 $\times$ network cycle time
Cache/Directory	cache size	64 Kbytes
	cache block size	16 bytes
	cache associativity	direct mapped
	cache update policy	write back
	directory pointer replace policy	random
	memory address size	32 bits
	base memory access time	6 $\times$ network cycle time
	directory access time	6 $\times$ network cycle time
Network	topology	Mesh
	network channel width	16 bits
	network message header size	32 bits

Table 4.4: Simulation parameter defaults for ASIM.

The ASIM Software

The Mul-T compiler is based on ORBIT [29], an optimizing compiler for a dialect of Lisp. Mul-T [28], a variant of Multilisp, uses the `future` construct to allow a programmer to explicitly designate tasks that can be executed in parallel. The compiler generates machine-language code that is compatible with Alewife's SPARCLE processor.

Before running an application, ASIM links the program's object code with a run-time system. Two run-time environments are currently available. One run-time system assigns tasks to processors, based on explicit instructions from the programmer. This environment requires a statically partitioned and scheduled program. The other run-time system dynamically partitions a program using a method called lazy task creation [35]. This partitioning method attempts to balance the number of parallel tasks created with the available processing resources in a machine. A dynamic scheduler performs the functions necessary to create tasks and to distribute them to the system's processors.

Processor Simulator

The processor simulator models the behavior of the SPARCLE processor at the register-transfer level. As the simulator runs the object code of a program and the run-time system, it gathers statistics that help evaluate both the software system and the architectural features of the SPARCLE processor. These statistics include the distribution of cycles between the boot sequence, the user code, and the scheduler's functions. The processor simulator also generates histograms that measure the effects of synchronization and context switching on system performance.

The SPARCLE module of ASIM runs the code generated by the Mul-T compiler and transmits data requests to the memory system using the processor-controller interface described in Section 3.3.1. Although the processor simulator maintains its own model of shared memory, it must receive permission from the cache/memory controller before completing a read or write request. It is the responsibility of the memory system simulator to ensure that the processor adheres to a valid shared-memory model.

Cache/Memory Simulator

The cache/memory simulator provides the base for experimenting with the implementation of cache coherence schemes. In particular, the mechanisms that support Alewife's context-switching processor were developed in this module of ASIM. The cache/memory simulator implements a range of cache coherence methods, including directory-based schemes, a scheme that only caches private data, and a software-controlled caching scheme. (Appendix B specifies the cache coherence protocols that are implemented in the simulator.) In order to simulate a "real" implementation of the Alewife architecture, the cache/memory module attempts to model a cache controller that could actually be implemented as a VLSI system. The model of the controller includes the interfaces to the processor and to the network, internal state machines, and network queues. In addition to modeling the cache controller, the cache/memory simulator also maintains the coherence state for all of the cache and memory blocks referenced by a program.

To help understand the relative performance of different coherence schemes, the cache/memory simulator gathers statistics that track the performance of each component of the memory system. There are three basic types of statistical output:

event histograms, locality arrays, and state summaries. The event histograms are graphs that display the number of times that certain events occur in the memory system. Locality arrays summarize the types of data access patterns between processors and memory controllers. The state summaries include basic statistics such as cache hit/miss ratios.

Network Simulator

The network simulator models the processor interconnect, which transports all of the cache coherence protocol messages. This module of ASIM is capable of simulating both packet-switched and circuit-switched networks in several different topologies, including mesh and Omega configurations. In fact, the network simulator was used to verify the network model used in conjunction with the decoupled simulation methodology. For the purposes of studying the Alewife architecture, the network simulator is set to model a packet-switched, two-dimensional mesh, with no end-around connections. The statistics available from the network module include average channel utilizations, switch load profiles, and a message delivery latency histogram.

4.3.2 Dynamic Post-Mortem Scheduling

As illustrated in Figure 4-1, the dynamic post-mortem scheduler receives the same input as the static scheduler. While the original version of the scheduler produces a static trace of processor requests to memory, the dynamic version of the scheduler allows a new data access only after the previous access has finished. By using the synchronization information encoded in the input, the dynamic post-mortem scheduler can correctly simulate the execution of a parallel program [32].

The dynamic post-mortem scheduler used in this study is compatible with the processor-controller interface defined in ASIM. This compatibility provides the opportunity to test the performance of the Alewife memory system on large applications such as Weather and Simple. All of the memory system statistics that are generated by the standard version of ASIM are also generated by the simulator when configured with the scheduler. In addition, the scheduler records information such as the average latency of access for different types of variables.

4.3.3 Sources of Error in Coupled Simulations

Since ASIM is capable of simulating the entire Alewife machine, different coherence schemes can be compared in terms of absolute execution time. While processor utilization (the metric derived from then decoupled methodology) gives an approximation of the average performance of a parallel system, simulated execution time gives an exact measure of the speed of a multiprocessor. The execution time metric emphasizes the bottom line of high-performance system design; namely, the speed of computation.

However, a large-scale multiprocessor is, by nature, a complicated beast. When a key feature of the memory system (such as the cache coherence protocol) is modified, the effects cascade through the multiprocessor as a whole. Thus, when using a coupled simulation technique to compare the relative performance of different coherence schemes, it is necessary to estimate the size of the effects due to artifacts of the programming environment, versus the effects due the memory system. For example, while it is easier to program with ASIM's dynamic run-time environment than with the static one, the non-determinism inherent in the dynamic scheme creates substantial differences between the behavior of a program running with different coherence protocols. Due to experimental problems in the current run-time system, numerical results are reported only for the dynamic post-mortem scheduler system. Although the simulations of the complete Alewife system are not used to generate quantitative data, initial experience with the SPARCLE processor and run-time environment confirms the qualitative conclusions about coherence schemes that are discussed in Chapter 5.

The analysis in this thesis also suffers from the benchmark problem. A benchmark is a program that is used to evaluate the performance of a computer system, because it exhibits either an average or a representative processing load. While standard suites of benchmarks have been developed for single-processor machines, no set of programs has been devised to test the performance of multiprocessors. The lack of benchmarks, and the more general dearth of multiprocessor programs, testifies to the youth of multiprocessor research. All of the evaluation in this thesis is conducted on programs that have both substantial size and interesting data access patterns; however, there is no guarantee that these applications exhibit average or representative loads on a multiprocessor memory system.

The slow speeds of simulation compound the benchmark problem. The fact that ASIM runs approximately 300,000 times slower than a hardware implementation of Alewife forces a trade-off between application size and simulated system size. Programs with enough parallelism to execute well on a large machine take an inordinate amount of time to simulate. This trade-off was resolved by simulating a 64-processor machine, which is large enough to require an interconnection network other than a bus, and small enough to simulate efficiently on the simulation engines available for use. It is important to note that this decision was made for practical, rather than theoretical, reasons.

Finally, the Alewife architecture has not been static during the evaluation of cache coherence schemes. It is not necessarily true that results derived for one point in the design space apply to other points. Thus, the numerical results presented in the next chapter should not be interpreted as definitive predictions of the future performance of the Alewife system. However, the major qualitative conclusions regarding the interaction between a multiprocessor's software and its memory system should hold as long as interprocessor communication latency remains an least an order of magnitude more expensive than processor cycle time.

4.4 Validating Decoupled Simulation

The coupled evaluation technique confirms the validity of the estimations of processor utilization by the decoupled methodology. In general, the results of complete system simulations verify the conclusions that may be drawn from the decoupled simulations. However, there are differences between the two methodologies in terms of absolute performance measurements that must be justified before trusting the results of the

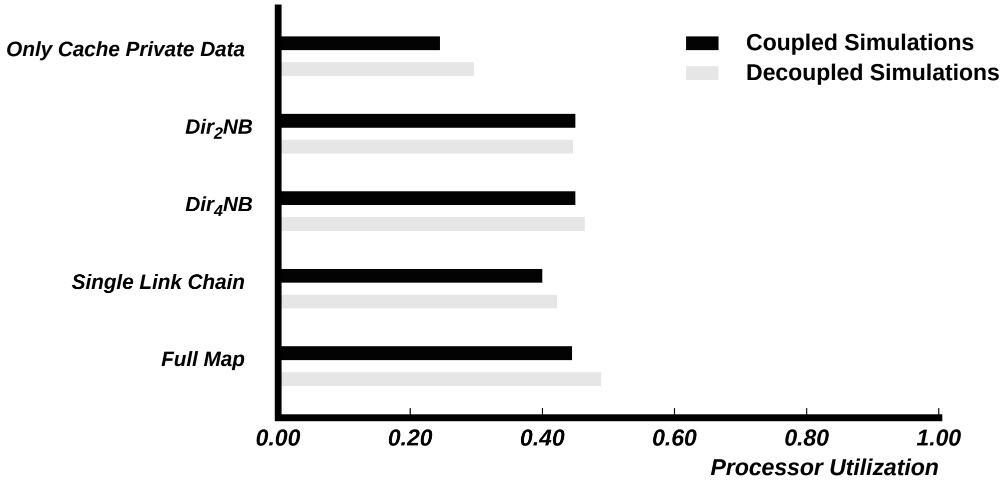


Figure 4-2: Comparison of processor utilization measurements for the Weather application, obtained from coupled and decoupled evaluation methodologies.

decoupled simulation technique.

Figure 4-2 shows the processor utilization results for the Weather application that are derived from both the coupled and the decoupled simulation techniques. The measurements from the different evaluation methodologies agree, due to two modifications that annul the differences between the simulation techniques: First, the fundamental system parameters are adjusted to be the same for both the decoupled and the coupled simulations. Section 4.4.1 discusses the parameter adjustment that is needed to reconcile the coupled and decoupled simulation results. Second, a variable in the Weather application that causes hot-spot access is optimized. Section 4.4.2 examines the effects of this variable.

In addition to validating the hybrid decoupled methodology, coupled simulations help evaluate how well processor utilization measures the performance of a multiprocessor. Although the metric usually provides good intuition about the behavior of some coherence schemes, it does not always accurately predict the actual behavior of a system. Section 4.4.3 investigates some of the discrepancies between the predicted and actual performance of the coherence scheme that only caches private data.

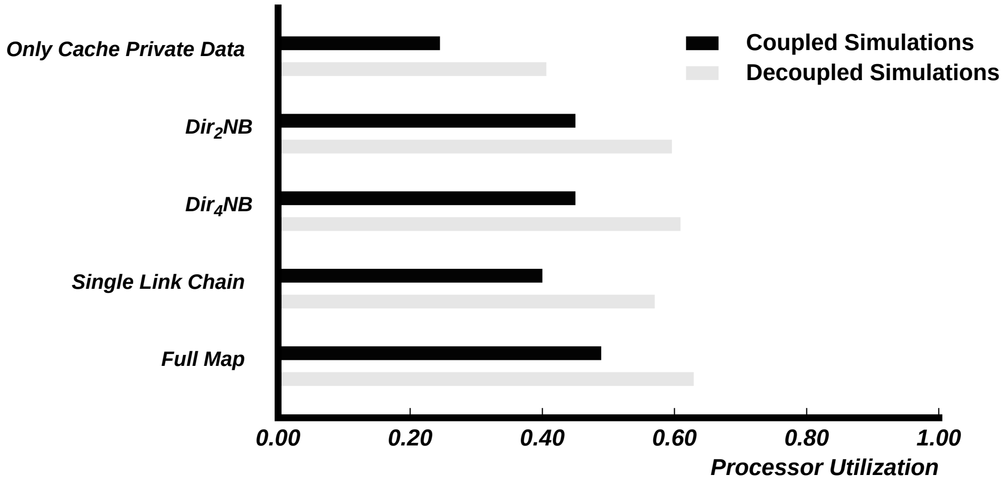


Figure 4-3: Comparison of processor utilization measurements for Weather, before adjusting the base memory access latency.

4.4.1 Parameter Adjustment

The coupled simulation technique models features of Alewife's cache/memory controller, including finite state machines, network buffers, and internal contention for resources. Since the decoupled methodology does not perform such a detailed simulation, Alewife's cache/memory controller runs slower in the coupled simulations than it does in the decoupled technique's network model. Figure 4-3 shows that the results from the two simulation strategies do not correlate, even when the coupled simulations are configured to match the parameters of the decoupled simulations, due to differences in the way that the controller is modeled.

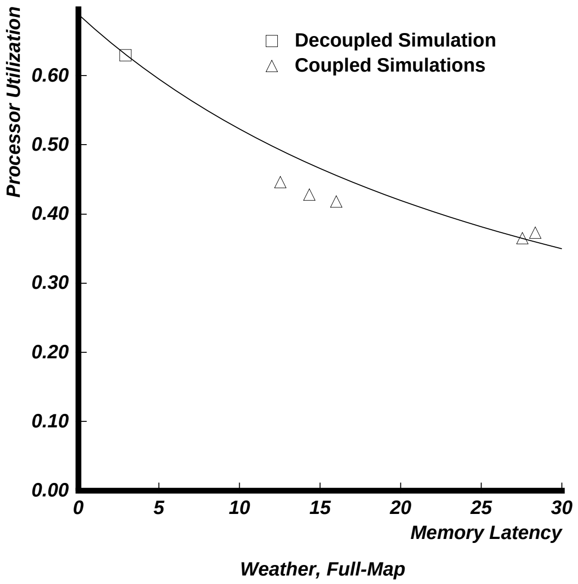
In order to reconcile the two evaluation techniques, the base memory access time (listed in Table 4.3) may be adjusted. By showing the relationship between base memory access time and processor utilization, Figure 4-4 extends the predictions of the network model into the range of memory latency observed for the Weather application with full-map and limited directory protocols. The curve on each of the graphs in the figure shows the prediction of the network model for a range of memory latencies, given the average request rate and the average block size calculated from the decoupled simulations. The square on each graph shows the prediction of the model for the memory latency assumed in the decoupled technique. Since this point is *calculated* from the network model, it sits on the prediction curve. The triangles label the *observed* processor utilizations and average memory latencies in coupled simulations of the Weather application.

The various memory latencies plotted for the coupled simulations do not correspond exactly to the memory access time parameter that is used by the network model. In the coupled simulations, different latencies can be created by changing parameters such as the time needed to read or write a directory entry. The reported latency values are calculated by subtracting twice the average network latency from the average total access latency of remote memory transactions. Thus, the reported memory latency values include all of the delay needed to service a transaction (including invalidations), except for the time needed to transport protocol messages through the network.

Comparing the predictions of the decoupled simulations to the processor utilizations measured in coupled simulations reconciles the differences between the results from two evaluation methodologies. When the base memory access time used in the decoupled technique is adjusted to correspond to the memory latency observed by coupled simulation, the different analysis methods yield comparable processor utilization measurements. The dependence of processor utilization on memory latency emphasizes an important (but perhaps obvious) conclusion: No matter what coherence scheme is used to implement shared-memory, increasing the efficiency of a memory system's design improves the performance of the system as a whole.

4.4.2 The Effect of Hot-Spot Contention

Although adjusting the base memory access time corrects for the absolute difference between the predicted and observed processor utilizations, Figure 4-5 shows that the adjustment does not completely reconcile the results of the coupled and decoupled simulation techniques. Specifically, the decoupled simulations predict that the limited directories perform almost as well as the full-map directory, but the coupled simulations demonstrate that the limited directories provide lower processor utilizations



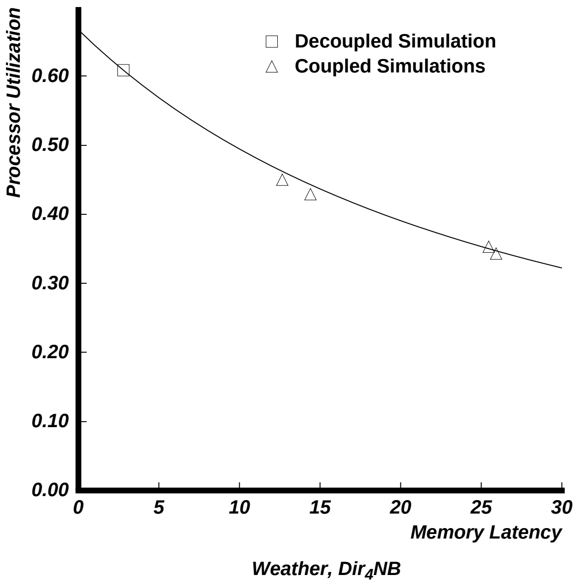


Figure 4-4: Processor utilization versus memory latency. The curve indicates the prediction of the network model. The individual points are data from simulations.

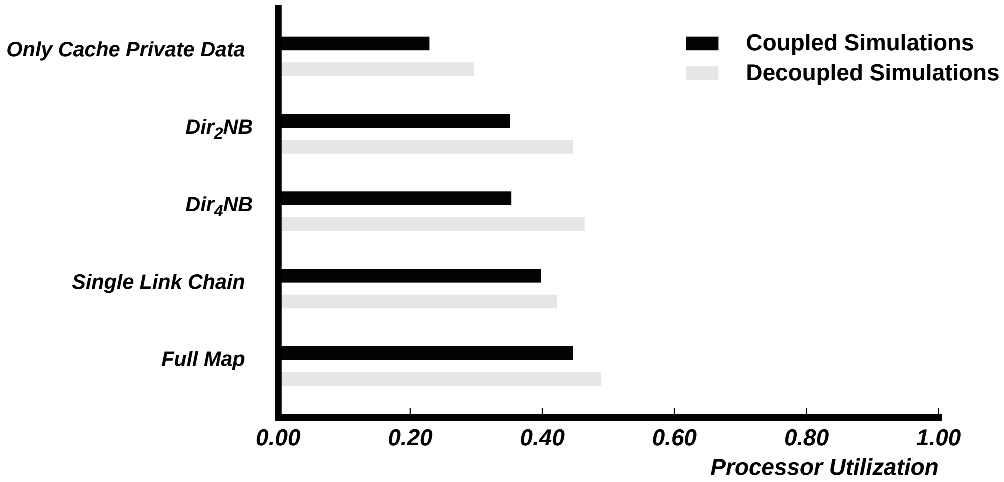


Figure 4-5: Comparison of processor utilization measurements for Weather, after adjusting the memory latency, but before eliminating the hot-spot.

than the full-map protocol.

The discrepancy between the predicted and the actual performance of limited directory protocols is due to an inaccuracy in the decoupled simulation technique. Recall that the decoupled evaluation technique averages the effects of data requests over the entire duration of a trace and over all of the components in the simulated multiprocessor. This methodology does not account for hot-spot contention, which results from a concentration of requests impinging on a single component.

The Weather application uses a variable that belongs to the class of *write-once* data. Section 5.2.3 explores the implications of this write-once variable on the balance between multiprocessor software and cache coherence schemes. For now, it is sufficient to note that the combination of write-once data and a limited directory protocol creates a constant flow of data requests from every processor in the system to the memory module that contains the variable. The constant flow of protocol messages causes hot-spot contention at the memory module. The decoupled methodology averages this hot-spot traffic over the entire multiprocessor. But in the coupled simulations, the one memory module that handles the write-once variable becomes a bottleneck, due to the extraordinary number of protocol messages. In addition to describing the semantics of the variable that causes the hot-spot phenomenon, Sec-

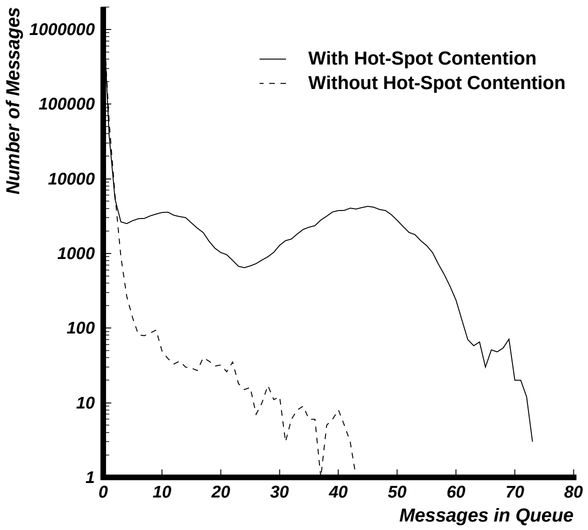


Figure 4-6: Cache controller queue sizes with Dir_4NB protocol.

tion 5.2.3 describes how to optimize multiprocessor software to eliminate this type of data.

