Archibald, James K.

THE CACHE COHERENCE PROBLEM IN SHARED-MEMORY MULTIPROCESSORS

University of Washington PH.D. 1987

The Cache Coherence Problem in Shared-Memory Multiprocessors

by

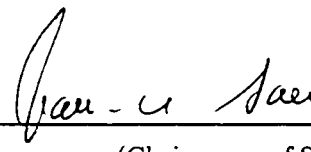
James K Archibald

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

University of Washington

1987

Approved by 

(Chairperson of Supervisory Committee)

Program Authorized
to Offer Degree Computer Science

Date 12/17/1986

Doctoral Dissertation

In presenting this dissertation in partial fulfillment of the requirements for the Doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that copying of this thesis is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to University Microfilms, 300 North Zeeb Road, Ann Arbor, Michigan 48106, to whom the author has granted "the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform."

Signature James K Archibald

Date December 18, 1986

University of Washington

Abstract

The Cache Coherence Problem in Shared-Memory Multiprocessors

by James K Archibald

Chairperson of the Supervisory Committee: Professor Jean-Loup Baer
Department of Computer Science

Shared-memory multiprocessors offer increased computational power and the programmability of the shared-memory model. However, sharing memory between processors leads to contention which delays memory accesses. Adding a cache memory for each processor reduces the average access time, but it creates the possibility of inconsistency among cached copies. The *cache coherence problem* is keeping all cached copies of the same memory location identical. This dissertation explores possible solutions to the cache coherence problem and identifies *cache coherence protocols*–solutions implemented entirely in hardware–as an attractive alternative.

Protocols for shared-bus systems are shown to be an interesting special case. Previously proposed shared-bus protocols are described using uniform terminology, and they are shown to divide into two categories: ***invalidation*** and ***distributed write***. In invalidation protocols all other cached copies must be invalidated before any copy can be changed; in distributed write protocols all copies must be updated each time a shared block is modified. In each category, a new protocol is presented with better performance than previous schemes, based on simulation results. The simulation model and parameters are described in detail.

Previous protocols for general interconnection networks are shown to contain flaws and to be costly to implement. A new class of protocols is presented that offers reduced implementation cost and expandability, while retaining a high level of performance, as illustrated by simulation results using a crossbar switch. All new protocols have been proven correct; one of the proofs is included.

Previous definitions of cache coherence are shown to be inadequate and a new definition is presented. Coherence is compared and contrasted with other levels of consistency, which are also identified. The consistency of shared-bus protocols is shown to be naturally stronger than that of non-bus protocols.

The first protocol of its kind is presented for a large hierarchical multiprocessor, using a bus-based protocol within each cluster and a general protocol in the network connecting the clusters to the shared main memory.

Table of Contents

List of Figures

Acknowledgements

I am grateful to Jean-Loup Baer for his insight and encouragement in guiding the completion of this work. His experience and wisdom have been instrumental at innumerable points in its progression. I would like to thank John Zahorjan and Ed Lazowska for their effort on the reading committee and for their comments and suggestions regarding the presentation of this dissertation and other papers. Finally, I would like to thank my wife Dixie for her constant support and unwavering patience.

The research for this dissertation was completed while the author was supported by NSF Grants MCS-8025616, MCS-8304534, and DCR-8503250.

Chapter 1

Introduction

1.1 Concurrency vs. Technology

Computer architects faced with the challenge of increasing the performance of a computer system have two alternatives: use a faster implementation technology, or introduce more concurrency in the operation of the system. While great technological breakthroughs have been made, it is unlikely that performance improvements as large as those sought for some compute-bound applications can result solely from improvements in the underlying technology. In addition, using the fastest available technology is generally very expensive.

Concurrency is a cost-effective alternative that can boost the performance of any system, independent of implementation technology, since it increases the amount of work that the computer can do in a given time interval. Concurrency can be obtained in many ways. Traditionally, systems of even moderate computing power have overlapped the completion of I/O operations with instruction execution by the CPU, allowing better utilization of system resources and increased system throughput.

Another common form of concurrency found in modern computers is *pipelining*–the overlapping of the various stages of execution of successive instructions. For example, it is possible to overlap the decoding of one instruction with the fetching of the next instruction. Overlap is possible at all stages of instruction execution: instruction address calculation, instruction fetch, decoding, operand address calculation, operand fetch, and execution [Bae80]. The performance of systems using this overlapping technique is dependent on the pipeline being kept full. This presents a

challenge in the cases of branches and instructions requiring operands that are being calculated by operations that are still in the pipe. Even if the pipeline remains full, the potential performance increase is limited by the number of stages in the pipeline, which limits the number of concurrent operations.

Another form of concurrency is the simultaneous operation of a particular instruction on a number of different data items by physically distinct processing elements. This is the type of parallelism or concurrency found in *array processors*. While it is an efficient technique for some applications using data in vector form, array processing is not practical for general applications because of the requirement that all processing elements perform exactly the same operation at the same time.

The most flexible form of concurrency is obtained by using physically distinct processing elements that each execute an independent instruction stream, exactly as processors in existing systems with a single processor. Each executing process may be a completely independent job, or processes may cooperate as different partitions of the same job that share information and are executed simultaneously. Since information sharing takes place, there must be some form of communication medium between the processing elements. The organization of this *interconnection network* between processor elements serves as the distinguishing feature in identifying and classifying different forms of computer systems utilizing this type of concurrency.

1.2 Interprocessor Connections

The organization of the interconnection network determines the overhead of interprocessor communication. When the overhead is quite high, the system is said to be *loosely-coupled*. If the overhead is very low, the system is *tightly-coupled*. An example of a loosely-coupled system organization is a distributed system consisting of self-contained computers communicating through computer networks. Since the interprocessor communication generally requires significant software overhead, such systems are classified as loosely-coupled *multicomputers*. Parallel applications requiring a great deal of computation with very little interprocessor communication would be appropriate candidates for such systems.

Applications with high interprocess communication requirements are best suited for tightly-coupled systems, where the communication overhead is much lower. The most efficient form of interprocess communication is through a shared memory, common to all processing elements. In such systems, usually referred to as *multiprocessors*, processes can communicate with each other simply by reading and writing shared variables in the common store.

Other multiprocessor or multicomputer system organizations are more difficult to classify, but they generally fall somewhere between the two extremes. It is possible, for example, for a distributed system to have an optimized message passing mechanism that greatly reduces the overhead of communication, making it more tightly-coupled than most distributed systems [Spe82]. Some 'multiprocessors' (such as the CHiP machine [Sny82] and the Cosmic Cube [Sei85]) do not share memory and communicate via messages through a mesh, grid, or nearest neighbor network, making them more loosely-coupled than shared-memory designs. However, these systems may also include hardware support for message passing that greatly reduces the overhead of remote references. A multiprocessor such as Cm* [SFD77] is also difficult to classify. In this experimental computer, all processors are allowed to address all of memory, but all memory is not equally distant from each processor. Local references may be serviced much more quickly than remote references, which must be routed through additional switches before they can be serviced. Remote data is requested and returned through a series of messages between controllers specially designed for the purpose, with substantially less overhead than software supported message passing, but more overhead than shared memory references in a machine with uniform memory access times.

Many of the terms used in the previous discussion have not yet found a consistent usage in published articles. For some, 'multiprocessor' is necessarily a configuration in which independent processors shared a global memory [Sat80]. Others choose to distinguish between 'multiprocessors' and 'shared memory multiprocessors', allowing the classification of CHiP-like machines as multiprocessors. Throughout this dissertation, the term 'multiprocessor' should be taken to mean 'tightly-coupled multiprocessor' or 'shared-memory multiprocessor'.

1.3 Tightly-Coupled Multiprocessors

Shared-memory multiprocessors offer the advantage of relatively efficient sharing of code and data through a common main memory. There are other important advantages of multiprocessors in general. The first is that they have the potential for increased reliability. For example, since there are several identical processors in the system, the failure of any single processor need not cause the entire system to fail. In the ideal case, the system should be able to dynamically reconfigure itself without the failed processor and continue operation. In practice, such reliability is difficult to obtain [JP80]. Another advantage of multiprocessors is that the sharing of resources between all executing job streams leads to a better utilization of those resources and increased system throughput [Sat80]. An important advantage claimed by multiprocessor designers is that both the design cost and the design time are reduced considerably (relative to a uniprocessor of comparable

power) due to the natural duplication of components [Wid79].

There are three noteworthy negative consequences of processors accessing a common store. The first and potentially most serious is that processors may now conflict with each other in accessing memory, regardless of how the processor-memory interconnection is organized. This interference increases the effective memory access time and decreases the effective bandwidth between a processor and main memory, already considered the main factor limiting performance in standard architectures. A second consequence is that the size of main memory must be very large to accomodate all the code and data for all processors. Since larger memories typically have slower access times, the speed of main memory is likely to be slower (relative to processor speed) than in comparable uniprocessor systems. A third consequence is that the interconnection network is likely to have a slower switching time than a corresponding uniprocessor-memory switch, since it must connect a number of processor elements with a number of memory modules. The increased size implies greater switch complexity and increased switching delay. These three factors combine to create a serious problem of accessing memory. Perhaps the greatest challenge to designers of multiprocessors is to provide an efficient memory organization that will not seriously limit the processor performance.

A common approach taken in high performance uniprocessors to increase the effective memory speed is the use of a *cache memory* in the design. A cache is a small, high speed memory used to store copies of recently referenced locations in main memory. Due to the locality exhibited by most computer programs, those locations recently accessed are likely to be accessed again in the future. When such references occur, the cache memory is then able to service the requests much more quickly than main memory. Writes to cached copies are permitted with the stipulation that the change eventually be reflected in main memory. In the *write-through* update policy, each write immediately updates both the cache and main memory. Using *write-back* (also called copy-back), all memory updates are delayed until the local copy is replaced in the cache.

In light of the difficulty of providing processors sufficient memory bandwidth, it should hardly be surprising that the use of caches in a high performance shared-memory multiprocessor is considered essential [SA86]. There are two general types of caches that may be used. The caches may be *shared* between two or more processors, or they may be *private*–accessed only by a single processor. The use of shared caches is quite limited, since processors may interfere with each other's accesses to the cache, and since the bandwidth of the cache must be very high to support the memory references of multiple processors. The second limitation is serious–according to Smith [Smi82, Smi85b], shared caches are normally unable to provide sufficient bandwidth for more than two processors. Therefore, private caches are the preferred alternative. If most of a processor's references can

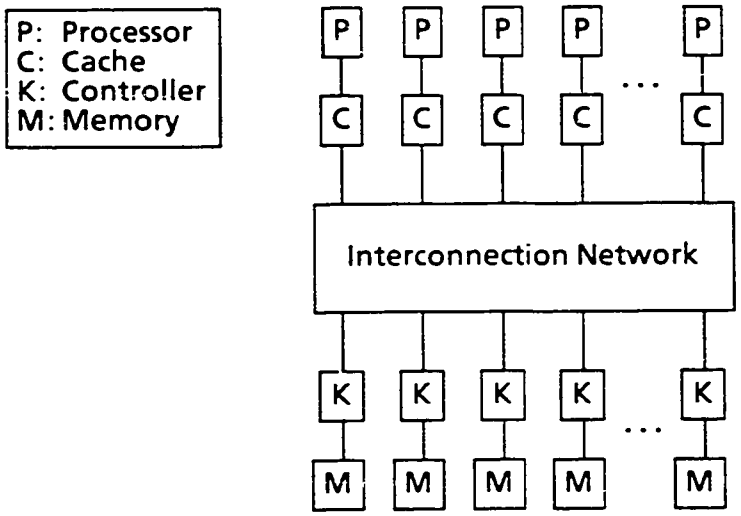


Figure 1.1: Multiprocessor organization

be serviced by its local cache (without traversing the interconnection network), the traffic on the interconnection network is reduced greatly, decreasing memory interference as well. Since the speed of a small memory can be matched quite closely with the processor speed, accesses to the cache can be completed much more quickly than accesses to main memory.

In this dissertation, we are concerned with shared-memory multiprocessors (as shown in Figure 1.3), consisting of a number of main memory modules connected to a number of processors, each with a private cache.

1.3.1 Cache Coherence

The inclusion of multiple caches (one per processor) in a multiprocessor design is not without its challenges. If each cache is allowed to have a copy of any given memory location, there can be as many copies in the system as there are processors. It is imperative that all copies remain identical when any copy of that memory word is modified. The task of keeping all cached copies consistent with each other is called the *cache coherence problem*. Intuitively, a system of caches is *coherent* if all copies of each memory location appear identical. An alternate definition, given by Censier and Feautrier [CP78], is that a system is coherent if every read of a memory location returns the value most recently written to that location. Although these definitions are useful in understanding cache coherence solutions, they are inadequate for an important class of protocols. A more precise

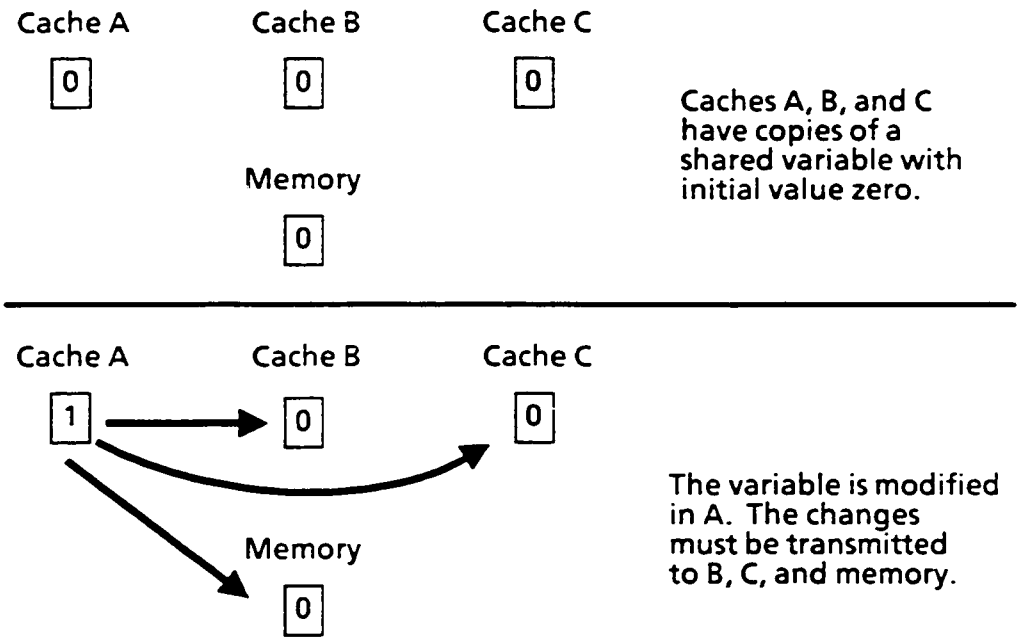


Figure 1.2: Inconsistency from shared data

definition of coherence will be given in Chapter 6.

In general, there are two possible causes of inconsistency in a multiprocessor with private caches. The most obvious is the result of several processors accessing writable shared data. For example, in Figure 1.2, caches A, B, and C have copies of a shared variable which has an initial value of zero. (The value of the main memory copy of the variable is also displayed.) If cache A (or any other cache) modifies the variable, those changes must also be observed by the other caches as well as main memory. The update of main memory may be delayed if a write-back cache is used, but even if a write-through cache is used, the changes must also be reflected in all otther cached copies. Therefore simply using write-through is not sufficient.

The second and more subtle cause of inconsistency is task migration. Assume that a process is executing on processor A and that it has loaded a copy of a local variable into cache A. At some point, the process migrates to processor B and modifies the local variable in cache B, as shown in Figure 1.3. If the state of the block in cache A is not affected, it is possible for the process to migrate back to processor A and find an obsolete value of the variable.

The purpose of this dissertation is to explore the cache coherence problem and to improve on the efficiency or economy of existing solutions. Chapter 2 begins with an overview of pos-

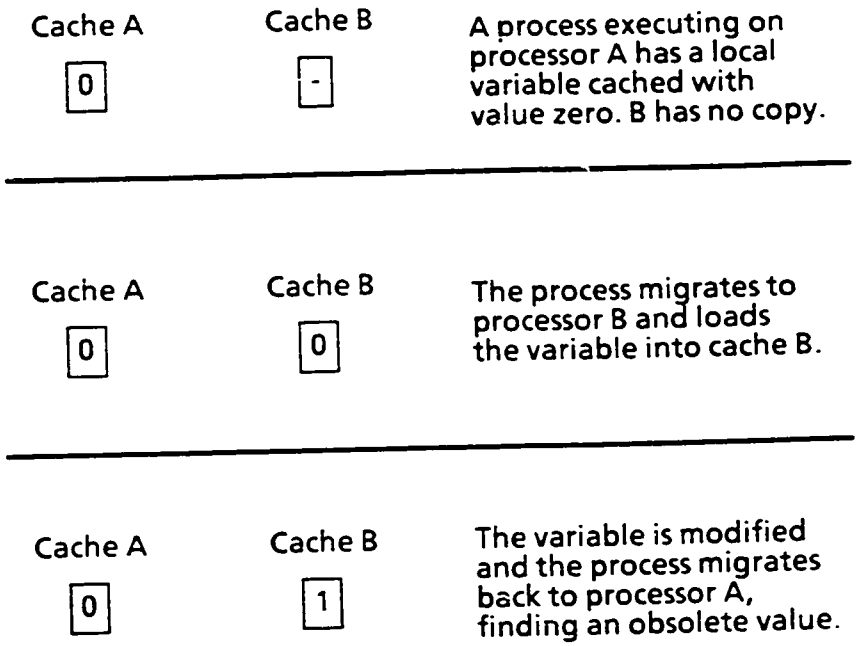


Figure 1.3: Inconsistency from task migration

sible approaches to the problem. (The most promising of these are explored in detail in later chapters.) Chapter 3 introduces a simulation model which was developed to evaluate the relative performance of alternative solutions. Chapters 4 and 5 discuss in detail previous proposals for a number of interesting classes of solutions. In addition, Chapters 4 and 5 also contain proposals for new solutions and a discussion of simulation results comparing the performance of the different approaches. Chapter 6 presents a precise definition of coherence and discusses its relationship to other forms of consistency. Chapter 7 presents a solution extending the approaches of Chapters 4 and 5 for a much larger, hierarchical multiprocessor organization. Chapter 8 includes a correctness proof for some of the new protocols. Conclusions and a summary are included in Chapter 9.

Chapter 2

General Approaches to the Cache Coherence Problem

Cache coherence solutions fall into one of two general categories. The first type are those solutions implemented entirely in hardware. Such solutions require no software support of any kind and are completely transparent to the software at all levels. The second category of solutions includes all those that require both hardware and software support. We begin with a general overview of hardware approaches and continue with an overview of software schemes. Hardware solutions will be examined in detail in subsequent chapters.

2.1 Hardware Solutions

Throughout the following discussion, it is helpful to distinguish between *global actions* that require the use of a resource shared between processor elements, and *local actions* that require the use of resources associated with a single processor. The addition of a cache to the memory hierarchy is useful because it reduces the number of global actions necessary in servicing processor memory requests. Cache hits may be serviced locally without the overhead of accessing shared memory, while cache misses and resulting block replacements require the use of global resources to service.

The overhead of all cache coherence solutions consists of extra local and global actions besides those required for normal cache operation. Local actions are usually quite simple and involve changing some information associated with the block in the cache directory. However, global

actions require signals to be sent through the interconnection network and therefore contribute to the traffic and contention in the system. The efficiency of a cache coherence solution depends greatly on the minimization of global actions.

In general, hardware schemes treat each block based entirely on its current state. They can therefore be classified as *dynamic* cache coherence solutions, in contrast with the *static* approach taken by the software schemes examined later in this chapter.

2.1.1 Write Broadcast

The simplest hardware approach to keep all cached copies identical is for each cache to inform all other caches each time a write occurs. Each cache that receives this *write broadcast* signal attempts to match the block address with its contents. If a successful match occurs, the cache must invalidate the local copy as it is no longer up-to-date. Although they differ widely in implementation cost and performance, all hardware cache coherence solutions are based on variations of this simple technique.

Consider two possible implementations of write broadcast, as illustrated in Figure 2.1. The simplest is to use a write-through memory update policy (in which each cache completes a write to main memory on each processor write) and to have each processor monitor the write-through traffic of the other processors. This assumes that the caches share a bus connecting them to main memory. The second implementation would work even if the memory switch is not a shared bus. In this case, all caches are connected via an auxiliary data path used exclusively to send invalidation signals for blocks that are about to be modified. Each cache constantly monitors the path and invalidates the local copies in the event of a match.

Note that there are two components of the overhead of these write broadcast schemes. The first is a global action for each write, necessary to inform all other caches of each modification. Because the write ratio usually ranges from 10 to 30 percent of all memory references, the traffic associated with write broadcasts can quickly saturate even high performance buses. The second component is matching overhead at each cache. Without the addition of special hardware, the cache must expend a cycle each time a match is attempted, which happens each time a broadcast signal is received. With as few as three or four caches in the system, it is possible that each cache spends more time servicing invalidation signals than it spends servicing processor requests.

There are two optimizations that will reduce the matching overhead at each cache, although they do not affect the invalidation traffic. The first, patented in 1979 [BLPS], is the addition of a *filter memory* associated with each cache. The filter memory filters out repeated invalidation

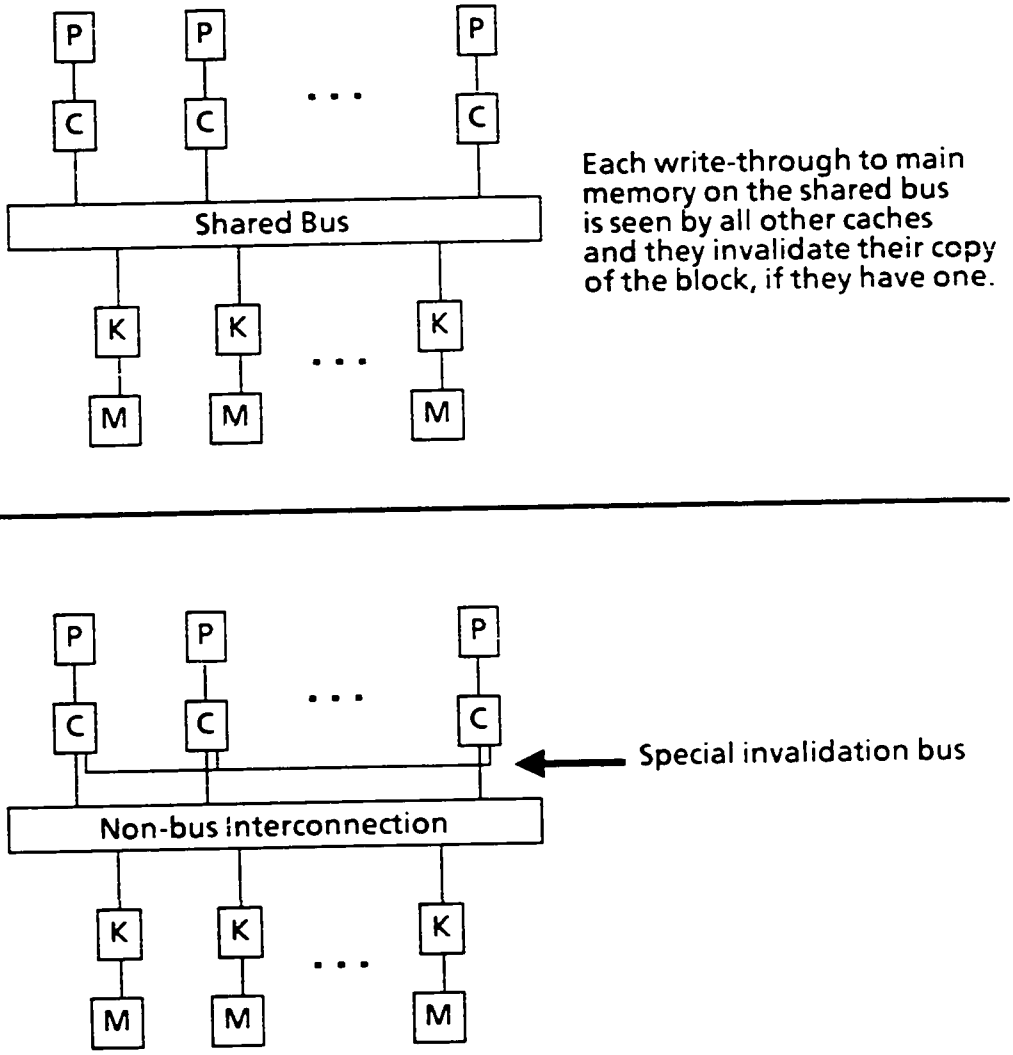


Figure 2.1: Implementations of write broadcast

signals for the same block, thus reducing the matching overhead. Even more effective is the dual directory cache controller approach. This involves the addition of dedicated bus-watching logic with a separate copy of the local cache directory. This specialized unit attempts to match on all the invalidation traffic, only interfering with the normal operation of the cache in the case of a successful match. This optimization is not limited to the write-broadcast technique and may be added to any of the hardware implemented techniques that are described here.

The obvious disadvantage of a global action taking place for each write issued by each processor is that little of the overhead is actually necessary to maintain consistency. In the next section, we examine techniques that reduce the number of broadcast signals that must be sent. Although write broadcast has been utilized in uniprocessors such as the VAX 11/780 to keep the cache contents consistent with ongoing I/O operations, it is not practical for multiprocessors with even a moderate number of processors and has never been implemented in systems with more than two caches [CP78]. Examples of systems using this approach include the UNIVAC 1100/80, using two caches with each cache shared between two processors, and the IBM 370/168 and 3033–both dual processor systems.

2.1.2 Directory Methods

Directory methods attain higher levels of performance by reducing the number of broadcasts through a combination of added global and local state information. Local state information is typically encoded in the tag bits associated with each block in the local cache directory. The global state information is maintained in logical tables or directories.

On the basis of the global and local state information it is determined whether or not an invalidation signal must be sent each time a block is modified. If the state information guarantees that no other cached copy exists, no invalidation is required. Schemes using a directory approach differ in the amount of information contained in the local and global states, as well as in the specified transitions from one state to another. Since the global and local states must remain consistent with each other, each directory scheme specifies the communication that must take place between the local cache controllers that maintain the local states and whatever units maintain the global states. Because this communication takes the form of a protocol, solutions using directories are often referred to as *cache coherence protocols*.

To illustrate the function of a protocol utilizing a directory, consider the general operation of a simple invalidation based protocol–one which, like most hardware based schemes, permits multiple readers but only a single writer at a time. Assume that the global state contains information about

whether the block is modified or not, and which caches have a copy of the block, if any. The local state of each block in each cache indicates whether the block is modified or not, with respect to main memory. The operation of the protocol can be described by considering each of the following cases:

- **Read hit.** In this case, no action is required of the protocol. The data is simply returned to the processor.
- **Read miss.** The global state must be queried to determine if another cache has a modified copy. If so, it must supply it and update the copy in main memory. If not, the block is loaded from main memory. In either case, the global state is updated to indicate that another cache has obtained a copy and that no cache has a modified copy. The local state is set to indicate that the block is unmodified in the requesting cache.
- **Write hit.** The modification may proceed only if the cache has write permission (indicated by having a modified copy of the block). If this is not the case, the global state must be queried, and all other caches with copies of the block (if any) must be sent invalidation signals. The global state is updated to show that only one cache has a copy and that the copy is modified. The cache may then complete the write and update the local state to indicate that the block has been written.
- **Write miss.** The global state is queried. If a cache has a modified copy, it must supply it. All caches with copies must be sent invalidation signals so that the cache with the miss may load an exclusive copy. The global state is updated to reflect this change. The block is loaded and the write is allowed to proceed with the local state indicating that the block has been modified.

Cache coherence protocols can be classified according to the way in which the directories maintaining the global state are organized. As shown in Figure 2.2, there are two general types of organizations. An entry may be maintained for each cache, listing the numbers of the blocks present in that cache, or an entry may be associated with each block in main memory, giving the global state of that block at any given time. Directory methods can also be classified according to the interconnection network used in the multiprocessor. In systems where all caches share the same datapath to main memory, it is possible for each cache to observe the memory transactions of all other caches in the system. This *snooping* feature allows the support for the coherence mechanism to be distributed across the caches in the system. If the caches are not able to observe

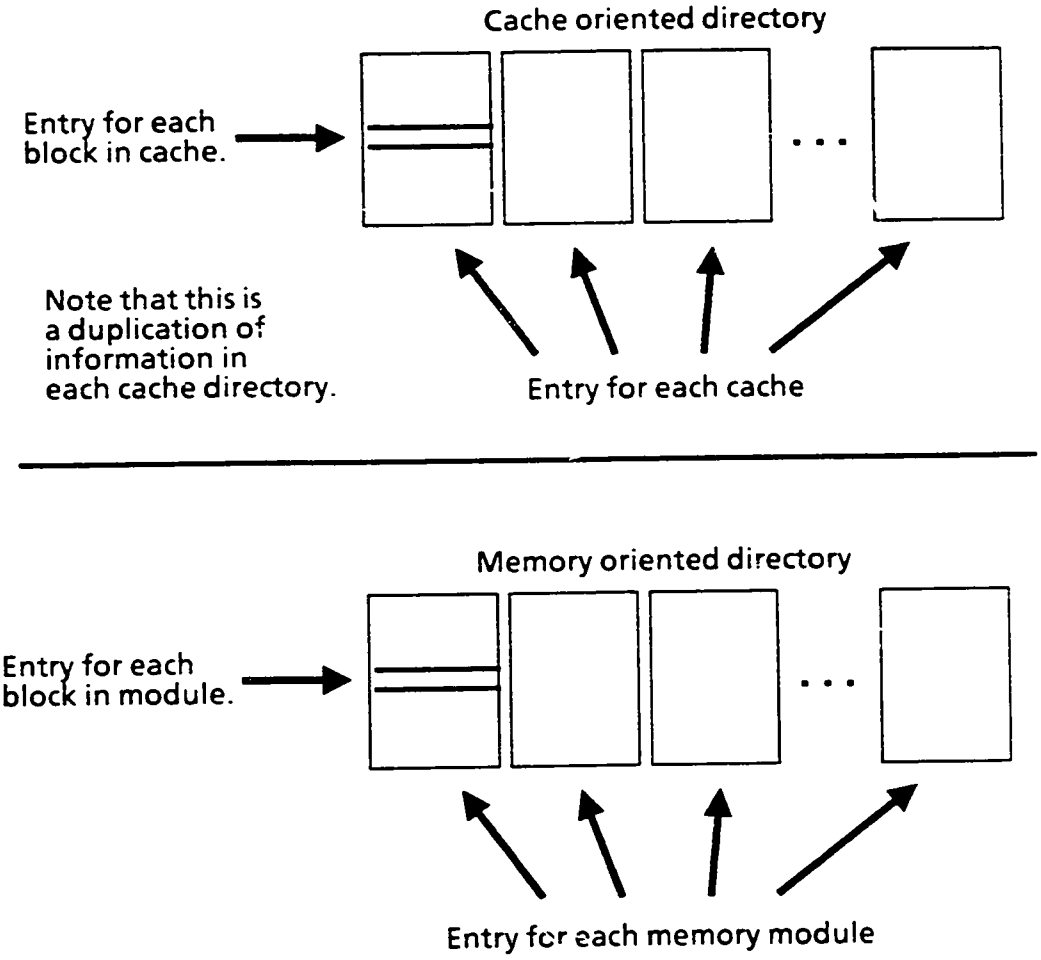


Figure 2.2: Logical directory organizations

each other's actions, as is the case in any network other than a shared bus, the support for the coherence mechanism must be supplied at a different level.

These classifications can be combined to create four categories of directory solutions, namely:

1. Cache oriented directory for general networks.
2. Memory oriented directory for general networks.
3. Cache oriented directory for shared bus.
4. Memory oriented directory for shared bus.

Cache Oriented Directory with General Network. Only one scheme has been published that fits this category–the first published coherence protocol proposed [Tan76]. Reportedly implemented in the IBM 30xx series, this scheme uses a centralized directory (accessible from the central memory controller) containing a copy of each cache directory maintained in parallel. The global state of the block can be determined by examining the contents of all the cache directories. In this way it can be determined whether or not any caches have copies of the block and if any cache has a modified copy. The local state is encoded in a single bit, indicating whether or not the block may be shared, which in this case is identical to the usual dirty bit used in normal write-back caches.

There are several problems with this technique that make it an unattractive alternative as soon as the number of processor-cache pairs is large. First, there is a single centralized directory that must be queried for the global state of the block on each cache miss, and on each request by a cache to modify a previously unwritten block. Since these queries occur frequently, they must be very efficient or the global state lookup can easily become a bottleneck to system performance. In particular, this would seem to imply that the cache directories would have to be searched in parallel, requiring a complicated controller design. In addition, the directory must be kept consistent with the directory of each cache. Thus, each block load, replacement, and local state change must be reflected in the central directory. A third consideration is that the technique is not expandible, since the addition of processor-cache pairs would necessarily add sections to the global directory. Such an addition would almost certainly be impossible unless provisions had been made in the design of the directory and controller. Because of these limitations, the use of a centralized directory in a multiprocessor with a general interconnection network appears to be restricted to systems with a small number of caches.

One possible improvement to this scheme would be to distribute the directory across all memory controllers. Each memory controller would record, for each cache, which copies of blocks from the local memory module were in that cache. The directories would still require a parallel search to be efficient, but they would no longer present a centralized bottleneck to the system. Although this would be more efficient than the basic scheme, it appears to be a less promising technique than the memory oriented directory approaches discussed in the next section, although they require more space for the global table.

Memory Oriented Directory with General Network. There have been at least two schemes proposed that are in this second category. All of these schemes associate a global state with each block in main memory, giving information about the shared status of that block at each instant

in time. Unlike the centralized directory of the previous protocol, the directories used in these schemes may be distributed over the physical memory modules. Each memory controller needs to access the global states only of those blocks contained in that module. To a large extent the accessing and updating of the bits encoding the global state can be completed in parallel with the access or updating of main memory.

Because of its more efficient operation, the memory oriented approach can be extended to many more processor-cache pairs than the centralized cache oriented scheme, and is therefore the preferred alternative. However, the space requirements for the encoding of the global state can be substantial. Previously proposed schemes (to be discussed in detail in Chapter 5) require several bits for every directory entry–one bit for every cache in the system (to indicate 'present' or 'absent') and an additional modified bit, to indicate whether or not the block is modified with respect to main memory. For example, in a multiprocessor system with 16 processors, 16 bytes per block, and a main memory of 32 Megabytes, 17 bits are required for each of the 2 million memory blocks, for a total of 34 Megabits, or about 13% of the total memory. Assuming a cache size of 32 Kbytes, with a 22 bit tag for each directory entry, the centralized directory scheme described in the previous section would require 22 bits for each of the 2K blocks in each cache, for a total of 720 Kbits, or about .3% of the total memory. Even if each cache directory were replicated in its entirety in each memory controller (as described in the previous section), with 16 memory modules the total number of bits required would be 11 Mbits, or about 4% of the total memory.

In Chapter 5 we examine these protocols in more detail and analyze their performance using a simulation model. In addition, new cache coherence protocols are proposed that use smaller, fixed length directory entries, significantly reducing the space requirements but also offering high levels of performance.

Cache Oriented Directory with Shared Bus. This category of protocols is based on the observation that caches can cooperate to maintain consistency by listening to each other's transactions. Since they share a common bus, each cache can *snoop* on the bus transactions of the other caches and take actions as necessary to maintain the consistency of those blocks of which it has a copy. The global state is obtained from information contained in the local caches, as is the local state. Conceptually, the directory containing the global state is distributed across the caches. In addition, the intelligence to support the protocol is distributed over all the caches. For example, if a bus transaction of another cache is a read miss on a block that is modified and present in the local cache, the valid data must be supplied by the local cache. If another cache has a write miss, the local copy may need to be invalidated. Other transactions may simply require a change in the

local state, or no action at all.

In Chapter 4, several previously proposed schemes are described and their performance is examined in detail. On the basis of observations of the performance of these schemes, two new shared bus protocols are proposed that surpass the performance of the best existing schemes.

Memory Oriented Directory with Shared Bus. There have been no schemes proposed specifically for a shared bus that use memory level maps, no do there appear to be advantages to such an approach. Of course, protocols proposed for a general network would also work for a shared bus, but it appears that snooping cache protocols (with cache oriented directories) are more appropriate for a shared bus multiprocessor. In general, the cache oriented shared bus protocols require only the addition of a special cache controller, while the memory level maps require special cache controllers and special memory controllers, in addition to the substantial memory requirements to implement the global table. Protocols in this category will not be further considered.

2.2 Software Solutions

All software approaches are based on the compile time tagging of all shared writable data. Static tags associated with each block indicate the way in which the block is to be treated at runtime. Blocks without the shared writable tag are referenced and cached in a normal fashion, while blocks with the tag will be treated differently so inconsistencies cannot arise. Proposals for implementation generally assume that tags are determined on the page level, with the tag stored with the page table entry in the TLB (Translation Look-aside Buffer), used to store recently referenced page table entries to perform the virtual to real address translation. It can then be assumed that the contents of the tag are known at the time of each cache access.

Unlike some schemes that will be considered in later sections, the correctness of the schemes discussed here relies on the correctness of the tags. If a shared block is incorrectly tagged as non-shared, there is no guarantee that inconsistencies will not arise.

2.2.1 Prohibit Caching of Shared Blocks

The simplest method of avoiding the coherence problem using tags is to prohibit the caching of shared writable blocks. All references to blocks with this tag are directed to main memory and the block is not loaded into the cache on a miss. Since no cached copies of this data will ever exist, no

inconsistencies can arise. If references to these shared blocks constitute a significant percentage of all references, the added overhead of accessing main memory will significantly reduce performance.

A major limitation of this approach, common to all software approaches, is the elimination (or severe restriction) of task migration. Since the results of task migration appear very much like actual sharing at runtime, it can only be permitted if the cache is flushed on each context switch. Prohibiting task migration in a multiprocessor forces the system designers to implement load balancing strategies in order to achieve satisfactory levels of processor utilization.

Another undesirable aspect of prohibiting caching is that performance suffers from the added overhead. It is likely that only a small percentage of references to tagged data will lead to inconsistencies, but this solution imposes the overhead of a global memory access on *all* shared accesses. Because the *possibility* of sharing exists, tagged blocks must be treated as if they are shared on every reference. If shared references constitute a significant percentage of all references, the additional overhead in accessing main memory can make this approach infeasible.

2.2.2 Access Shared Blocks Only in Critical Sections

A second approach, described by Smith as 'the standard software solution', is to allow the caching of all blocks, but to maintain consistency by prohibiting multiple copies of all tagged data. In the case of writable shared blocks, accesses are possible only within a critical section, enforced through lock and unlock operations implemented with atomic operations on non-cacheable synchronization variables in main memory. This requires a special tag value (generated by the compiler) indicating that the block containing the synchronization variables may not be cached. Before any references to a shared variable are permitted, the lock permitting access to that variable must be obtained. After all references in the critical section have been completed, the blocks in the local cache containing copies of the relevant shared variables must be removed from the cache by marking as invalid and writing-back to main memory if modified. At this point, the lock may be released, allowing the next process to enter the critical section. To ensure the consistency of non-shared data, task migration is prohibited, unless all blocks are removed from the cache at each context switch. By using a write through memory update policy, memory is kept up to date so that misses can always be serviced by main memory. Smith has proposed a variation of this approach which introduces several enhancements and optimizations to the basic scheme[Smi85b]. Because it is the only comprehensive software solution described in the literature, we examine the scheme in detail.

A Software Solution Using 'One Time Identifiers' The main advantage of this scheme is a significant reduction in the number of cache flushes required in the standard approach described

above. This is accomplished by the addition of an identifier that is unique for each write shared page accessed within a critical section. Since the identifier is unique for each access, it can be determined whether copies of the block remaining in the block are *stale*–loaded in the last critical section accessing that block–or if they are up-to-date. A second advantage is that write through is performed only on write shared blocks. Blocks without the tag are cached with a write back policy, eliminating much of the memory traffic.

The first reference to a page tagged as write shared loads the TLB in the normal fashion with the new translation entry for that page. Each TLB has associated with it an 'OTI register' to generate *one time identifiers*. As the new entry in the TLB is created, the current contents of the OTI register are copied to an OTI field in the new TLB entry, and the contents of the OTI register are incremented by one. Whenever a block must be loaded into the cache, the contents of the OTI field in the TLB entry of the corresponding page are copied to an OTI field in the cache directory entry associated with the block. On every cache access, the contents of the OTI entry from the TLB are compared against the OTI entry from the cache directory for that block. Only if the identifiers match is the access permitted. If the OTI fields do not match, the reference is treated as a cache miss–the block was loaded in a previous critical section and is presumed to be stale.

When the access to a write shared page is about to be given up, the TLB entry for that page is invalidated, using a special *invalidate TLB entry command*. This ensures that all block copies from this page are now inaccessible, since in the next critical section accessing that page, it will have a new OTI. In the rare event that the OTI register overflows, the entire cache must be purged. Since this is a rare occurrence, the cache purge mechanism need not be especially efficient. In the standard approach, the efficiency of the cache purging mechanism is far more critical, since it will be used more frequently.

Although this scheme eliminates much of the overhead of the standard software approach, it retains some of the same limitations. In particular, task migration is not permitted unless the task is marked in its entirety as a shared data structure, forcing all accesses to be write through and all relevant TLB entries to be invalidated on a context switch.

2.2.3 Hybrid Schemes

There exist a number of alternative approaches (not previously considered) that could make use of the software tag. For example, consider a write broadcast approach in which only writes on tagged blocks require a broadcast. In general, any hardware protocol could be used to maintain

the coherence of the tagged blocks with no coherence overhead on the unmarked blocks.

These approaches do not appear promising in small to moderate multiprocessors for a number of reasons. First, as with all schemes ignoring the coherence of unmarked blocks, task migration must be limited. Secondly, it requires the full hardware support necessary to implement the protocol, in addition to the hardware and software overheads of the standard software approach. This means that it would cost more to implement than either solution by itself, and it is unlikely that there would be any resulting performance increase. High performance directory solutions are able to dynamically detect the sharing of blocks quite efficiently–the static tag would be of little if any value to these schemes.

In systems with a large number of processors, it is possible that hybrid approaches might lead to interesting results. For example, the use of the shared tag could lead to a more space efficient memory oriented directory scheme if the full state information were only maintained for blocks with the tag. Smith observes that his OTI scheme might be useful in a hierarchical system consisting of clusters of processors using a shared bus protocol locally and the OTI scheme globally. In such a system, the limitation of task migration of the OTI scheme would not be as critical, since it could be allowed between processors sharing a bus, although it would not be permitted between clusters. These and other alternatives are considered in Chapter 6 in the discussion of protocols for multiprocessors with hierarchical networks.

An interesting alternative is a hybrid scheme using a software tag and a hardware protocol that does not rely on the accuracy of the tag for correctness. In such a scheme, the value of the software tag is said to be a *hint* that the protocol can use to increase the performance. A few protocols using this technique will be discussed briefly in the following chapters.

2.3 Hardware Or Software?

Proponents of software coherence approaches have claimed a number of advantages over hardware based protocols. According to Smith [Smi85b], software solutions are cheaper and faster than hardware solutions, and they are better suited for systems with large numbers of processors. We consider each argument in turn.

It might appear that a software approach would be a natural result of the recent RISC movement which advocates assigning traditional hardware tasks to the compiler, but it must be recognized that the efficient software schemes still require significant software support. The extra hardware required to support Smith's OTI scheme is not trivial. For example, it requires extra storage in the TLB and cache address tags for the OTIs, the OTI register itself, the logic to increment

the register and detect overflows, an instruction to selectively invalidate TLB entries, a cache supporting both write-through and write-back, and a mechanism to purge the cache. He states that the hardware cost will not, in any case, be worse than that required for some hardware based protocols, but 'a sharp reduction in hardware cost is not the primary advantage of the OTI scheme' [Smi85b]. An estimate of the actual relative design and implementation cost is beyond the scope of this dissertation, but the cost of the OTI scheme must include the design and implementation of special TLBs, caches, and memory controllers. In addition, the cost of modifying the system software to accurately generate the tags must be included. Smith notes that it is not always trivial to determine which blocks are shared, and that 'it is easy to designate far too much as shared', which would result in lower performance.

The relative performance of hardware and software schemes is very much an open question. An accurate comparison of the two is beyond the scope of this dissertation and it is an important area for further research. However, the main tradeoffs in performance will be stated here. If an application requires vast amounts of shared data (e.g., a number of huge arrays manipulated by a number of processes in parallel), the hardware protocols would be expected to have the advantage for the following reason. Although there are many blocks that are *shared* (in the sense that they are referenced by more than one process), the blocks might only rarely be present in more than one cache at a time. (Other processes may be operating on a different portion of the array.) Because the hardware protocol treats blocks according to their dynamic state, these 'shared' blocks will be treated as efficiently as private blocks for those periods of time that they are not actually shared. In the software solutions, all references to the shared data will incur additional overhead, even if the data are only rarely present in more than one cache at a time. On the other hand, if there is very little shared data, the main factor in performance is the overhead in processing private blocks. Because the hardware protocols must check and update local and global states (and maintain consistency between the two), the relative performance of the software schemes would likely improve.

A serious performance consideration is the limitation of task migration in all software approaches. One of the main advantages of a multiprocessor is the uniform utilization of system resources shared between all job streams. In particular, it is easy to maintain a uniform level of processor utilization if there is a central pool of ready processes that can execute on any processor in the system. Eliminating task migration forces tasks to resume on the same processor after they are suspended. This can lead to a system state where some processors are saturated and other processors are idle, but unable to share the load. To efficiently use the processing elements, some type of load balancing must be used. This complication is avoided by allowing unrestricted task

migration. It would be difficult to quantify the difference in performance resulting from elimination of task migration without multiprocessor runtime data, but it is an important factor to consider.

Smith's third statement, that his OTI scheme is more appropriate for large numbers of processor, is correct concerning existing hardware schemes. As he observes, this is the most significant aspect of his proposal. Solutions requiring a shared bus are limited by the bus traffic to perhaps a few dozen processors. The memory required to implement previously proposed protocols for general interconnection networks grows linearly with the number of processors, limiting their practicality to the same general range. However, the new protocols for general interconnection networks proposed in Chapter 5 significantly reduce the amount of memory required for the global map, which might make the approaches feasible for a much larger number of processors. And in Chapter 7, we examine a protocol suitable for a very large multiprocessor using a hierarchical interconnection network.

There are several clear advantages of a hardware implemented protocol. The primary ones are listed below.

- Hardware based solutions do not complicate any level of software in any way.
- The traditional transparency of the cache is preserved. No processor instructions regarding explicit block loading or flushing are necessary.
- Applications using very high levels of shared data references have a potential for higher performance. Data acesses by different processors are not mutually exclusive.
- Inconsistencies resulting from task migration are prevented using the same mechanism that prevents inconsistencies in shared variables. Task migration is not restricted in any way, allowing uniform utilization of all processors.

On the basis of this information, it is our belief that hardware cache coherence protocols are preferable to software solutions whenever their use is not prohibited by cost or performance considerations. It is the intent of this dissertation to identify those protocols whose efficiency and hardware cost make them feasible alternatives over the largest range of system sizes.

Chapter 3

Multiprocessor Simulation Model

The purpose of the model described in this chapter is to accurately reflect the quantitative differences in performance between the protocols that will be presented and discussed in later chapters. To accomplish this purpose it is necessary for the model to reflect the normal operations of a shared-memory multiprocessor, including such things as network contention. The most critical element of the model is the stream of references to shared data, since the interaction between processor reference streams has tremendous impact on the performance of all coherence protocols. This chapter describes all aspects of the model in detail.

3.1 Advantages of Simulation Approach

Although there have been many cache coherence solutions proposed, there has been only one other published study comparing the relative performance of different approaches. In that study, Vernon and Holliday evaluate the performance of protocols for shared buses using Generalized Timed Petri Nets[VM86]. They do not, however, model the protocols exactly as they are proposed. Instead, they evaluate the effects of various modifications to a particular protocol. In addition, the workload model chosen for their study makes use of several parameters which have subtle relationships with each other and are therefore challenging to find accurate values for. The simulation study proposed here does not require as many parameters and therefore avoids the main problems resulting from subtle dependencies. In section 3.5, an example of one such problem is presented that casts some doubt on the accuracy of their results.

Papers proposing cache coherence protocols have often included some sort of performance

estimate, but expressions derived are often overly simplified, and do not apply to other schemes. In addition, it may be difficult to select accurate values for the parameters used, since there is little, if any, published multiprocessor runtime data.

Consider one example that illustrates the difficulty in comparing the performance of two different schemes. One way in which protocols for shared buses differ is the action taken on each write to a block shared between multiple caches. Most protocols require all other copies to be invalidated, but some protocols update the other copies by distributing the new data over the bus. Subsequent references in other caches will result in cache misses in an invalidation scheme. In an update protocol, they result in cache hits. An evaluation of the relative merits of an invalidation scheme versus an update scheme must necessarily weigh the overhead of any misses resulting from invalidations against the overhead of the updates that prevent the invalidations from occurring. Any performance estimates must explicitly or implicitly include the relative frequency of these and other actions.

In light of these challenges, the method we have chosen to model the schemes is *trace driven simulation*, although the traces used are artificially generated. Each of the protocols is faithfully simulated and the overall system performance using each protocol is compared while running an identical workload. The use of actual traces is not possible, for no multiprocessor traces are available. While multiprocessor traces could be created from existing uniprocessor traces, they would be no more realistic than the technique used in this study. As the example in the previous paragraph shows, protocol performance is very dependent on the nature of shared references. In the absence of actual measurements, an accurate method of generating a synthetic reference stream is the only alternative. The synthetic reference model, described in the next section, is the main contribution in the simulation model described in this chapter.

3.2 Workload Model

The choice of workload representation is critical, since it determines the nature of data sharing, and since the performance of all coherence solutions depends heavily on the level of sharing. The model selected is based on one proposed by Dubois and Briggs [DF82]. In that paper, the authors note that a workload model for a shared memory multiprocessor is generally much more complex than for uniprocessors, since it must include a specification of each processor reference stream as well as a specification of the interaction between streams. They state that a stochastic reference model is appropriate for this application because it is difficult to obtain parallel traces on multiprocessors, and because separate traces of the execution of the same program may be different due to the

indeterminate nature of process execution.

Another potential problem with multiprocessor traces–one not previously noted–is that they are likely to exhibit different levels of sharing depending on the underlying architecture and system software. For example, a trace obtained from a multiprocessor with a lot of added overhead in accessing shared blocks is likely to have less actual sharing than a multiprocessor in which shared block accesses are very efficient. Application and system programmers are likely to be aware of the shared-reference overhead and structure their programs accordingly, restricting the amount of shared writable data when accesses are costly, and using it freely when accesses are very efficient. Therefore, realistic parallel traces would best be obtained on a system similar to the one being modelled. Clearly, this would be a serious limitation, considering the limited number of commercial multiprocessors available today.

In the simulation model, just as in the Dubois and Briggs model, the reference stream of each processor is viewed as the merging of two reference streams. The first stream consists of references to S-blocks, which are blocks that are shared between processes. The second stream is the sequence of references to P-blocks, or blocks that are private to a process. S-blocks are referenced by all processors, including write references, and may therefore be present in any cache. The model does not include effects resulting from task migration, although its effects would be similar to increasing the frequency of S-block references. Nor does the model include a class of shared read-only references, since the overhead in shared read-only accesses is generally very similar to the overhead in accessing private blocks. While P-blocks are never present in more than a single cache and cannot therefore lead to inconsistencies, they are an important part of the simulation because they are handled differently by the protocols. Because P-blocks are referenced frequently, slight differences in efficiency of P-block handling can have a tremendous impact on the overall performance of a scheme. Together, the P-block and S-block references provide an accurate model of a multiprocessor workload.

Our model differs substantially from the model of Dubois and Briggs in the way specific S-blocks and P-blocks are referenced. In their model an independent reference model (IRM) is used to determine the references to S-blocks, and a least recently used stack model (LRUSM) is used to generate P-blocks references. They observe, however, that such a model of P-block references is not required if the hit ratio can be determined by other means. It is assumed that there are a fixed number of S-blocks and P-blocks. In the IRM model, S-blocks references are made according to a fixed probability distribution, with a fixed proportion of the references to each block. This choice of S-block reference model was made on the assumption that S-blocks accesses exhibit less locality than P-block references. Although it may be true that the S-blocks show less locality, the

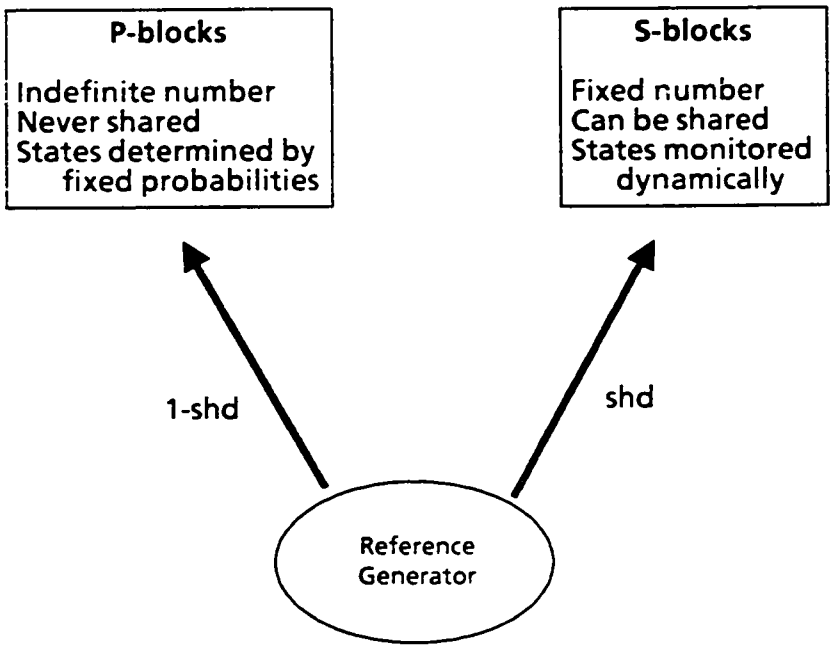


Figure 3.1: Reference stream composition

IRM does not provide *any* temporal locality of reference.

For our simulation model, an LRUSM is used to generate S-block references. Like the LRUSM in the model of Dubois and Briggs for P-blocks, this allows for some locality of reference. Stated simply, those S-blocks recently referenced by a particular processor are more likely to be referenced in the near future than S-blocks not referenced for some time. This would be the case if shared variables tend to be referenced as local variables for some length of time. S-blocks are represented explicitly, with a fixed number in each simulation run. Tables are used in the simulation to maintain global and local state information about each individual S-block. The P-block references are characterized by fixed probabilities obtained from uniprocessor measurements. It is assumed that P-block reference behavior is essentially identical to reference behavior in a uniprocessor environment. There is no explicit representation of P-blocks, nor is the number of P-blocks explicit in the simulation.

Each time a memory reference is generated, it is an S-block reference with probability *shd* and a P-block reference with probability 1-*shd*. Similarly, the probability that the reference is a read is *rd* and the probability that it is a write is 1-*rd*. P-block references are all treated identically–no distinction is made as to which block is referenced. S-block references include a specific block number, generated using an LRU stack that is unique to each processor. The contents of the stack reflect the past reference pattern of the processor, with the most recently accessed S-block on top, and the least recently accessed S-block on the bottom. The probability of accessing each S-block depends on the depth of the block in the LRU stack at each reference. The probability that an S-block reference is to the S-block at stack depth i is given by the equation:

$$g[1/(b+i) - 1/(b+1+i)]$$

where g is a normalizing factor, and b is a parameter to fine tune the probability distribution. For the simulation results presented in this dissertation, a value of 5 was used for b, because, with a single processor, it results in an S-block hit ratio comparable to the P-block hit ratio of 95%. (The S-block hit ratio is slightly less to reflect the reduced locality of reference with respect to P-blocks.)

Using the LRU stack and the formula given above, the blocks near the top are much more likely to be referenced than those blocks near the bottom. After each reference the contents of the stack are updated, with the most recently referenced block being moved to the top and the others shifting down one position. The stack is initialized uniquely for each processor in such a way that the average depth over all stacks is approximately the same for each S-block.

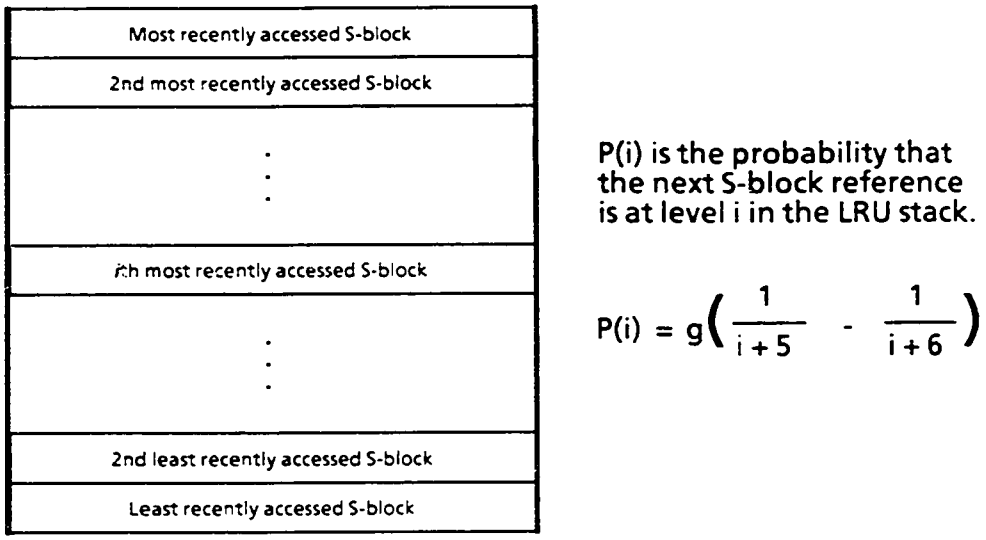


Figure 3.2: Organization of LRU stack for S-block references

3.3 Processors and Caches

Our simulation model has been programmed in Simula, with each processor and each cache represented by a Simula process. Each processor performs useful work for some w cycles (picked from a specified distribution) and then generates a memory reference. The nature of that reference is determined according to the workload model given above. The processor then places the memory request in the service queue of its cache and waits for a response, during which time no work is done. Processor utilization is measured by the ratio of the total time doing useful work to the total simulation time. The sum of all processor utilizations in the system is used as the main metric in determining system performance.