Figure 4-6 shows the effect of the hot-spot caused by Weather's write-once variable. The graph shows a histogram of the size of the cache controller network queues for coupled simulations with and without hot-spot contention. Since network queues store protocol messages that memory modules need to transmit through the network, the histogram gives a sense of the amount of time that data requests have to wait to be serviced. The solid curve on the histogram shows the behavior of the system with the hot-spot data accesses, and the dashed curve shows the performance once the write-once variable has been optimized. Note that the vertical axis is in a logarithmic scale. Figure 4-6 illustrates the fact that hot-spot contention causes thousands of protocol messages to wait in long queues. However, optimizing for the write-once location effectively removes the hot-spot.

After the hot-spot has been removed, the processor utilizations observed for the limited directory schemes in ASIM conform to the prediction of the decoupled simulation technique. In fact, this is the last step necessary to reconcile the differences between the two evaluation methodologies. Figure 4-2 shows the processor utilizations of the Weather program, with special code in the dynamic post-mortem sched-

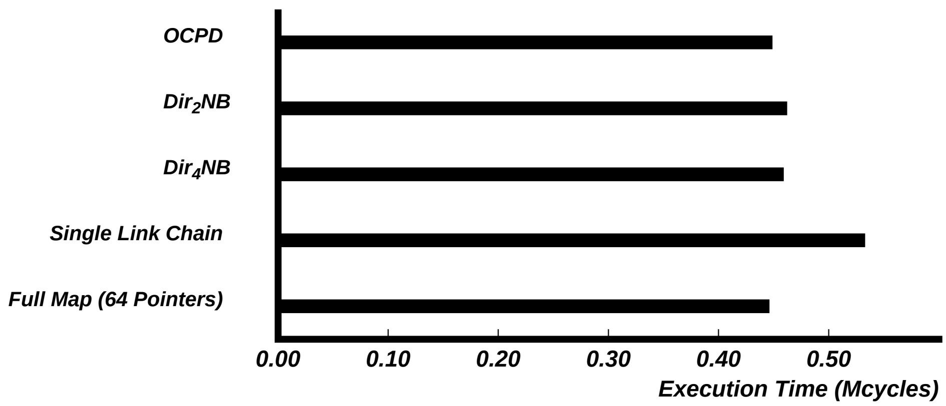


Figure 4-7: Comparison of coherence schemes using coupled simulations.

uler that marks the write-once data location. Although the results from the two simulation techniques correlate well, the performance of the protocols in coupled simulations remains slightly below the predictions of the decoupled methodology, due to the non-uniform distribution of requests to memory modules.

4.4.3 The Processor Utilization Metric

In general, processor utilization provides a good indication of the performance of a multiprocessor. However, the processor utilization metric does not always accurately indicate the performance of a program running on a multiprocessor. Figure 4-7 shows the execution times that correspond to the processor utilizations reported in Figure 4-2. While the directory-based coherence schemes perform as predicted by the decoupled simulations, the scheme that only caches private data (OCPD) performs better, relative to the other schemes, than the decoupled method predicts.

The difference between the predicted and the actual performance of the OCPD scheme is explained in [32] by examining the way that processor utilization is defined. The utilization metric estimates the contribution of the memory system to the time needed to run a program on the system, without actually measuring the forward progress of an application. When shared data is cached, synchronization requests from the processors are satisfied in the cache and raise processor utilizations values, even if the synchronizations fail and do not contribute to making forward progress. The processor utilization for the OCPD scheme does not profit from failed synchronizations, so the metric gives a more realistic value for the absolute performance of OCPD than it does for the other coherence schemes. Section 5.1.2 discusses other factors that impact the behavior of the OCPD scheme.

Despite the problems with predicting the bottom-line performance of the OCPD scheme, the hybrid decoupled simulation technique gives a good indication of the contribution of a memory system to the overall performance of a multiprocessor. Coupled simulations complement the decoupled simulations by supporting detailed analysis of the implementation of a cache coherence scheme and the interaction between a multiprocessor's memory system and its software.

Chapter 5

Analysis

The decoupled and coupled simulation techniques generate a plethora of raw statistics, including invalidation histories, cache hit-miss profiles, memory latency histograms, and other records that describe the low-level behavior of cache coherence protocols. Taken as a whole, this body of information demonstrates the interactions between a multiprocessor's software and its shared memory implementation. In order to present the key results of the simulations, the statistics are refined into two different parameters: processor utilization and execution time. Processor utilization, as derived from the hybrid decoupled methodology, quantifies the contribution of a memory system to the time needed to run a program on a multiprocessor. Execution time, the total number of cycles needed to run a program in a coupled simulation, directly measures the speed of a system. When necessary, these metrics will be supplemented with other statistics to present a complete analysis of the behavior of cache coherence protocols.

This chapter uses the decoupled and coupled simulation techniques to analyze the behavior of various cache coherence protocols. Section 5.1 establishes the performance of the coherence schemes. Section 5.2 discusses system-level techniques that improve the performance of directory-based protocols, and Section 5.3 shows that the scalable LimitLESS protocol achieves high performance by using the integrated systems approach. Section 5.4 evaluates additional coherence protocol features that can be used to improve the efficiency of a shared memory system. Finally, Section 5.5 draws conclusions about the design of large-scale cache-coherent multiprocessors.

5.1 Performance of Cache Coherence Schemes

Before engaging in discussing how to optimize directory-based cache coherence protocols, it is necessary to establish that this class of protocols is worth studying. Once the simulations prove that directory protocols show potential for realizing a high-performance shared-memory system, further effort is spent to show how to modify multiprocessor software to interact well with coherent caches. Note that many of the programs evaluated below are not written especially for cache-based shared memory systems. Since this section shows that caches work well for most applications that use no knowledge about the implementation of the memory system, it indicates that programs that are optimized to take advantage of caches will perform even better.

5.1.1 Analysis of Directory Schemes

The data from decoupled simulations is presented in graphs that plot various combinations of applications and cache coherence schemes on the vertical axis and processor utilization on the horizontal axis. Since the data reference characteristics vary significantly between applications and trace gathering methods, results from the different traces are not averaged. The results that are presented here concentrate on the Weather, Speech, and P-Thor applications. Other applications are discussed when they exhibit significantly different behavior.

5.1.2 Are Caches Useful for Shared Data?

Figure 5-1 shows the processor utilizations realized for the Weather, Speech, and P-Thor applications using each of the coherence schemes. The long bar at the bottom of each graph gives the value for "no cache coherence." This number is derived by considering all addresses in each trace to be non-shared. Processor utilization with no cache coherence gives, in a sense, the effect of the native hit/miss rate for the application. The number is artificial, because it does not represent the behavior of a correctly operating system. However, the number does give an upper bound on the performance of *any* coherence scheme and helps determine the component of processor utilization that is lost due to sharing between processors.

To assess the potential of shared data caching schemes in general, compare the optimal (full-map) directory scheme to the scheme that only caches private data (OCPD). For most applications (including the ones shown in Figure 5-1), the full-map directory gives significantly better processor utilization than the scheme that only caches private data. Generally good performance of the full-map scheme in 16 and 64 processor machines implies that caches are useful for shared data, even when applications are not written or compiled specially for a system with directory-based cache coherence.

However, for two traces (FFT and MP3D), processor utilization for a full-map directory is worse than the utilization for the private data cache scheme. Examining the network model shows the reason why it is possible for private data caches to perform better than full-map directories: Even though the private cache scheme has a higher network message rate, it uses smaller message block sizes. In the model, network latency is proportional to the square of the message block size, but is linearly dependent on the message rate. The fact that for FFT and MP3D the private data cache scheme performs better than the full-map directory scheme indicates that the average time between writes by different processors to each shared location is low. For these traces, the full-map directory scheme does not perform significantly better than the limited directory schemes.

It is important to note that the coupled and decoupled simulation techniques incorporate two features that artificially improve the performance of the OCPD scheme, relative the other protocols. First, the network model assumes a low overhead for message transmission. A memory system that supports input/output and error correction mechanisms requires information to be wrapped around protocol messages. Since the OCPD scheme transmits relatively short messages through the network, it can not amortize the cost of the transmission overhead as well as the other coherence schemes. Second, since the OCPD scheme treats every memory word as a separate entity, it does not suffer when unrelated data objects are allocated to consecutive addresses in memory. This type of data organization reduces the performance of the

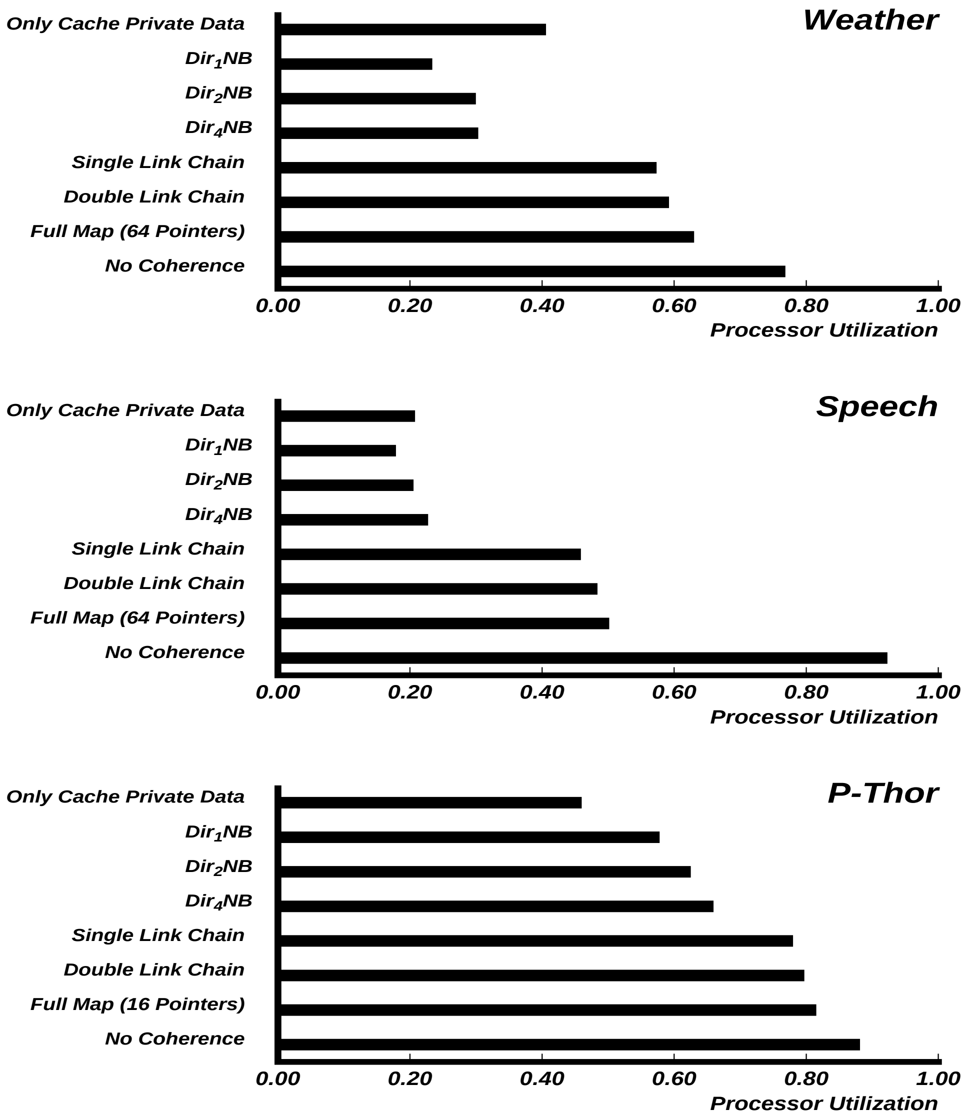


Figure 5-1: Comparison of coherence schemes.

other cache coherence schemes, because allocating more than one shared data object per cache block tends to reduce processor locality, thereby reducing the effectiveness of caches.

Laying performance issues aside, there is a compelling reason to use hardware-enforced cache coherence, rather than the OCPD scheme. In practice, OCPD requires compile-time differentiation between private and shared data. In some programming environments, it is difficult (if not impossible) to accomplish the separation of shared and private data at compile-time. In these cases, the directory protocols allows caches to take advantage of locality, without requiring support from the the compiler or the programmer.

5.1.3 Limited Directory Performance

How well do limited directories perform compared to the full-map directory scheme? The answer depends on the amount of shared data, the number of processors that access each shared data location, and the method of synchronization. The P-Thor application was written to minimize the communication between processors by reducing the number of synchronization points and the number of processors that read each shared location. It is not surprising that all of the directory schemes perform well for this application.

On the other hand, four traces show significantly worse processor utilization for limited directories than for a full-map directory due to naive synchronization techniques (Weather, SIMPLE, and SA-TSP) or widespread sharing of a large read-only data structure (Speech). Section 5.2 investigates methods for ameliorating the effects of widespread sharing on limited directory protocols.

5.1.4 Chained Directory Performance

When applications use data structures that are widely shared and accessed frequently, a limited directory performs significantly worse than a full-map directory. However, Figure 5-1 shows that both singly and doubly-linked directories perform almost as well as the full-map directory protocols. While the doubly-linked scheme always performs slightly better than the singly-linked scheme, the small increase in performance may not justify the additional resources needed for the doubly-linked scheme. The difference between the schemes is small because the number of replacements as a percentage of total memory accesses is very small, even though the memory system uses direct-mapped caches.

In general, the decoupled simulation technique indicates that chained directory schemes provide higher utilization than limited directory protocols. However, as discussed in Section 4.2.5, the decoupled methodology does not account for the difference between the sequential invalidation patterns in chained directories and the parallel invalidations of limited and full-map directories. Thus, chained directory protocols have potentially longer write latency than limited directory protocols.

Coupled simulations provide evidence that the longer write latency impacts the performance of chained directories. Table 5.1 shows the average latencies for various data request types during the simulations of the Weather application. The chained directory protocol performs worse than the full-map and limited directories, because it suffers from a higher average latency for write transactions to shared data than the other protocols. Furthermore, the full-map and the limited directories can use the modify request (MREQ) and modify grant (MODG) protocol messages to decrease the average shared data write latency (see Section 3.4.1). The chained protocol can not use the modify request/grant message pair, while the optimization reduces the shared data write latency of both the full-map and limited directories to about 28 cycles.

The difference between the shared data write latencies explains why the full-map and limited directory protocols perform better than the chained directory, assuming the optimizations described in Section 5.2. Since the average directory chain length grows with the number of processors in a system, the gap between the shared data write latencies of chained directory protocols and other protocols becomes wider as processors are added to the system. Thus, due to the latencies caused by the structure of a chained directory, the coherence protocol does not scale as well as a the other

	Average Latency		
Memory Request Type	Chain	Full-Map	Dir_4NB
All Data	3.42	3.03	3.09
Private Data	0.39	0.39	0.39
Shared Data	26.09	21.51	21.90
Shared Data Read	18.54	18.50	18.92
Shared Data Write	61.34	35.59	35.77

Table 5.1: Average latency statistics (in processor cycles) for the Weather application with combining tree synchronization and write-once data optimization.

directory protocols.

5.2 Improving the Performance of Directories

The results presented in the previous section show that limited directory schemes suffer from data types that are both widely shared and frequently referenced. The Weather and Speech applications serve as case studies that demonstrate three methods for ameliorating the effects of this type of data. These methods are examples of *system-level optimizations*, because they involve contributions from several components of a multiprocessor system. In addition to improving the performance of limited directory schemes, the methods also enhance the performance of the other coherence schemes.

5.2.1 Optimizing Synchronization Variables

The Weather application uses barriers as the primary method of synchronization. In the straightforward implementation of barriers, each processor increments a barrier variable and then spin-locks on a barrier flag. The last processor to reach the synchronization point increments the barrier variable to its final value N and writes into the barrier flag, thereby releasing the spinning processors. The memory accesses from many processors spin-locking on a single location cause pointer thrashing (repeated evictions) in the limited directory.

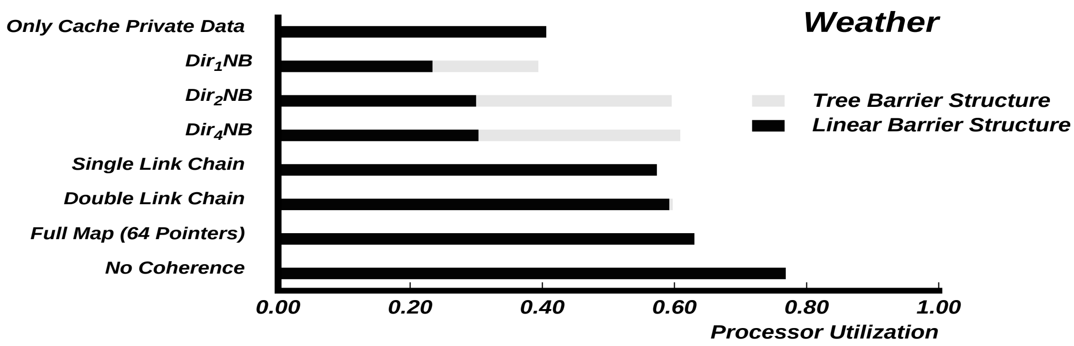
A software solution, called a *combining tree* [46], can alleviate this problem in directories. Instead of implementing barrier synchronizations with a single barrier variable and barrier flag, a balanced tree structure of nodes can be used for each. To demonstrate the benefits of this barrier implementation, the post-mortem scheduler was modified to implement combining tree synchronization. The resulting trace was virtually identical to the original trace, except with respect to the distribution of synchronization address accesses. In the original trace, all of the synchronization addresses were accessed by all of the processors; while in the combining tree trace, almost all of the synchronization addresses were accessed primarily by one processor, with just one access by one other processor.

The top graph in Figure 5-2 shows that the combining tree dramatically improves the performance of the limited directory schemes. The darker colored bars show the processor utilization of the application with linear barrier synchronization, and the lighter bars show the enhanced utilization when using the combining tree structure. The two and four pointer directories give nearly the same processor utilization as the full-map scheme. The one pointer directory suffers from sharing of other data between processors. However, this data sharing must exist only between processor pairs, because it does not affect the two pointer directory. Thus, combining tree structures and limited directory schemes provide an efficient implementation of barrier synchronization.

5.2.2 Optimizing Read-Only Data

The Speech application provides an example of both a different programming model and a different type of widely-shared data. There are two primary data structures in the Speech application: an utterance (the sentence to be identified) and a dictionary (the algorithm's vocabulary). For the duration of the application, these data structures are only read, but they are shared by all the processors in the system. This type of data reference pattern causes pointer thrashing in limited directories.