The cache process takes the incoming request, including the information describing it as a read or a write, and as an S-block or P-block reference (including a specific block number if it is an S-block reference). The two types of references are treated differently by the cache, but all types of requests incur the specified overhead of the coherence protocol for the necessary actions. If a global transaction is necessary, it is completed as part of the normal cache operation. Such global actions may be cache misses, requiring the loading of a block from memory, or they might involve actions unique to a particular protocol. All global transactions are initiated by the cache in the form of a request to the interconnection network. Depending on the type of multiprocessor system simulated, this might be one or more additional processes, and the cache will have to wait if the

network is busy. Once the servicing of the request is complete, the cache sends a signal to the processor to continue. Cache operations that can be processed locally require a single cache cycle to service.

P-block references are serviced in a random fashion based on fixed probabilities. Each P-block reference results in a cache hit with probability *h* and a miss with probability 1-*h*. If the request is a write hit, the P-block is already modified with probability *wmd*, and the block is not yet modified in the local cache with probability 1-*wmd*. Note that the workload model for P-blocks reflects steady state behavior and not behavior including a cold start, since at the beginning of the simulation the cache is effectively loaded with most of the P-blocks it will access in the next several references and the hit ratio has leveled out.

S-block references include a specific block number and are serviced by the cache according to the actual local state of the block at that instant in time. If the block is present and the protocol specifies no global action, the request is processed in a single cycle and the processor then is allowed to proceed. If the block is absent, or the protocol otherwise requires it, a global transaction is completed as part of the servicing of the cache request. The local state of each S-block is maintained in a table associated with each cache. Other tables maintain the global state of S-blocks for those schemes that require a global directory. Explicit local and global states for each S-block are maintained dynamically (including each processor's LRU stack used to generate S-block references).

If a cache miss occurs, a block must be ejected to make room for the new block. The probability that the replaced block is an S-block is equal to the percentage of S-blocks in the cache at that point in time. If an S-block is to be replaced, one is chosen at random. The local state of any S-block selected for replacement determines whether or not a write back is required to update main memory. The simulation tables are modified to indicate that the replaced block is no longer present in the cache. If a P-block is selected, it is modified and needs to be written back with probability *md*, and with probability 1-*md* no action need be taken.

In addition to servicing requests from the processor, each cache also receives commands through the interconnection network requiring the completion of certain actions on S-blocks that it has copies of. These commands have a higher priority for service by the cache than processor memory requests. Generally, the necessary action is a simple state change (requiring a single cycle to service) or a request to supply the contents of a block (requiring as many cycles to service as there are words in the block). After the required action is completed, the cache is again free to respond to processor requests. In all simulations it is assumed that the cache is organized with the dual directory controller described in Chapter 2. Hence, the cache only responds to those

commands which involve a block currently in the cache. Since this takes place only in the relatively infrequent case of actual sharing, the servicing of these commands has little impact on the servicing of processor memory requests.

3.4 Interconnection Networks

Two different types of interconnection networks have been simulated for the studies described in this work. A shared bus multiprocessor was simulated to compare schemes intended specifically for shared bus systems. To compare protocols for general interconnection networks, a crossbar switch was simulated. The multiprocessor simulations use the same workload and are otherwise identical except as noted.

3.4.1 Shared Bus

In the shared bus multiprocessor simulation, a single Simula process is used to model the bus and memory. Both resources can be combined into a single process because, in all the schemes modelled, the bus is held until the memory cycle is completed. Shared bus schemes allowing the overlapping of bus operations (while the memory cycles, for instance) would require a more complex organization. This bus and memory server process receives service requests from all caches and services them in FIFO order. Conceptually, the bus process includes the added cache logic responsible for matching addresses, and so can determine the location of all cached copies of shared blocks. If, in servicing a request, the bus process determines (from the global state of the block) that other caches need to supply the data or that a local state change is required, commands are sent to the appropriate caches. When the transaction is complete, the bus signals to the cache that it has completed, and the bus then begins to process the next request, if one is waiting for service. In the shared bus simulation, modified blocks selected for replacement are written back before the requested block is loaded.

3.4.2 Crossbar

In a crossbar switch connecting multiple memory modules with multiple caches, concurrent operations are possible as long as neither the source nor destination of one transaction are involved in another transaction at the same time. Communication is assumed possible in only one direction at a time. The normal operation of the switch is as follows. A request from a cache is put into the service queue of the Simula process associated with the port of the switch connected to that

cache–each cache has its own switch connection. If the port is already busy, either sending a previous request or receiving a signal from any memory module port, the cache request must wait until it becomes available.

To service a cache request, the port process determines the destination and determines if the destination port is busy or not. If it is busy, the transfer is delayed until the next available time slot that the destination is free. During any time waiting for the destination to become available, the cache port is free to receive incoming packets. When the destination becomes available, a connecting circuit is established and the packet is sent. The length of service time is a single cycle for any requests not involving the transfer of an entire block. Otherwise, it requires as many cycles as there are words in the block. During the transmission, both the source and destination ports of the switch are busy and unable to participate in any other transactions. At the completion of the transfer, the request is inserted in the service queue of the memory module connected to the destination switch port, where it will be processed in FIFO order by the memory process.

Sending messages from the memory to the cache is very similar, with one difference. In this direction, it is possible for a memory module to broadcast to all caches in a single cycle. Before the broadcast is allowed to take place, the source must wait until *all* destination units are available before it can be sent. It is assumed that broadcasts have a higher priority than the transmission of other signals.

In the crossbar simulation, blocks selected for replacement are written back *after* the cache miss request is sent. It is felt that this *buffered write back* approach would be more appropriate for a high performance multiprocessor using an expensive crossbar switch.

3.5 Memories

Each memory module is represented by a Simula process. Memories are represented explicitly only in the crossbar simulation; in the shared bus simulation they are implicitly included in the bus process. The controller process begins to process the next request in its service queue only after the previous request has completed in its entirety, including memory and global map cycle time. The first step in processing most requests is reading the state of the block from the global table. (Depending on the protocol and the request type, this is not always necessary.) For those requests involving a main memory access, it is assumed that both accesses may be started simultaneously and overlapped. (The logical organization of each memory module is that displayed in Figure 3.3 where the map and the memory are physically distinct.) It is assumed that the map and memory cycle times are the same–the global state is known at the same time that the first word in the

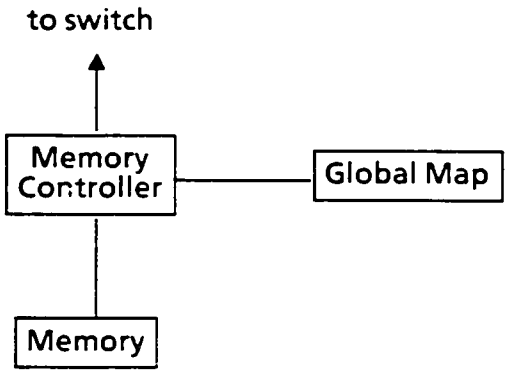


Figure 3.3: Logical organization of memory module

block is read from the memory. Therefore, the time required to read the block always takes less than the time required to read the entire block, assuming the block size is greater than one word.

Following the global map access, the controller may be required to send signals to other caches, instructing them to invalidate their copies or to provide main memory with a valid copy. If a single signal or a broadcast to all caches is required, the signal may be sent in a single cycle, during which the request is placed in the service queue of the crossbar port associated with the memory module. If signals must be sent to more than one cache (but not broadcast to all), sequential handling is required. These alternatives are illustrated in Figure 3.4.

Following the sending of any necessary signals, the controller must update the global map, assuming that the transaction caused a change in the global state of the block. (If no change occurred, the update portion may be omitted from the examples in Figure 3.4.) If the transaction involves the transmission of a block, it is sent as soon as it becomes available, assuming that it is permitted by the protocol with the current global state. If the current global state does not permit the data to be sent, the main memory access in progress is aborted and the word of the block that has already arrived is ignored.

Other types of requests may not require the reading (or writing) of a block from main memory. They do, however, require accesses to the global map. Depending on the protocol (and the type of transaction), this may be both a read and a write, or just one of the two (as shown in Figure 3.5).

As can be seen, for the parameters used in the simulation, the time required to read and write the global map dominates the overall service time of memory requests, even though it is overlapped as fully as possible with each memory access.

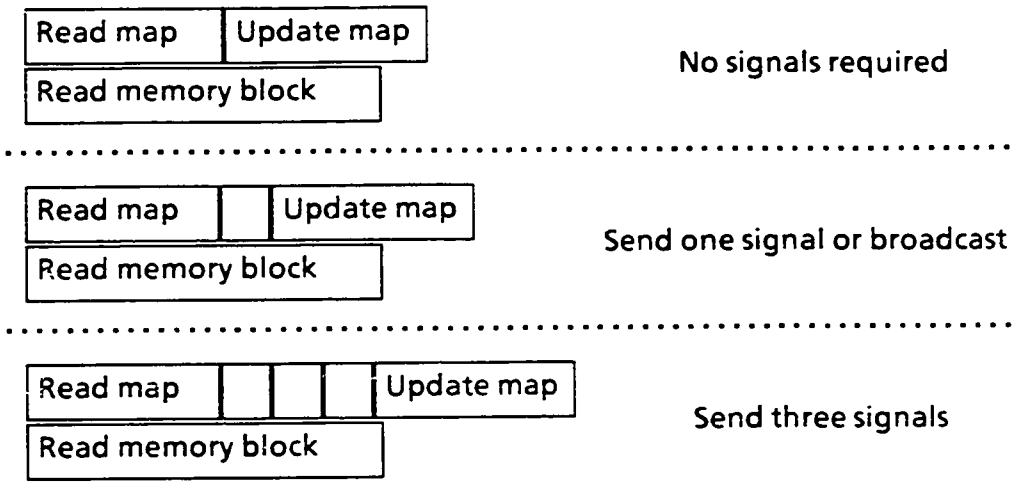


Figure 3.4: Overlap of memory and map accesses

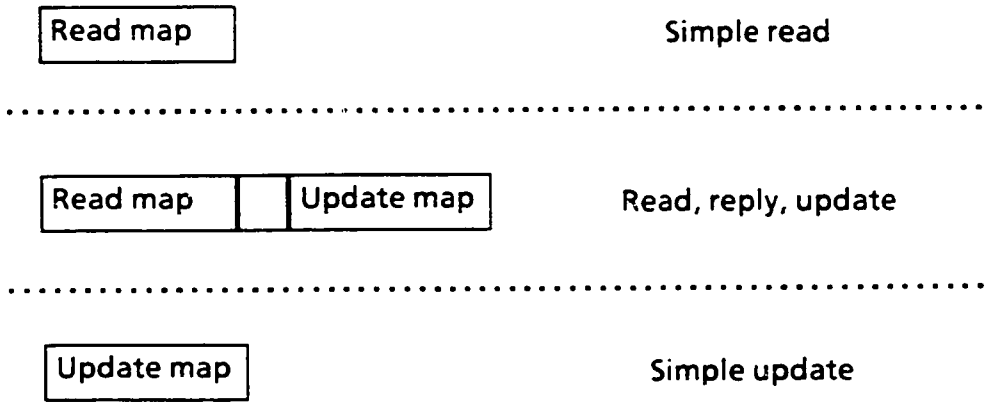


Figure 3.5: Map accesses without memory accesses

3.6 Simulation Parameters

The probabilities *md*, *wmd*, and *rd* are not independent. An example of inconsistent parameter values used in an early version of this simulation study provides evidence of the potential difficulty of selection of parameter values. An initial draft of a report describing the performance of shared bus protocols included results obtained with the following values: $md = 0.3$, $wmd = 0.7$, $rd = 0.7$, $h = .95$, and $shd = 0.05$. It may not be readily apparent that there is a problem with these values, but the inconsistency can be demonstrated as follows. Consider a sequence of 100 processor references in steady state. About 5 of the 100 references will result in cache misses. Approximately 95 of these references will be to P-blocks, with about 64 read hits, 3 read misses, 27 write hits, and 1 write miss. Of the 27 P-block write hits, about 19 will be to blocks already modified, and the remaining 8 are to blocks that are not yet modified.

But it is impossible to have 9 write hits on unmodified blocks and only 3 read misses–unmodified blocks cannot be modified faster than they are brought into the cache. Even if the rate of write hits on unmodified blocks were reduced to the rate of read misses, this implies that all the blocks loaded into the cache on a read miss are eventually modified. Therefore, all blocks would be dirty on replacement. Since *md* and *rd* are unlikely to differ greatly from the uniprocessor environment, and since the values in the example are consistent with published measurements [Smi85a], it is the value of *wmd* that is at fault. Clearly, *md* must be at least as large as 1-*rd*, and the difference depends on the value of *wmd*. A value of 70% for *wmd* is far too low. In practice, only a small percentage of blocks loaded on a read miss will eventually be modified, since normal values of *md* range from 30% to 40% and normal values of *rd* range from 15% to 30% [Smi85a]. Not surprisingly, the simulation results obtained with these faulty parameter values greatly overemphasized the overhead of writes to unmodified blocks. Simulation runs with more appropriate parameter values resulted in quite different relative performance of shared bus protocols.

Other studies have also used incorrect values for *wmd*. In a paper by Papamarcos and Patel describing a shared bus protocol, performance estimates are obtained using a value of 70% for *wmd*[PJ84]. In fact, it was from this paper that the value for our simulation study had originally been obtained. (It is only fair to note that it was Patel himself who, after reading the draft of our report, told us of the problem with the parameters, having discovered it after his paper was published.) A value of 70% for *wmd* is also used in the study of Vernon and Holliday[VM86], raising some questions about their results. It should be noted that, unlike the simulation model described here, their model characterizes both shared and private block references using fixed probabilities. In the case of shared references, these probabilities are difficult, if not impossible, to verify, since

so little is known about the runtime reference behavior of shared variables. In the simulation workload, these difficult to obtain parameter values are implicit. No explicit characterization of shared references is required, except for the number of shared blocks and the percentage of shared references.

To correct the inconsistency in the early simulation results, the following approach was taken. First, observe that the probability that a block is dirty when it is replaced is equal to the probability that it was loaded on a write miss, plus the probability that it was loaded on a read miss and later modified. (The probability of loading on a read miss (or write miss) can be approximated by the percentage of read requests (or write requests), assuming that the miss ratios on reads and writes are nearly identical to the overall miss ratio.) Stated in an equation, where x is the percentage of blocks loaded on a read miss that are eventually modified,

$$md = (1 - rd) + x(rd).$$

In steady state, the probability of writing to an unmodified block present in the cache must equal the probability of loading a clean block into the cache times the percentage of blocks loaded on a read miss and eventually modified (or x above). That is,

$$(1 - rd)(h)(1 - wmd) = x(1 - h)(rd).$$

These two equations define a relationship that is assumed for the simulation runs in this study. Although these approximations are not exact, they serve as good estimates of the relative magnitude of the simulation parameters *md*, *wmd*, and *rd*. Typical values of *wmd* range from 95% to 99%, far different from the 70% used elsewhere.

A complete list of simulation parameters and ranges of values used is given in Figure 3.6.

3.7 Simulation Output

Output of the simulation includes bus utilization figures, processor utilization, and a result referred to as the *system power*. This is simply the sum of all processor utilizations in the system, multiplied by 100. Although a metric of effective number of processors might be more common, we use system power as the performance measure, because the uniprocessor utilization varies between coherence solutions and between simulation runs (with new parameters) of the same scheme. For example, it would be possible to have two protocols, A and B, where A is more efficient than B with a single processor, but, evaluated with ten processors, the resulting 'effective number of processors' for both protocols are identical–perhaps at eight times the uniprocessor performance. Protocol A is

	Parameter	Range
shd	% S-block references	0.1% - 5%
rd	% reads	70% - 85%
h	P-block hit ratio	95% - 98%
1-wmd	% write hits to unmodified P-blocks	1.75% - 5.26%
md	% P-blocks written back	30% - 40%
w	Distribution of inter-reference time	Uniform [0..5]
	Main memory cycle time	4 cache cycles
	Block size	4 words
	Cache size	2K - 16K bytes
	Number of S-blocks	16 - 1024
	Number of processors	1 -32
	Simulation length	25000 cycles

Figure 3.6: Summary of parameters and ranges

actually more powerful and more efficient than B, but a metric using a multiple of the uniprocessor power does not reflect this difference. Defining a common uniprocessor power for all schemes and dividing the system performance of each protocol by this constant would not alter the relative position of the curves in our figures–only the labels on the vertical axis would change. Since our intent here is to determine the relative performance of the schemes, rather than the maximum number of processors possible in a particular system, we use the system power metric.

Chapter 4

Shared Bus Protocols

4.1 Invalidate or Update?

Shared bus protocols are based on the principle that each cache can observe all bus transactions of other caches that would affect the status of blocks of which it has a copy. For efficiency, this observation is typically performed by a dedicated bus *snoop* that has a logical copy of the local cache directory (either a duplicate copy or access to the directory itself) and attempts to match on the block addresses of all bus transactions of other caches. On a match, the protocol specifies the action to be taken based on the local state of the block and the type of transaction observed. Although a match must be attempted by every cache on every bus operation, the added logic of the bus snoop is able to eliminate the overhead of all unsuccessful matches.

Shared bus protocols can be divided into two categories, based on the actions taken when a shared block is modified. The first approach maintains consistency by marking all other cached copies in the system invalid. The second technique is to keep the copies in other caches valid by updating them with the new data. For the remainder of this dissertation, these will be called the *invalidation* and *distributed write* approaches respectively. Note that it is not necessary in either approach to keep the copy in main memory current, provided that caches with dirty copies (relative to main memory) supply the data to other caches which request it, and that they eventually update main memory when the block is replaced in the cache. Since the write-back method typically results in reduced bus traffic relative to write-through [Smi82], it is preferred in a system where the bus is the main performance limiting resource.

	Processor References			
INVALIDATION	i	j	k	DISTRIBUTED WRITE
Read miss cache i	R			Read miss cache i
Read miss cache j		R		Read miss cache j
Invalidate i, write j		W		Update i, write j
Local write j		W		Update i, write j
Local write j		W		Update i, write j
Local write j		W		Update i, write j
Read miss cache k			R	Read miss cache k
Invalidate j, write k			W	Update i&j, write k
Local write k			W	Update i&j, write k
Local write k			W	Update i&j, write k
Local write k			W	Update i&j, write k

For purposes of illustration, assume misses require 8 bus cycles to service, and updates and invalidations each require 1 cycle. (Local writes and read hits require no bus cycles.)
Total bus cycles required: invalidation 26, distributed write 32.

Figure 4.1: Access patterns of shared variable similar to local variables

Consider two examples that demonstrate the general advantages and disadvantages of the invalidation and distributed write methods. Figures 4.1 and 4.2 list hypothetical reference patterns to a single block shared between three caches–i, j, and k. The three center columns indicate the sequence of references of the three processors–R indicates a read to the shared block and W indicates a write. For each successive reference, the actions of a simplified invalidation scheme are listed on the left and the actions of a simplified distributed write scheme are given on the right. Since the bus is the limiting resource in a shared bus multiprocessor, the comparison of the two approaches is made by the bus traffic each generates (based on simplistic timing assumptions for illustration purposes). In Figure 4.1, the invalidation approach has slightly less overhead, while the distributed write approach is much better in the example given in Figure 4.2. In general, invalidation may yield better performance than distributed write if access patterns to shared variables are quite similar to those for local data–as if each shared variable becomes a local variable for some period of time. On the other hand, if there is significant contention for writable shared data, the invalidation approach can incur much more overhead than distributed write due to a ping-pong effect or *thrashing*–the frequent movement of the block from cache to cache.

In this chapter several published shared bus schemes are described. Proposed invalidation schemes are outlined first, followed by a section describing a new invalidation protocol. Then

	Processor References			
INVALIDATION	i	j	k	DISTRIBUTED WRITE
Read miss cache i	R			Read miss cache i
Read miss cache j		R		Read miss cache j
Read miss cache k			R	Read miss cache k
Invalidate i&k, write j		W		Update i&k, write j
Read miss cache i	R			Read hit cache i
Write miss cache k, inv i&j			W	Update i&j, write k
Write miss cache i, inv k	W			Updatej&k, write i
Read miss cache j		R		Read hit cache j
Write miss cache k, inv i&j			W	Update i&j, write k
Write miss cache i, inv k	W			Update j&k, write i

(Using the same service requirements as the previous example)
Total bus cycles required: invalidation 73, distributed write 29.

Figure 4.2: Access patterns of shared variables with read and write contention

existing distributed write approaches are discussed, followed by the description of a new distributed write protocol. Results comparing the performance of the simulated protocols are presented, including results from the new protocols. Implementation issues are discussed briefly, noting the additional hardware necessary to obtain varying levels of performance. Finally, additional shared bus alternatives are mentioned that might prove to be interesting areas for future research.

4.2 Previously Proposed Invalidation Protocols

All schemes presented in this section are described using a uniform terminology allowing easy comparison. Each of the states required for each protocol are clearly specified, with state names and their abbreviations always given in capital letters. To describe the actions of the protocols, the essential actions of each scheme are listed for each of the following cases: read miss, write hit, and write miss. (The case of read hit requires no protocol action and so is not considered.) In addition, those blocks requiring a write-back to main memory when replaced in the cache are indicated. State diagrams are also included for each protocol for easy reference.

4.2.1 Synapse

This approach was used in the Synapse N+1, a multiprocessor for fault-tolerant transaction processing[Fra84]. The N+1 differs from other shared bus designs considered here in that it has

two system buses. The added bandwidth of the extra bus allows the system to be expanded to a maximum of 28 processors. Another noteworthy difference is the inclusion of a single bit tag with each cache block in main memory, indicating whether or not main memory is to respond to a miss on that block. If a cache has a modified copy of the block the bit tells the memory it need not respond. This prevents a possible race condition if a cache does not respond quickly enough to inhibit main memory from responding.

The Synapse protocol uses only three states, the fewest of any invalidation schemes we examine here. These three states form the basic set of states used in all invalidation protocols. Other schemes typically use four states, since four states can be encoded as cheaply as three states using two bits in the directory, and since a fourth state can bring an improvement in performance of the protocol. The local states (with abbreviations used in the text given in parentheses) are:

1. INVALID. (INV) Copy is not up to date.
2. UNMODIFIED-SHARED. (UNMOD-SHD) Copy is not modified with respect to main memory. Other caches *may* have a copy.
3. MODIFIED-EXCLUSIVE. (MOD-EXC) No other copies exist. Block must be written back on replacement.

A state transition diagram for this protocol is given in Figure 4.3. In Synapse terminology, any cache with a MOD-EXC copy of a block is called the *owner* of that block. If no MOD-EXC copy exists, memory is the owner. The Synapse protocol works as follows:

- **Read miss.** If another cache has a MOD-EXC copy, the cache submitting the read miss receives a negative acknowledge. The owner then writes the block back to main memory–simultaneously resetting the bit tag and changing the local state to INV. The requesting cache must then send an additional miss request to get the block from main memory. In all other cases the block comes directly from main memory. Note that the block is always supplied by its owner, whether memory or a cache. The loaded block is always in state UNMOD-SHD.
- **Write hit.** If the block is MOD-EXC the write can proceed without delay. If the block is UNMOD-SHD the procedure is identical to a write miss (including a full data transfer) because there is no invalidation signal.
- **Write miss.** Like a read miss, the block always comes from memory–if the block was MOD-EXC in another cache it must first be written to memory by the owner. Any caches with a

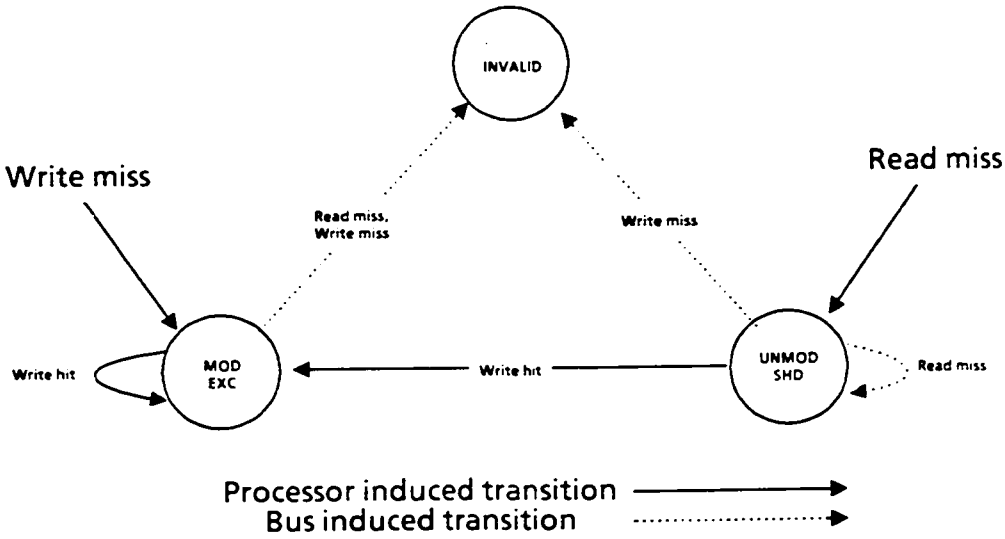


Figure 4.3: Synapse state transition diagram

UNMOD-SHD block copy set their state to INV and the block is loaded in state MOD-EXC. The block's tag in main memory is set so the memory ignores subsequent requests for the block.

As in all invalidation protocols, the Synapse scheme allows multiple caches to have read access to a block, but only a single cache is allowed write access at a time. All other copies must be invalidated before a write is allowed to proceed.

4.2.2 CMU Read Broadcast (RB)

This protocol introduces the concept of *validation*, or updating invalid copies whenever possible when the valid contents of the block are available on the bus [RZ84]. The scheme uses the following states:

1. INVALID. (INV)
2. UNMODIFIED-SHARED. (UNMOD-SHD)
3. MODIFIED-EXCLUSIVE. (MOD-EXC)

Although this scheme uses the same three states as the Synapse protocol, it assumes a block size of one word (the width of the data bus), resulting in significant differences between the two protocols. Figure 4.4 contains a state transition diagram for the protocol, which works as follows:

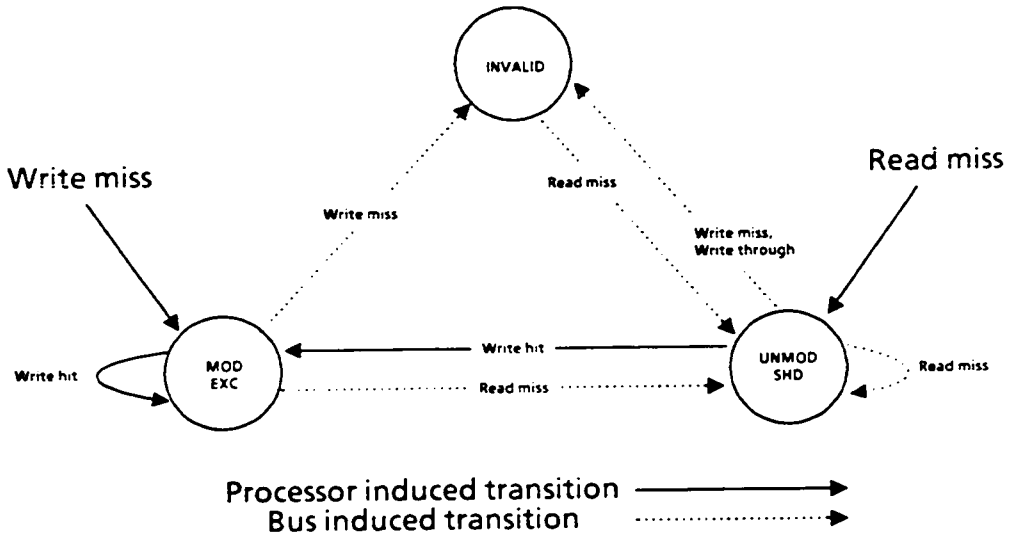


Figure 4.4: CMU RB state transition diagram

- **Read miss.** When the read miss signal is sent, any cache with a MOD-EXC copy must interrupt the bus activity and update main memory with a write back, setting its local state to UNMOD-SHD. Immediately following this action, the read miss is serviced, loading the block from main memory in the UNMOD-SHD state. Any caches in the system with an INV copy will also take the valid data and update their block copy, setting the local state to UNMOD-SHD. If no cache has a MOD-EXC copy, the block is loaded directly from main memory in state UNMOD-SHD.

- **Write hit.** If the block is MOD-EXC the write can be serviced immediately by the cache. If the block is UNMOD-SHD, a write-through takes place, updating main memory and invalidating all other caches copies. The state is changed to MOD-EXC.

- **Write miss.** Because the block size is one word, blocks do not need to be loaded on a write miss. An entry for the block is created in the cache and modified directly, accompanied by a write-through that invalidates all other cached copies of the block and updating main memory. Note that this is exactly the same action as a write hit on UNMOD-SHD, except that a cache entry might need to be created.

All blocks in state MOD-EXC must be written back to main memory upon replacement, although for those blocks written exactly once, the copy in memory will already be updated from the write through on the first (and only) write. There is no way of distinguishing between one

and more than one write using only these three states. The next scheme described, write-once, provides a method for eliminating these unnecessary write backs through the addition of a new state.

Because of the fixed one-word block, the read broadcast scheme is not simulated, although its performance will be discussed qualitatively. All of the simulated schemes are analyzed with block sizes of multiple words, making comparison difficult, since having identical block sizes is an important assumption in the workload model. The other schemes were not intended for one-word blocks, although adaptions could certainly be made. Alternately, the RB scheme could be modified to work with block sizes of multiple words. However, it was felt important to simulate the protocols *exactly* as they were proposed.

4.2.3 Write-once

Chronologically the first scheme described in the literature[Goo83], Goodman's write-once scheme was designed for single-board computers using Multibus. The requirement that the scheme work with an existing bus protocol was a severe restriction but one that results in implementation simplicity. In the write-once scheme, blocks in the local cache can be in one of four states.

1. INVALID. (INV)
2. UNMODIFIED-SHARED. (UNMOD-SHD)
3. UNMODIFIED-EXCLUSIVE. (UNMOD-EXC) No other caches have a copy. Block is not modified with respect to memory.
4. MODIFIED-EXCLUSIVE. (MOD-EXC)

Write-once adds the UNMOD-EXC state to the basic set of three used in the Synapse and RB protocols. A state transition diagram for the scheme is given in Figure 4.5. The write-once protocol works as follows:

- **Read miss.** If another copy of the block exists that is in state MOD-EXC, the cache with that copy inhibits the memory from supplying the data and supplies the block itself as well as writing the block back to main memory. If no cache has a MOD-EXC copy the block comes from memory. All caches with a copy of the block set their state to UNMOD-SHD.
- **Write hit.** If the block is already MOD-EXC the write can proceed locally without delay. If the block is in state UNMOD-EXC, the write can also proceed without delay, and the state is changed to MOD-EXC. If the block is in state UNMOD-SHD, a write through takes place

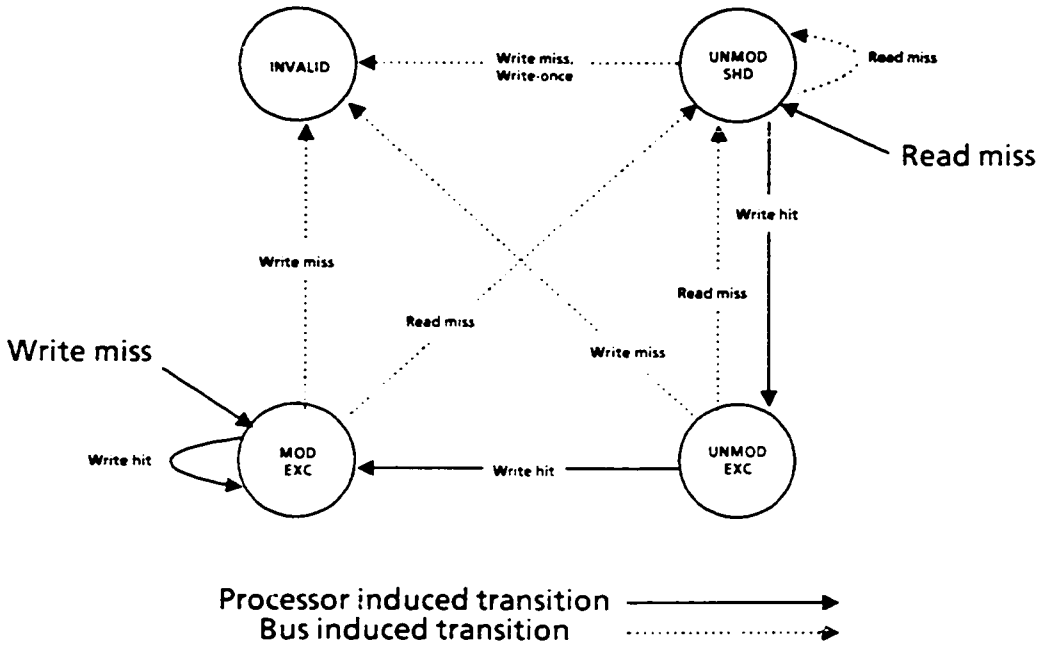


Figure 4.5: Write-once state transition diagram

and the local state is set to UNMOD-EXC. Other caches with a copy of that block (if any) observe the bus write and change the state of their block copies to INV.

- **Write miss.** Like a read miss, the block is loaded from memory, or, if the block is MOD-EXC, from the cache which has the MOD-EXC copy, which must then set its local state to INV. Upon seeing the write miss on the bus, all other caches with the block also change their local states to INV. Once the block is loaded, the write takes place and the state is set to MOD-EXC.

The write-once scheme gets its name from the single write-through performed on the first write on a previously unmodified block. Each write-through functions very much like an invalidation signal, but because memory is updated, blocks written exactly once do not require a write back when replaced, unlike the other schemes considered here.

4.2.4 Berkeley

This approach is to be implemented in a RISC multiprocessor currently being designed at the University of California at Berkeley[KEW*85]. The following states are used:

1. INVALID. (INV)
2. UNMODIFIED-SHARED. (UNMOD-SHD)
3. MODIFIED-SHARED. (MOD-SHD) Other caches may have copies. Block needs to be written back when replaced.
4. MODIFIED-EXCLUSIVE. (MOD-EXC)

Figure 4.6 contains a state transition diagram for this protocol. In Berkeley terminology, a cache with a MOD-SHD or MOD-EXC copy is called the owner. There can be at most one such cache. Although a MOD-SHD copy indicates that other caches may have copies, the protocol ensures that they will be UNMOD-SHD copies. Thus, there is at most one owner cache, and the owner is responsible for writing the block back to memory. If a block is not owned by any cache, memory is the owner. The protocol works as follows:

- **Read miss.** If the block is MOD-EXC or MOD-SHD, the cache with that copy must supply the block contents directly to the other cache and set its local state to MOD-SHD. If the block is in any other state or not cached, it is loaded from main memory. In any case, the block state in the requesting cache is set to UNMOD-SHD. Note that the block always comes directly from its owner.
- **Write hit.** If the block is already MOD-EXC, the write proceeds with no delay. If the block is UNMOD-SHD or MOD-SHD, an invalidation signal must be sent on the bus before the write is allowed to proceed. All other caches invalidate their copies upon matching the block address and the local state is changed to MOD-EXC in the originating cache.
- **Write miss.** Like a read miss, the block comes directly from the owner. All other caches with copies change the state to INV and the block in the requesting cache is loaded in state MOD-EXC.

In the Berkeley scheme, the MOD-SHD state is used to allow efficient cache-to-cache transfers of MOD-EXC blocks on read misses in other caches. The state allows a cache to retain ownership (and write-back responsibility) without a costly update of main memory. Note that, despite the presence of a MOD-SHD state, the Berkeley protocol always requires that the block be in state MOD-EXC for a write, ensuring an exclusive copy.

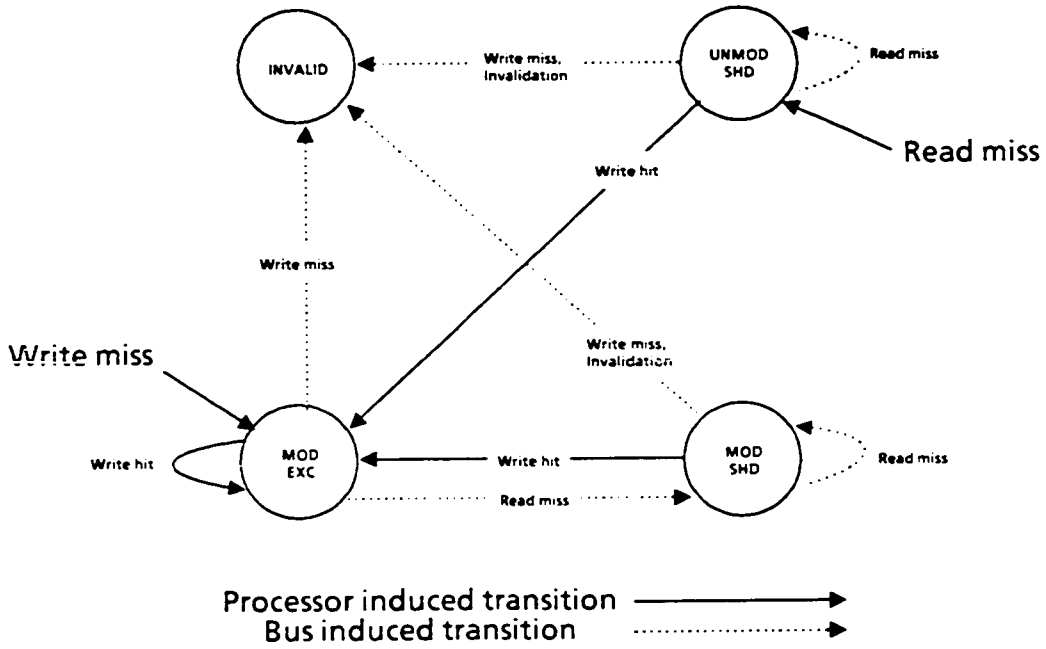


Figure 4.6: Berkeley state transition diagram

4.2.5 Illinois

This approach [PJ84] assumes that missed blocks always come from other caches, if any copies are cached, and from memory if no cache has a copy, and it is also assumed that the requesting cache will be able to determine the source of the block. Each time a block is loaded it can therefore be determined if it is shared or not. This information can eliminate invalidation signals for write hits on unmodified blocks that are not shared, boosting system performance. The scheme uses the same four states for cached blocks as the write-once protocol, namely:

1. INVALID. (INV)
2. UNMODIFIED-SHARED. (UNMOD-SHD)
3. UNMODIFIED-EXCLUSIVE. (UNMOD-EXC)
4. MODIFIED-EXCLUSIVE. (MOD-EXC)

The operation of the Illinois protocol is described below. Figure 4.7 contains a state transition diagram.

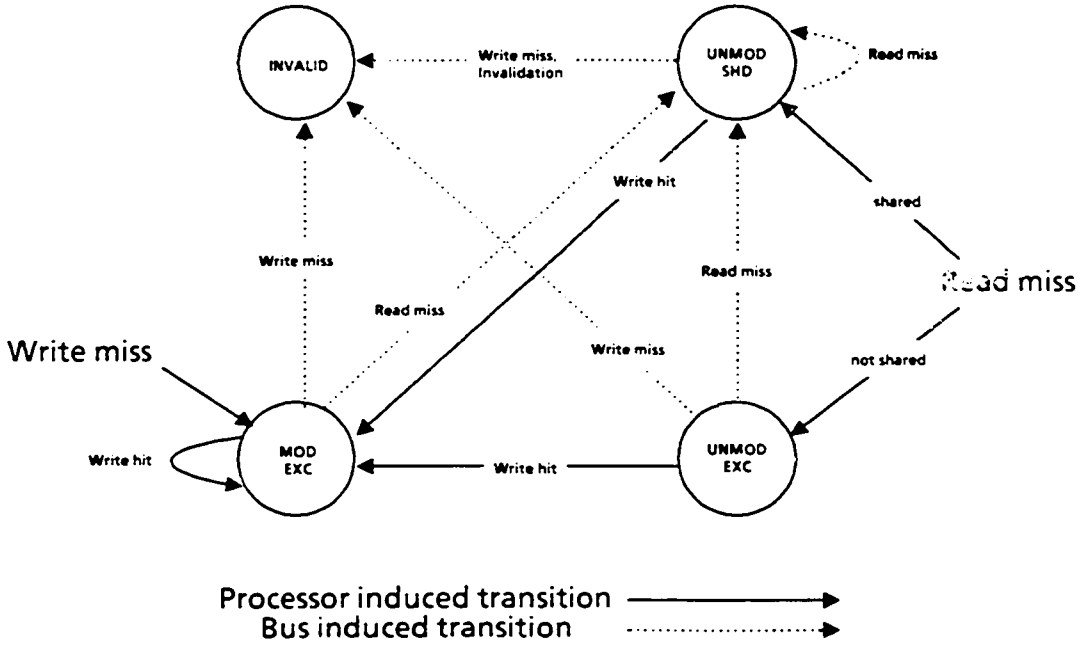


Figure 4.7: Illinois state transition diagram

- **Read miss**. If any other cache has a copy of the block, it tries to put it on the bus. If the block is MOD-EXC, it is also written to main memory at the same time. If the block is UNMOD-SHD, one cache (decided by a bus arbiter) will succeed in putting the block on the bus. All caches with a copy of the block will observe the miss and set their local states to UNMOD-SHD, and the requesting cache sets the state of the loaded block to UNMOD-SHD. If the block comes from memory, no other caches have the block, so the block is loaded in state UNMOD-EXC.

- **Write hit**. If the block is MOD-EXC, it can be written without delay. If the block is UNMOD-EXC, it can be written immediately with a state change to MOD-EXC. If the block is UNMOD-SHD, the write is delayed until an invalidation signal can be sent on the bus, which causes all other caches with a copy to set their state to INV. The writing cache can then write the block and set the local state to MOD-EXC.

- **Write miss**. Like a read miss, the block comes from a cache, if any cache has a copy of the block. All other caches with copies change their state to INV and the block is loaded in state MOD-EXC.

Although the Illinois scheme has the same states as write-once, the UNMOD-EXC state is used very differently. In the write-once protocol blocks are in this state only after being written exactly once. In this scheme any block obtained directly from memory on a read miss (and therefore not shared between caches) is loaded in this state.

4.2.6 Futurebus Write-Once

This protocol is an adaptation of the write-once scheme using the added capabilities of the Futurebus [Goo86]. The states used are identical to those of the basic write-once and Illinois schemes, namely:

1. INVALID. (INV)
2. UNMODIFIED-SHARED. (UNMOD-SHD)
3. UNMODIFIED-EXCLUSIVE. (UNMOD-EXC)
4. MODIFIED-EXCLUSIVE. (MOD-EXC)

The state transitions for this protocol are illustrated in Figure 4.8. Although it uses the same states that are used in the basic write-once protocol, the use of the UNMOD-EXC state is very different. The Futurebus version uses the UNMOD-EXC state the same as the Illinois protocol, although the actual implementation is different. The Futurebus provides a *SHARED* line on the bus that is used to detect sharing on read misses. The protocol functions as follows:

- **Read miss.** The block comes from memory unless some cache has a MOD-EXC copy. If such a copy exists, the cache with the copy supplies the data, raises the *SHARED* line, updates main memory, and changes its state to UNMOD-EXC. If the block is supplied by memory, all caches with valid copies set their local states to UNMOD-SHD and raise the *SHARED* line, indicating to the cache with the miss that the block is shared. The cache loading the block tests *SHARED*. If it is high the block is loaded in state UNMOD-SHD. If it is low the block state is set to UNMOD-EXC.
- **Write hit.** If the block is MOD-EXC, the write can proceed directly. If the block is UNMOD-EXC, it can be written immediately with a state change to MOD-EXC. If the block is UNMOD-SHD, the write is delayed until a write-through can be performed (invalidating other copies), following which the block is modified and the local state is set to MOD-EXC.
- **Write miss.** The block comes from memory unless a MOD-EXC copy exists, in which case the cache supplies the data and sets its state to INV. If the block comes from memory, all

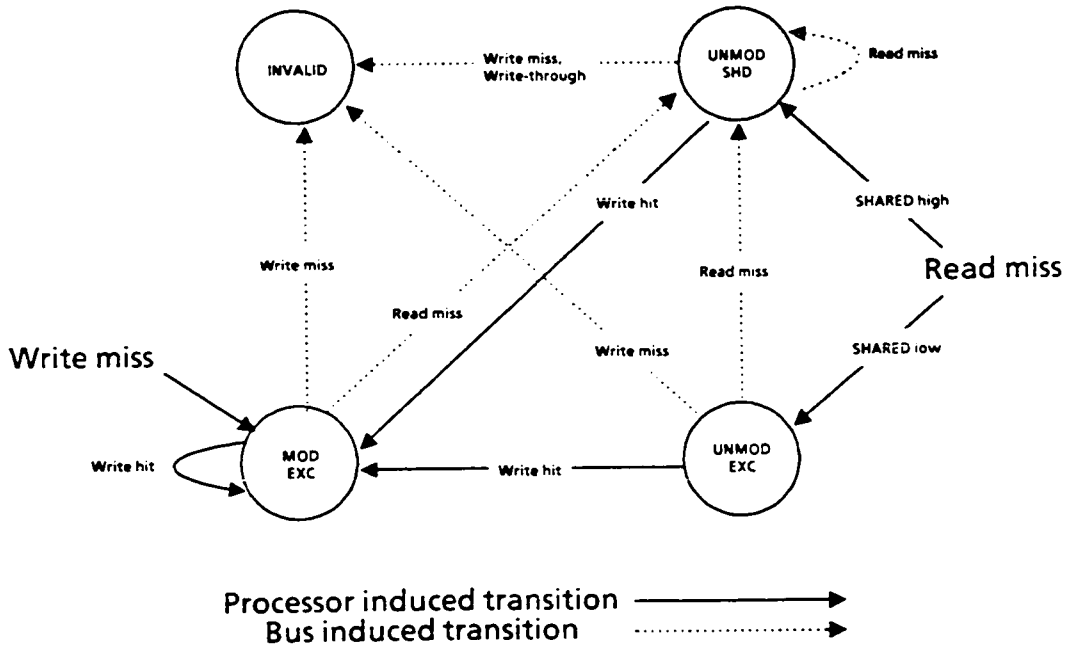


Figure 4.8: Futurebus write-once state transition diagram

caches with copies observe the write miss and mark their copies INV. The block is loaded in state MOD-EXC and the write may take place immediately.

The Futurebus provides a *SHARED* bus line that was not assumed in the basic write-once protocol. Note that blocks written exactly once in this version of write-once require a write-back as in other schemes, although a write-through occurs on the first write.

4.3 EIP: An Efficient Invalidation Protocol

The EIP protocol is a new invalidation protocol that we have designed that combines features of other protocols and new features intended to reduce the overhead of other invalidation protocols. A proof of correctness for the protocol is given in Chapter 8. EIP uses the following set of states:

1. INVALID. (INV)
2. UNMODIFIED-EXCLUSIVE. (UNMOD-EXC) Guaranteed only cached copy.
3. UNMODIFIED-SOURCE. (UNMOD-SRC) Possibly other copies cached. Cache provides data on read miss.

4. UNMODIFIED-SHARED. (UNMOD-SHD) Possibly other copies cached.

5. MODIFIED-SHARED. (MOD-SHD) Possibly other copies cached. Block must be written back.

6. MODIFIED-EXCLUSIVE. (MOD-EXC) Guaranteed only cached copy. Must be written back.

As can be seen, the states used are a combination of states from other schemes with the addition of the UNMOD-SRC state. This state, independently proposed by Bitar [Bit85, BA86], allows for efficient cache to cache transfers of unmodified blocks by guaranteeing that a unique source will respond to any miss. Its operation will be outlined in the following discussion. In order to better understand how the protocol works, consider the following definitions:

- Dirty owner: A cache with a copy in state MOD-SHD or MOD-EXC is called the *dirty owner*. There is at most one such cache–the last cache to write the block. If the block is in state MOD-SHD all other cached copies must be in state UNMOD-SHD. If the block is in state MOD-EXC no other cached copies exist. All blocks that are modified with respect to main memory have a dirty owner, and it is the dirty owner who is responsible to update main memory.

- Clean owner: A cache with a copy in state UNMOD-EXC or UNMOD-SRC is called the *clean owner*. There is at most one such cache–the last cache with a read miss on that block, unless it has since replaced the block, or unless the block has a dirty owner. If the block is UNMOD-SRC all other valid cached copies must be in state UNMOD-SHD. If the block is UNMOD-EXC no other cached copies exist. If no cache has a copy in either of these two states, memory is the clean owner. Note that all blocks in the system that are not modified in a cache have a unique clean owner, and that memory can be the clean owner although there are copies of the block in caches.

Correct operation of the protocol requires the bus to have two special lines to exchange information about shared blocks. The *SHARED* line indicates whether a block is shared or not. The *MODIFIED* line indicates whether a shared block is modified with respect to main memory and is meaningless unless the *SHARED* line is high. Their operation is described below. The operation of the protocol is described below. (The state transitions for EIP are summarized in Figure 4.9.)

- **Read miss.** If a modified copy of the block exists in the system, the block will be supplied by the dirty owner. Otherwise the block comes from the clean owner. If the block was

present in state MOD-EXC, that cache changes its copy to MOD-SHD. If the block was in state UNMOD-SRC or UNMOD-EXC it is changed to state UNMOD-SHD. All caches with a copy of the block raise the *SHARED* line, and if the block had a dirty owner, the dirty owner raises the *MODIFIED* line. In addition, all caches with INV copies take the block, raise the *SHARED* line, and set the state of their copies to UNMOD-SHD. (This validation feature is used in the CMU RB protocol.) The requesting cache tests the value of both lines to determine the state of the newly loaded block. If the *SHARED* line is raised, but the *MODIFIED* line is not, the block is loaded in state UNMOD-SRC. If both the *SHARED* and *MODIFIED* lines are raised the block is loaded in state UNMOD-SHD. If the *SHARED* line is not raised the block is loaded in state UNMOD-EXC. Note that the requesting cache becomes the clean owner unless there is a dirty owner.

- **Write hit.** If the block is in state MOD-EXC the write may proceed with no overhead. If the state is UNMOD-EXC the write may proceed immediately with a state change to MOD-EXC. If the block is in any other state an invalidation signal must be sent to all other caches, causing a state change to INV. The local state is changed to MOD-EXC and the write is allowed to proceed.

- **Write miss.** As in the case of a read miss, the block comes from the clean owner or dirty owner. All caches with a copy change their state to INV and the requesting cache loads the block in state MOD-EXC, thus becoming the dirty owner.

When a MOD-EXC or MOD-SHD block is replaced in the cache, it needs to be written back. During the write-back, while the valid data is on the bus, all caches with an INV copy take the valid data and update their copies, setting the state to UNMOD-SHD. Main memory becomes the clean owner, just as the case when a block is replaced in the clean owner cache.

The EIP, by virtue of the UNMOD-EXC state and the *SHARED* line, eliminates the coherence overhead on private blocks except that arising from task migration. The MOD-SHD state allows for efficient cache-to-cache transfers of modified blocks on read misses. The clean owner concept and the UNMOD-SRC state (along with the required *MODIFIED* line) allow for efficient cache-to-cache transfers of most unmodified blocks on read or write misses. Updating copies in the INV state on read misses or write-backs reduces the bus traffic resulting from misses caused by invalidations. Note that INV copies may not be updated on a write miss since the requesting cache must have an exclusive copy.

The advantage of clean ownership is that cache to cache transfers of unmodified blocks are simplified. If another cache is able to supply a block (in response to a read or write miss) much faster

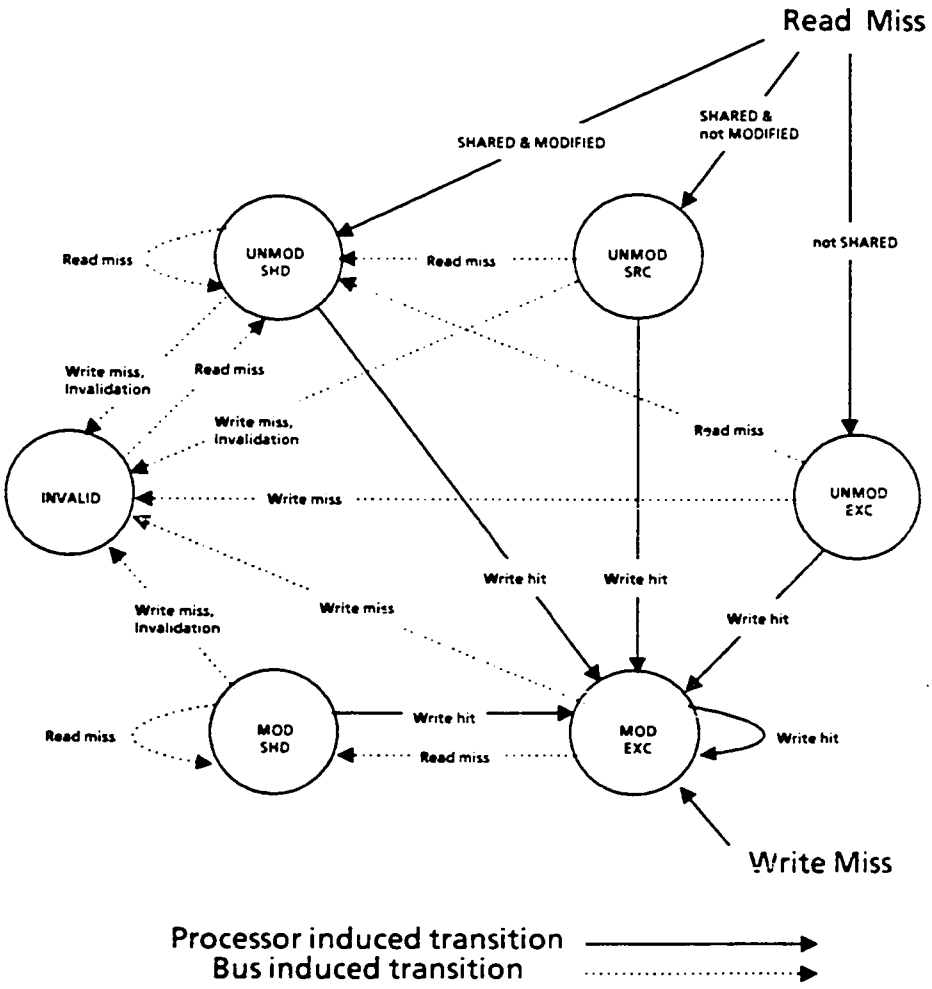


Figure 4.9: EIP state transition diagram

than memory, loading the block from a cache whenever possible will improve overall performance. In all other protocols, a single cache is responsible for supplying valid data when the block is modified with respect to memory, but only in this protocol is a single cache responsible for supplying valid data when the block is not modified–the cache with the UNMOD-SRC copy. It is possible that memory is the clean owner, but this occurs only if the last cache with a read miss has already replaced the block, and this is unlikely. Since there is always a unique block source on every miss, any overhead of bus arbitration is eliminated.

Figure 4.10 lists the processor-to-cache commands, bus transactions generated by caches, and a summary of all actions resulting in a state change in any cache. Note that the list of actions causing state changes includes all bus transactions and two events that are local to the cache. The first of these is the state change from UNMOD-EXC to MOD-EXC in response to a write. The second is the result of clean replacement and simply removes the block from the cache.

As shown in Figure 4.10, it is assumed that the processor has a test-and-set (TAS) instruction to provide synchronization, but this can be implemented without an additional type of bus transaction. For example, one possible implementation is based on the use of busy-wait and test-and-test-and-set (TATAS) as proposed by Rudolph and Segall[RZ84]. The first step is a simple read, testing the contents of the synchronization variable. Only if the contents indicate that the TAS could succeed will the cycle continue–otherwise the processor busy-waits until the contents are modified. If the contents permit, a write to the variable is attempted. If the copy is in an EXCLUSIVE state the lock can be modified without a bus transaction. If the copy is in a SHARED state, the write can take place only after an invalidation signal is sent on the bus. If the local copy is invalidated (and hence written elsewhere) while the cache is waiting to use the bus, the TATAS cycle must be restarted, beginning with a read miss to reload the block and a busy-wait until the modification may again be attempted. Thus, if multiple caches attempt to write a lock, the first to obtain the bus will succeed. The advantage of this approach is that the processor *spins* on reads to a locally cached copy and does not generate bus traffic, except to reload the locks modified elsewhere and to sucessfully modify locks. As far as the cache coherence protocol is concerned, the TATAS cycle appears as a read followed by a write to a previously unmodified block.

4.4 Performance of Invalidation Protocols

The performance of cache coherence protocols is very dependent on the sharing present in the workload. In the simulation model, the amount of *actual sharing* (defined as the percentage of processor references to blocks present in another cache at the time the reference is generated)

Processor-to-Cache Commands

Read word
Write word
Test-and-set word

Bus Transactions

Read block [Read miss]
Exclusive-read block [Write miss]
Invalidate block
Write-back block

Actions Causing State Change

Read block
Exclusive-read block
Invalidate block
Write-back block
Write on UNMOD-EXC block
Replace unmodified block

Figure 4.10: Specification of EIP system actions

depends mainly on three parameters: the number of S-blocks in the simulation, the percentage of S-block references, and the number of caches in the system. For a fixed percentage of S-block references, the actual sharing may vary significantly as the other parameters vary.

Figure 4.11 displays the actual sharing measured in simulation runs of the EIP and Illinois protocols with 5% of all processor references to S-blocks. With 16 S-blocks the actual sharing approaches 5%, indicating that S-blocks are generally present in another cache at each reference. With 1024 S-blocks only about one in ten S-block references is to a block that is present in another cache. All other invalidation protocols would be expected to exhibit nearly identical levels of actual sharing. (EIP might be expected to demonstrate slightly more sharing due the validation feature, but the results show little difference.) For all experiments with 5% S-block references, the actual sharing is very similar to that shown in Figure 4.11. For experiments using 0.1% S-block references, there is virtually no sharing; at most one reference in each simulation run (with over 48,000 total references) was to an S-block present in another cache.

Figures 4.12 through 4.19 display the results of several simulation experiments that have been run. Each figure includes a curve for each simulated invalidation protocol. (For reasons that will be explained shortly, there are two write-once curves included in each figure.) Figure 4.12 shows the performance of the protocols with a negligible amount of sharing. Figures 4.13 through 4.15 indicate the performance of the protocols with what might be called the *normal* parameter values. These three figures differ from each other only in the number of S-blocks used in the simulation. As indicated in Figure 4.11, there is little actual sharing when 1024 S-blocks are used. As a result, Figure 4.15 (using 1024 S-blocks and 5% S-block references) is very similar to Figure 4.12 (using 1024 S-blocks and 0.1% S-block references). In general, all experiments using 1024 S-blocks yielded similar results, so far as the relative performance of the curves is concerned.