Given the nature of the Speech application, it is fair to assume that all the read-only variables can be identified by the programmer. To assess the potential benefits of marking read-only data, the trace was post-processed to find all the data locations that were only read for the duration of the trace. The read-only locations were



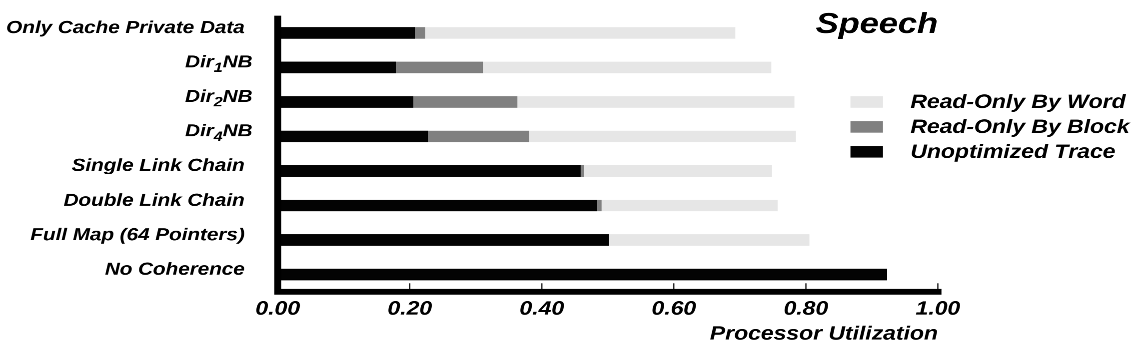


Figure 5-2: System-level optimizations.

then marked as private to prevent the cache and directory simulator from executing coherence transactions for this data. When these locations were identified on a block-by-block basis, the system showed moderate improvement for the limited directory schemes. However, when the post-processor identified the read-only locations on a word-by-word basis and relocated the data to a special segment of memory, the improvement was more pronounced. The bottom graph in Figure 5-2 demonstrates the increase in processor utilization realized by specially processing read-only data. The darkest bars show the unoptimized performance of the Speech application, the lighter bars show the gains due to processing read-only data.

The boost in processor utilization due to read-only data detection on a word-by-word basis can be explained by the reduction of sharing due to cache blocks that contain unrelated data words that are accessed by different processors. The Mul-T runtime system ignores the boundary of cache blocks and allocates read-write data words in the same cache blocks as read-only data words. This data allocation policy prevents the block-by-block post-processor from properly identifying read-only data words and lowers processor utilization by creating unnecessary shared data traffic in the network.

5.2.3 Optimizing Write-Once Data

As discussed in Section 4.4.2, a write-once variable can cause performance degradation due to hot-spot contention at a memory module. In the case of the Weather application, there is one variable that is initialized at the beginning of the program, read frequently by all of the processors in the system, and not modified for the duration of the application. Since all of the processors in the system frequently read the memory location that stores the variable, the limited directory entry continuously thrashes. That is, almost as soon as the memory module containing the location transmits a read-only copy of the variable to a processor, the module transmits an invalidation so that another processor can receive a copy.

The problems caused by this variable can be eliminated quite easily. Since the variable is modified only in an initial, sequential portion of the program, a programmer can make a local copy of the data (perhaps in a stack segment) for each processor in the system. To remove the effect of this variable, ASIM simulates a protocol mechanism that handles write-once data. When a memory location is marked by a compiler as *write-once*, indicating that it is initialized by one processor and then only read by other processors, the protocol uses a special data fetch mechanism to distribute the location without thrashing its limited directory entry.

The coupled simulation technique reveals the effects of the hot-spot contention caused by Weather's write-once variable. Figure 5-3 shows the performance of the Weather application with combining tree synchronization, before and after the variable has been optimized. Since the figure's horizontal axis measures the execution time (as opposed to processor utilization) for each coherence protocol, the longer bars are the simulations with the unoptimized variable, and the shorter bars show the increased performance with special code in the dynamic post-mortem scheduler that marks the write-once data location.

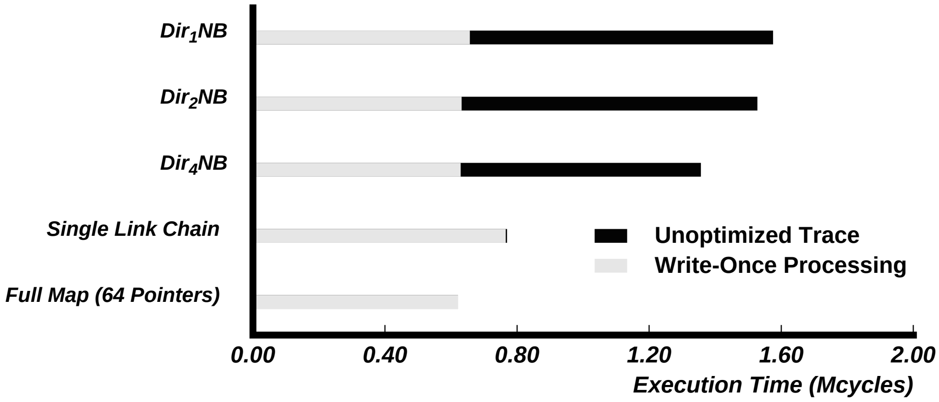


Figure 5-3: Comparison of coherence schemes after write-once optimization.

Although the hot-spot contention caused by the write-once variable seriously degrades the performance of the limited directory protocols, after write-once data optimizations, the coherence schemes show the same relative behavior as in Figure 5-2. Thus, the write-once variable in Weather provides yet another example of a system-level optimization that makes limited directory protocols a viable alternative for implementing scalable shared memory systems.

5.2.4 Implications of Directory Optimization Techniques

When multiprocessor algorithms and software are optimized for caches, large-scale cache-coherent systems realize their execution potential. In the case of the Weather and Speech applications, system-level optimizations resulted in processor utilizations between 0.6 and 0.8 for scalable cache coherence protocols. Coordinating multiprocessor hardware and software requires some subset of programmer specifications, new language primitives, special compile-time analysis, support in the runtime system, specialization in the processor-to-cache interface, and additional states in the cache coherence protocol. Combining tree synchronization, read-only data optimization, and write-once data optimization are archetypes of system-wide efforts to improve multiprocessor performance.

5.3 LimitLESS Directory Protocol Performance

The results in Figures 5-2 and 5-3 show that while it is possible for a limited directory protocol to perform as well as a full-map directory protocol, limited directories are extremely sensitive to the optimization of multiprocessor software. In order to satisfy Alewife's programmability goal, the performance of a memory system should not degrade so quickly when the worker-set of a single memory location overflows its directory entry. The LimitLESS directory protocol, which is described in Section 3.2.1, avoids the sensitivity of limited directories to software optimization. When a directory entry overflows, the associated memory module requests its local processor to handle the exception in software. Preliminary results from ASIM confirm this strategy.

As shown in Figure 5-4, the LimitLESS protocol avoids the sensitivity displayed by limited directories. This figure compares the performance of a full-map directory, a four-pointer limited directory (Dir_4NB), and the four-pointer LimitLESS (LimitLESS$_4$) protocol with several values for the additional latency required by the LimitLESS protocol's software (T_s = 25, 50, 100, and 150). The execution times show that the LimitLESS protocol performs about as well as the full-map directory protocol, even in a situation where a limited directory protocol does not perform well. Furthermore, while the LimitLESS protocol's software should be as efficient as possible, the performance of the LimitLESS protocol is not strongly dependent on the latency of the full-map directory emulation. The current estimate of this latency in the Alewife machine is between 50 and 100 cycles.

Surprisingly, the LimitLESS protocol, with a 25 cycle emulation latency, actually performs better than the full-map directory. This anomalous result is caused by the participation of the processor in the coherence scheme. By interrupting the Weather application software and slowing down certain processors, the LimitLESS protocol produces a slight back-off effect that reduces contention in the interconnection network.

The number of pointers that a LimitLESS protocol implements in hardware interacts with the worker-set size of data structures. Figure 5-5 compares the performance

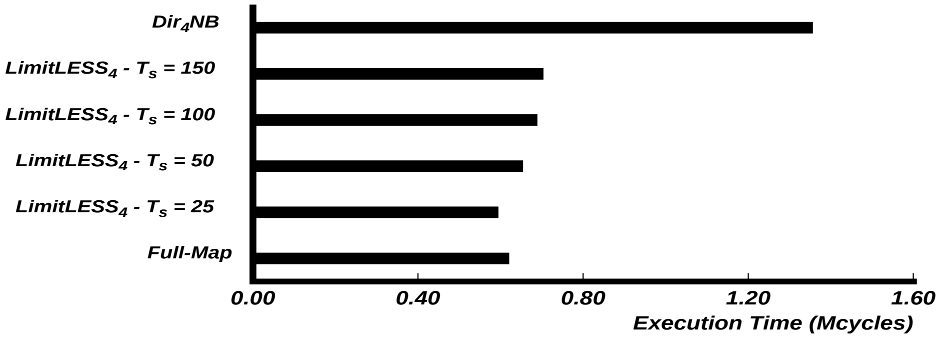


Figure 5-4: Weather, 64 Processors, LimitLESS with 25 to 150 cycle directory emulation latencies.

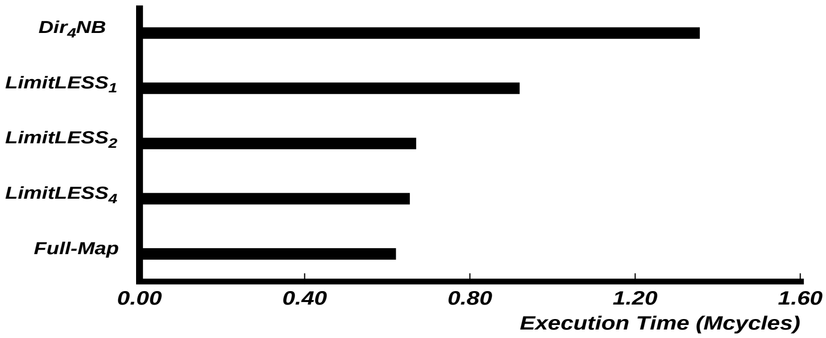


Figure 5-5: Weather, 64 Processors, LimitLESS scheme with 1, 2, and 4 hardware pointers.

of Weather with a full-map directory, a limited directory, and LimitLESS directories with 50 cycle emulation latency and one (LimitLESS$_1$), two (LimitLESS$_2$), and four (LimitLESS$_4$) hardware pointers. The performance of the LimitLESS protocol degrades gracefully as the number of hardware pointers is reduced. The one-pointer LimitLESS protocol is especially bad, because some of Weather's variables have a worker-set that consists of exactly two processors.

This behavior indicates that multiprocessor software running on a system with a LimitLESS protocol will require some of the optimizations that would be needed on a system with a limited directory protocol. However, the LimitLESS protocol is much less sensitive to programs that are not perfectly optimized. Moreover, the software optimizations used with a LimitLESS protocol should not be viewed as extra overhead caused by the protocol itself. Rather, these optimizations might be employed, regardless of the cache coherence mechanism, since they tend to reduce hot-spot contention and to increase communication locality.

5.4 Implementation Issues

In addition to the issue of directory implementation, there are several other protocol features that have the potential to significantly affect the performance of a coherence scheme. The coupled simulation technique yields performance results for some of these features. Section 5.4.1 compares two methods that allow a processor to tolerate memory access latency, and shows that both have significant potential to improve multiprocessor performance. Section 5.4.2 analyzes several protocol features that have only a weak effect on the performance of a shared-memory system.

5.4.1 Weak Ordering versus Multiple Contexts

To confirm the assumption that multiple contexts are a viable method for masking the latency of shared-memory transactions, the performance of multiple contexts can be compared with that of weak ordering, another method for hiding the latency of memory transactions.

With multiple contexts per processor, it is possible to quickly switch between threads of control when a memory request must be transmitted over the interconnection network. The ability to switch from a thread of control that needs to wait for a response from a remote memory module allows a processor to overlap the latency of any type of shared-memory transaction with useful execution cycles. Such a system can overlap cache read misses, cache write misses, and attempts to write to read-only cache blocks, while still providing sequential consistency (the strongest form of cache coherence).

Proponents of weak ordering take a slightly different approach to the problem of shared-memory latency. Architectures that provide weak ordering assume that there is a contract between the programmer and the memory system: The programmer agrees to obey strict synchronization semantics, and the memory system ensures only that all transactions are complete at synchronization points. In other words, if the programmer agrees to write multiprocessor code using critical sections, then the memory system will be able to overlap some types of shared-memory transactions and still provide sequential consistency. However as discussed in Section 2.2, if the programmer violates the rules imposed by the memory system, then the system's behavior is undefined and programs may not operate correctly or even deterministically.

Since the two schemes approach the memory latency issue by using different architectural mechanisms, they are not mutually exclusive. However, the comparison of the performance of weak ordering and multiple contexts affords insight into methods for masking the delay of shared memory latency.

In order to evaluate the two methods, ASIM is instrumented to determine an upper bound on the performance of a system with weak ordering. ASIM collects statistics about a weakly ordered memory system, while simulating a single-context processor with a memory system that provides sequential consistency. The algorithm records the latency of the memory transactions that may be overlapped with execution; namely, write misses and attempts to write to read-only cache blocks. At every point in the simulation, the algorithm can determine the latest time that overlapped transactions will be completed for each processor. So when a thread that is running on a processor performs a synchronization, the algorithm is able to determine how long a weakly ordered memory system would have to stall the processor to ensure that all of the outstanding memory transactions were complete. ASIM calculates the performance of weak ordering for each processor by subtracting overlapped cycles from the strongly coherent execution time, and adding the extra wait time at synchronization points. The performance of weak ordering for the entire system is calculated by averaging the execution time over all of the processors.

It is important to emphasize that ASIM determines only an *upper bound* for the performance of weak ordering (by determining a *lower bound* on the execution time of a weakly ordered system). Two assumptions that are implicit in the analysis cause the performance of weak ordering to appear better than it would be on an actual system:

- The software has not been written for a weakly ordered system. While the code that is executed by ASIM probably does conform to the programmer/memory-system contract described above, it is not required to do so. Recoding an algorithm to run on a weakly ordered system means adding extra synchronization, which would increase the overall execution time.

- The latency for remote accesses is assumed to be the same for both the single-context system that provides sequential consistency and for the system that provides weak ordering. In general, the latency of remote accesses will be longer for the weakly ordered system, since the higher processor utilization afforded by weak ordering also causes more contention in the interconnection network and at the cache controllers.

ASIM's upper bound is valid only for the original definition of weak ordering [19]. The approximation method does not properly simulate the behavior of other definitions of weak ordering, such as the one proposed in [1]. Furthermore, ASIM does not account for techniques that can improve the performance of weakly ordered systems. For instance, a data prefetch mechanism could be used to allow a weakly ordered system to reduce the effects of read misses. On the other hand, a data prefetch would also improve the performance of a protocol that guarantees sequential consistency, so such optimizations have been eliminated from the following performance analysis.

Table 5.2 shows that the 64 processor system with two contexts per processor requires less execution time in processor cycles than the lower bound on the system that provides weak ordering. This conclusion is independent of the cache coherence protocol. Thus, even if the fast context-switch were useful for no other purpose, it would be a good mechanism for masking the latency of shared-memory transactions.

As a final note, the Weather and Simple applications do very little synchronization, so the weakly ordered system does not need to force its processors to wait at synchronization points. Weak ordering would not perform as well for applications that must synchronize more often than the applications used for the results above,

Weather Application			
	Dir_4NB	Full-Map	Chain
1 Context	1,356,447	620,874	769,518
2 Contexts	1,234,554	459,156	554,541
Weak Ordering	1,256,069	535,814	570,830

Simple Application			
	Dir_4NB	Full-Map	Chain
1 Context	613,209	590,276	614,378
2 Contexts	385,564	388,496	408,035
Weak Ordering	453,166	426,769	440,509

Table 5.2: Comparison of Lower Bound on Weak Ordering and Exact Simulation of Multiple Contexts. Units are in Processor Execution Cycles.

while multiple contexts may be used to overlap synchronization delays as well as the latency of shared-memory transactions. However, the performance measurements show that weak ordering and multiple contexts are both viable methods for tolerating memory access latency.

5.4.2 Second-Order Effects

The implementation of a cache coherence protocol contributes to the absolute performance of the shared-memory system. Figure 5-6 shows the effect of several of the implementation issues discussed in Section 3.4 on the performance of Weather running with a limited (Dir_4NB) and a full-map directory protocol. The graphs compare the performance of the base protocols with the performance of the protocols modified in three different ways.

The *One Ack Counter* bar indicates the execution time of the system with one acknowledgment per controller, instead of one counter per directory entry. Using only one counter per controller reduces the effective bandwidth of the memory system, so the execution time is slightly higher for the full-map protocol. The counter implementation interacts in an interesting way with the limited protocol hot-spot contention (see Section 4.4.2). With only one counter per controller, the directory (implemented in slow DRAM modules) does not need to be read to process acknowledgment messages, thereby reducing the time spent processing the eviction of pointers from a limited directory. Since pointer eviction is the dominant effect in this particular application, the counter implementation actually increases the bandwidth of the controller and decreases the execution time of the system. This anomalous interaction further emphasizes the sensitivity of limited directory protocols to widely-shared data locations.

The protocol with the REPU message notifies the memory controller whenever a Read-Only copy of data is replaced in a processor's cache. While this modification is intended to reduce the number of unnecessary invalidation messages, it actually creates more network traffic than it saves, thereby increasing the execution time. The modify grant (MODG) message prevents memory modules from sending redundant data to caches when processors write data soon after reading it. This is the most beneficial of the unessential protocol messages, and tends to decrease execution times.

The most significant conclusion that can be made from the investigation into protocol implementation is that unessential messages and other implementation details have a second-order effect compared to issues such as directory structure and software optimizations. This relationship does not indicate that the implementation should be haphazard; however, it does justify the time invested in examining the higher-level issues.

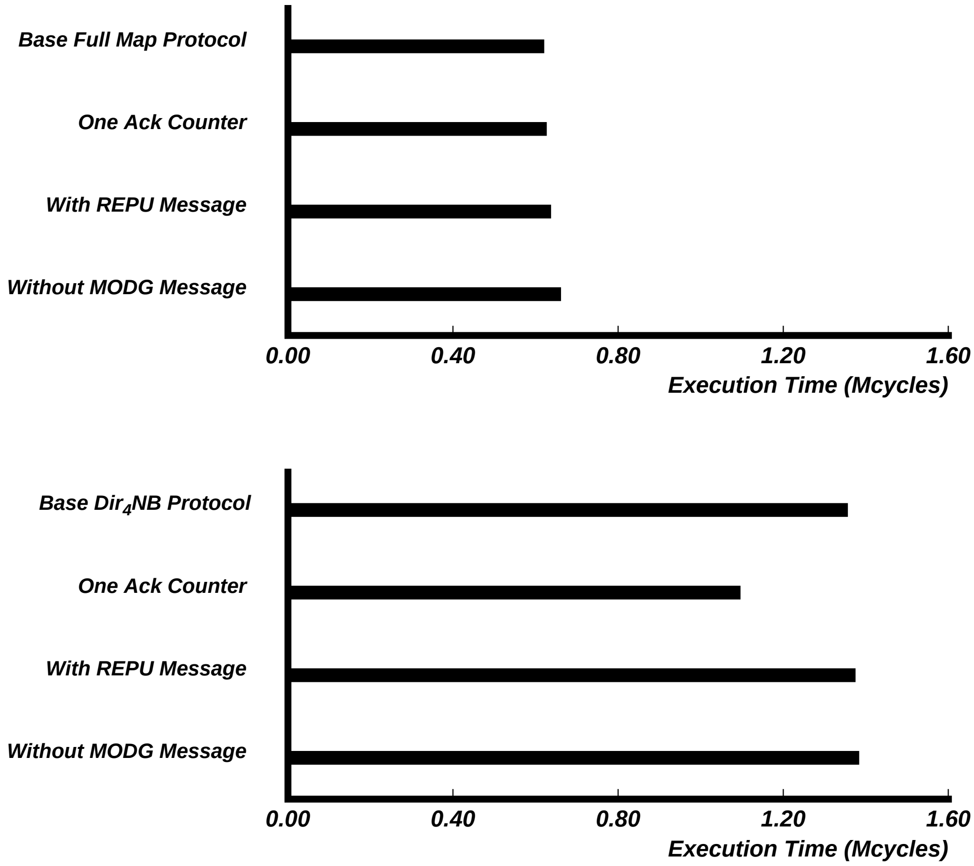


Figure 5-6: The effect of protocol implementation on the performance of Weather.