The remaining figures were obtained using 16 S-blocks (and the associated high level of sharing) to emphasize the differences among the protocols. Figure 4.16 indicates the results obtained when the frequency of writes is increased from 15% to 30%. Figure 4.17 shows the performance when the cache size is increased significantly and the P-block hit ratio is increased from 95% to 98%. Figure 4.18 demonstrates the performance when the block size is reduced from four words to two words, and Figure 4.19 shows the results with the block size increased to eight words.

As can be seen, the performance of the invalidation protocols can be quite different, but the ordering of the schemes is rather consistent over all the simulation experiments. EIP is consistently the best, followed by Berkeley and Illinois. Below these two come the write-once protocols. The Synapse protocol is consistently at the bottom. The observed differences in performance are the

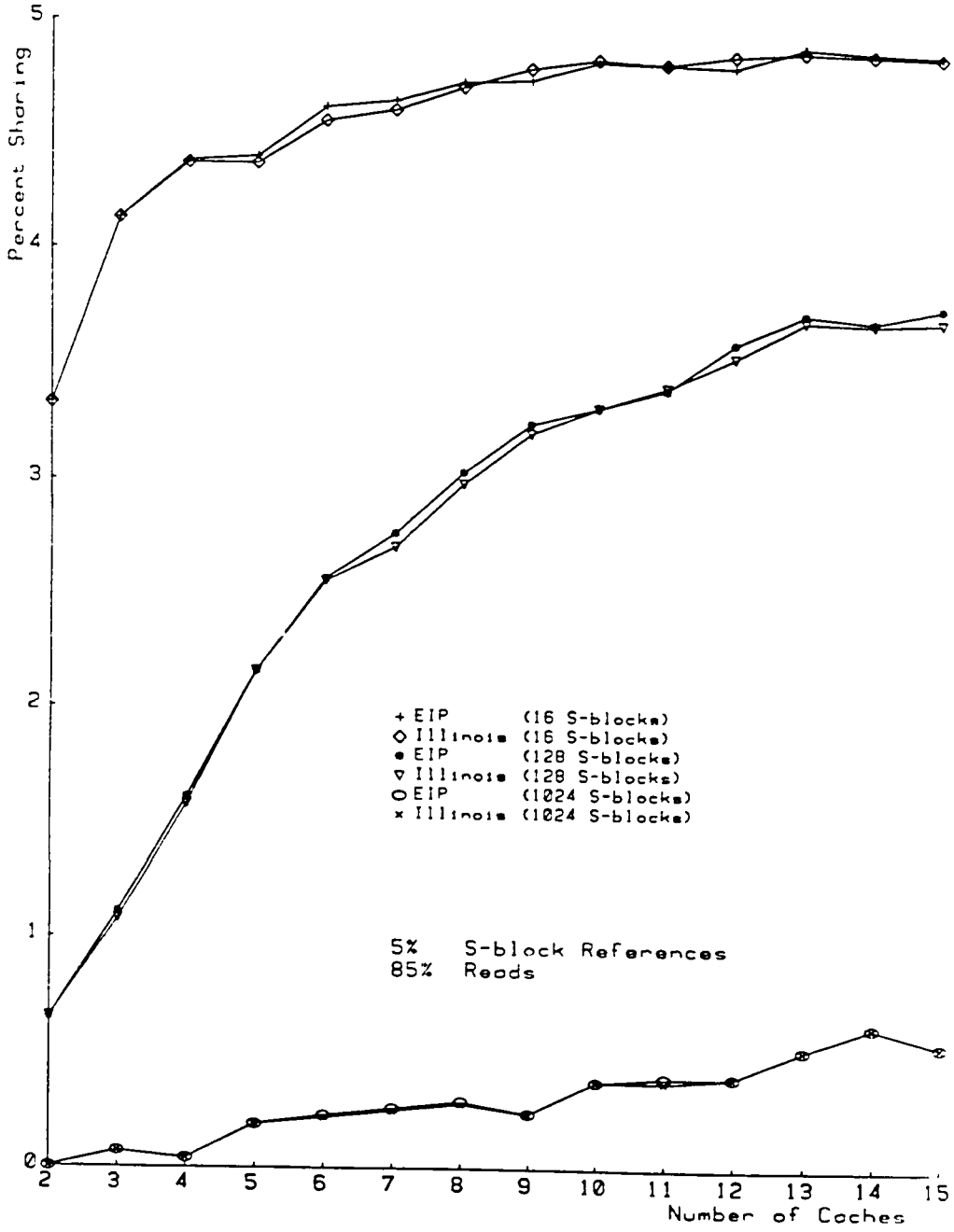


Figure 4.11: Inv: actual sharing

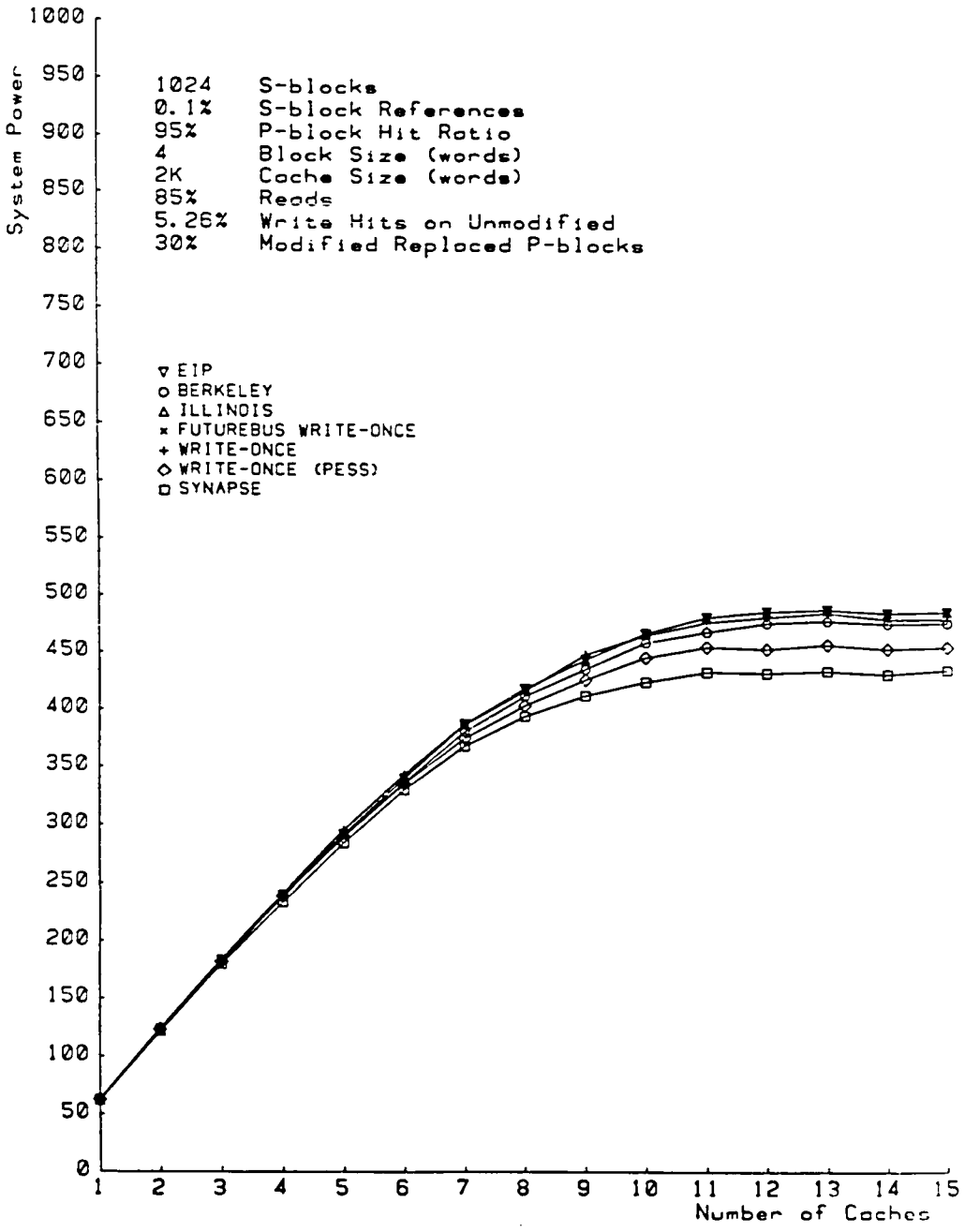


Figure 4.12: Inv: performance with negligible sharing

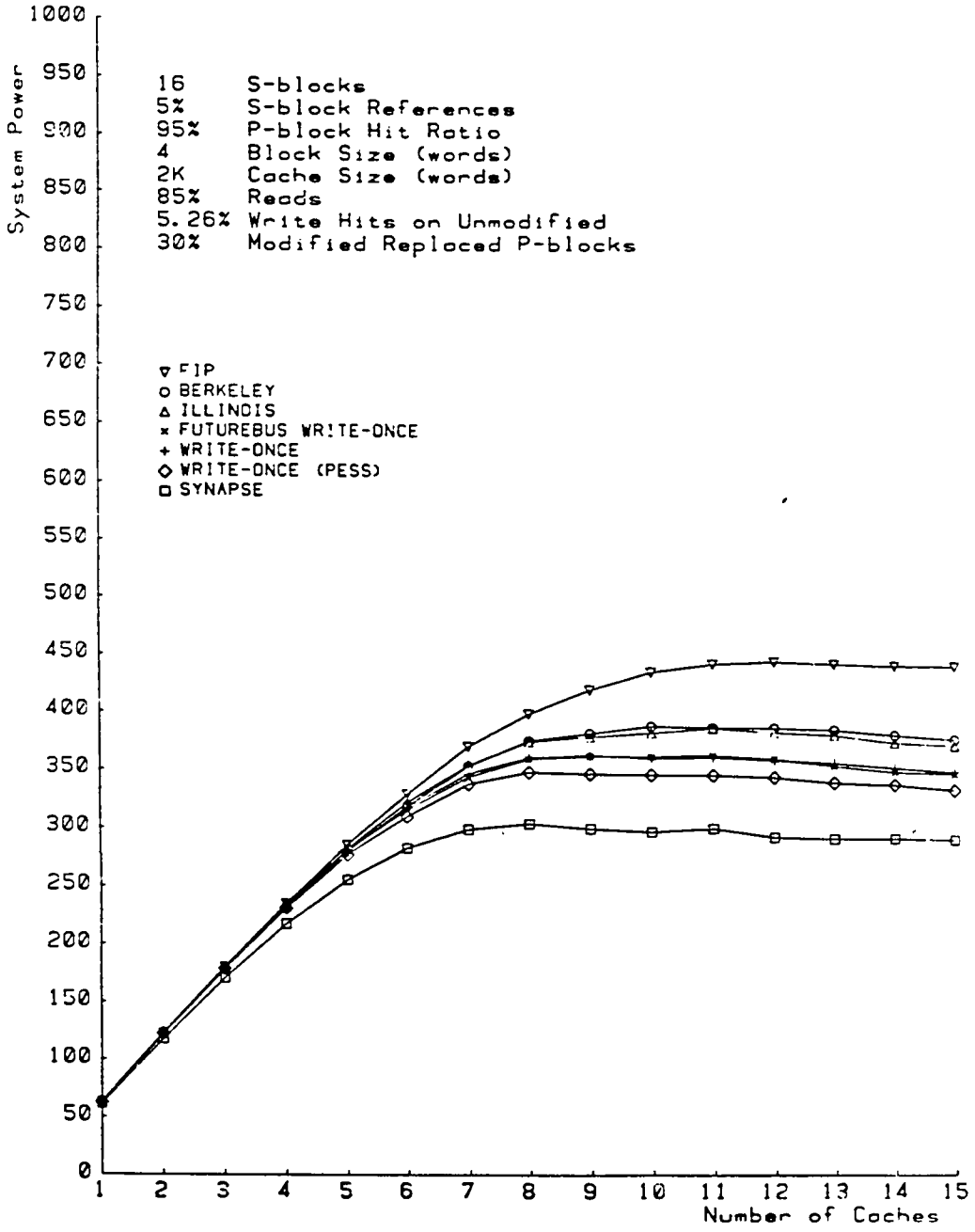


Figure 4.13: Inv: basic model, 16 S-blocks

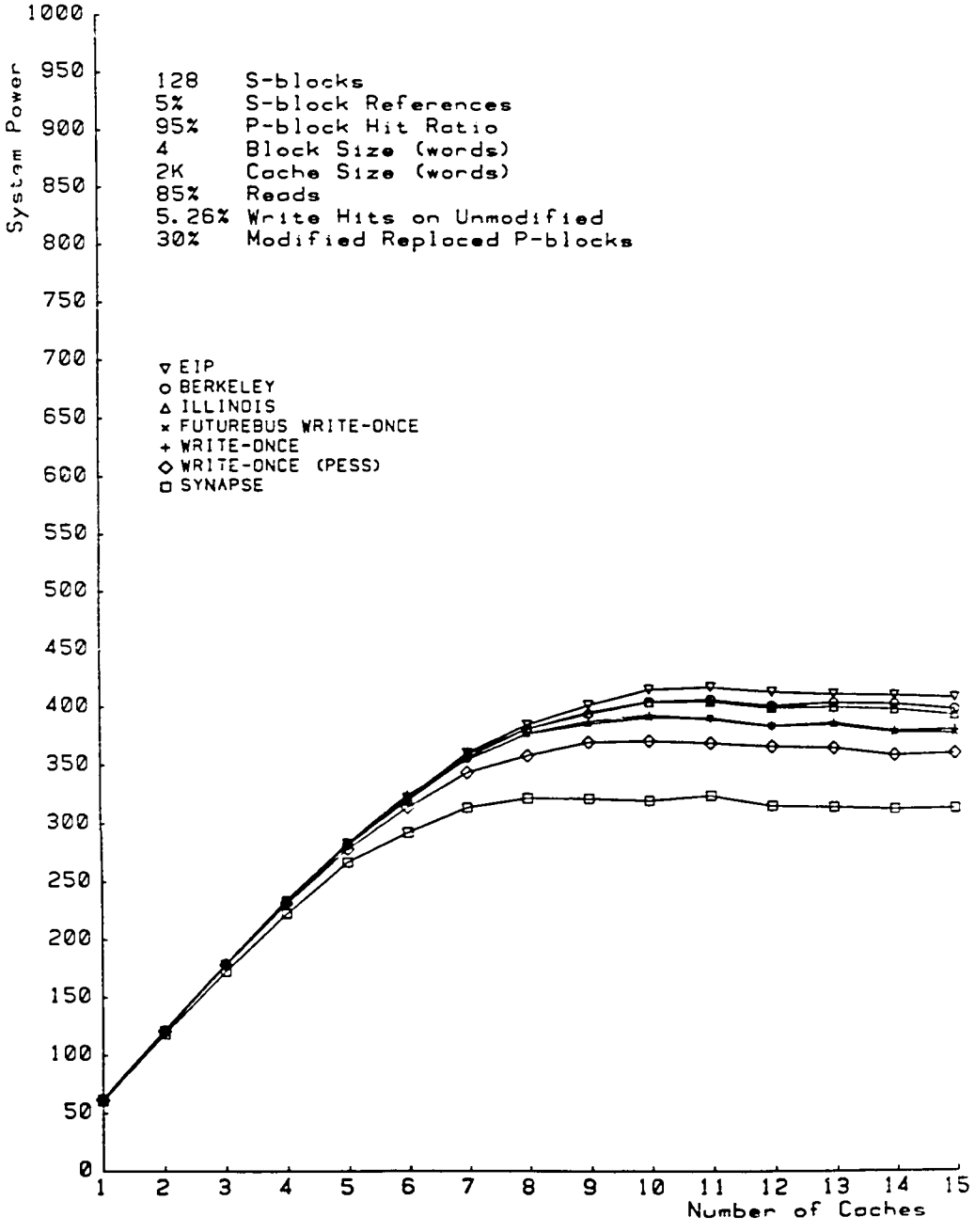


Figure 4.14: Inv: basic model, 128 S-blocks

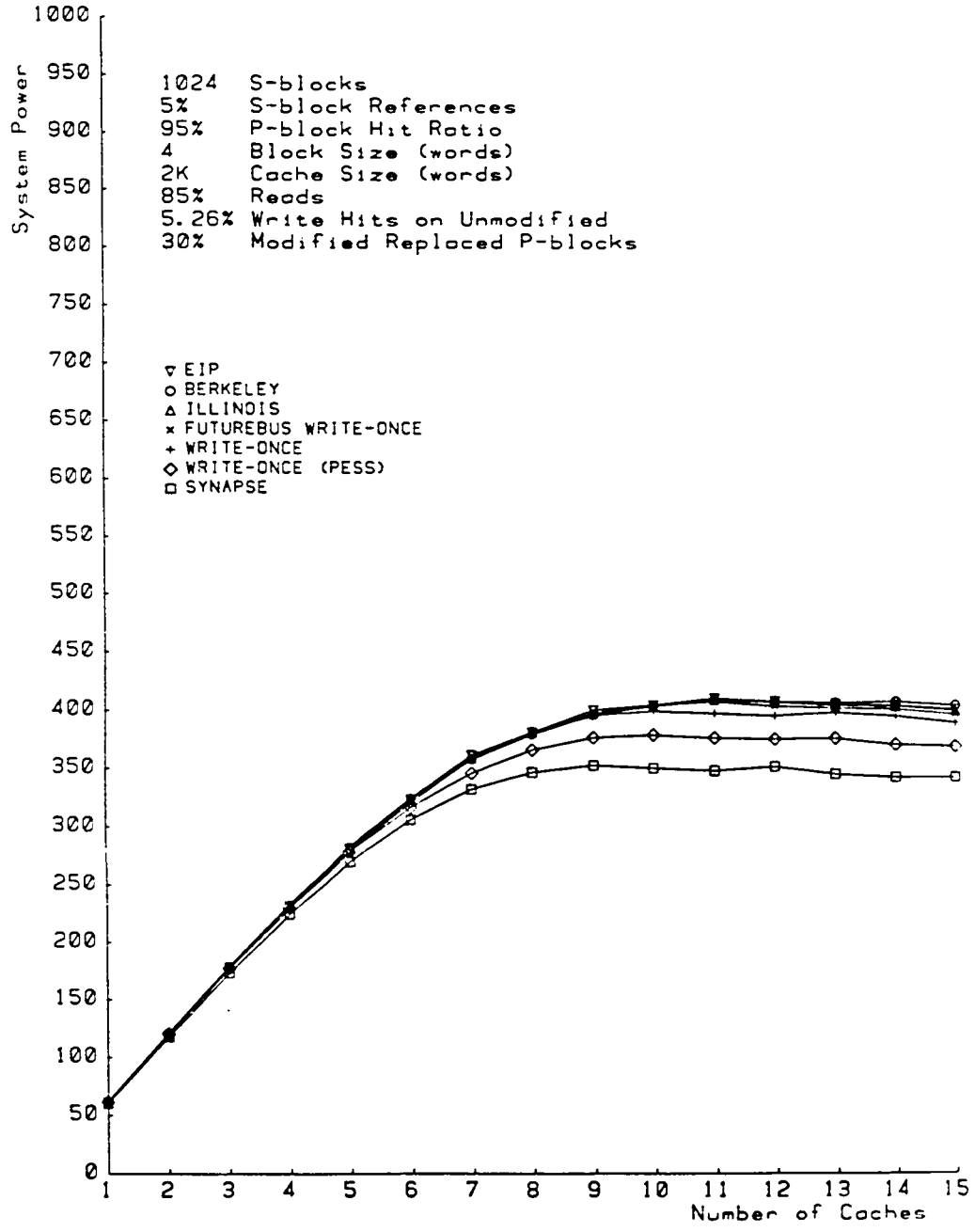


Figure 4.15: Inv: basic model, 1024 S-blocks

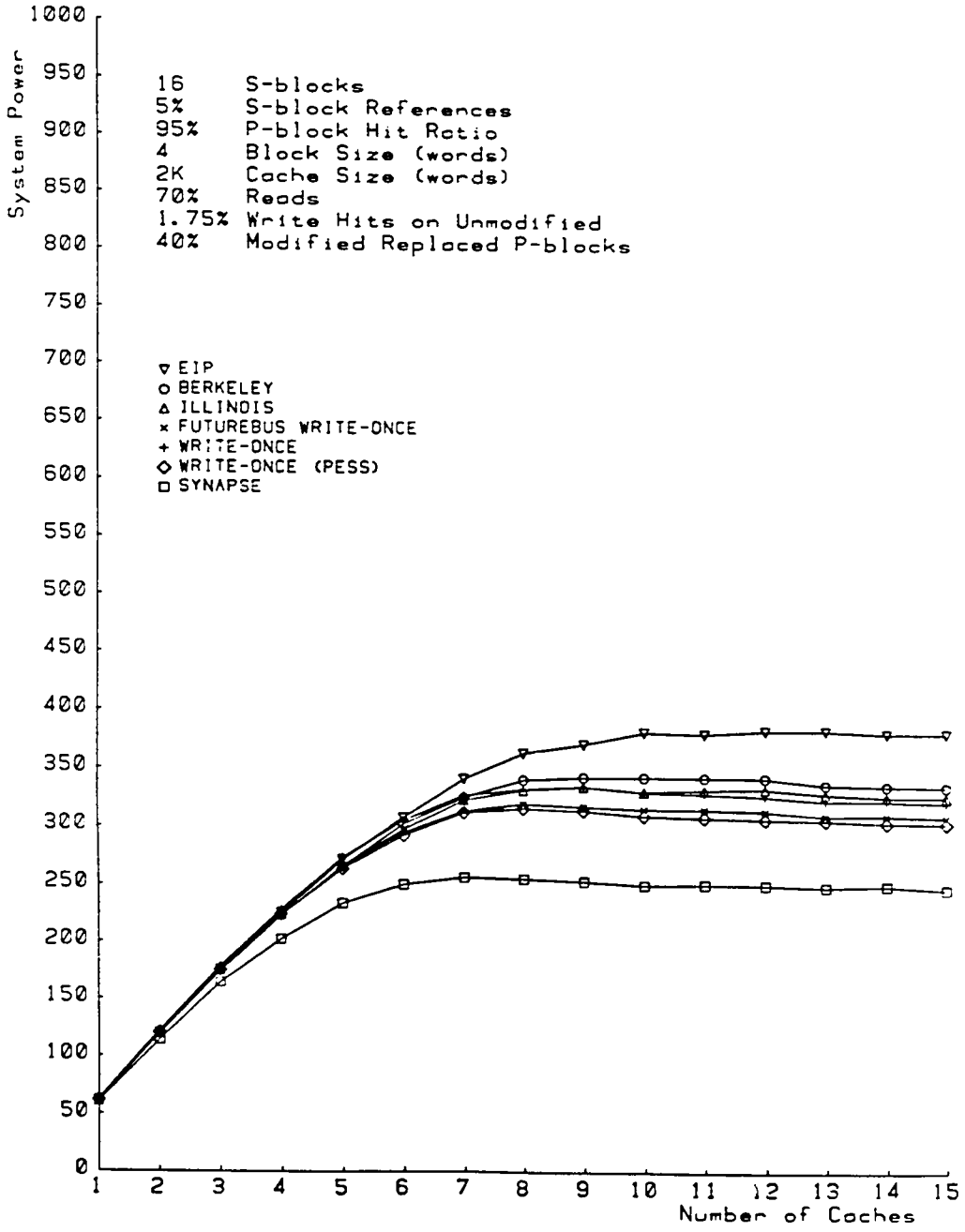


Figure 4.16: Inv: increased write ratio

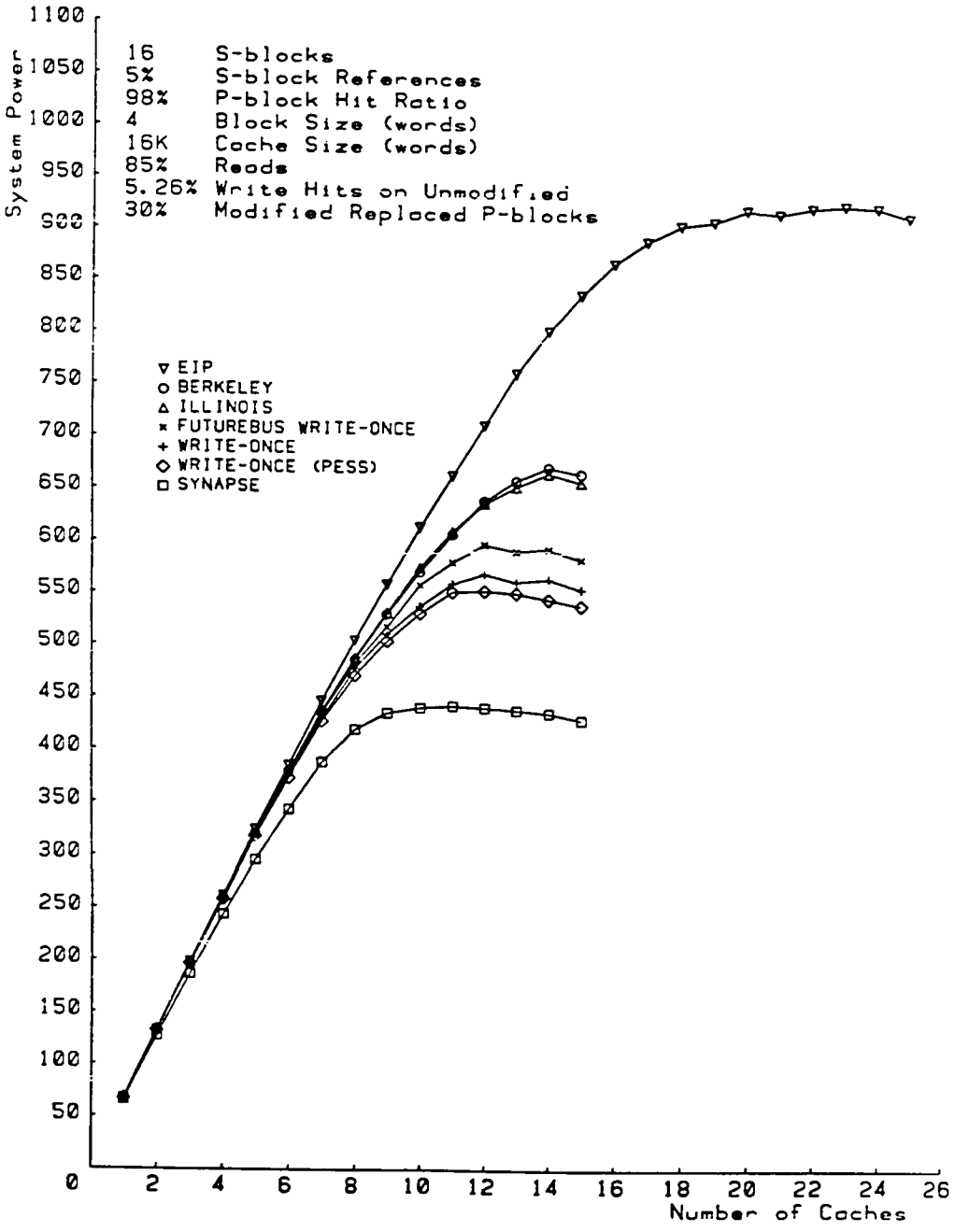


Figure 4.17: Inv: increased cache size

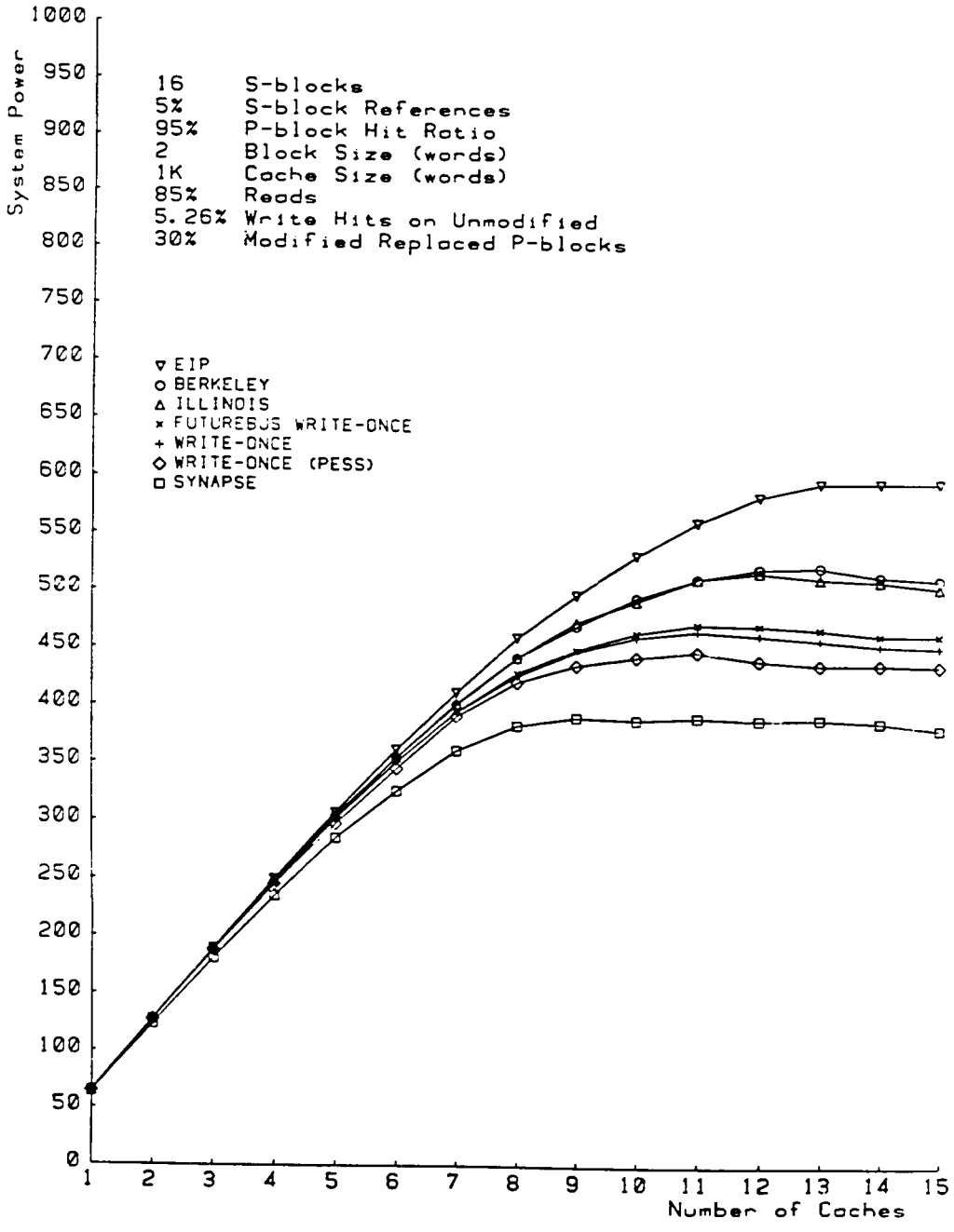


Figure 4.18: Inv: 2 word block size

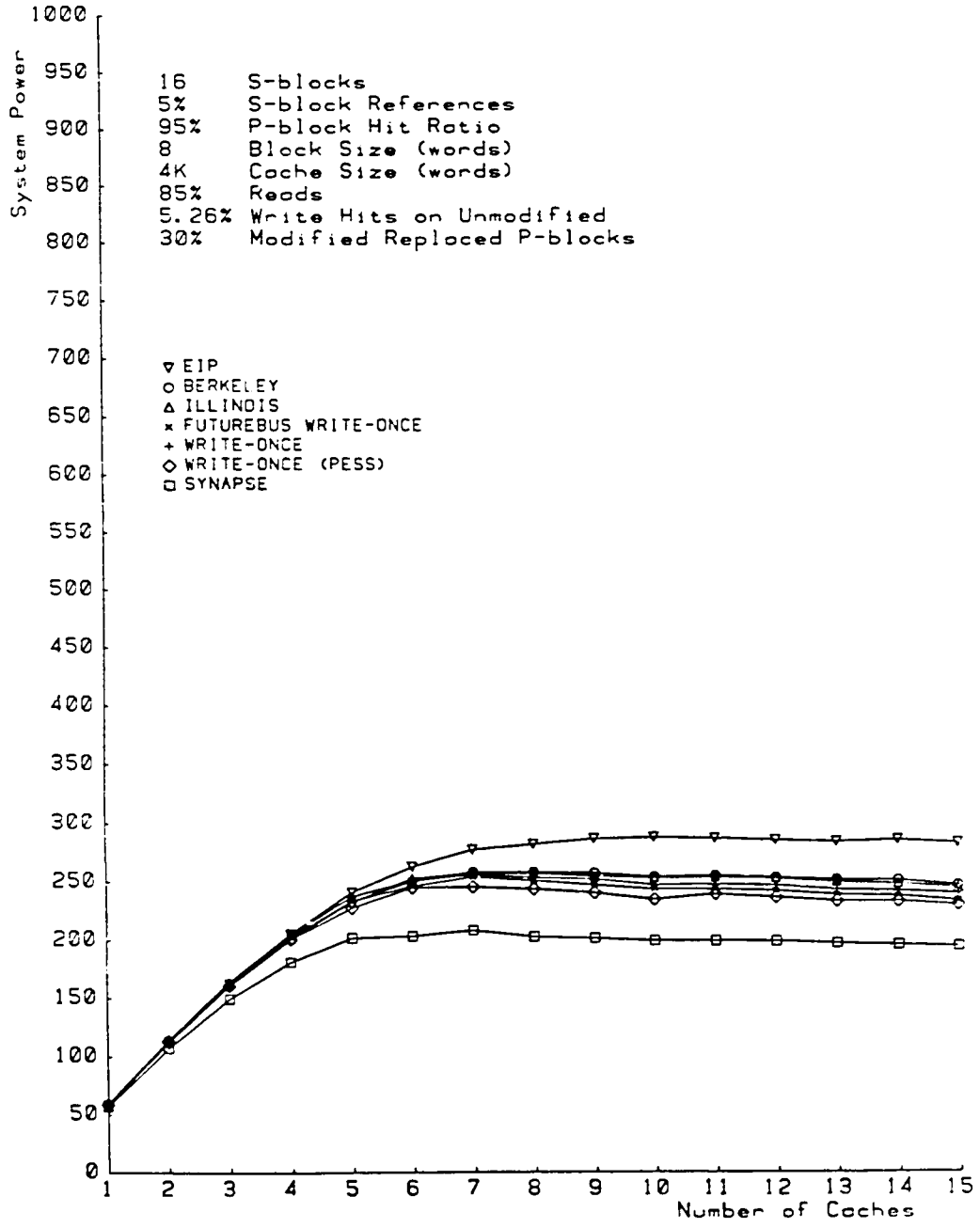


Figure 4.19: Inv: 8 word block size

results of protocol differences in the handling of *private* and *shared* blocks. For purposes of our discussion here, a *shared* block is actually present in at least one other cache, and *private* blocks are not present in other caches. Note that the terms 'shared' and 'private' are defined in terms of the instantaneous state of the block and not in terms of their reference patterns. Thus, blocks referenced only by a single process might be 'shared' temporarily if the process migrates to a new processor (although this overhead is not reflected in the simulation results). In addition, blocks referenced by several processes but not actually present in another cache will be 'private' until they are loaded into a second cache. The handling of private and shared blocks will be examined in detail in the following sections.

4.4.1 Efficiency of Private Block Handling

Since most references are to private blocks, efficiency in the handling of private blocks can be more critical than the efficiency in the handling of infrequently accessed shared blocks. Figure 4.12 shows the performance of the protocols with virtually no sharing. All differences between the curves are the result of differences in the handling of private blocks.

There are only two differences between the invalidation protocols in the handling of private blocks. These are the actions required with write-hits on unmodified blocks and actions required when a block is replaced in the cache. Any overhead associated with write hits on unmodified blocks is logically unnecessary since they are never present in another cache. However, only the EIP, Illinois, and Futurebus write-once are able to detect that a block is not shared when it is loaded into the cache on a read miss. Note that each of the three has an UNMOD-EXC state used for this purpose. (The basic write-once scheme also has an UNMOD-EXC state but it is not able to use it in this fashion.) Without this dynamic sharing detection the overhead can be substantial. The Synapse scheme performs a complete block load from memory as if a write miss had occurred. Write-once (and CMU RB) require a single word write to main memory. Berkeley requires a single bus cycle for an invalidation signal.

The difference in block replacement is a reduction of write-backs in the write-once protocol. With the exception of write-once, all other schemes are identical to each other. In write-once, private blocks modified exactly once do not require a write-back at replacement. The overall performance of write-once is very sensitive to the amount of write-back reduction, since it comes at the expense of increased overhead on write hits on unmodified blocks. Because there are varying estimates of the percentage of write backs that will be eliminated, results from two different versions of write-once are displayed in each figure. The first assumes that 33% of the write-back traffic of

private blocks is eliminated. The second, labelled write-once (pess), uses the more pessimistic assumption that the reduction in write-backs required is only 5%. It is likely that the normal operation of write-once would fall somewhere in between the two extremes.

Figure 4.12 indicates that those schemes with no overhead on write hits on private blocks are identical in the handling of private blocks. If 33% of the writebacks are eliminated, the performance of write-once is just slightly lower than the best. However, if the reduction is only 5%, the performance falls below that of Berkeley. With a 33% reduction, the additional overhead of first time writes pays off with greatly reduced write-back traffic. With a 5% reduction, the savings in write-backs is not sufficient to compensate for the increased write through overhead. The performance of Synapse is well below all other schemes because of the tremendous overhead of treating write hits on unmodified as write misses.

Although it was not simulated, the private block efficiency of the CMU RB scheme falls between the pessimistic write-once curve and Synapse. On write hits on unmodified blocks the RB scheme requires a write-through (as does write-once) but with no corresponding reduction in write-backs. The scheme therefore has somewhat more overhead than write-once, but less than Synapse.

4.4.2 Efficiency of Shared Block Handling

The remaining cause of performance differences between the schemes is overhead in the handling of shared blocks. A comparison of Figure 4.12 with Figures 4.13 through 4.15 shows the impact of increased sharing. The only difference between the four figures is the amount of sharing. As the level of sharing increases, the differences in performance between the protocols become more pronounced.

There are several differences in the handling of shared blocks. These include: read miss, write miss, write hits on unmodified, and replacement. On read and write misses with at least one unmodified copy in another cache (but no modified copy cached), the Illinois scheme always obtains a copy from another cache, while other schemes (write-once, Berkeley, and Synapse) always load the block from memory. EIP usually obtains the block from another cache, and occasionally it is obtained from memory. Read misses on blocks modified in another cache require a memory update at the same time in those schemes without a MOD-SHD state (Illinois, write-once, and Synapse). In addition, in EIP, other caches with INVALID copies will take the data and update their local states. Write hits on unmodified blocks require either an invalidation signal (EIP, Berkeley, Illinois), or a write-through (write once, CMU RB), or a complete block load (Synapse). The actions required when blocks are replaced in the cache are affected by the other actions of the

protocol. For example, the write-once protocol requires no write-back of blocks written exactly once although other protocols will require a write-back. In those schemes which update memory on read misses on modified blocks, there is no write-back required when the block is replaced unless it is again modified. Those schemes with a MOD-SHD state will not update memory (on read misses on modified blocks) and must therefore perform a write-back when the block is replaced.

Overall, the best performance is given by the EIP scheme. Its improvement over the other protocols is greatest with 16 S-blocks (and therefore with a higher level of sharing). Note that the performance of EIP actually declines as the number of S-blocks increases. For all other invalidation schemes, the performance improves as the number of S-blocks increases (and as the contention for shared blocks decreases). For high levels of sharing, normal invalidation protocols experience a severe reduction in the hit ratio on S-blocks. As the sharing declines, this invalidation effect decreases and overall system performance improves. In the EIP, the validation of INVALID copies on read misses and write-backs drastically reduces the negative effects of invalidations on S-blocks. System performance with 16 S-blocks is better than with 1024 because the S-block hit ratio is higher with 16 because of the increased locality of S-block references. For the other invalidation schemes the S-block hit ratio is actually lower with 16 S-blocks than with 1024 because of the invalidations.

Although the Illinois private block efficiency was superior to Berkeley, the overall performance of Berkeley surpasses the Illinois at high levels of sharing. In the Berkeley scheme read misses on blocks present and modified in other caches do not require an update of main memory. In the Illinois, blocks are obtained from caches whenever possible (requiring two less cycles to service in the simulation) while in Berkeley they come from memory unless a modified copy exists. Apparently, the Berkeley MOD-SHD advantage outweighs those of the Illinois protocol at high levels of sharing.

The write-once scheme yields performance below the Berkeley because it must update memory on each read miss on a modified block. In addition, the single word write to main memory (on write hits on unmodified) appears to create more overhead than it saves in reducing write-backs. However, with a high write percentage and a high write-back percentage (as shown in Figure 4.16), the relative performance of write-once increases, as the 33% write-back reduction becomes much more significant.

The performance of the Futurebus write-once protocol is comparable to write-once with 33% write-back reduction. It is usually at least slightly higher, but it is also occasionally lower. Like basic write-once, the Futurebus write-once must update memory on read misses on modified blocks and perform a write-through on write hits on unmodified blocks, although there is no corresponding reduction in write-backs.

The performance of Synapse is considerably lower than the other protocols. This is, in part, due to the increased overhead of requiring the requesting cache to resubmit the read miss when a cache with a modified copy interrupts the first miss and performs a write-back to update main memory. The overhead of treating write hits on unmodified blocks as write misses also contributes to the lower performance.

EIP performance exceeds that of the other protocols for a number of reasons (including a higher implementation cost, cf. section 4.8). It is the only protocol that includes both an UNMOD-EXC state and a MOD-SHD state, providing the benefits of writing all exclusive copies without delay *and* efficient cache to cache transfers of modified blocks. It also offers the advantage of reduced invalidation overhead, achieved through the mechanism of validation of shared blocks. Figure 4.20 shows the effect of validation on the S-block hit ratio of EIP compared with the normal invalidation protocols (represented here by Berkeley and Illinois). As can be seen, validation has a tremendous impact on the efficiency of shared block handling.

EIP also offers performance advantages over schemes that always load blocks from memory (unless modified), and it offers implementation advantages over schemes that always load shared blocks from other caches. To demonstrate the performance gain resulting from clean ownership, two modified versions of EIP were simulated and compared with EIP using clean ownership. Figure 4.21 shows curves representing a version that always loads shared blocks from other caches (ignoring any arbitration overhead) and a second version in which blocks are always loaded from memory unless modified. These are both compared against the EIP using clean ownership. Results are shown using main memory cycle times of four and eight cycles.

As can be seen, it is somewhat slower getting blocks from memory than from a cache. In the simulation, each shared block loaded from another cache saves two cycles with a four cycle memory (compared with loading from main memory), and blocks loaded from caches with an eight cycle memory save six cycles. Surprisingly, the difference between loading from cache and loading from memory remains about the same even when the memory cycle is doubled. Apparently this is a sufficiently rare event that increasing its efficiency has little impact on system performance. It is possible that a high percentage of blocks missed are modified in another cache, in which case all versions obtain the block directly from the owner cache. The addition of a shared read-only data class to the simulation would increase the difference between the cache and memory approaches. Based on the simulation results, the performance of EIP using clean ownership is indistinguishable from the scheme always loading from the cache. Very few, if any, shared blocks are loaded from memory when cached copies exists.

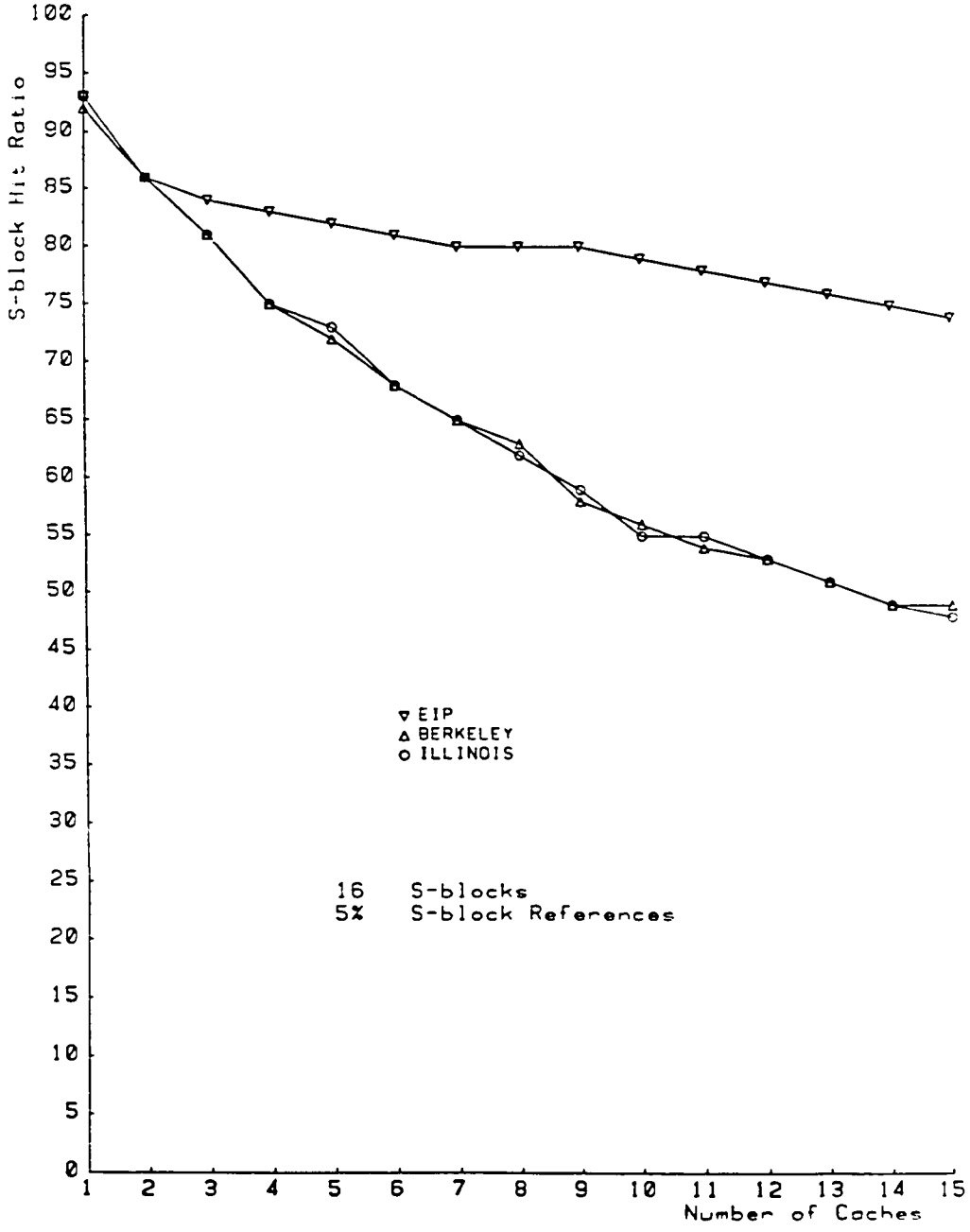


Figure 4.20: EIP: advantage of validation

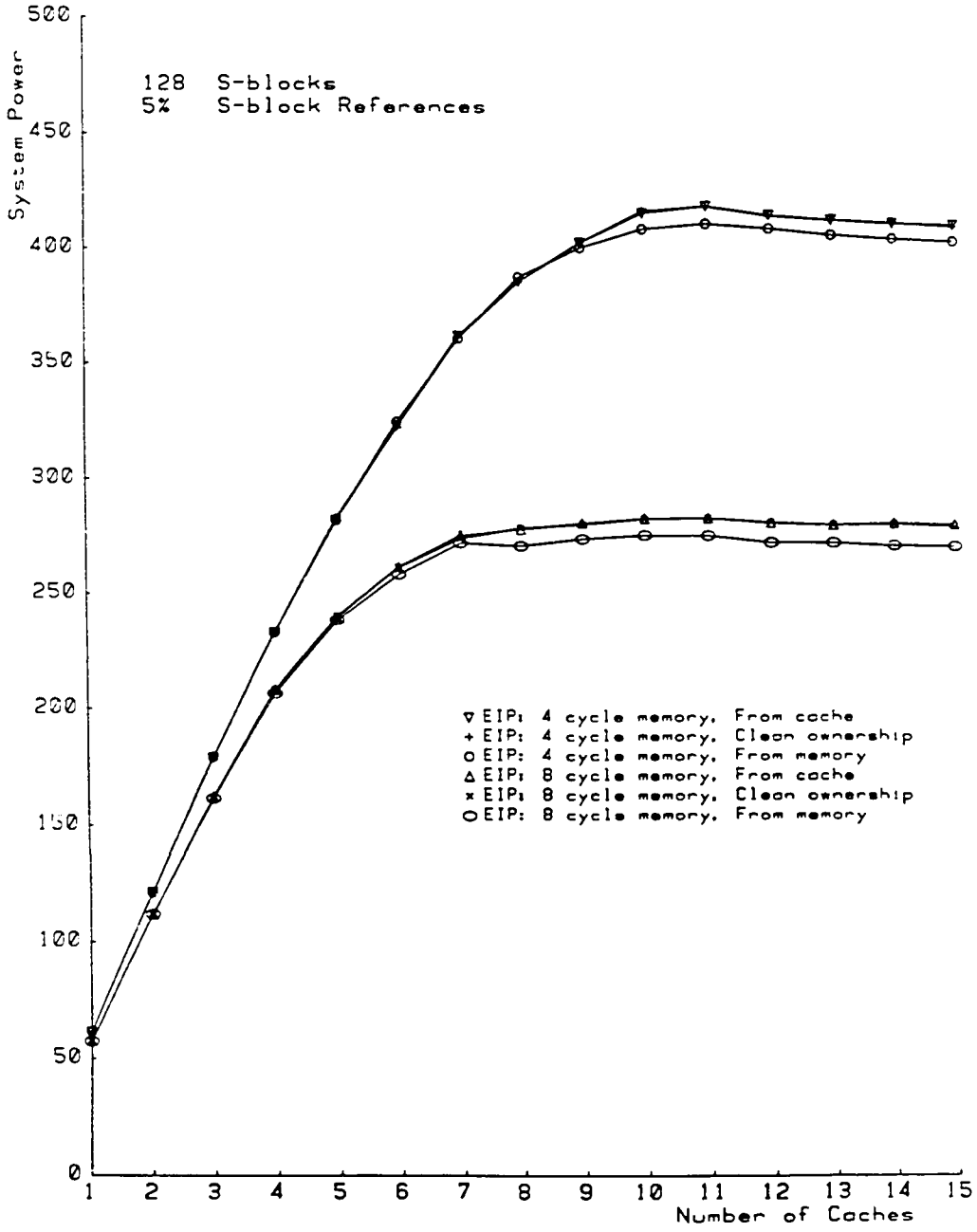


Figure 4.21: EIP: advantage of clean ownership

4.5 Previously Proposed Distributed Write Protocols

As in the section describing the invalidation protocols, the distributed write schemes are described in a uniform fashion, including a specification of the states and the actions of the protocol for the three cases mentioned. The same state naming conventions and abbreviations are used. For each of the distributed write schemes state transition diagrams are included.

4.5.1 Firefly

This scheme is used in the Firefly[Tha84], a multiprocessor workstation developed by Digital Equipment. Possible states for blocks in local caches are:

1. UNMODIFIED-EXCLUSIVE. (UNMOD-EXC)
2. UNMODIFIED-SHARED. (UNMOD-SHD)
3. MODIFIED-EXCLUSIVE. (MOD-EXC)

The state transitions of the scheme are described in Figure 4.22. This protocol never causes an invalidation so the INVALID state is not included (although it might actually be implemented for other reasons, e.g., a cold start in the cache). There is a special bus line used to detect sharing which we will refer to as the *SHARED* line. The protocol is described as follows:

- **Read miss**. If another cache has the block, it supplies it directly to the requesting cache and raises *SHARED*. All caches with a copy respond by putting the data on the bus–the bus timing is fixed so they all respond in the same cycle. All caches, including the requesting cache, set the state to UNMOD-SHD. If the owning cache had the block in state MOD-EXC, the block is written to main memory at the same time. If no other cache has a copy of the block, it is supplied by main memory, and it is loaded in state UNMOD-EXC.
- **Write hit**. If the block is MOD-EXC, the write can take place without delay. If the block is in state UNMOD-EXC, the write can be performed immediately and the state is changed to MOD-EXC. If the block is in state UNMOD-SHD, the write is delayed until the bus is acquired and a write-word to main memory can be initiated. Other caches with the block observe the write-word on the bus, take the new data, and overwrite that word in their copy of the block. In addition, these other caches raise *SHARED*. The writing cache can determine if sharing has stopped by testing this line. If it is not raised, no other cache has a

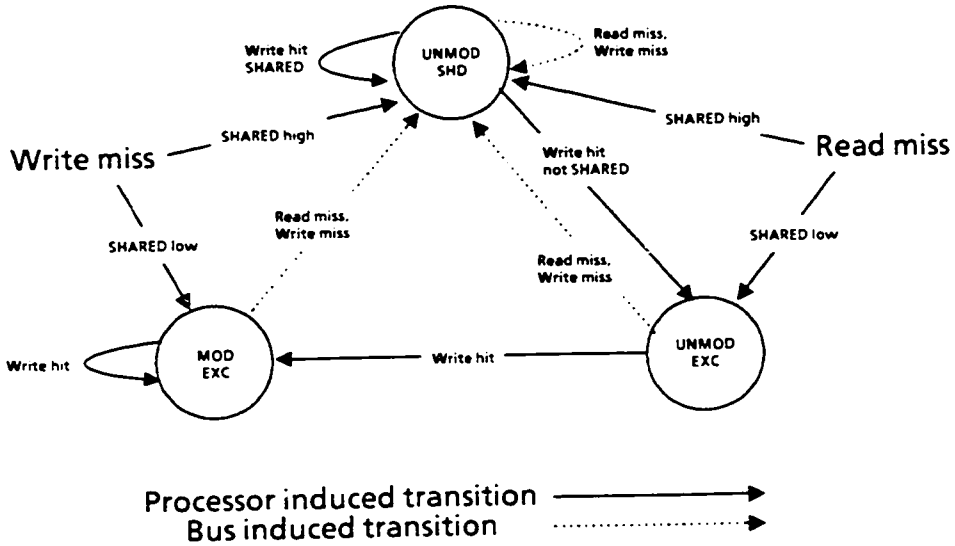


Figure 4.22: Firefly state transition diagram

copy and writes need no longer be broadcast–allowing a state change to UNMOD-EXC (and then to MOD-EXC on the next local write). If the line is high, sharing continues and the block remains in state UNMOD-SHD.

- **Write miss.** As with a read miss, the block is supplied by other caches, if any other caches have a copy. The requesting cache determines from the *SHARED* line whether the block came from other caches or not. If it came from memory, it is loaded in state MOD-EXC and written to without additional overhead. If it came from a cache, it is loaded in state UNMOD-SHD and the requesting cache must write the word to memory. Other caches with a copy of the block will take the new data, overwrite the old block contents with the new word, and set their states to UNMOD-SHD.

The Firefly protocol, like all distributed write protocols, allows multiple writers provided that each writer transmits the new data to all other caches with a copy. Unlike the invalidation schemes using a *SHARED* line, the Firefly uses the line on both read and write misses, and also on distributed writes to determine when sharing ceases.

4.5.2 Dragon

The Dragon[McC84] is a multiprocessor being designed at Xerox PARC. The coherence solution employed is similar to the Firefly scheme described above. The scheme employs the following states

for blocks present in the cache:

1. UNMODIFIED-EXCLUSIVE. (UNMOD-EXC)
2. UNMODIFIED-SHARED. (UNMOD-SHD)
3. MODIFIED-SHARED. (MOD-SHD)
4. MODIFIED-EXCLUSIVE. (MOD-EXC)

As with the Firefly scheme, the INVALID state is not included in the description, since the protocol does not require it. Also, a *SHARED* line on the bus is assumed. Figure 4.23 contains a state transition diagram for the protocol, which works as follows:

- **Read miss.** If another cache has a MOD-EXC or MOD-SHD copy, that cache supplies the data, raises *SHARED*, and sets its block state to MOD-SHD. Otherwise the block comes from main memory. Any caches with a UNMOD-EXC or UNMOD-SHD copy raise *SHARED* and set their local state to UNMOD-SHD. The requesting cache loads the block in state UNMOD-SHD if *SHARED* is high, otherwise it is loaded in state UNMOD-EXC.

- **Write hit.** If the block is MOD-EXC, the write can take place locally without delay. If the block is in state UNMOD-EXC, the write can also take place immediately with a local state change to MOD-EXC. Otherwise, the block is UNMOD-SHD or MOD-SHD and a distributed write must take place. When the bus is obtained, the new contents of the written word are distributed to all other caches with a copy of that block. The other caches take the new data, overwrite that word of their copy of the block, set the local state of their copies to UNMOD-SHD, and raise *SHARED* indicating that the data is still shared. By observing this line on the bus, the cache performing the write can determine if other caches still have a copy and hence, if further writes to that block must be broadcast. If the *SHARED* line is raised, the block state is changed to MOD-SHD, otherwise it is set to MOD-EXC.

- **Write miss.** As with a read miss, the block comes from a cache if it is MOD-EXC or MOD-SHD and from memory otherwise. Other caches with copies set their local state to UNMOD-SHD. Upon loading the block, the requesting cache sets the local state to MOD-EXC if *SHARED* is not raised. If *SHARED* is high, the requesting cache sets the state to MOD-SHD and performs a distributed write.