5.5 Conclusions

The search for a cache coherence protocol for the Alewife machine has led to the following conclusion: By balancing the responsibility for supporting a cache coherence protocol between a system's hardware and software components, it is possible to build a large-scale cache-coherent multiprocessor. Both coupled and decoupled simulation methods reveal design strategies that help achieve such a balance. Two basic strategies have been identified; namely, optimizing a multiprocessor's software for its memory system and following the integrated systems approach when designing a cache coherence protocol. These two strategies address the problem of scalable cache coherence protocol design by mitigating the effects of widely-shared data.

The software optimization method uses techniques such as distributed synchronization structures and identification of special data types to reduce the effects of a program's widely-shared data. In doing so, the strategy permits limited directory cache coherence schemes to be a viable option for implementing a scalable shared-memory system. That is, an optimized program performs as well with a scalable limited directory protocol as with a non-scalable full-map directory protocol. However, limited directories exhibit excessive sensitivity to the limitations of software optimization. Chained directory protocols are also candidates for the basis of a scalable shared-memory system, but may incur excessive write latencies in very large systems.

The integrated systems approach relies on the observation that once a multiprocessor's software has been modified to reduce data sharing, accesses to shared variables are relatively rare in a cache-based system. By handling uncommon requests for widely-shared data in software, the LimitLESS directory protocol approaches the performance of a full-map directory protocol, with the memory efficiency of a limited directory protocol. Furthermore, the LimitLESS protocol provides a migration path toward a future in which cache coherence is handled entirely in software.

While the analysis of cache coherence schemes lays the foundation for the Alewife machine's shared-memory system, the protocol design effort is far from over. Once the details of the LimitLESS directory implementation are solidified, it will be necessary to verify the correctness and fairness properties of the protocol. Even after the protocol is bound into hardware, finding new design strategies that balance hardware scalability and software complexity will be an ongoing topic of research.

Appendix A

Tables of Statistics

The following tables contain numerical results from the simulations described in Chapters 4 and 5.

A.1 Trace-Driven Simulation Results

Table A.1 lists the results for the FORTRAN application suite. Table A.2 lists the results for the C and T-Mul-T suites. Table A.3 lists the results for the Weather application with combining tree synchronization. Finally, Table A.4 lists the results for the Speech application with read-only data processing. Section 4.2 discusses the parameters listed in these tables.

Application	Coherence Scheme	Block Size	Request Rate	Memory Latency	Processor Utilization
FFT	Dir_1NB	5.919	0.1420	2.968	0.3300
	Dir_2NB	4.825	0.1546	2.301	0.3554
	Dir_4NB	4.831	0.1354	2.448	0.3864
	Full-Map	4.852	0.1318	2.468	0.3922
	OCPD	3.093	0.1453	3.000	0.4281
	No Coherence	6.000	0.0103	3.000	0.8939
Weather	Dir_1NB	4.507	0.2853	2.087	0.2337
	Dir_2NB	4.307	0.2162	2.003	0.2997
	Dir_4NB	4.337	0.2114	2.023	0.3033
	Full-Map	5.491	0.0485	2.954	0.6301
	Singly-Linked	4.731	0.0660	3.000	0.5733
	Doubly-Linked	5.012	0.0593	3.000	0.5921
	OCPD	3.037	0.1599	3.000	0.4059
	No Coherence	6.000	0.0248	3.000	0.7683
Simple	Dir_1NB	4.701	0.3512	2.189	0.1902
	Dir_2NB	4.400	0.2632	2.019	0.2539
	Dir_4NB	4.608	0.2124	2.149	0.2905
	Full-Map	5.532	0.0958	2.751	0.4479
	OCPD	3.046	0.2066	3.000	0.3423
	No Coherence	6.000	0.0645	3.000	0.5373

Table A.1: Simulation results for FORTRAN application suite.

Application	Coherence Scheme	Block Size	Request Rate	Memory Latency	Processor Utilization
P-Thor	Dir_1NB	4.881	0.0760	2.396	0.5779
	Dir_2NB	4.347	0.0709	2.068	0.6251
	Dir_4NB	4.434	0.0606	2.141	0.6595
	Full-Map	5.131	0.0240	2.770	0.8151
	Singly-Linked	4.721	0.0303	3.000	0.7799
	Doubly-Linked	4.999	0.0266	3.000	0.7970
	OCPD	3.036	0.1498	3.000	0.4599
	No Coherence	6.000	0.0129	3.000	0.8813
LocusRoute	Dir_1NB	4.786	0.0244	2.365	0.8255
	Dir_2NB	4.459	0.0155	2.239	0.8901
	Dir_4NB	4.869	0.0096	2.609	0.9229
	Full-Map	5.087	0.0081	2.814	0.9317
	OCPD	3.087	0.0511	3.000	0.7234
	No Coherence	6.000	0.0051	3.000	0.9506
SA-TSP	Dir_1NB	5.768	0.3726	2.928	0.1707
	Dir_2NB	4.007	0.3857	1.824	0.2245
	Dir_4NB	4.006	0.3851	1.824	0.2248
	Full-Map	4.031	0.0854	2.085	0.5898
	OCPD	3.019	0.1950	3.000	0.3928
	No Coherence	6.000	0.0013	3.000	0.9871
MP3D	Dir_1NB	4.863	0.2807	2.580	0.2462
	Dir_2NB	4.024	0.2628	2.075	0.2983
	Dir_4NB	4.027	0.2394	2.109	0.3189
	Full-Map	4.051	0.1690	2.268	0.4007
	OCPD	3.029	0.1761	3.000	0.4182
	No Coherence	6.000	0.0132	3.000	0.8792
Speech	Dir_1NB	4.680	0.3770	2.279	0.1787
	Dir_2NB	4.031	0.3666	1.893	0.2053
	Dir_4NB	4.042	0.3229	1.919	0.2274
	Full-Map	4.187	0.0979	2.337	0.5016
	Singly-Linked	4.224	0.1082	3.000	0.4587
	Doubly-Linked	4.623	0.0932	3.000	0.4838
	OCPD	3.010	0.4010	3.000	0.2076
	No Coherence	6.000	0.0074	3.000	0.9228

Table A.2: Simulation results for the C and Mul-T application suites.

Coherence Scheme	Block Size	Request Rate	Memory Latency	Processor Utilization
Dir_1NB	5.007	0.1280	2.483	0.3939
Dir_2NB	5.014	0.0599	2.664	0.5957
Dir_4NB	5.179	0.0554	2.792	0.6087
Full-Map	5.328	0.0497	2.920	0.6294
Singly-Linked	4.597	0.0679	3.000	0.5704
Doubly-Linked	4.993	0.0583	3.000	0.5972
OCPD	3.035	0.1598	3.000	0.4062

Table A.3: Results for Weather with combining tree synchronization.

Read-Only Data Unit	Coherence Scheme	Block Size	Request Rate	Memory Latency	Processor Utilization
Cache Block	Dir_1NB	5.675	0.1584	3.042	0.3106
	Dir_2NB	4.111	0.1704	2.086	0.3629
	Dir_4NB	4.114	0.1585	2.113	0.3808
	Full-Map	4.187	0.0979	2.337	0.5016
Memory Word	Dir_1NB	5.999	0.0274	3.096	0.7475
	Dir_2NB	5.392	0.0251	2.746	0.7825
	Dir_4NB	5.409	0.0247	2.757	0.7846
	Full-Map	5.748	0.0208	2.994	0.8052
	Singly-Linked	4.701	0.0318	3.000	0.7484
	Doubly-Linked	4.744	0.0303	3.000	0.7570
	OCPD	3.754	0.0465	3.000	0.6930

Table A.4: Results for Speech with read-only data processing.

A.2 Dynamic Post-Mortem Scheduler

The following two tables consolidate data from the dynamic post-mortem scheduler, coupled with the ASIM memory system. Table A.5 lists the results for the Weather application, and Table A.6 lists the results for the Simple application.

Coherence Scheme	Protocol Options	Execution Time
Full-Map	None.	620874
	without MODG	661573
	with BUSY loop-back	619976
	with REPU	637474
	1 ack counter/module	627043
	write-once optimized	621393
Single-Linked	None.	769518
	write-once optimized	765544
OCPD	None.	617464
Dir_4NB	None.	1356447
	without MODG	1383866
	with BUSY loop-back	1359043
	with REPU	1375051
	1 ack counter/module	1096390
	write-once optimized	629086
Dir_2NB	None.	1527552
	write-once optimized	631953
Dir_1NB	None.	1575031
	write-once optimized	656811
LimitLESS$_4$	25 cycle latency	594716
	50 cycle latency...	654444
	without MODG	694706
	with BUSY loop-back	653099
	with REPU	659739
	write-once optimized	653930
	100 cycle latency	689113
	150 cycle latency	703801
LimitLESS$_3$	50 cycle latency	614585
LimitLESS$_2$	50 cycle latency	669784
LimitLESS$_1$	50 cycle latency	920283

Table A.5: Results for Weather with combining tree synchronization.

Coherence Scheme	Protocol Options	Execution Time
Full-Map	None.	202545
	without MODG	202700
	with BUSY loop-back	
	with REPU	204419
	1 ack counter/module	205519
Single-Linked	None.	205918
OCPD	None.	241614
Dir_4NB	None.	212441
Dir_2NB	None.	220749
Dir_1NB	None.	233823
$\mathrm{LimitLESS}_4$	25 cycle latency	202697
	50 cycle latency	203307
	100 cycle latency	204440
	150 cycle latency	205035
$\mathrm{LimitLESS}_3$	50 cycle latency	203974
$\mathrm{LimitLESS}_2$	50 cycle latency	229382
$\mathrm{LimitLESS}_1$	50 cycle latency	300598

Table A.6: Results for Simple with combining tree synchronization.

Appendix B

Cache Coherence Protocol Specification for ASIM

This appendix specifies the cache coherence protocol that is implemented in ASIM, the Alewife System Simulator. The heart of the specification is a series of protocol state transition diagrams, so the text presents enough information to make the diagrams intelligible. Given these diagrams, it is possible to hand-simulate any sequence of memory transactions. Since the protocol is intended to be experimental, it contains many features that will not be implemented in Alewife's hardware. The purpose of documenting the protocol is to illustrate the different options that have been considered during the implementation of Alewife's shared memory system.

The cache coherence protocols that are specified below either ensure *sequential consistency* on a transaction-by-transaction basis or provide mechanisms for multiprocessor software to enforce cache coherence. A protocol that ensures cache coherence on a transaction-by-transaction basis is called a *hardware coherence protocol*, because the multiprocessor hardware presents a consistent model of shared memory to its software. A protocol that provides mechanisms for software enforcement of cache coherence is called a *software coherence protocol*, because it relies on the programmer, the compiler, and/or the runtime system to guarantee correct program behavior.

The communication between a coherence protocol and the multiprocessor software is confined to the processor/controller interface. When a processor needs to perform a load, store, or other memory access, it executes one of the hardware or software coherent accesses listed in Table B.1. When the processor issues one of the above access types, the cache/memory controller must respond with one of the three signals listed in Table B.2. Section 3.3.1 describes the processor/controller interface.

B.1 The State Transition Diagrams

A cache coherence protocol consists of the set of possible states in the local caches, the states in the shared memory, and the state transitions caused by the messages that are transported through the interconnection network to keep memory coherent. So, the protocols are specified by state transition diagrams that illustrate cache or

Coherence Type	Name	Unit of Access	Processor Access Type
Hardware	**Read**	Word	Load
	Write	Word	Store
Software	**ReadSoft**	Word	Load
	WriteSoft	Word	Store
	FlushSoft	Block	Flush block from cache
	Fence	Control Thread	Wait until all flushes complete

Table B.1: Processor Request Types

Name	Controller Response
READY	Access complete at the end of the current cycle
SWITCH	Context switch
WAIT	Repeat the same access on the next cycle

Table B.2: Controller Response Types

memory states, transitions between the states, processor requests that cause transitions, protocols messages that cause transitions, controller to processor responses, and protocol message transmissions.

Each diagram displays all of the possible cache or memory states for a block of data. It is important to note that this state is stored on a block-by-block basis. Every processor request (except **Fence**) and every protocol message contain a block address, and affect only the block of data with the corresponding address. On any given cycle, every block of data in the system has a protocol state that is defined by the block's state in shared memory, the block's state in each processor's cache, the protocol messages that contain the block's address, and the processor requests that contain the block's address. Each cache or memory state is represented as an ellipse with the name of the state in boldface type. Since the state transition diagrams are complicated, they are split over several pages, which may be viewed as overlays. That is, each page of a diagram shows all of the states but only a logical subset of the possible transitions. The entire state transition diagram would be visible if all of the individual pages were transparencies, and they were displayed simultaneously on an overhead projector. Table B.3 gives the figures that compose each of the complete state transition diagrams.

On the diagrams, every transition is represented as an arrow with a label that indicates the inputs, side-effects, and output that are associated with the transition.

Transition Diagram	Figure Numbers	Protocol Described		
		Software	Limited	Chained
Cache State	B-1–B-8	√	√	√
Memory State	B-9–B-13	√	√	
Memory State	B-14–B-18	√		√

Table B.3: Correspondence between Transition Diagrams and Figures.

The cache state transitions have two possible label formats. The first format, used in Figures B-1 through B-3, indicates transitions that are caused by the processor requests:

Processor Access Type / Output Message (Controller Response)

The possible values for the **Processor Access Type** are listed in Table B.1 and the **Controller Response** values are listed in Table B.2. The **Output Message** is an optional field in the format that is used to specify a protocol message that the controller may send over the interconnection network to satisfy the processor access. The second format, used in Figures B-4 through B-8, indicates cache state transitions that are caused by messages received from the interconnection network:

Input Message / Output Message

The **Input Message** is the protocol message that causes the state change, and the **Output Message** is the response that the controller sends back through the network. If no response is necessary, the **Output Message** field contains a tilde (~). The memory state transitions in Figures B-9 through B-18 use a similar format:

processor id: Input Message / Side Effects / Output Message

The **processor id:** is an optional field that specifies the identifier of the node that sends the **Input Message**. If the transition is accompanied by changes to the directory, then they are specified as **Side Effects**, otherwise the field contains a tilde. As in the cache state transition diagrams, if a response is necessary, it is specified in the **Output Message** field.

Any (processor access type, cache state) or (input message, state) combinations that are not specified by the transition diagrams are considered error conditions. Such error conditions are caused by states that are disallowed by the protocols or by interactions between hardware and software coherence (see Section B.4). If an illegal combination is detected by ASIM, the protocol module terminates the simulation run and reports the error condition.

Some local state changes are implicit in the state transitions, but may not be obvious. For instance, when a controller responds with a **READY** signal to a processor **Write** request, this implies that the controller has stored the data from the data bus into the appropriate word in the cache. Or, when a controller processes an **UPDATE** message, it stores the data block that is contained in the message into the appropriate block of shared memory. To reduce the complexity of the state transition diagrams, these types of implicit state changes are not specified.

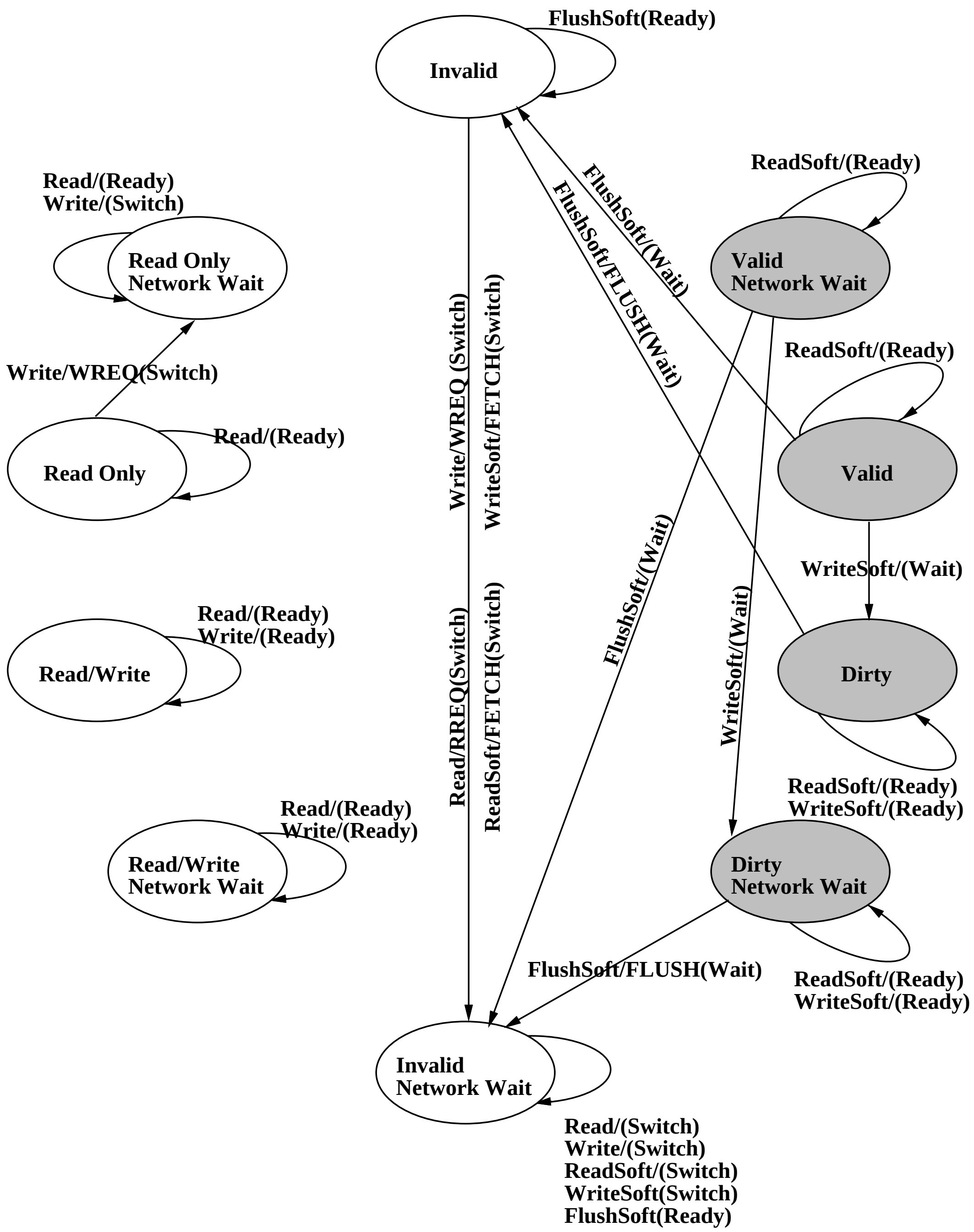


Figure B-1: Cache state transitions for processor requests when the tag matches.

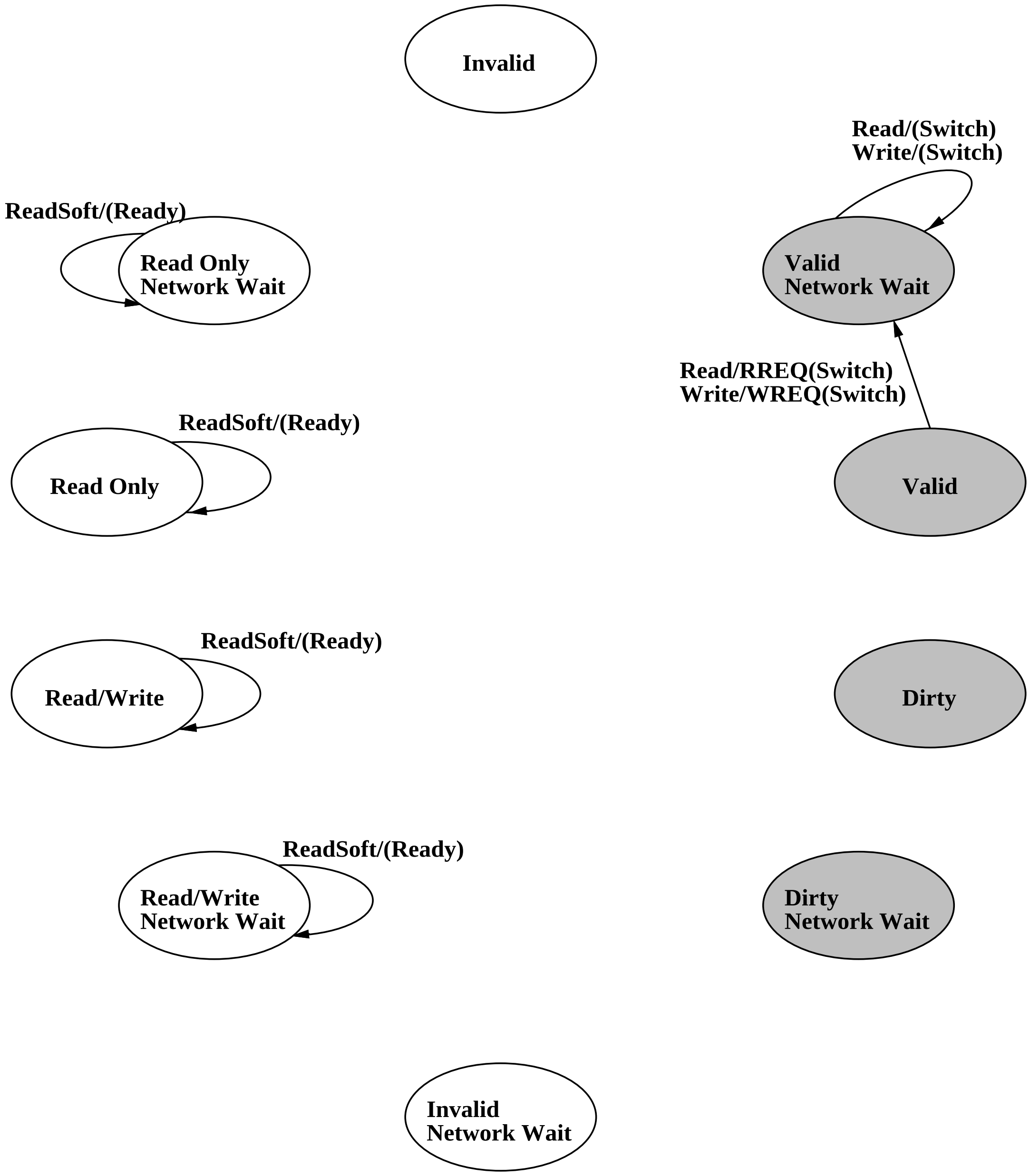


Figure B-2: Cache state transitions for processor requests when the tag matches, hardware/software coherence interaction.

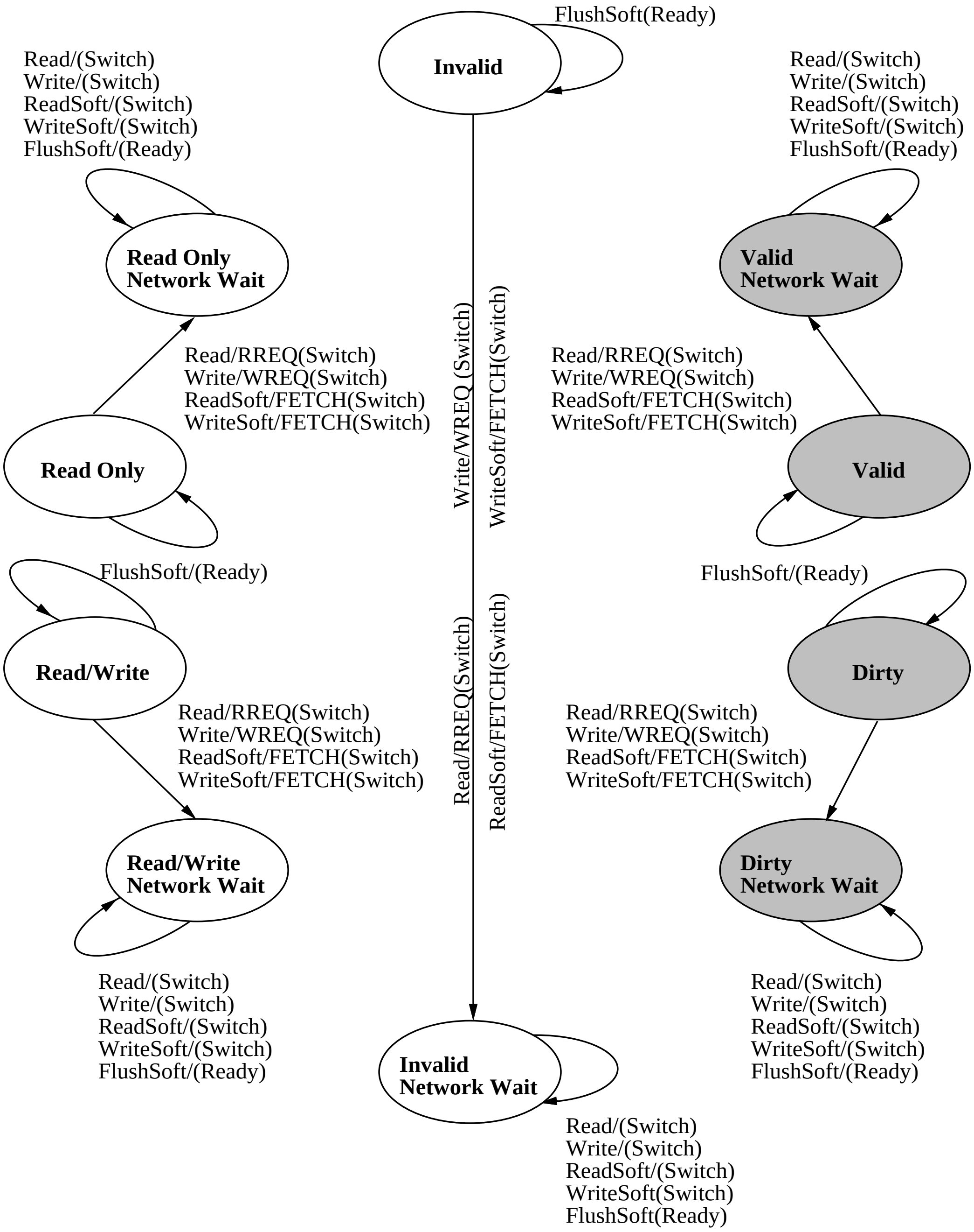


Figure B-3: Cache state transitions for processor requests when the tag does not match.

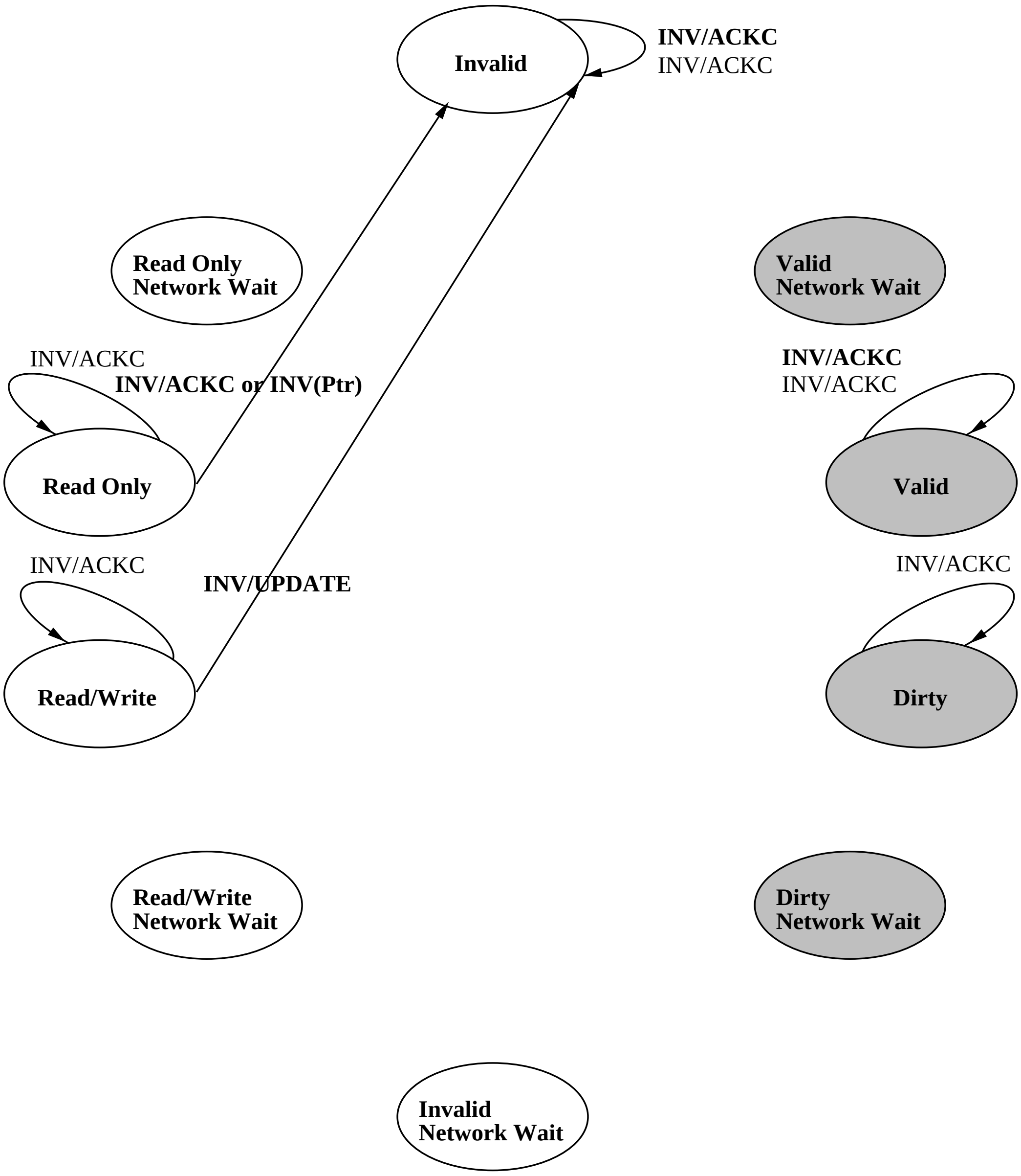


Figure B-4: Cache state transitions for messages: Non-network-wait states.

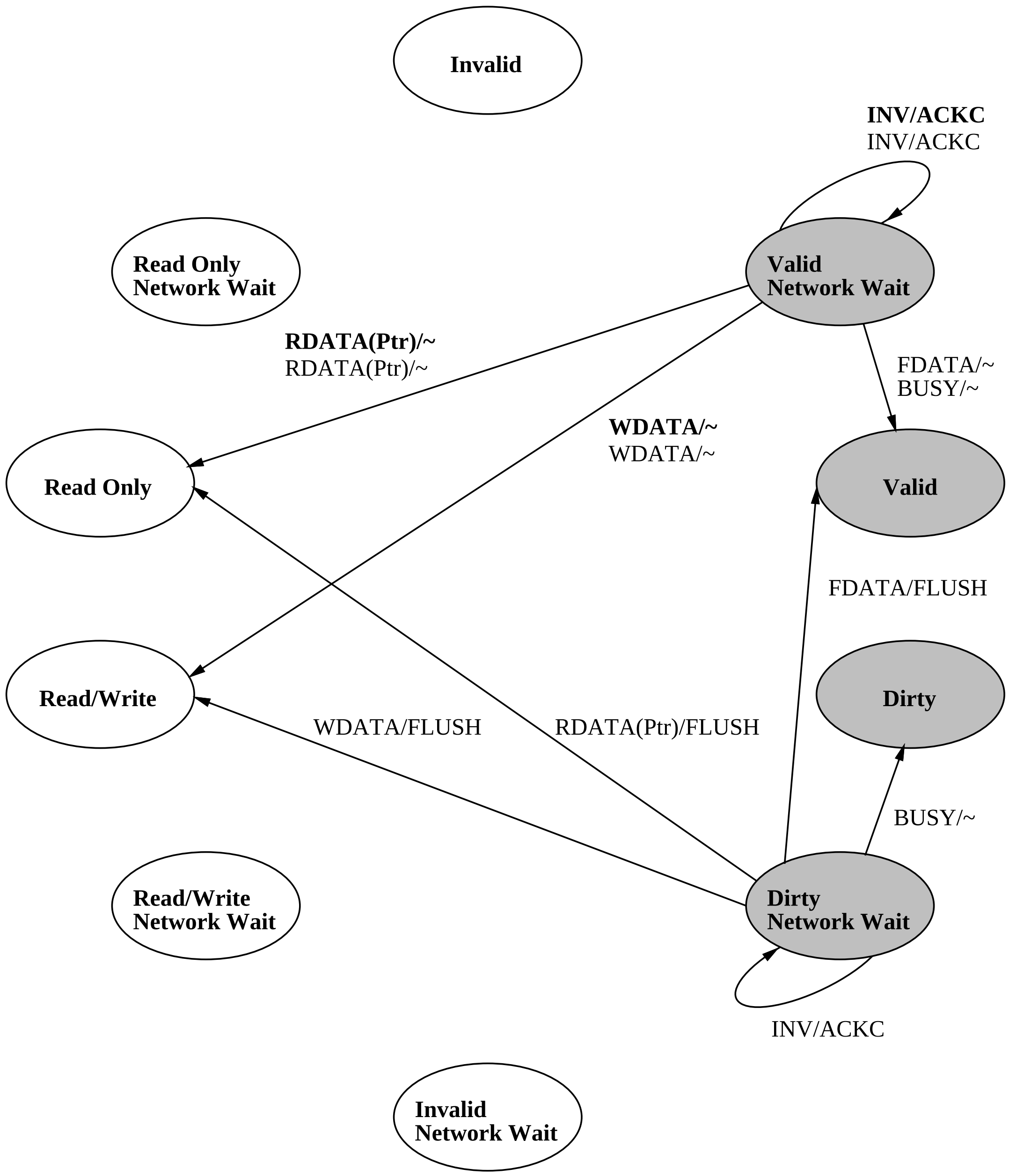


Figure B-5: Cache state transitions for messages: Software coherence **Network Wait** states.

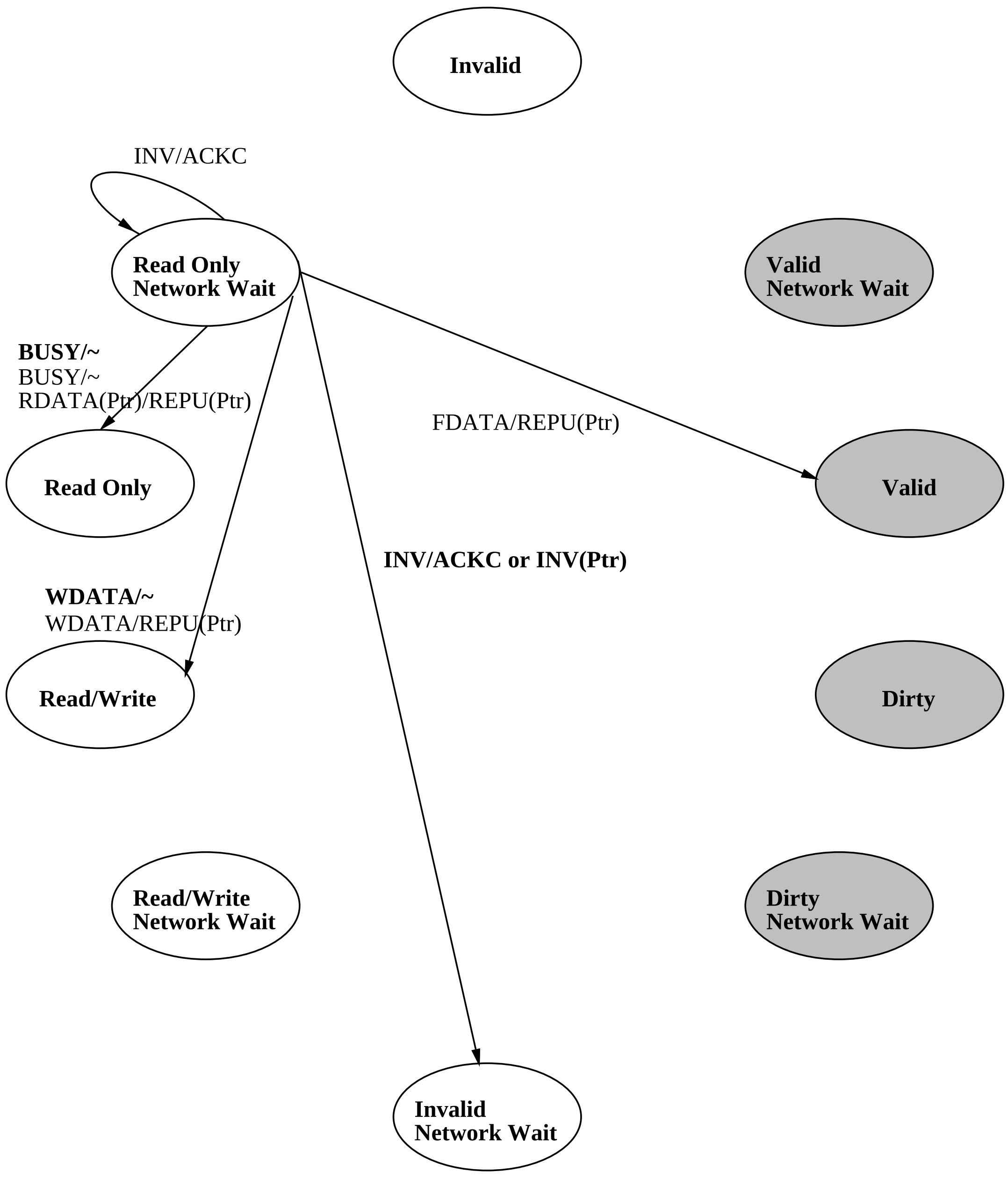


Figure B-6: Cache state transitions for messages: **Read Only Network Wait** state.