In the Dragon scheme, distributed writes update only other caches and not main memory as in the Firefly protocol. The addition of the MOD-SHD state makes this possible, since it marks the

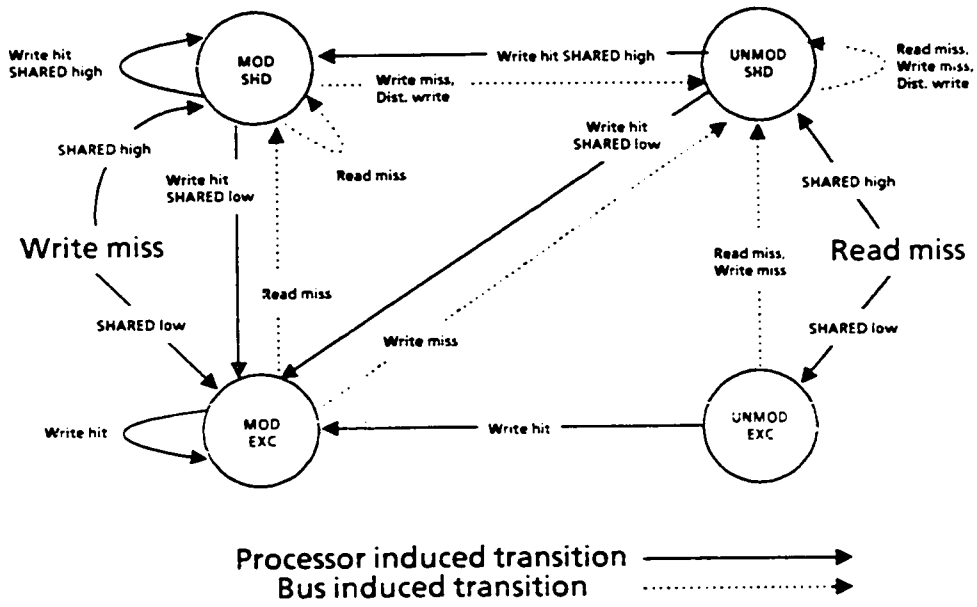


Figure 4.23: Dragon state transition diagram

last cache to write and gives that cache the responsibility to update main memory (with a write back when the block is replaced). While a block may be in MOD-SHD and present in other caches, the other copies must be UNMOD-SHD. Thus, at most one cache has a MOD-EXC or MOD-SHD block.

4.5.3 CMU Read & Write Broadcast (RWB)

This protocol is an extension of the CMU RB scheme described with the invalidation schemes. In the RWB protocol, caches update INV copies not only when data is on the bus in response to a read miss (as in the RB scheme), but also when data is on the bus for a write through. In the RB scheme, a write through served as an invalidation signal–it serves as a distributed write in the RWB scheme. The RWB is the first protocol proposed that uses both distributed writes and invalidations. Initially distributed writes are performed, followed by an invalidation when the scheme detects that the block has become local to a particular cache. (The criteria for this decision is outlined below.) Like the RB scheme, a one word block size is assumed, and hence the scheme is not simulated. The following states are used:

1. INVALID. (INV)

2. UNMODIFIED-SHARED. (UNMOD-SHD)

3. FIRST-WRITE. (FW) The most recent write to this block by any cache in the system occurred in the cache with the block in this state.

4. MODIFIED-EXCLUSIVE. (MOD-EXC)

The FW state is added to allow dynamic detection of write reference patterns to shared blocks, which can help eliminate logically unnecessary overhead in distributed writes. Should two consecutive writes to a block occur in the same cache, the block is deemed to be local. The FW state indicates that the first of the two necessary writes has taken place. The state transitions of the protocol are illustrated in Figure 4.24. The RWB scheme works as follows:

- **Read miss.** Unless a MOD-EXC copy exists, the block is loaded directly from main memory in state UNMOD-SHD. If a MOD-EXC copy exists, the servicing of the read miss is interrupted and the cache with the MOD-EXC copy performs a write back to main memory, in addition to setting its own state to UNMOD-SHD. Immediately following the write back, the read miss is serviced, with the data being loaded from memory in state UNMOD-SHD. While the valid data is on the bus (either during the write-through or in loading the cache from main memory), all caches with INV copies take the data and update their copies, setting the state to UNMOD-SHD.

- **Write hit.** If the local copy is MOD-EXC, the write may proceed immediately. If the state is UNMOD-SHD, it is changed to FW and a distributed write takes place, updating all other cached copies and main memory. All other caches with copies (in any of the four states) take the new data and change their local state to UNMOD-SHD. In this way, the last cache to write has the block in FW. All other cached copies are UNMOD-SHD. If the local state is already FW, then two successive writes have taken place by the same processor, and the block is assumed to be local to that processor. An invalidation signal is sent, setting all other cached copies to the INV state, and the local state is set to MOD-EXC.

- **Write miss.** The block is entered in the cache, the local state is set to FW, and a distributed write takes place, setting all other cached copies to UNMOD-SHD. Note that this is exactly the action taken on a write hit on UNMOD-SHD.

The authors state that 'two writes to a variable without any intervening references to the variable' by another processor are sufficient to indicate local usage. However, their scheme does not provide this. Two writes to a block without any intervening *write* references by any other

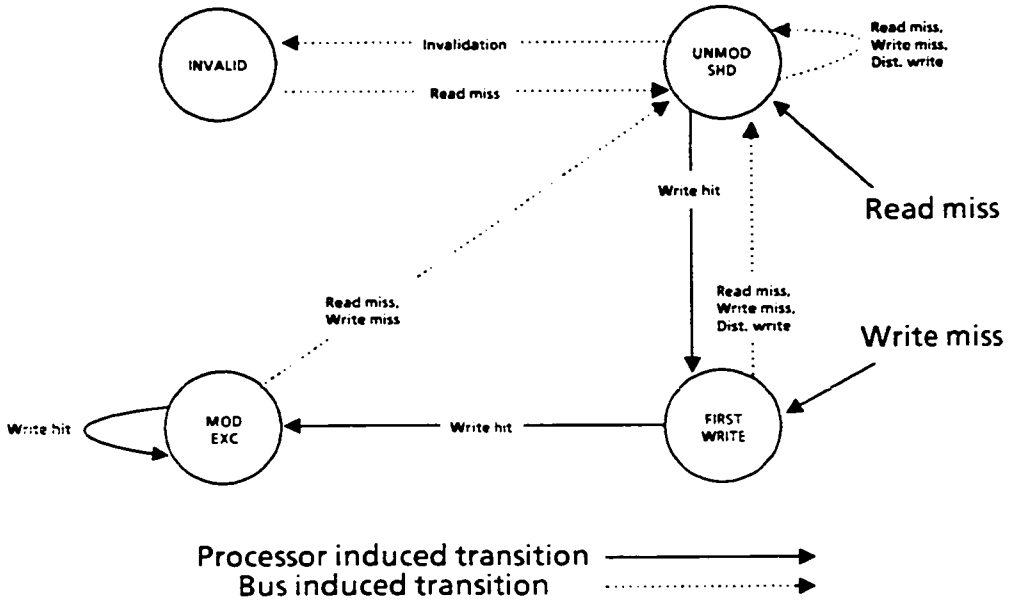


Figure 4.24: CMU RWB state transition diagram

processor cause an invalidation–intervening reads have no effect. The EDWP scheme described in the next section introduces a technique to determine local usage that includes both read and write references.

4.6 EDWP: An Efficient Distributed Write Protocol

The EDWP is a new distributed write protocol that we have designed that uses the following set of states:

1. INVALID. (INV)
2. UNMODIFIED-EXCLUSIVE. (UNMOD-EXC) Only cached copy.
3. UNMODIFIED-SOURCE. (UNMOD-SRC) Possibly shared.
4. UNMODIFIED-SHARED. (UNMOD-SHD) Possibly shared.
5. MODIFIED-SHARED. (MOD-SHD) Possibly shared, must be written-back.
6. MODIFIED-EXCLUSIVE. (MOD-EXC) Only cache copy, must be written-back.

7. REMOTE-WRITE 1. (RW1) Not referenced locally since last distributed write.

8. REMOTE-WRITE 2. (RW2) Not referenced locally since last two distributed writes.

The first six states are the same as those in the EIP protocol but the associated actions are very different. The REMOTE-WRITE states are added to detect unnecessary updates. Their function will be explained in the following discussion. (A diagram summarizing all state transitions is given in Figure 4.25.) Like EIP, EDWP requires the *SHARED* and *MODIFIED* lines in the bus, and it also uses the ideas of clean ownership and dirty ownership exactly as presented in the EIP discussion. A proof of correctness for the scheme is given in Chapter 8.

The EDWP is unique among all schemes presented here in that the coherence protocol requires some actions in the case of a read hit. The actions of the protocol are the following:

- **Read hit.** If the state is RW1 or RW2, the state is changed to UNMOD-SHD. No other state changes are made.

- **Read miss.** If a modified copy of the block exists in the system, the block will be supplied by the dirty owner. Otherwise the block comes from the clean owner. If the block was present in state MOD-EXC, that cache changes its copy to MOD-SHD. If the block was in state UNMOD-SRC or UNMOD-EXC it is changed to state UNMOD-SHD. All caches with a copy of the block raise the *SHARED* line, and if the block had a dirty owner, the dirty owner raises the *MODIFIED* line. The requesting cache tests the value of both lines to determine the state of the newly loaded block. If the *SHARED* line is raised, but the *MODIFIED* line is not, the block is loaded in state UNMOD-SRC. If both the *SHARED* and *MODIFIED* lines are raised the block is loaded in state UNMOD-SHD. If the *SHARED* line is not raised the block is loaded in state UNMOD-EXC.

- **Write hit.** If the block is in state MOD-EXC the write may proceed immediately. If the block is in state UNMOD-EXC, it is changed to MOD-EXC and the write may proceed. If the block is in any other state, a distributed write must be performed. The cache performing the distributed write must first obtain the bus and send the new data to all other caches. It then tests the value of the *SHARED* line to determine if the updates must continue. If the line is high, the block continues to be shared and the local state is set to MOD-SHD. If the line is low, the state is set to MOD-EXC and subsequent writes will proceed locally.

  Each cache receiving the distributed write takes the new data and overwrites its copy of the block. In addition, a change may be required in the local state and the *SHARED* line may be raised–done *only* if a valid copy of the block is present in some state other than RW2. If the

local copy is UNMOD-SRC, UNMOD-SHD, or MOD-SHD, (UNMOD-EXC and MOD-EXC are impossible since the block is shared), the state is changed to RW1. If it is already RW1, it is changed to RW2. If the copy is RW2, the state change depends on the value of the *SHARED* line. If the line is high (raised by another cache), the state remains RW2. If the line remains low, all cached copies must be RW2, and the state of each is changed to INV. It is important to note that all copies either remain valid together, or they become invalidated together, based on the value of the *SHARED* line, which in turn depends on the local states. The addition of states RW1 and RW2 allows the detection of local references since the last distributed write. Using this technique, EDWP causes an invalidation of all cached copies if there are three distributed writes by a single processor with no intervening reference by any other processor.

- **Write miss.** The actions are identical to a read miss followed by a write hit. If the block was loaded in a non-exclusive state, a distributed write must be performed on the bus, with the associated actions described above. Following the write, the block will be present in either state MOD-EXC or state MOD-SHD. Note that it is possible for the block to be loaded in a shared state (*SHARED* high on the miss) and for the block to be in MOD-EXC after the write (*SHARED* low on the update) if all other copies are in state RW2.

Unlike the Firefly and Dragon protocols, EDWP does not continue to update other caches with a distributed write as long as other cached copies exist. In the case of task migrations in a system with very large caches, references to blocks present in another cache might be a frequent occurrence. However, the distributed write overhead is unnecessary since the blocks are not being referenced in the other cache. Unlike the CMU distributed write scheme, EDWP bases its determination of whether to continue distributed writes on both read and write references in other caches. The data will continue to be supplied to all caches as long as it continues to be referenced by any one of them. In a producer-consumer relationship between two or more processes, where one process frequently updates a shared variable and other processes read the variable, the CMU scheme will result in an invalidation on the second write, although the location continues to be referenced.

The protocol uses the *SHARED* line to eliminate all coherence overhead on unshared blocks, except that resulting from task migration. The handling of shared blocks is made more efficient by the use of clean ownership which reduces the overhead of cache misses on unmodified blocks. The EDWP uses the REMOTE-WRITE states as counters to determine the reference behavior of those blocks updated by a distributed write. Three consecutive distributed writes by any single processor without any intervening references by any other processor cause the block to be invalidated, and

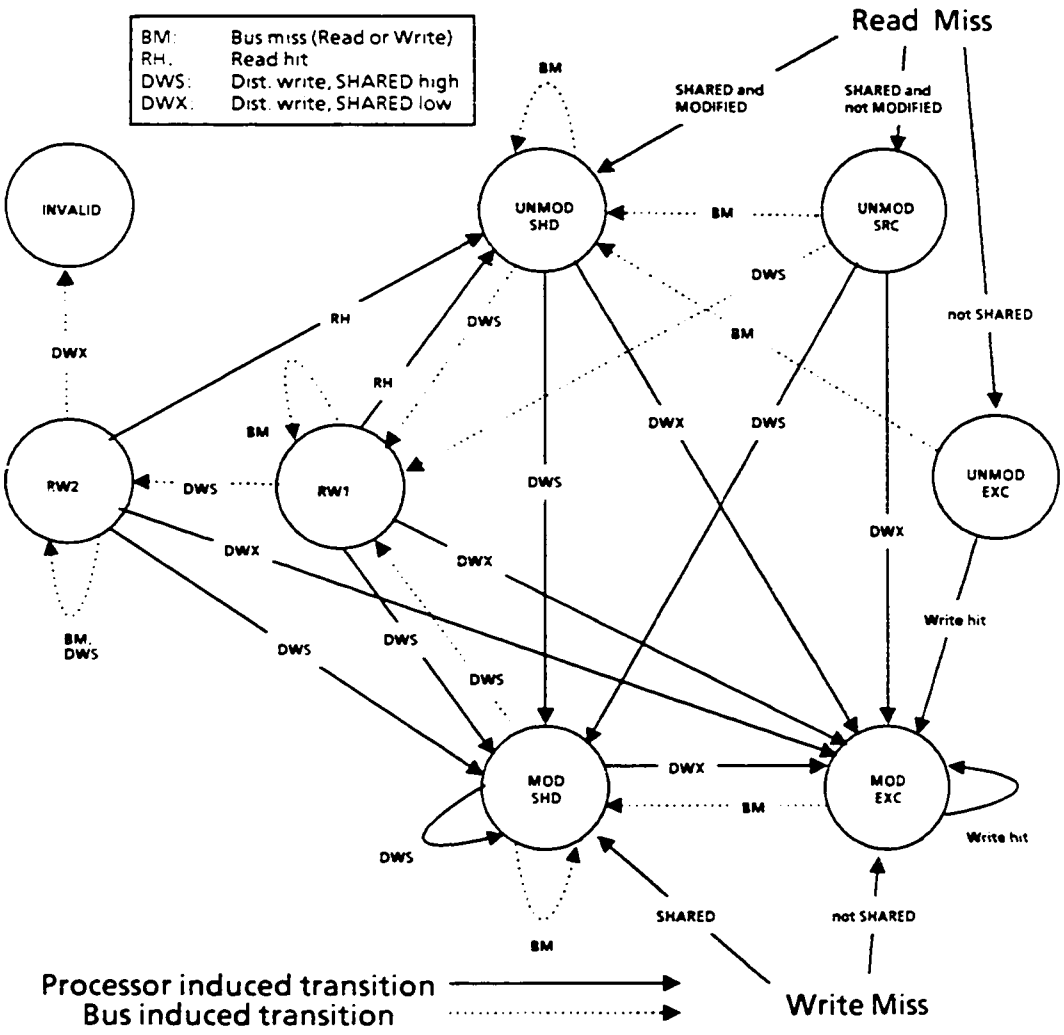


Figure 4.25: EDWP state transition diagram

Processor-to-Cache Commands

Read word
Write word
Test-and-set word

Bus Transactions

Read block [Read or write miss]
Distributed write
Write-back block

Actions Causing State Change

Read block
Distributed write
Write-back block
Read hit [on REMOTE-WRITE]
Write on UNMOD-EXC block
Replace unmodified block

Figure 4.26: Specification of EDWP system actions

the writer's copy becomes exclusive, allowing subsequent writes to proceed locally. The copies remain valid together or are invalidated together since the overhead of a distributed write is not dependent on the number of caches with copies–if the updates must continue to a single processor it is no additional overhead to supply the data to all.

Figure 4.26 gives a summary of all required system transactions. As with EIP, it is assumed that the processor has a TAS instruction, but that no additional bus transaction is required. Note that there is a single bus transaction for loading data on both read miss and write miss. The list of actions causing state changes includes all three types of bus transactions and three local events–resetting the REMOTE-WRITE state on a read hit, writing a previously unmodified exclusive copy, and replacing an unmodified block or removing it from the cache.

4.7 Simulation Results

Figure 4.27 shows the actual sharing using a distributed write scheme with 5% S-block references. (Results from the Dragon are displayed, but other distributed write protocols are nearly identical.) For comparison purposes, the results from the Illinois protocol are also included. In general, the

distributed write protocols encourage sharing and result in a higher level of actual sharing than the invalidation protocols. In experiments using 0.1% S-block references the sharing is virtually non-existent.

Figures 4.28 through 4.36 indicate the simulations results for the distributed write protocols using the same experiments used in the discussion of invalidation protocol performance. Figure 4.28 shows results with negligible sharing. Figures 4.29 through 4.31 display the performance with normal parameter values and varying the number of S-blocks. Figure 4.32 indicates the protocol performance when the write frequency is doubled. Figures 4.33 and 4.34 demonstrate the performance using a large cache and an increased P-block hit ratio. Figure 4.35 and 4.36 show the results obtained when the size of the block is two words and eight words, respectively.

The ordering of the schemes is very consistent, with EDWP the most efficient, followed closely by the Dragon and, somewhat further behind, the Firefly. As with the invalidation protocols, the efficiency of private block and shared block handling will be examined in greater detail. Consistent with their usage in the discussion of invalidation schemes, *shared* refers to blocks that are actually present in another cache, and *private* refers to blocks that are not present in other caches.

4.7.1 Efficiency of Private Block Handling

Figure 4.28 shows simulation results with virtually no sharing. As can be seen, the performance of the three protocols is indistinguishable. All schemes include an UNMOD-EXC state and are equipped to detect sharing dynamically on read misses. All private blocks are therefore loaded in an exclusive state (UNMOD-EXC on read miss, MOD-EXC on write miss), avoiding all logically unnecessary overhead.

Although it was not simulated, the performance of the CMU RWB scheme can be shown to be much less efficient in the handling of private blocks. Specifically, the scheme has no UNMOD-EXC state, and first time write hits on unmodified private blocks result in a write-through to main memory. The next write also requires a global action–an invalidation signal. While this might be a reasonable approach to take for shared blocks, the resulting overhead on private blocks is excessive.

4.7.2 Efficiency of Shared Block Handling

The only difference in the performance of the simulated distributed write protocols arises in the handling of shared data. Figures 4.29 through 4.36 indicate the protocol performance with higher levels of sharing. There are many causes of the observed differences in performance. Read and