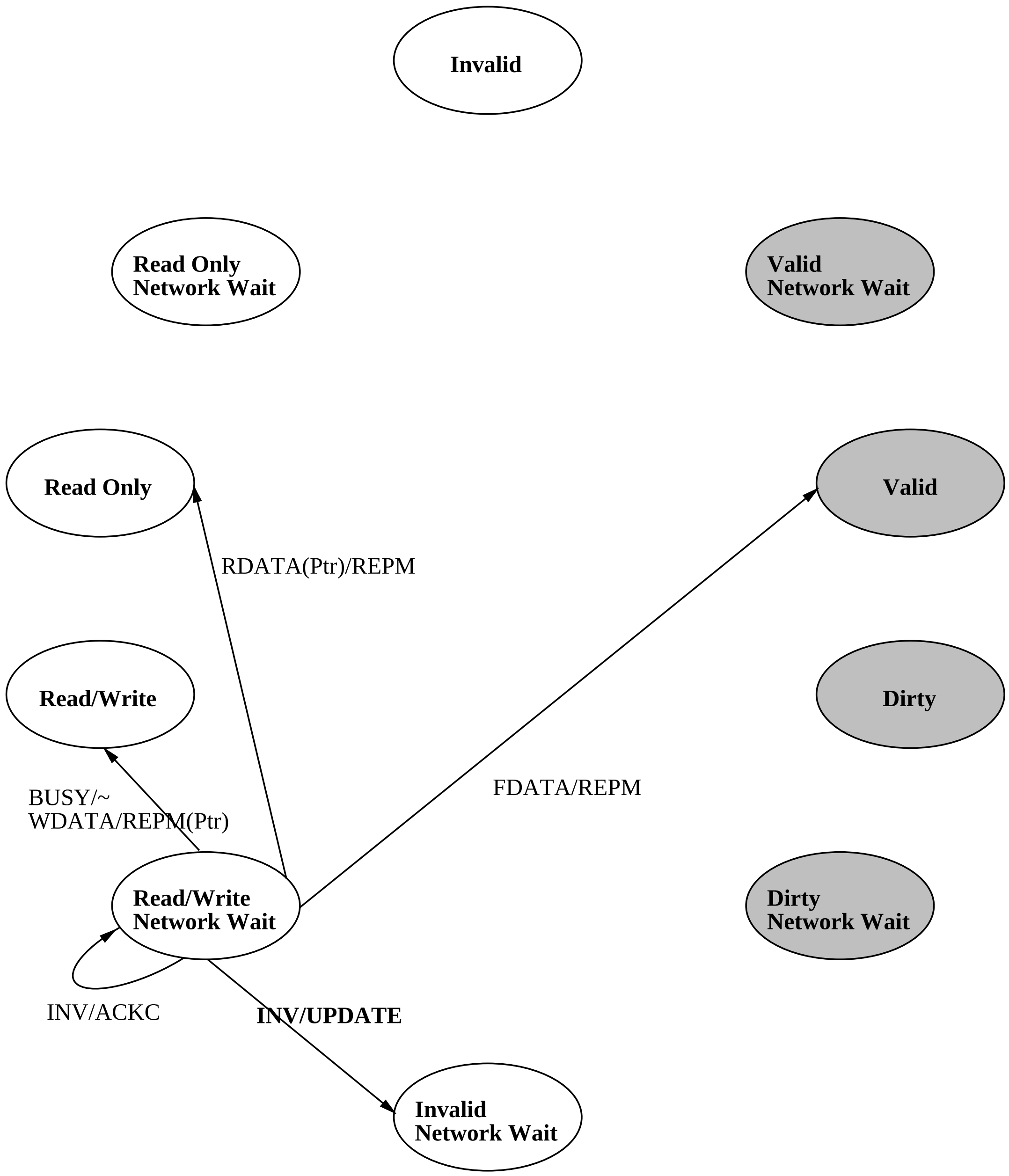


Figure B-7: Cache state transitions for messages: **Read/Write Network Wait** state.

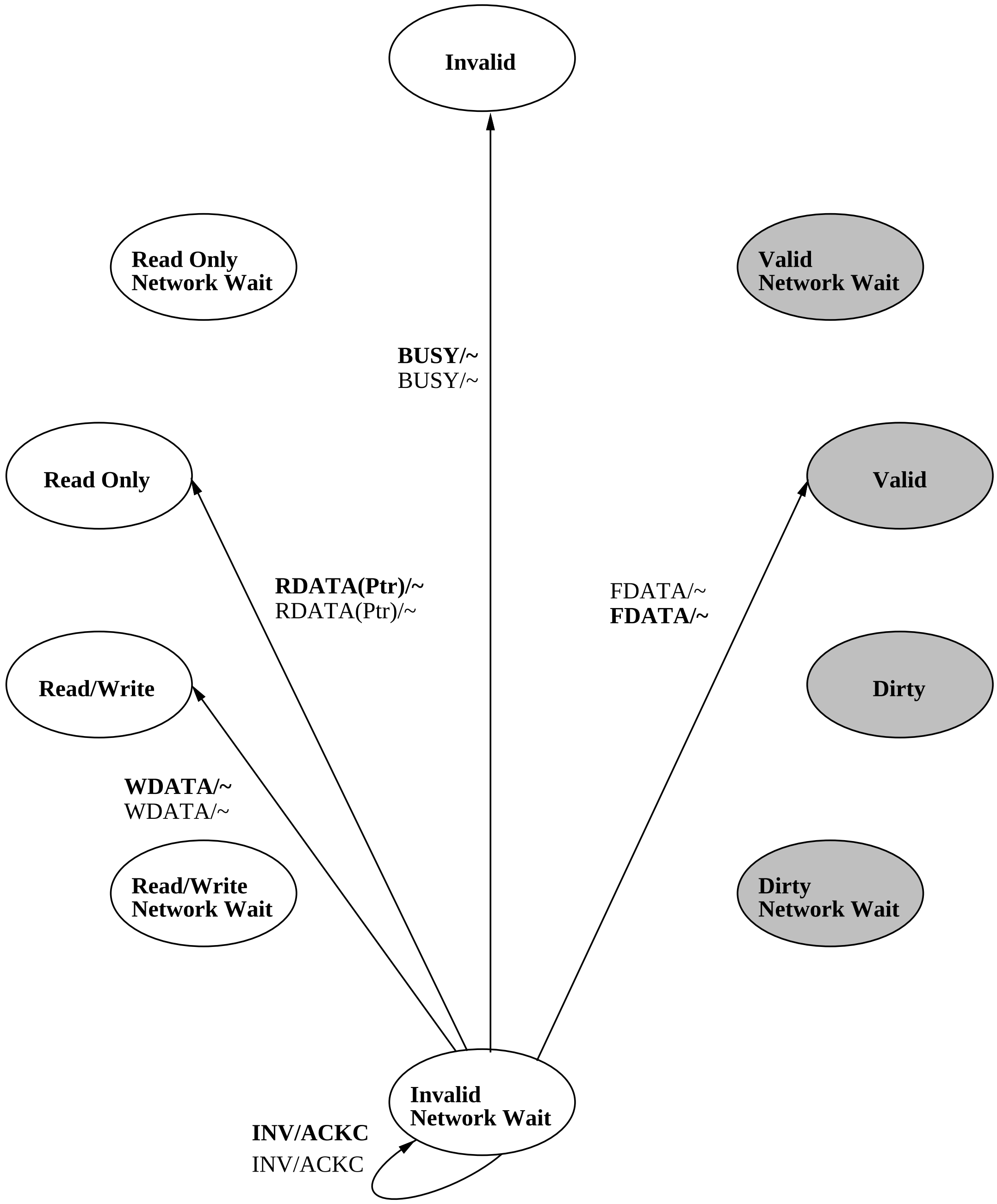


Figure B-8: Cache state transitions for messages: **Invalid Network Wait** state.

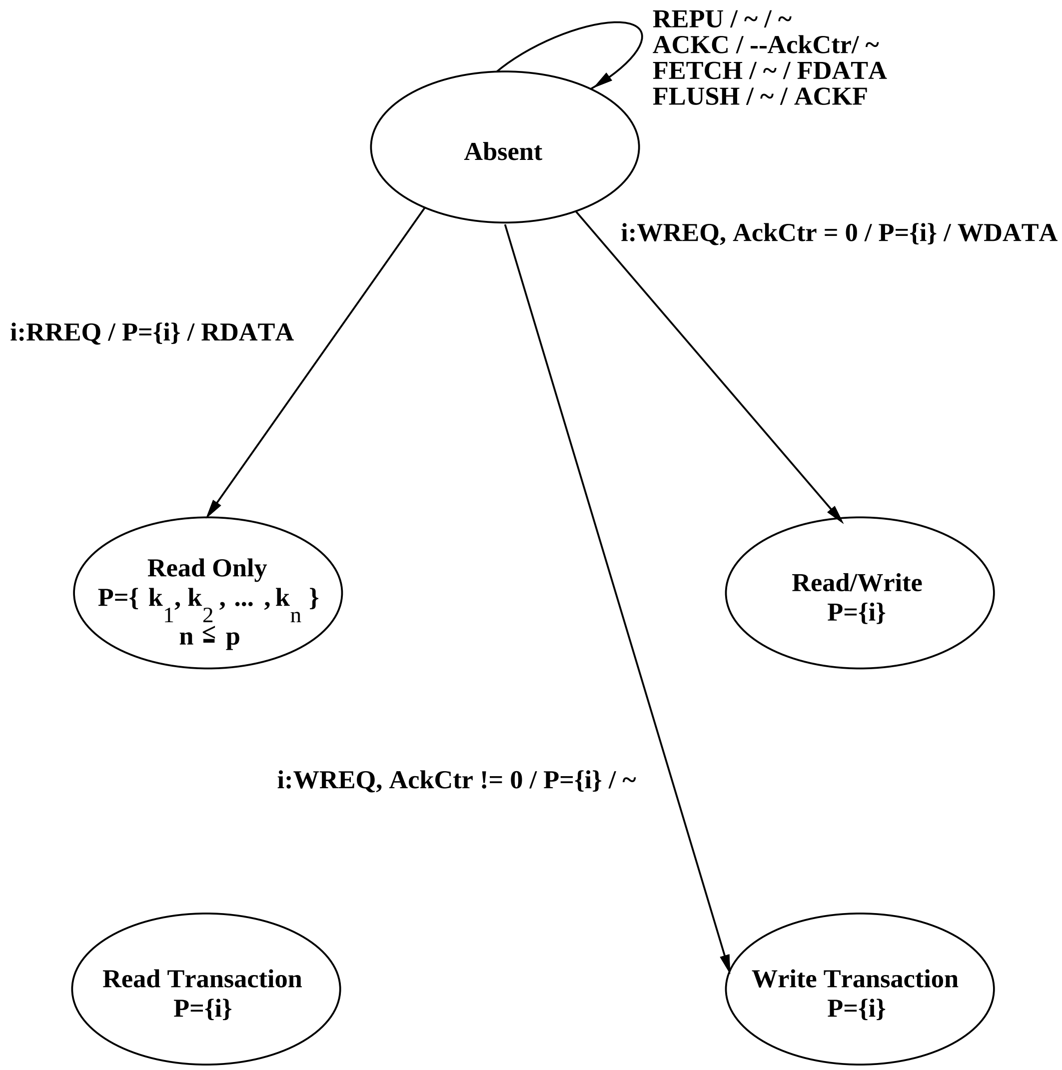


Figure B-9: Memory state transitions for limited directory: **Absent** state.

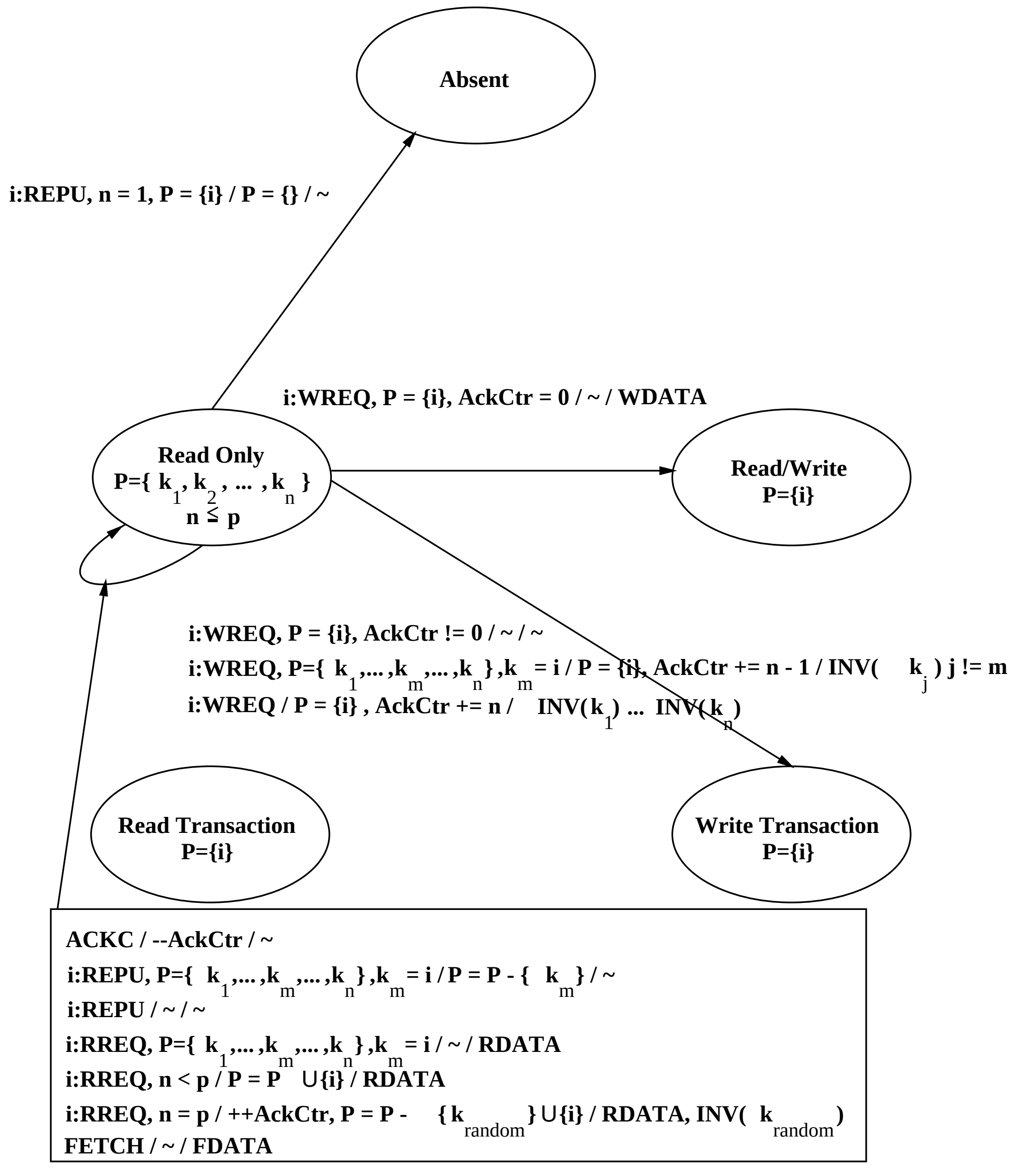


Figure B-10: Memory state transitions for limited directory: **Read Only** state.

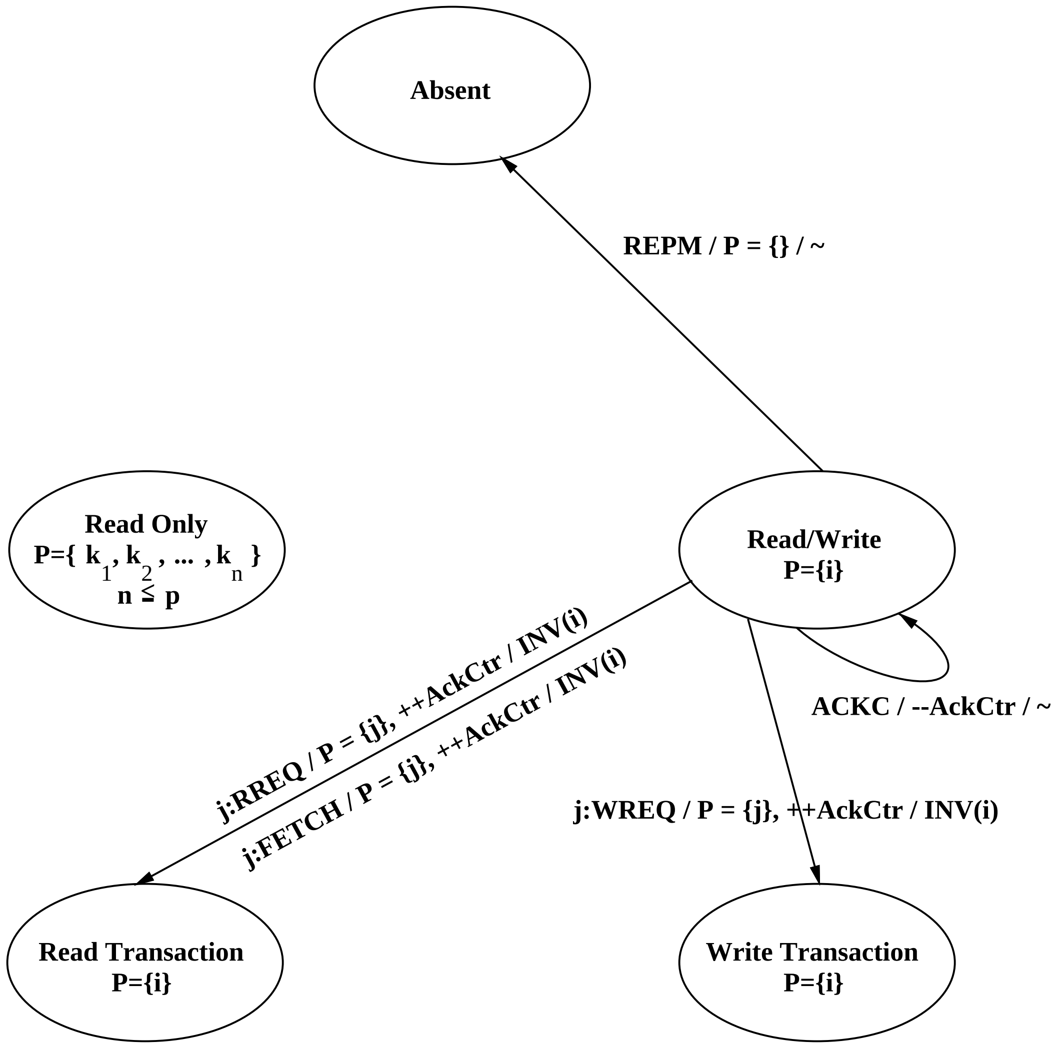


Figure B-11: Memory state transitions for limited directory: **Read/Write** state.

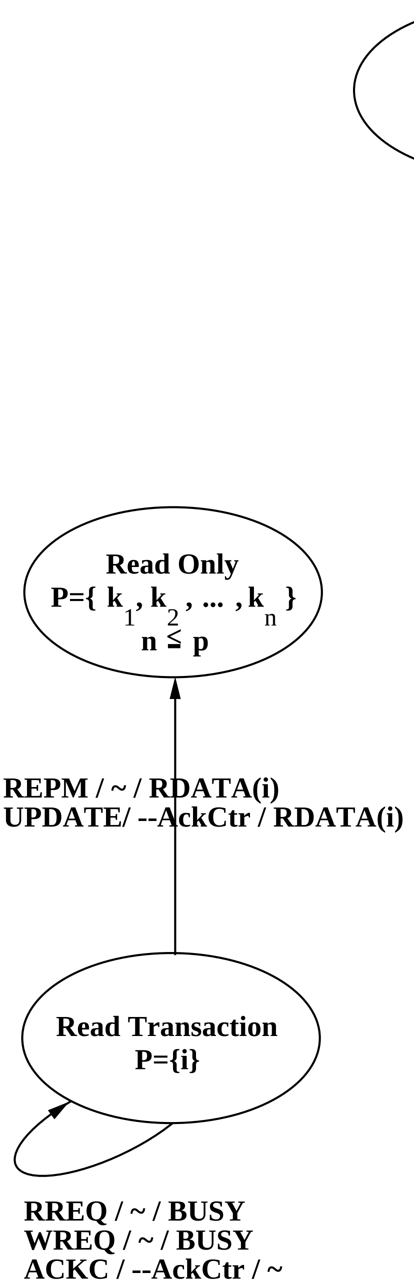


Figure B-12: Memory state transitions for limited directory: **Read Transaction** state.

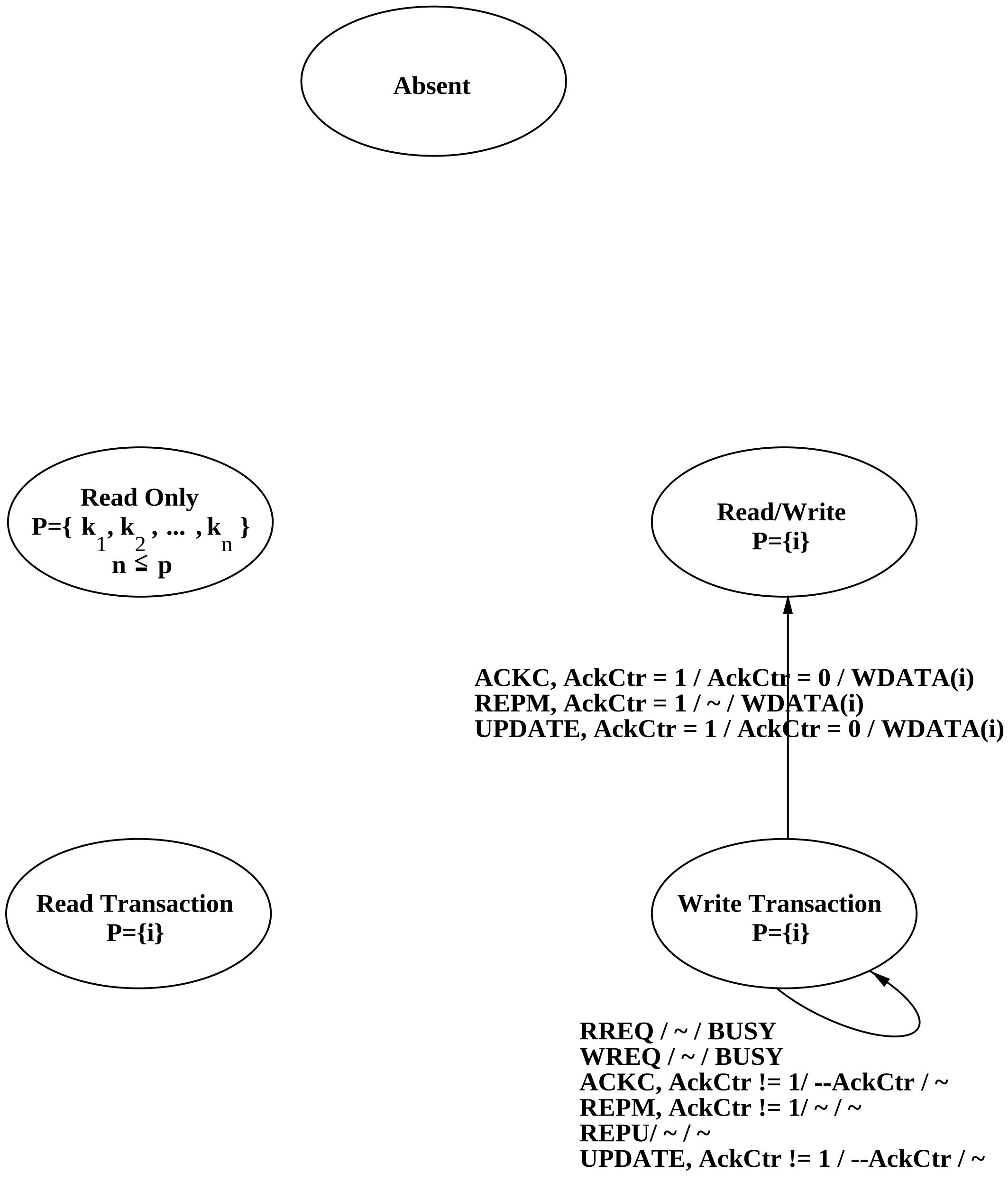


Figure B-13: Memory state transitions for limited directory: **Write Transaction** state.

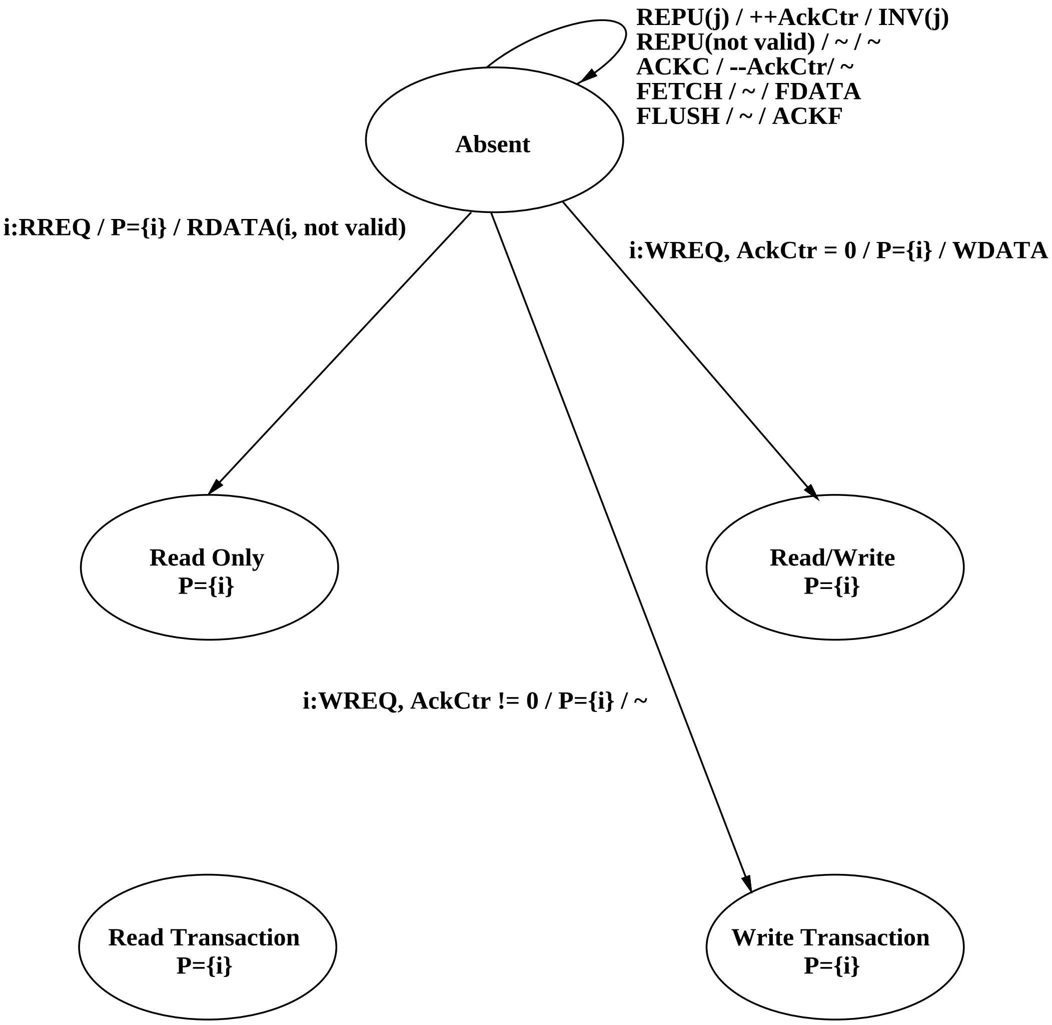


Figure B-14: Memory state transitions for chained directory: **Absent** state.

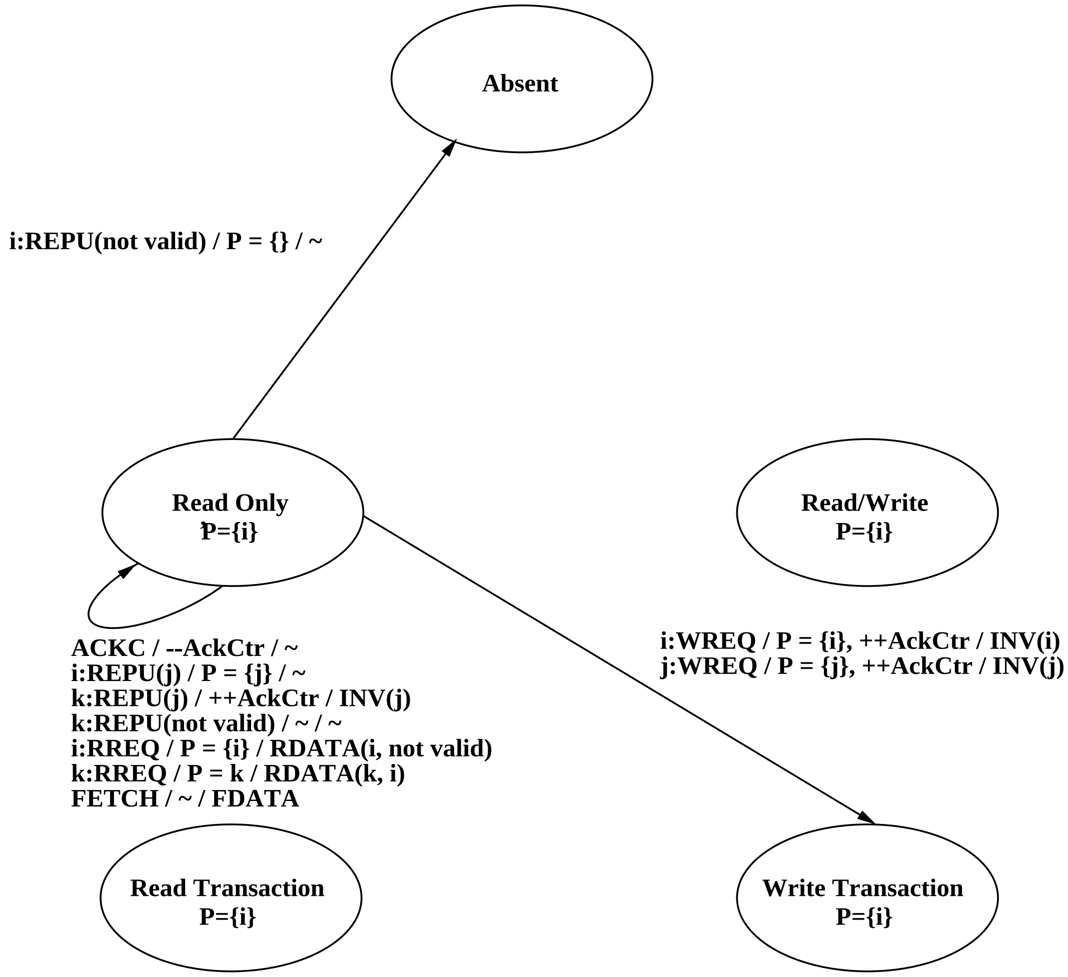


Figure B-15: Memory state transitions for chained directory: **Read Only** state.

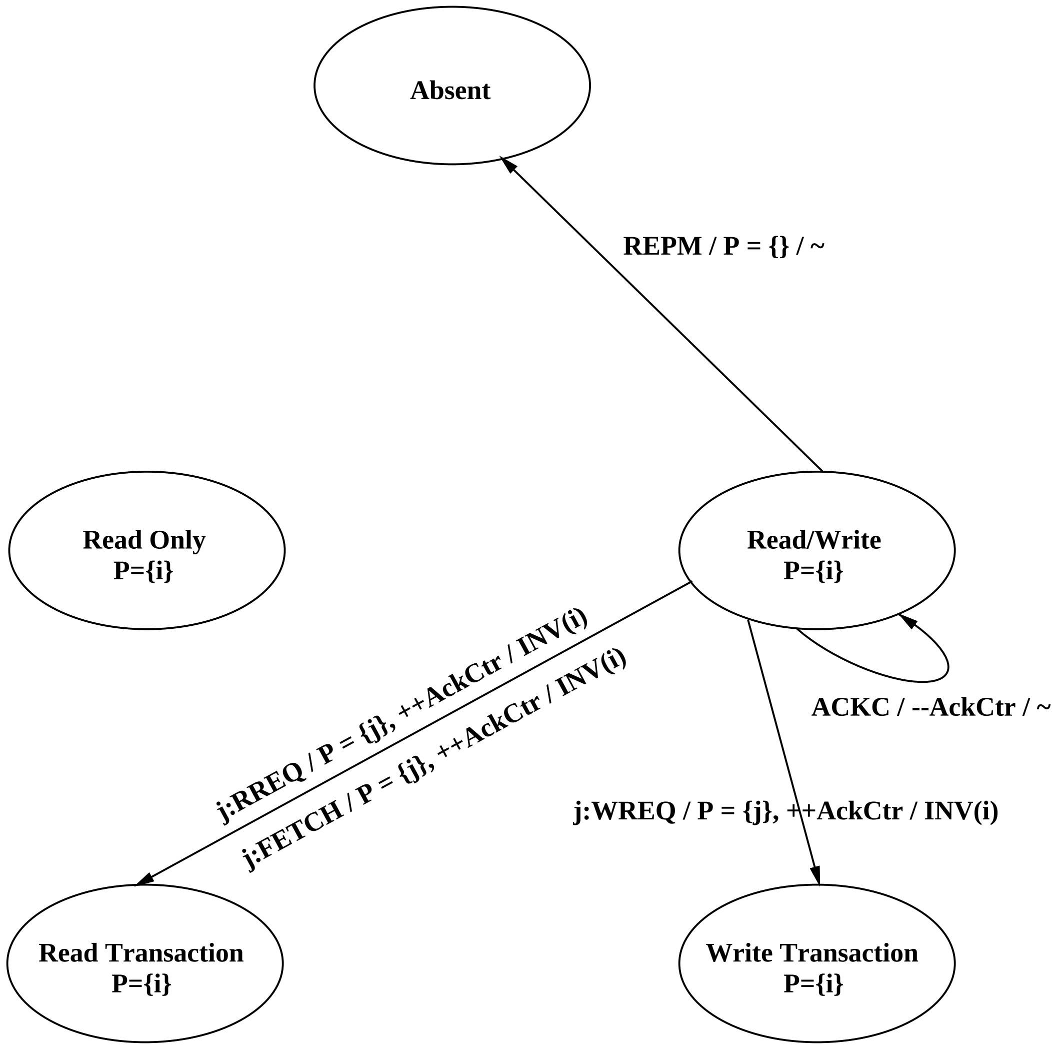


Figure B-16: Memory state transitions for chained directory: **Read/Write** state.

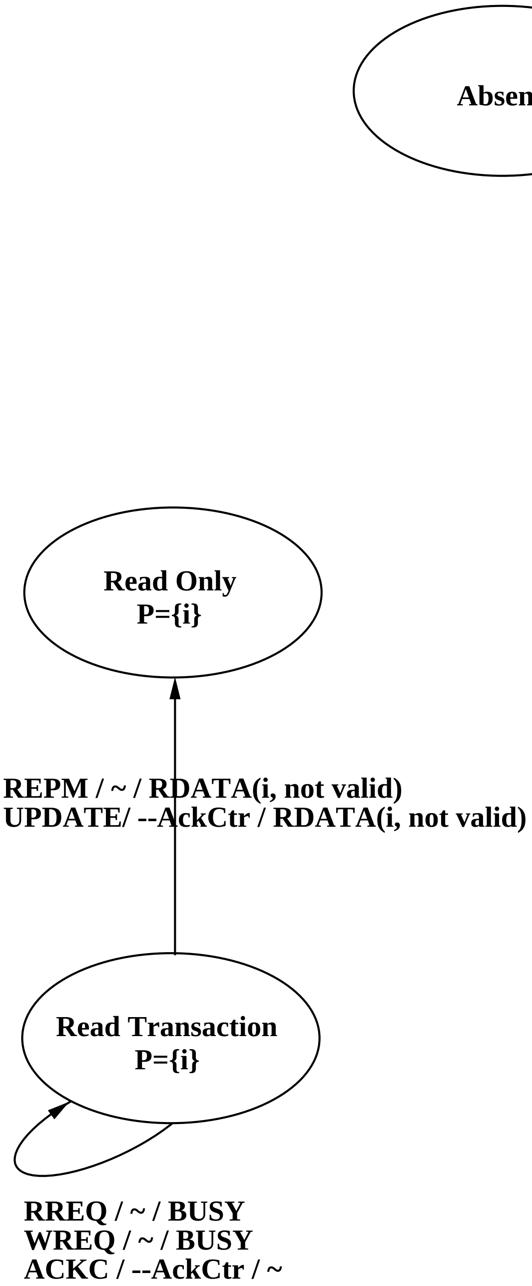


Figure B-17: Memory state transitions for chained directory: **Read Transaction** state.

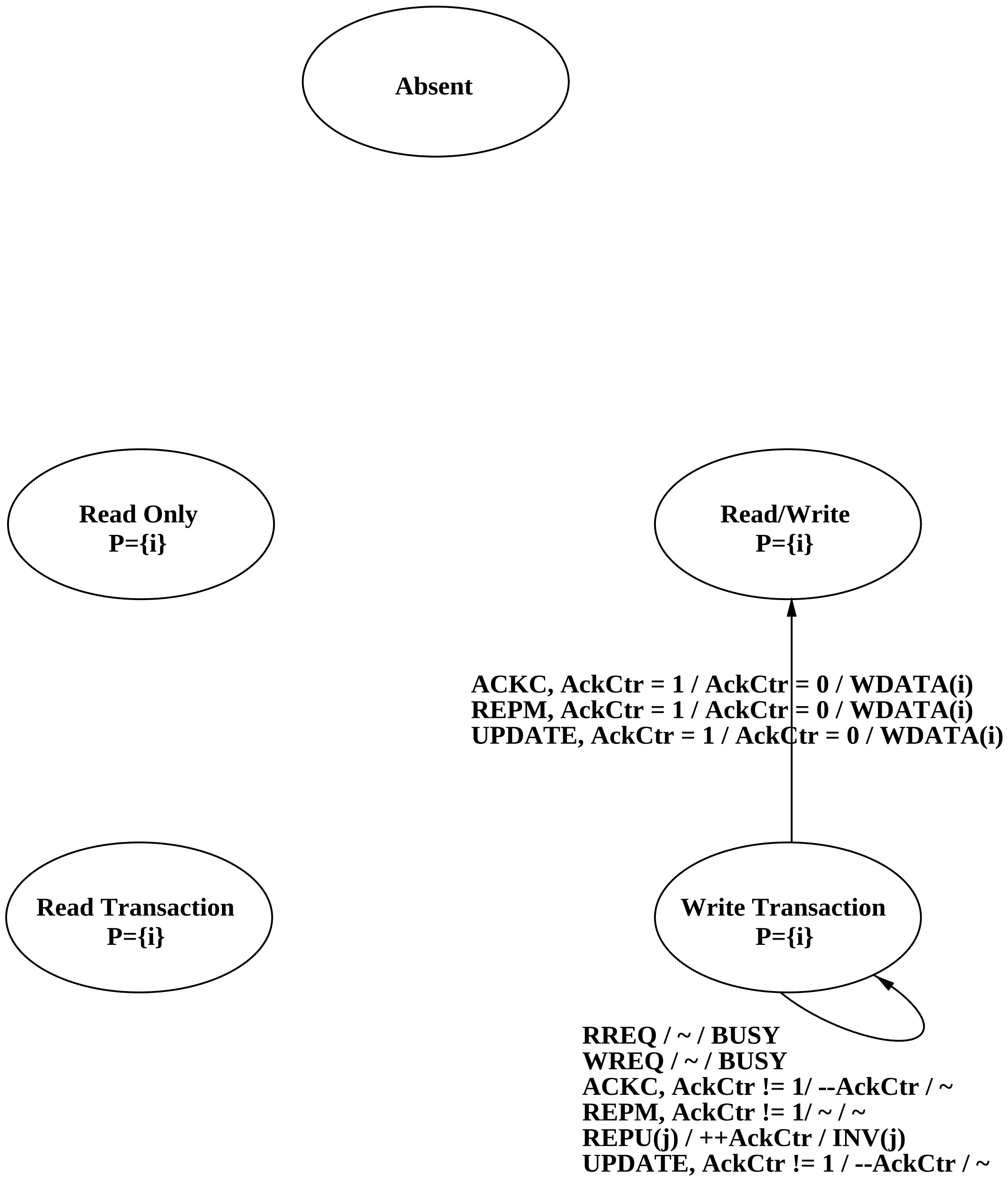


Figure B-18: Memory state transitions for chained directory: **Write Transaction** state.

While the protocol diagrams show transitions for both hardware and software coherence, the protocols may be separated to make them easier to understand. In practice, each of the protocols was implemented as a separate ASIM module. However, the experimental implementation does not necessarily impact the final controller design!

B.2 Hardware Coherence Protocols

Two classes of scalable hardware coherence protocols are operational: a limited directory protocol and a singly-linked chain directory protocol. The goal of both hardware coherence protocols is to allow several processors to simultaneously read a block of data, but to permit only one processor to access a block of data while it is being modified.

B.2.1 Protocol States

The basic states of a cache block (listed in Table B.4) are the same for both the limited and the chained protocol. In addition to the basic state information, the chained protocol **Read Only** and **Read Only Network Wait** states are augmented by a directory pointer, which contains either a processor identifier or a chain termination symbol.

For both protocols, each cache block has an associated *tag*, which serves the same purpose as a uniprocessor cache tag: Any shared memory address may be decomposed into a cache block identifier and a cache tag. If the data in a cache block is valid, then the tag associated with the block specifies the address that is being cached. Every processor request and protocol message sent to a controller must contain a shared memory address, because the response to the request or protocol message depends on whether the address *matches* the tag associated with the cache block. On the cache state diagrams, transitions that are executed in the case of a tag match are printed in boldface type, while normal type is used to represent transitions that are executed when an address does not match a cache tag. To emphasize the fact that the tag is used only when the cached data is valid, transitions from the **Invalid** and **Invalid Network Wait** states are printed in both boldface and normal type.

The **Network Wait** states ensure that a controller does not issue multiple requests for a cache block, while allowing access to a cache line with an outstanding request. For example, if a cache block is in the **Read/Write Network Wait** state, then at least one context is attempting to replace the current data in the cache (which is modifiable) with some other block of data that happens to map to the same cache line. The first time that the context attempts to access the data, the controller injects the appropriate request into the network and changes the cache block's state from **Read/Write** to **Read/Write Network Wait**. Subsequent accesses by the context do not cause the controller to flood the network with additional requests, since the block is in a **Network Wait** state. However, it may be the case that some other context is currently accessing the data in the block. This effect is called *replacement*

Name	Meaning
Invalid	Cache block may not be read or written.
Read Only	Cache block may be read, but not written.
Read/Write	Cache block may be read or written.
Invalid Network Wait	Invalid, network request is pending.
Read Only Network Wait	Read Only, network request is pending.
Read/Write Network Wait	Read/Write, network request is pending.

Table B.4: Cache states for hardware coherence.

Name	Meaning
Absent	No cache has a copy of the data.
Read Only	Some number of caches have read-only copies of the data.
Read/Write	Exactly one cache has a read-write copy of the data.
Read Transaction	Holding read request, update is in progress.
Write Transaction	Holding write request, invalidation is in progress.

Table B.5: Directory states for hardware coherence.

thrashing. For a more detailed example, see Section 3.3.2.

On the shared memory side, each block of data has a state that is stored in its directory entry. Table B.5 lists the five basic memory block states, which are just names for different logical subsets of the possible states of a directory entry's pointers and state bits. These are the same states used in Figure 3-3, except that Figure 3-3 combines the **Absent** and **Read Only** states.

The state transition diagrams use an implementation-independent notation to model the set of directory pointers. That is, the protocol specification is independent of both the number of directory pointers, and how the pointers are actually implemented. The meaning of the set of pointers ($\mathbf{P}$) depends on the state of the memory block. In the **Absent** state, $\mathbf{P} = \emptyset$, because no cache has a copy of the data. In the **Read Only** state of the limited directory protocol, $\mathbf{P} = \{k_1, k_2, \ldots, k_n\}$, where $n \leq p$, and p is the maximum number of pointers. Note that the only difference between the **Absent** and the **Read Only** states is in the size of $\mathbf{P}$. In ASIM, the limited protocol can be configured with directory entries that contain any number of pointers (p). If p is equal to the number of processors in the system, then the limited protocol becomes a full-map protocol. The **Read Only** state of the chained protocol uses only one directory pointer ($\mathbf{P} = \{i\}$), and the rest of the pointers are distributed to the caches with the data stored in the memory block. The **Read/Write**, **Read Transaction**, and **Write Transaction** states in both the chained and limited protocols use only one directory pointer ($\mathbf{P} = \{i\}$). In the **Read/Write** state, the pointer identifies the cache that has permission to write the data block. In the **Transaction** states, the pointer indicates the cache that is waiting to receive a response from the memory module.

Type	Symbol	Name	Data?	Pointer?
Cache to Memory	**RREQ**	Read Request		
	WREQ	Write Request		
	MREQ	Modify Request		
	REPU	Replace Unmodified		$\surd$
	REPM	Replace Modified	$\surd$	
	UPDATE	Update	$\surd$	
	ACKC	Invalidate Acknowledge		
Cache to Cache	**INV**	Chained Invalidation		
Memory to Cache	**RDATA**	Read Data	$\surd$	$\surd$
	WDATA	Write Data	$\surd$	
	MODG	Modify Granted		
	INV	Invalidate		
	BUSY	Busy Signal		

Table B.6: Protocol messages for hardware coherence.

Name	Meaning
Invalid	Cache block may not be read or written.
Valid	Data is valid for software coherent accesses.
Dirty	Data is valid and has been modified.
Invalid Network Wait	Invalid, network request is pending.
Valid, Network Wait	Valid, network request is pending.
Dirty, Network Wait	Dirty, network request is pending.

Table B.7: Cache states for software coherence.

B.2.2 Protocol Messages

The messages that are used by the hardware coherence protocols to keep the cache and the memory states consistent are listed in Table B.6. The **Data?** column indicates the four messages that contain the data of the shared memory block, and the **Pointer?** column indicates the two messages that carry a directory pointer for the chained protocol. See Section 3.4.1 for a description of the functions of each message.

B.3 Software Coherence Protocol

ASIM's software coherence protocol implements the processor access types specified in the bottom half of Table B.1. The protocol consists of six cache states (see Table B.7) and four messages (see Table B.8). Since this coherence protocol only provides mechanisms for allowing the software to ensure sequential consistency, no directory states are necessary for this protocol.

The **ReadSoft** and **WriteSoft** access types emulate the load and store functions on a uniprocessor write back cache. When a cache block is **Valid** or **Dirty**, it may be read or written. Writing to a **Valid** location using **WriteSoft** causes the block to

Type	Symbol	Name	Data?
Cache to Memory	**FETCH**	Fetch Data Request	
	FLUSH	Flush Data	$\surd$
Memory to Cache	**FDATA**	Fetch Data Response	$\surd$
	FACK	Flush Acknowledgment	

Table B.8: Protocol messages for software coherence.

become **Dirty**. Either a **ReadSoft** or a **WriteSoft** to an **Invalid** location causes a miss in the cache. Upon a cache miss, the controller transmits a message to **FETCH** the data into the cache. The **FDATA** response contains the data for the cache line.

The **FlushSoft** and **Fence** directives can be by the software to enforce sequential consistency by forcing blocks to be removed, or *flushed*, from the cache. Studies such as [40] propose methods for using a fence mechanism to guarantee correct execution of parallel programs. If a cache block is **Valid** when it is flushed or replaced by another block, then its cache line may be invalidated immediately. Otherwise, the **Dirty** block must be written back to memory using the **FLUSH** message. The **Fence** operation is used to ensure that all previous flush and replace operations have been completed. A controller increments its *fence counter* for every **FLUSH** that it injects into the network, and decrements the counter for every **FACK** that it receives from the network. If a control thread issues a **Fence**, then it is not permitted to continue until the fence counter reaches zero.

B.4 Interaction between Hardware and Software Coherence

Although it is useful to study the hardware and software coherence protocols independently, the protocol implemented in ASIM is actually a combination of both types of protocol. Not only does the composite protocol allow each application to select its own flavor of coherence, but it enables the programmer to get the best of both worlds. For example, a typical program sequence could perform the following accesses:

1. Use hardware coherent **Read** and **Write** operations on a synchronization variable to enter a critical section of code.

2. Use software coherent **ReadSoft** and **WriteSoft** operations to read a data structure, do some calculations, and write the data structure back to memory.

3. **Flush** all of the cache lines occupied by the data structure in the last step.

4. **Fence** to make sure that the data structure is no longer cached, and to make sure that all changes made by the software coherent **Write** operations are written back to shared memory.

5. Use hardware coherent **Read** and **Write** operations on a synchronization variable to exit the critical section.

This program sequence uses the hardware coherent protocol to synchronize processors, but relies on software coherence (enforced by critical sections) to guarantee the correctness of the data structure. The combination of the **Fence** operation and hardware coherent synchronization may be used to tolerate memory access latency and to increase processor locality.

The above example uses hardware coherence for certain variables and software coherence for other variables. With a composite protocol, it is also possible to construct data types by using both coherence types to access a single location. For example, the ASIM programming environment implements write-once data by first performing a hardware coherent **Write** access, and then using software coherent **ReadSoft** accesses. The protocol guarantees that on the first **ReadSoft** to a location, the processor will receive the latest-and-greatest version of the data. However, the software must make sure that the data is still valid for subsequent **ReadSoft** accesses by assuring that the data is actually written only once.

Bibliography

[1] Sarita V. Adve and Mark D. Hill. Weak Ordering - A New Definition. In *Proceedings 17th Annual International Symposium on Computer Architecture*, June 1990.

[2] Yehuda Afek, Geoffrey Brown, and Michael Merritt. A Lazy Cache Algorithm. In *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, ACM, New York, June 1989.

[3] Anant Agarwal. Overview of the Alewife Project. July 1990. Alewife Systems Memo #10.

[4] Anant Agarwal and Anoop Gupta. *Temporal, Processor, and Spatial Locality in Multiprocessor Memory References*, chapter 8, pages 271–295. Plenum Press, 1989. Stu Tewksbury Ed. Also appears as MIT VLSI Memo, 1989.

[5] Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing. In *Proceedings 17th Annual International Symposium on Computer Architecture*, June 1990.

[6] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. An Evaluation of Directory Schemes for Cache Coherence. In *Proceedings of the 15th International Symposium on Computer Architecture*, IEEE, New York, June 1988.

[7] James K. Archibald. *The Cache Coherence Problem in Shared-Memory Multiprocessors*. PhD thesis, University of Washington, Department of Computer Science, 1988.

[8] Michael C. Browne, Edmund M. Clarke, David L. Dill, and Bud Mishra. Automatic Verification of Sequential Circuits Using Temporal Logic. *IEEE Transactions on Computers*, C-35(12), December 1986.

[9] Lucien M. Censier and Paul Feautrier. A New Solution to Coherence Problems in Multicache Systems. *IEEE Transactions on Computers*, C-27(12):1112–1118, December 1978.

[10] David Chaiken. Personal communication with David Ford of Encore Computer Corporation and Ken McMillan of CMU.

[11] David Chaiken, Craig Fields, Kiyoshi Kurihara, and Anant Agarwal. Directory-Based Cache-Coherence in Large-Scale Multiprocessors. *IEEE Computer*, June 1990.

[12] David Chaiken, John Kubiatowicz, and Anant Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. August 1990. Submitted for publication.

[13] Mathews Cherian. *A Study of Backoff Barrier Synchronization in Shared-Memory Multiprocessors.* Technical Report, S.M. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1989.

[14] David R. Cheriton, Gert A. Slavenberg, and Patrick D. Boyle. Software-Controlled Caches in the VMP Multiprocessor. In *Proceedings of the 13th Annual Symposium on Computer Architecture*, pages 367–374, IEEE, New York, June 1986.

[15] Edmund M. Clarke, Soumitra Bose, Michael C. Browne, and Orna Grumberg. *The Design and Verification of Finite State Hardware Controllers.* Technical Report CMU-CS-87-145, Computer Science Dept., Carnegie Mellon University, Pittsburgh, PA, July 1987.

[16] William W. Collier. *Reasoning about Parallel Architectures.* (under review by) Prentice-Hall, Inc., 1990.

[17] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* The MIT Press, 1990.

[18] Michel Dubois and Faye A. Briggs. Effects of Cache Coherence in Multiprocessors. In *Proceedings of the 9th International Symposium on Computer Architecture*, pages 299–308, IEEE, New York, May 1982.

[19] Michel Dubois, Christoph Scheurich, and Faye A. Briggs. Synchronization, coherence, and event ordering in multiprocessors. *IEEE Computer*, 9–21, February 1988.

[20] S. J. Eggers and R. H. Katz. A Characterization of Sharing in Parallel Programs and Its Application to Coherency Protocol Evaluation. In *Proceedings of the 15th International Symposium on Computer Architecture*, IEEE, New York, June 1988.

[21] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings 17th Annual International Symposium on Computer Architecture*, June 1990.

[22] James R. Goodman. Using Cache Memory to Reduce Processor-Memory Traffic. In *Proceedings of the 10th Annual Symposium on Computer Architecture*, pages 124–131, IEEE, New York, June 1983.

[23] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer – Designing a MIMD Shared-Memory Parallel Machine. *IEEE Transactions on Computers*, C-32(2):175–189, February 1983.

[24] David V. James, Anthony T. Laundrie, Stein Gjessing, and Gurindar S. Sohi. Distributed-Directory Scheme: Scalable Coherent Interface. *IEEE Computer*, 74–77, June 1990.

[25] Harry F. Jordan. Performance Measurements on HEP - A Pipelined MIMD Computer. In *Proceedings 10th Annual International Symposium on Computer Architecture*, IEEE, New York, June 1983.

[26] 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 International Symposium on Computer Architecture*, pages 276–283, IEEE, New York, June 1985.

[27] R. H. Katz et al. *Memory Hierarchy Aspects of a Multiprocessor RISC: Cache and Bus Analyses*. Computer Science Division 85-221, University of California, Berkeley, January 1985.

[28] D. Kranz, R. Halstead, and E. Mohr. Mul-T: A High-Performance Parallel Lisp. In *Proceedings of SIGPLAN '89, Symposium on Programming Languages Design and Implemenation*, June 1989.

[29] David Kranz. *ORBIT: An Optimizing Compiler for Scheme*. PhD thesis, Yale University, February 1988. Technical Report YALEU/DCS/RR-632.

[30] Clyde P. Kruskal and Marc Snir. The Performance of Multistage Interconnection Networks for Multiprocessors. *IEEE Transactions on Computers*, C-32(12):1091–1098, December 1983.

[31] John Kubiatowicz. Special Mechanisms for Multi-Model Support. March 1990. Alewife Systems Memo #4.

[32] Kiyoshi Kurihara. *Performance Evaluation of Large-Scale Multiprocessors*. Technical Report, S.M. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, September 1990.

[33] Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9), September 1979.

[34] Nancy A. Lynch and Mark R. Tuttle. *An Introduction to Input/Output Automata*. Laboratory for Computer Science MIT/LCS/TM-373, M.I.T., November 1988.

[35] Eric Mohr, David A. Kranz, and Robert H. Halstead. Lazy task creation: a technique for increasing the granularity of parallel tasks. In *Proceedings of Symposium on Lisp and Functional Programming*, June 1990.

[36] Dan Nussbaum and Anant Agarwal. Scalability of Parallel Machines. July 1990. Alewife Systems Memo #9. Submitted for publication.

[37] Brian W. O'Krafka and A. Richard Newton. An Empirical Evaluation of Two Memory-Efficient Directory Methods. In *Proceedings 17th Annual International Symposium on Computer Architecture*, June 1990.

[38] Janak H. Patel. Analysis of Multiprocessors with Private Cache Memories. *IEEE Transactions on Computers*, C-31(4):296–304, April 1982.

[39] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, A. Norton, and J. Weiss. The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture. In *Proceedings ICPP*, pages 764–771, August 1985.

[40] Dennis Shasha and Marc Snir. *Efficient and Correct Execution of Parallel Programs that Share Memory.* Research Report 58037, Courant Institute, New York University and IBM T.J. Watson Research Center, T.J. Watson Research Center, POB 218, Yorktown Heights, NY 10598, July 1987.

[41] Alan Jay Smith. Cache Memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[42] C. K. Tang. Cache Design in the Tightly Coupled Multiprocessor System. In *AFIPS Conference Proceedings, National Computer Conference, NY, NY*, pages 749–753, June 1976.

[43] Ingmar Vuong-Adlerberg. On Cache consistency (Draft). March 1989. Unpublished work done in the Laboratory for Computer Science.

[44] Wolf-Dietrich Weber and Anoop Gupta. Analysis of Cache Invalidation Patterns in Multiprocessors. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, April 1989.

[45] Wei C. Yen, David W. L. Yen, and King-Sun Fu. Data Coherence Problem in a Multicache System. *IEEE Transactions on Computers*, C-34(1):56–65, January 1985.

[46] Pen-Chung Yew, Nian-Feng Tzeng, and Duncan H. Lawrie. Distributing hot-spot addressing in large-scale multiprocessors. *IEEE Transactions on Computers*, C-36(4):388–395, April 1987.