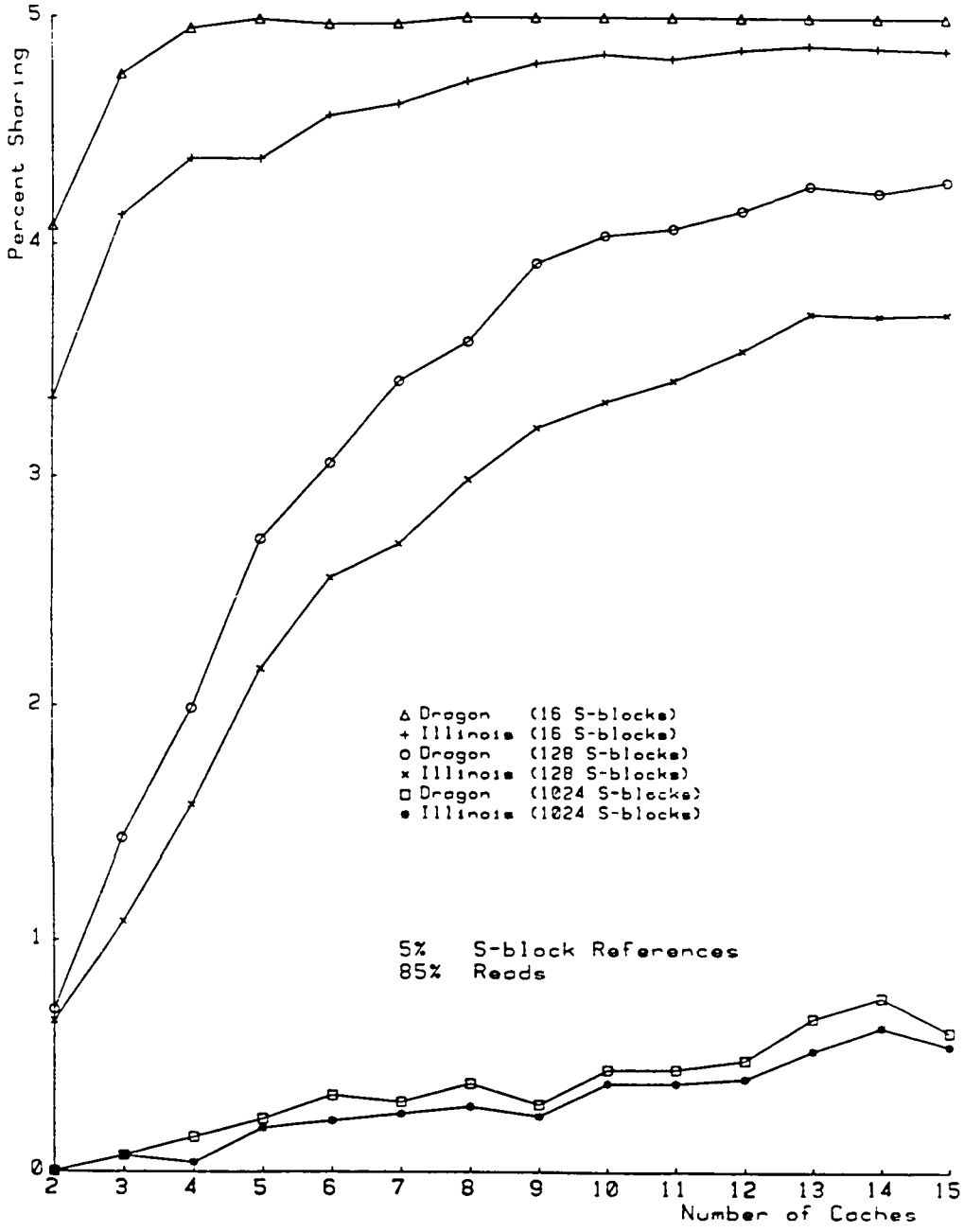


Figure 4.27: Dist. write: actual sharing

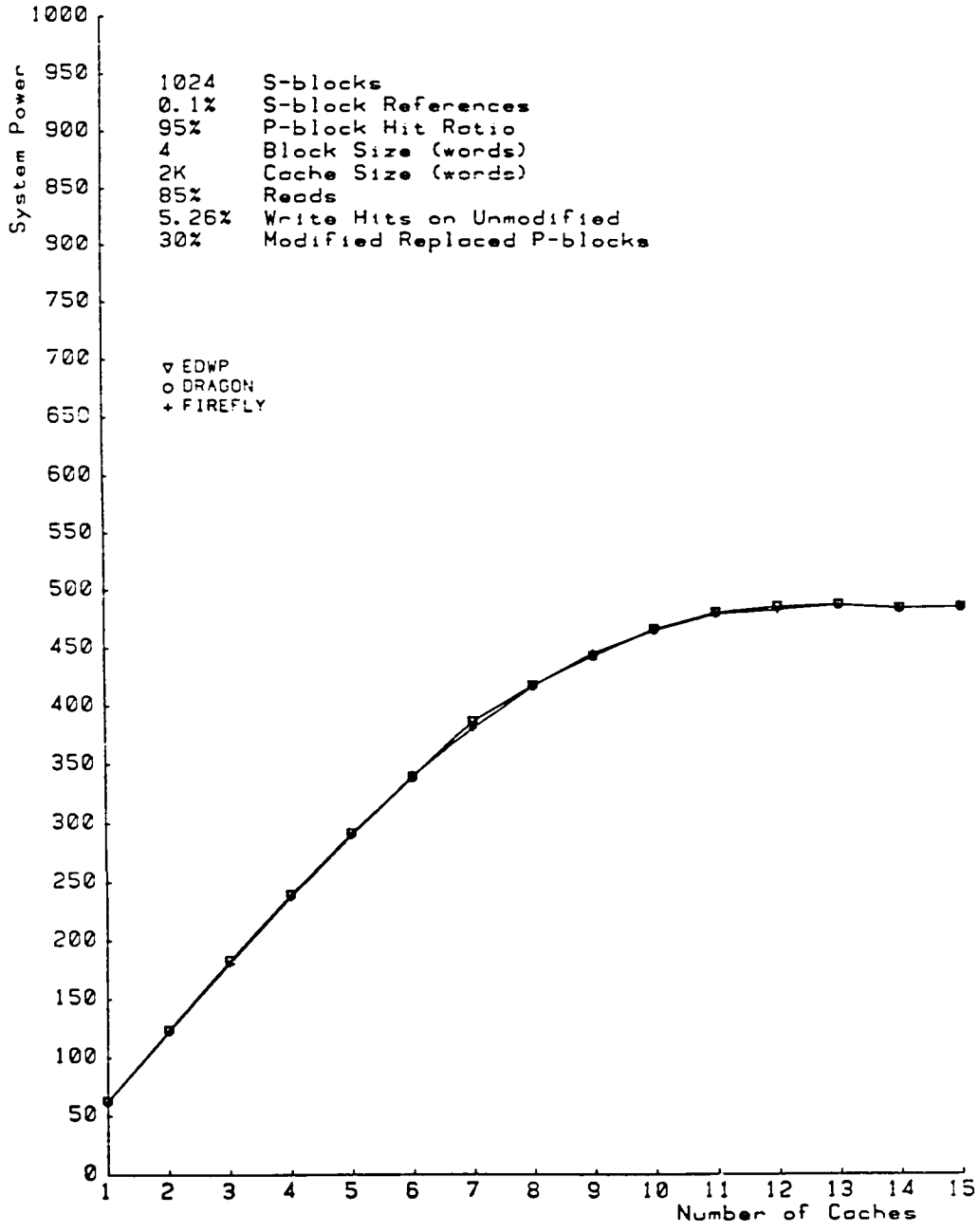


Figure 4.28: Dist. write: negligible sharing

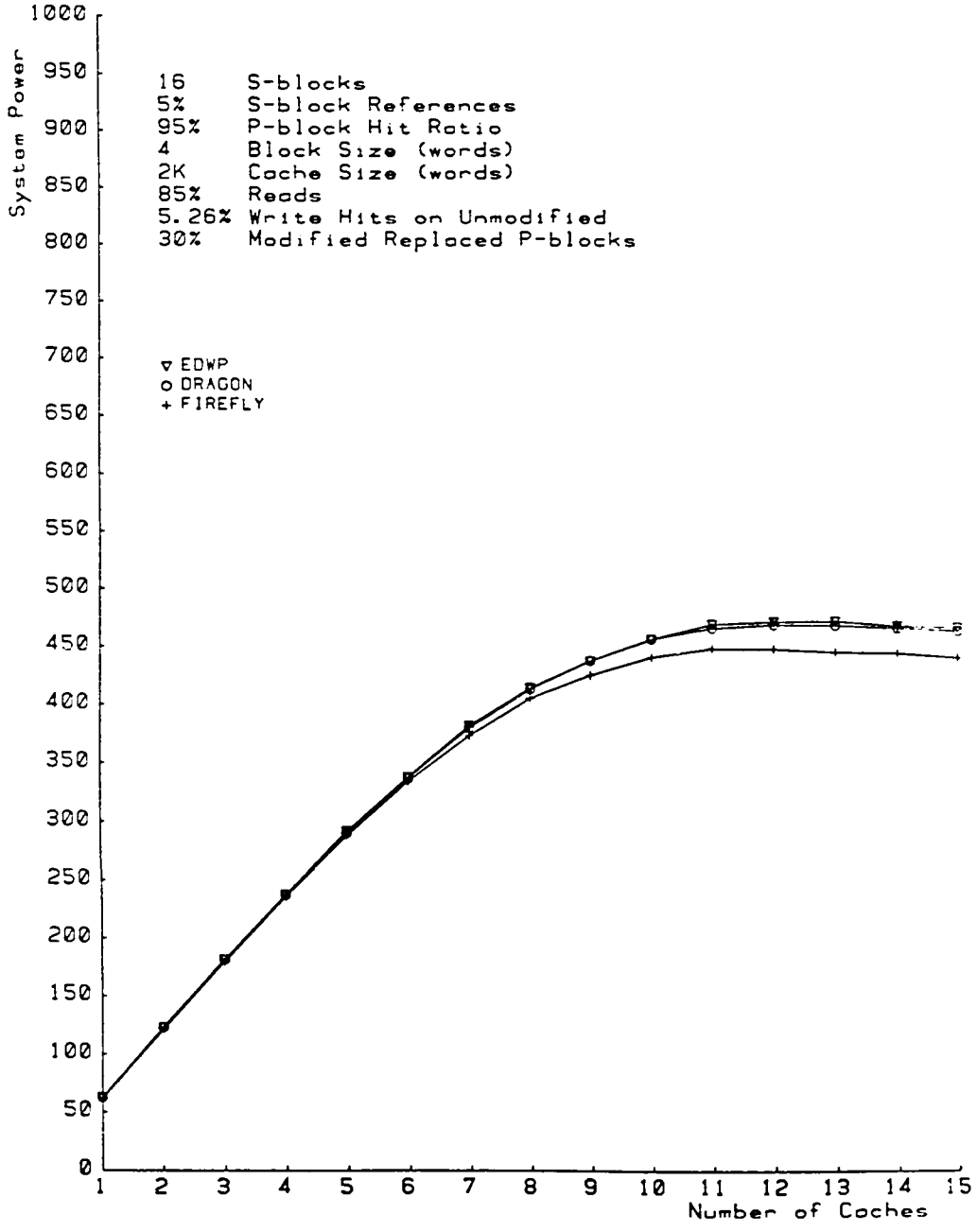


Figure 4.29: Dist. write: basic model, 16 S-blocks

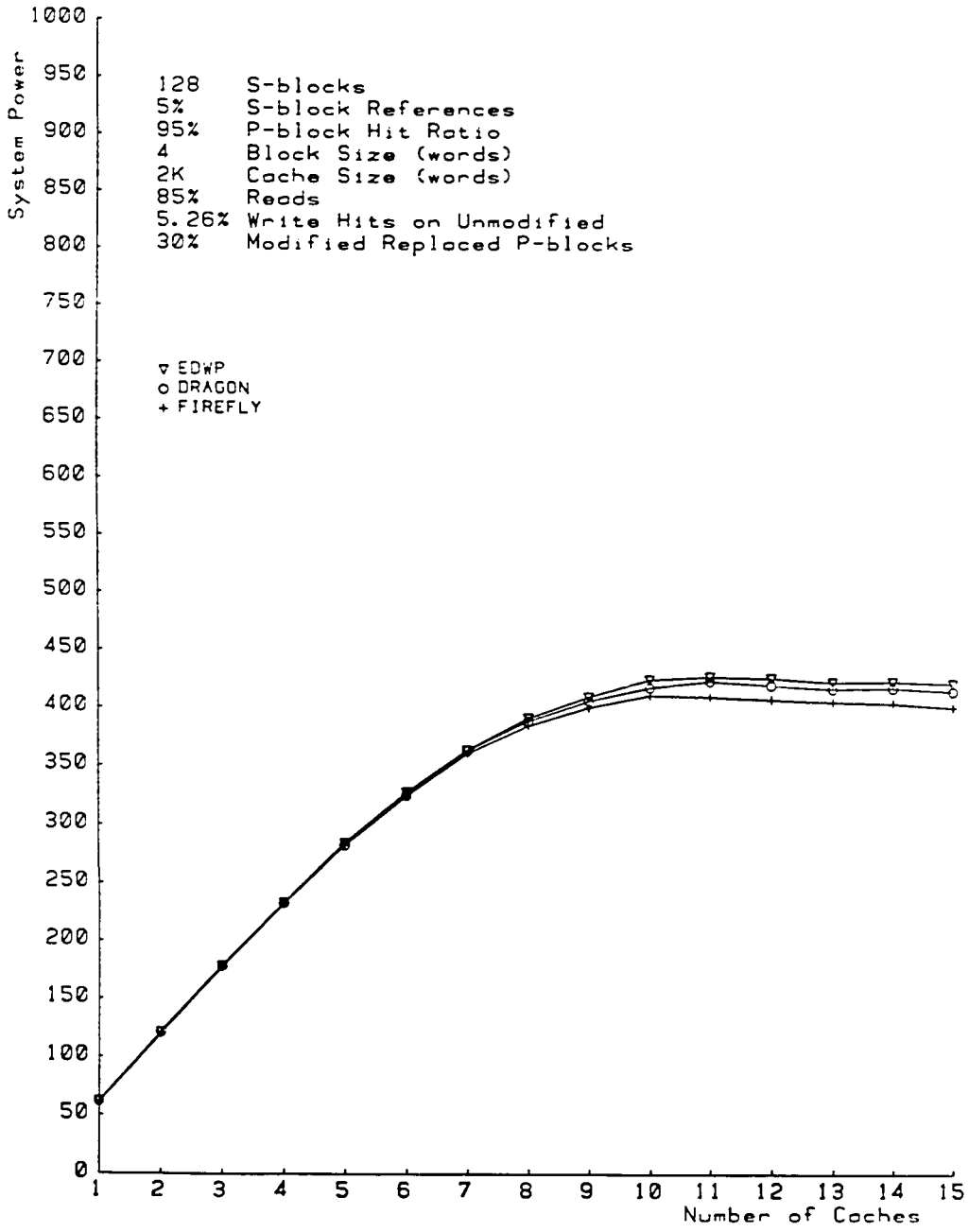


Figure 4.30: Dist. write: basic model, 128 S-blocks

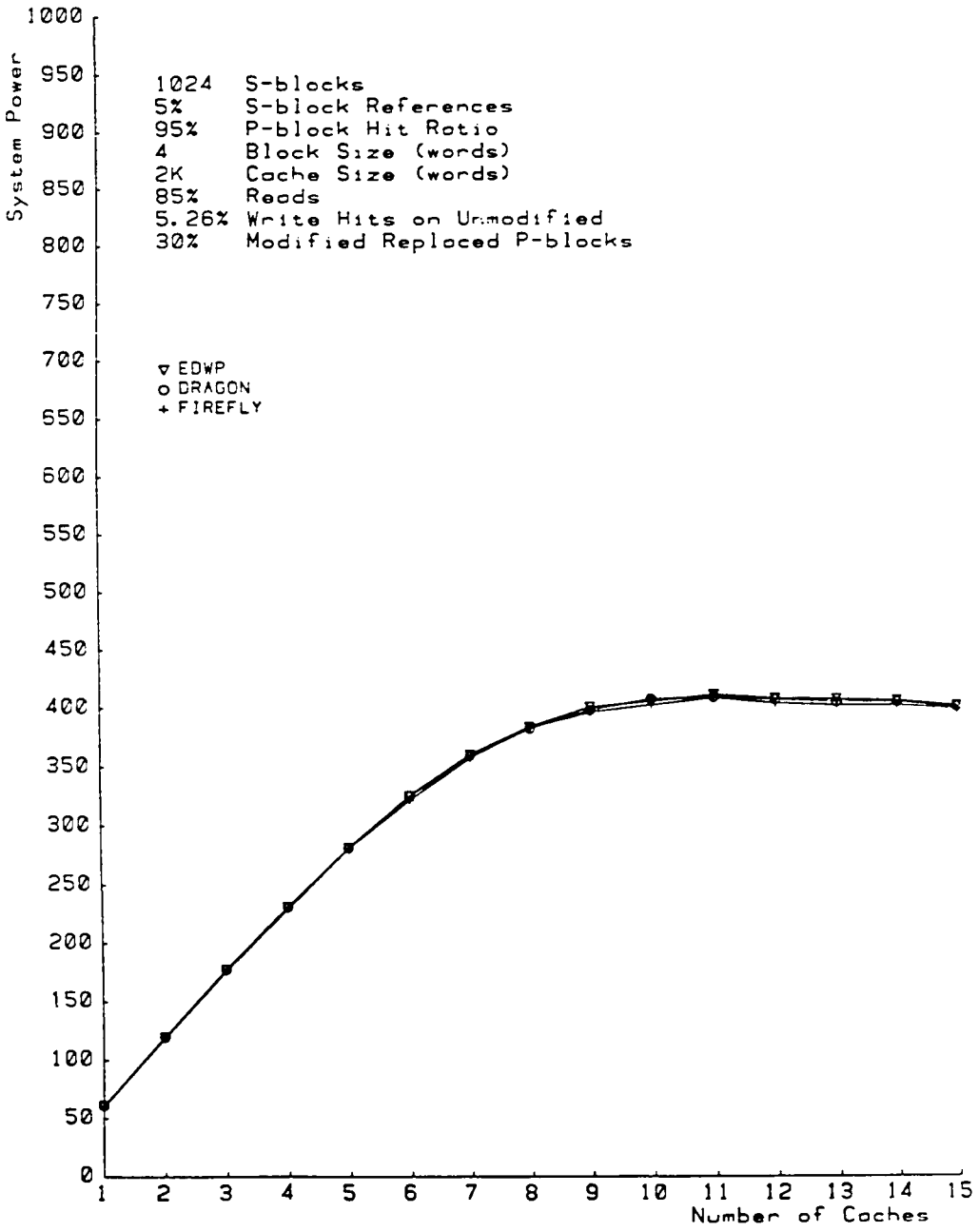


Figure 4.31: Dist. write: basic model, 1024 S-blocks

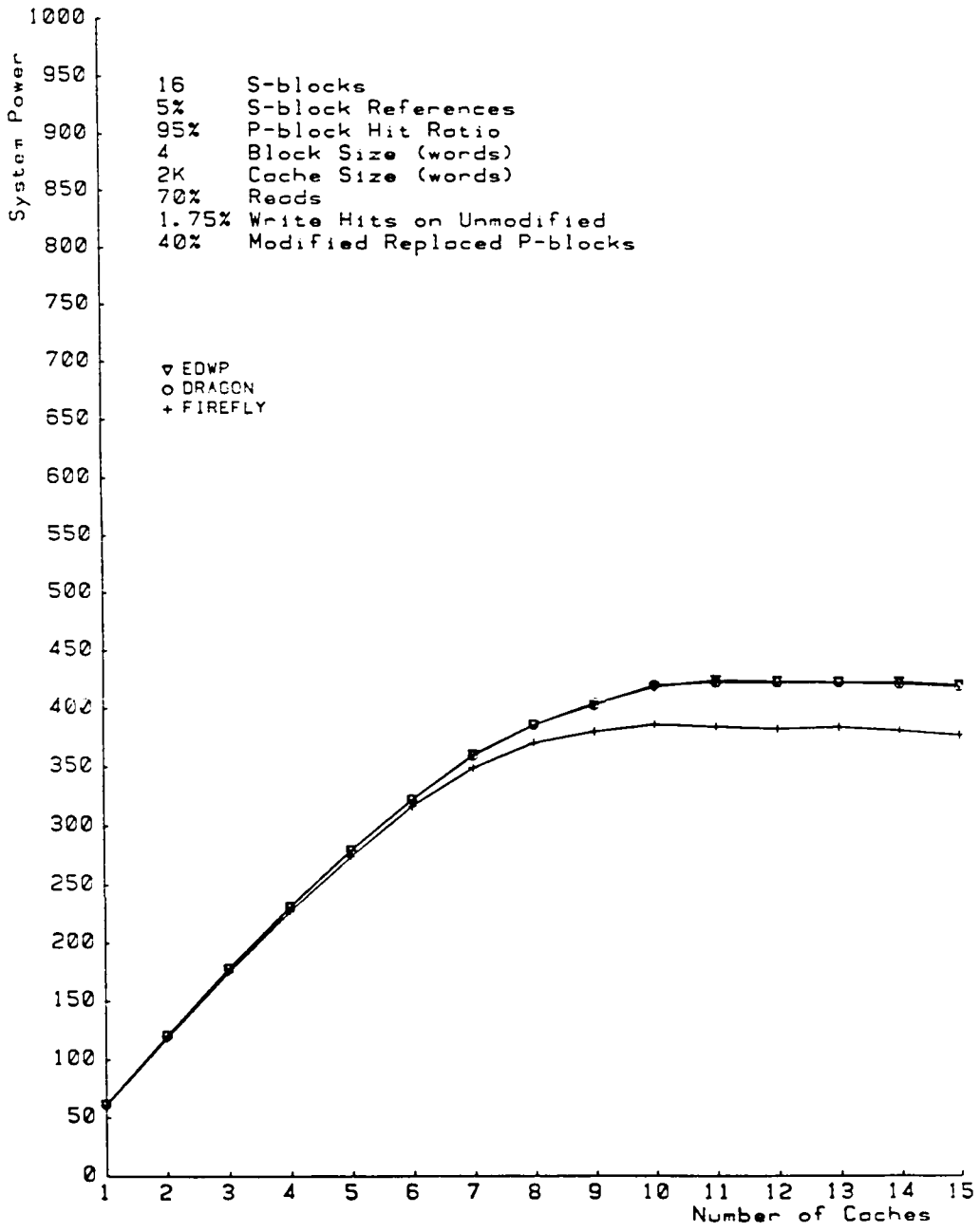


Figure 4.32: Dist. write: increased write ratio

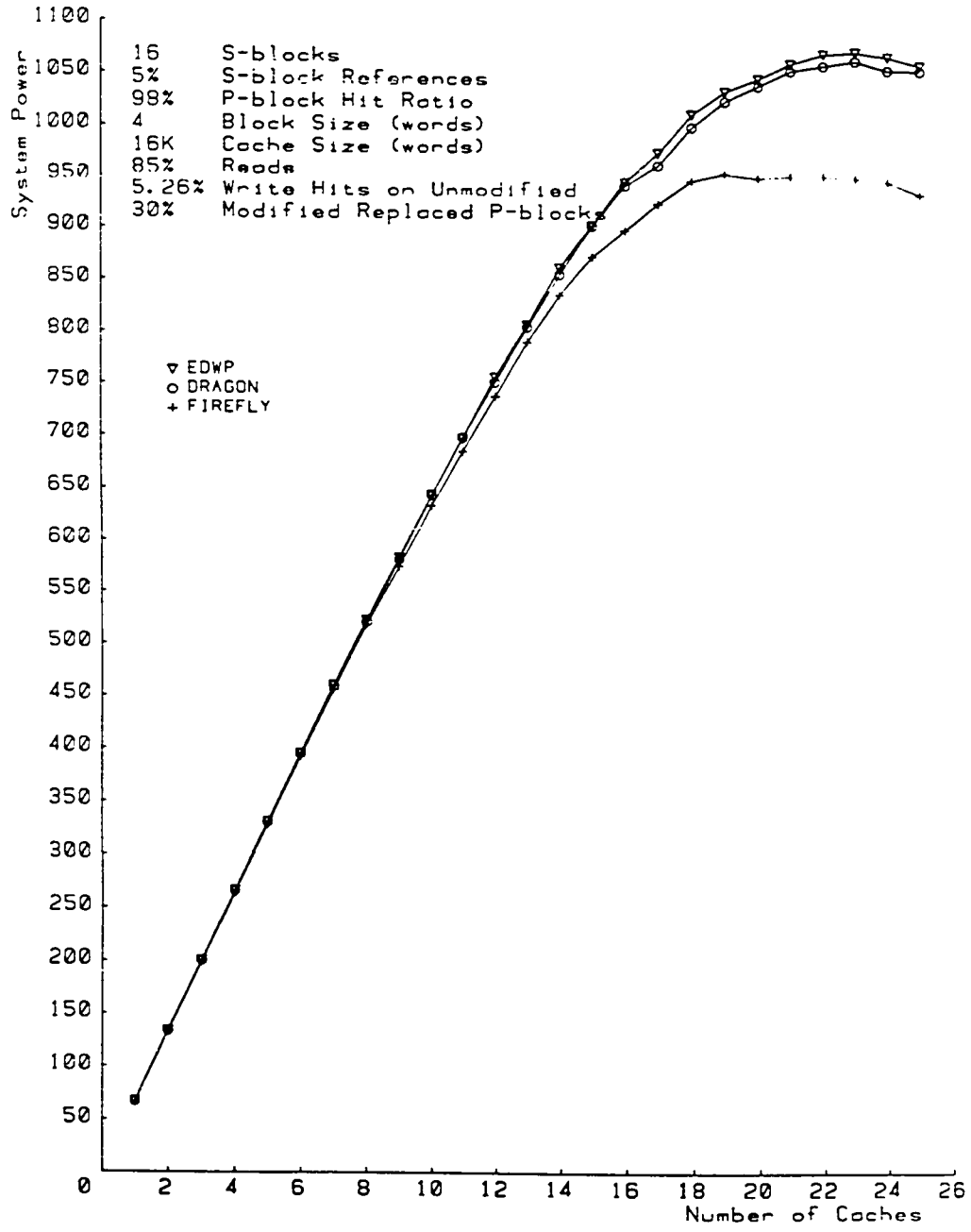


Figure 4.33: Dist. write: increased cache size, 16 S-blocks

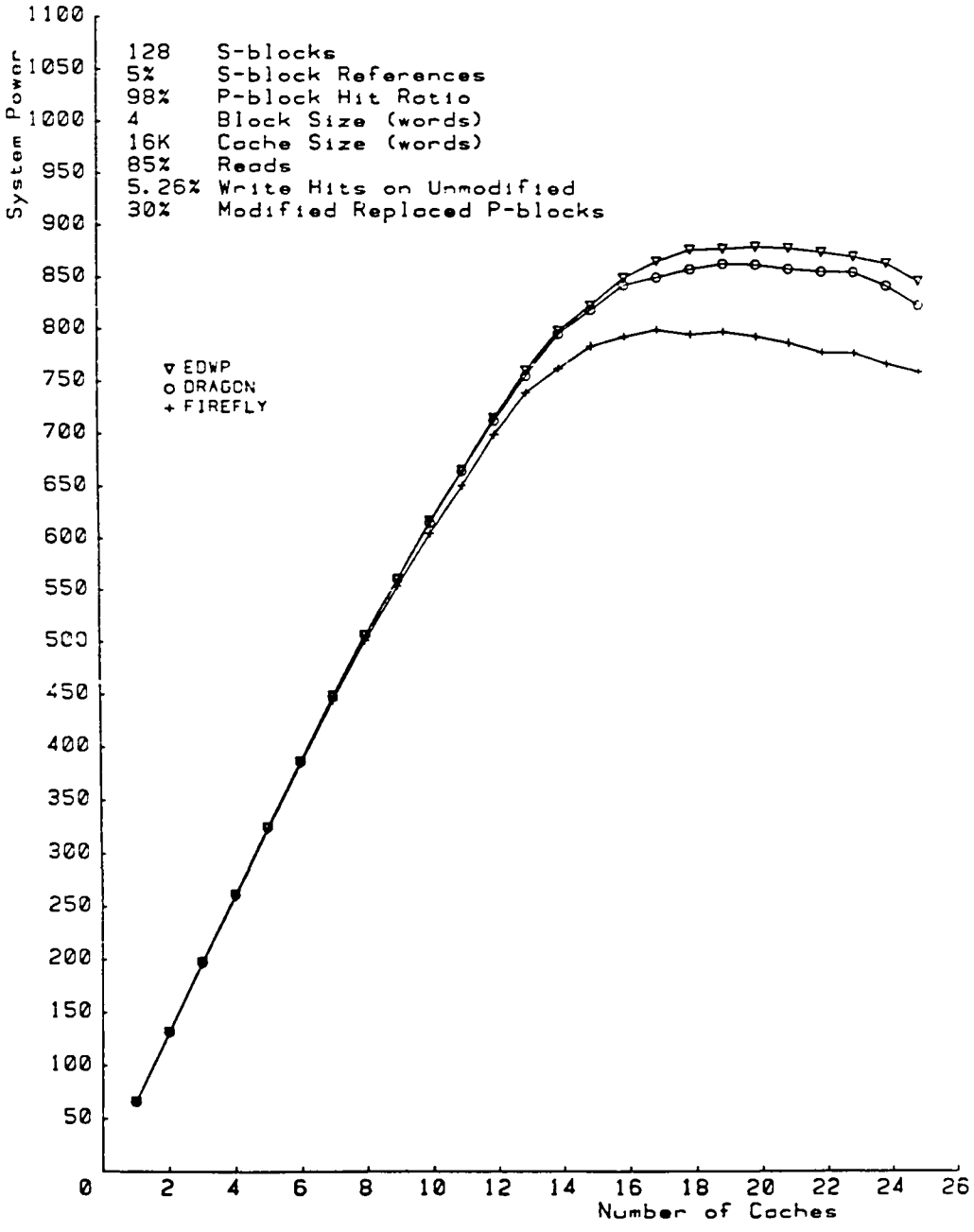


Figure 4.34: Dist. write: increased cache size, 128 S-blocks

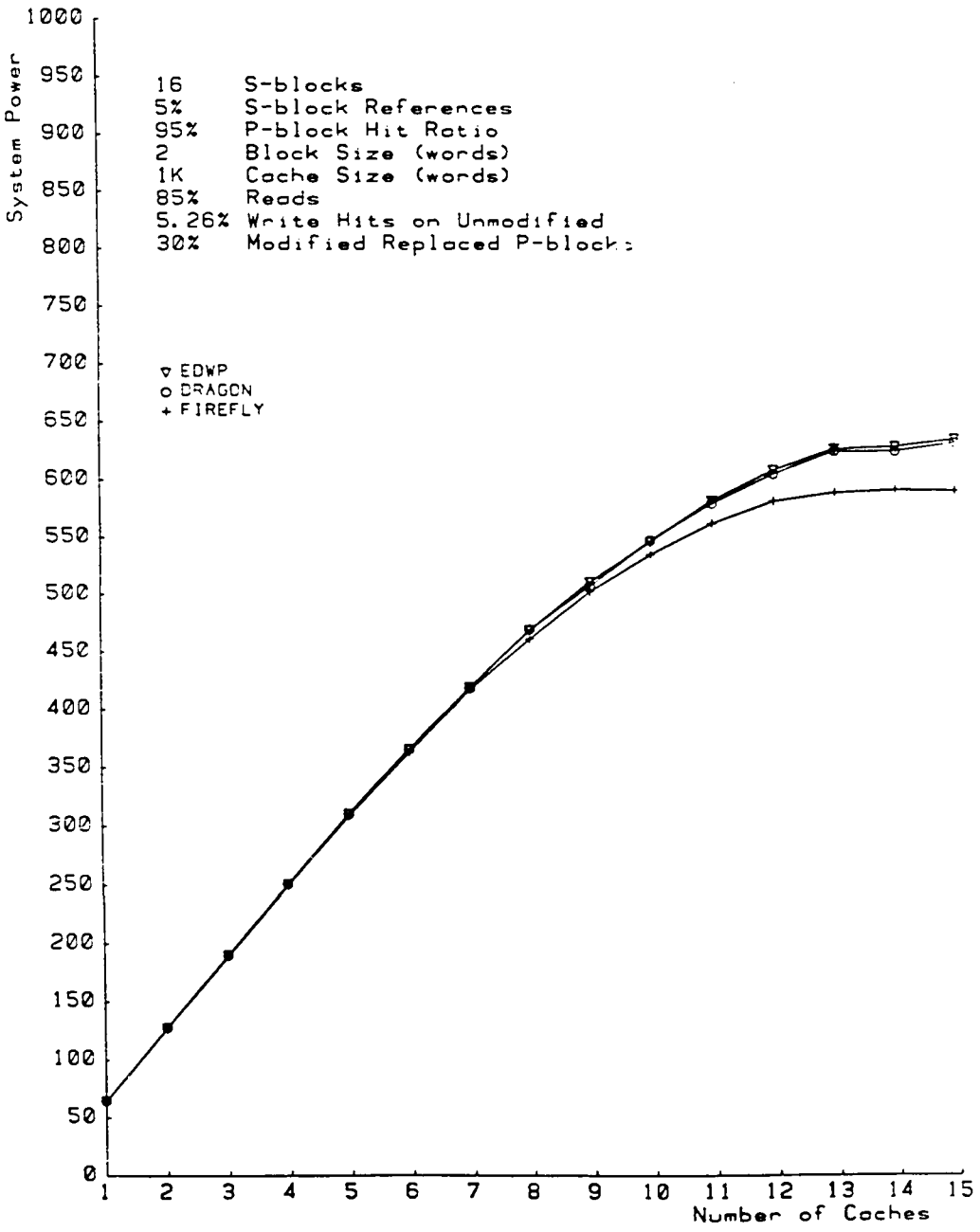


Figure 4.35: Dist. write: 2 word block size

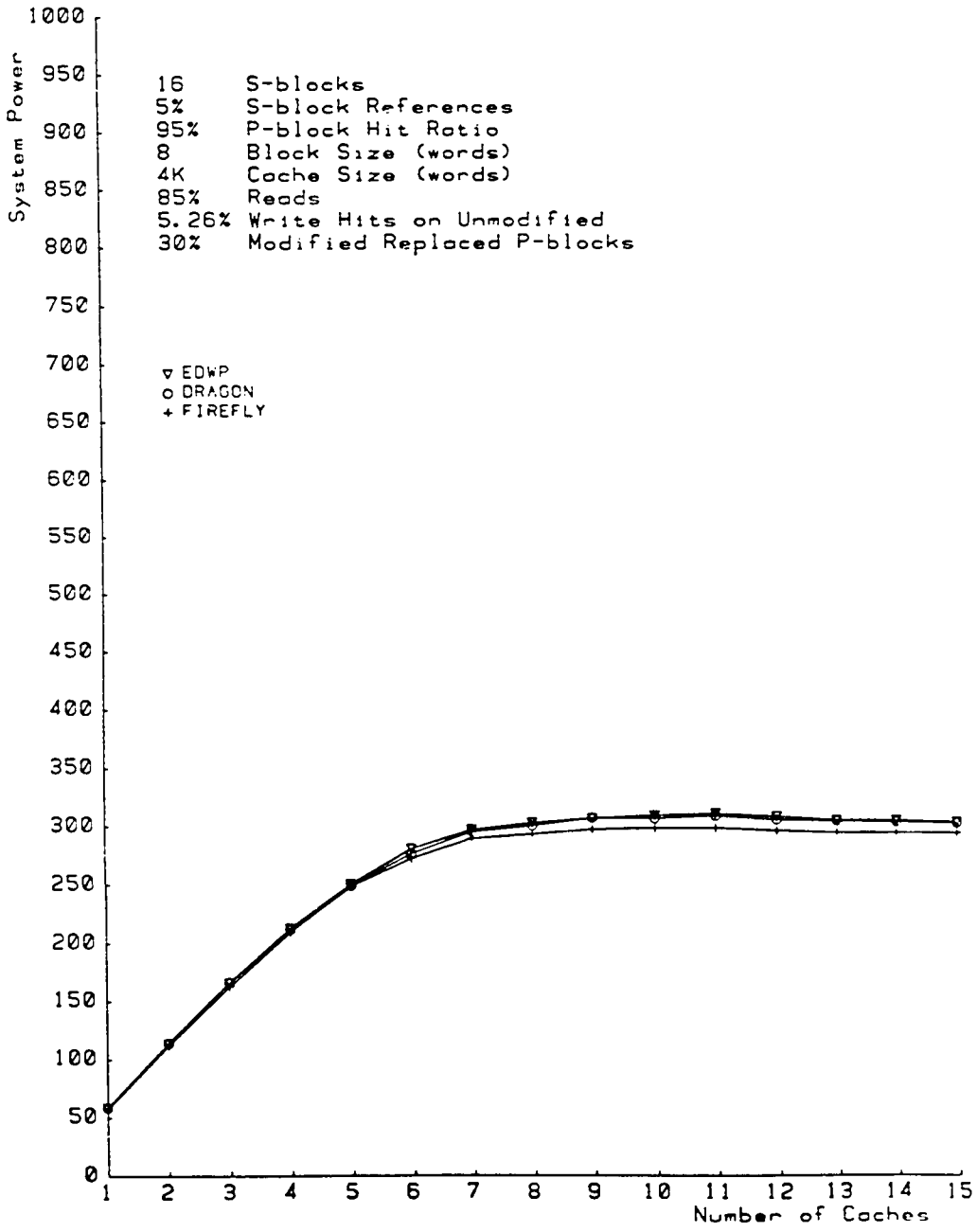


Figure 4.36: Dist. write: 8 word block size

write misses always load shared blocks from other caches in the Firefly, from memory (unless modified) in the Dragon, and from other caches in EDWP (unless memory is clean owner). In the Dragon and Firefly, distributed writes continue until the block is no longer present in other caches. In the EDWP, distributed writes are discontinued if the block does not continue to be referenced in other caches. On each distributed write, EDWP and Dragon update only the other caches, while the Firefly must also update main memory since it has no MOD-SHD state. However, this causes a difference in the replacement of shared blocks. In the Firefly, shared blocks will never require a write-back because memory is always up-to-date. Write-back are required of all modified shared blocks in the Dragon and EDWP.

The best overall performance is given by EDWP, which proves to be just slightly better than the Dragon. The performance of the Firefly is somewhat lower, due to the overhead of updating memory on every distributed write. Despite its feature to dynamically detect local usage of shared blocks, the CMU RWB scheme would probably not exhibit better shared block performance than the Firefly because of the overhead of updating memory on each update. The overall performance, including private block overhead, would be far less than the other distributed write protocols.

Note that the overall performance of the three protocols decreases as the number of S-blocks increases, as was the case with the EIP protocol. Because there are no invalidations in the Dragon and Firefly, and few invalidations in EDWP, the hit ratio on shared blocks depends only on the locality of references. With 16 S-blocks, each cache has a high hit ratio and performance is very good, despite the overhead of distributed writes. With 1024 S-blocks, the S-block references are more widely spread and the hit ratio declines. This results in lower performance even though the overhead of distributed writes is lower since they occur less frequently.

The slight improvement of EDWP over the Dragon is the result of clean ownership and the elimination of unnecessary distributed writes. Note that both features require additional states to implement. Figure 4.37 shows the performance improvement resulting from clean ownership. Two modified versions of EDWP were simulated and compared against clean ownership. One scheme always loads blocks from memory unless dirty, and the other always loads shared blocks from caches. The figure shows the results using memory cycle times of four and eight cycles. As was the case with the EIP scheme, there is a slight performance advantage of getting blocks from caches whenever possible, and the performance of clean ownership is as good as always loading from cache. As mentioned in the EIP discussion, the improvement would be increased if the simulation included a class of read-only data.

The elimination of unnecessary distributed writes in EDWP comes at the expense of two REMOTE-WRITE states, allowing invalidation after three consecutive writes in the same cache.

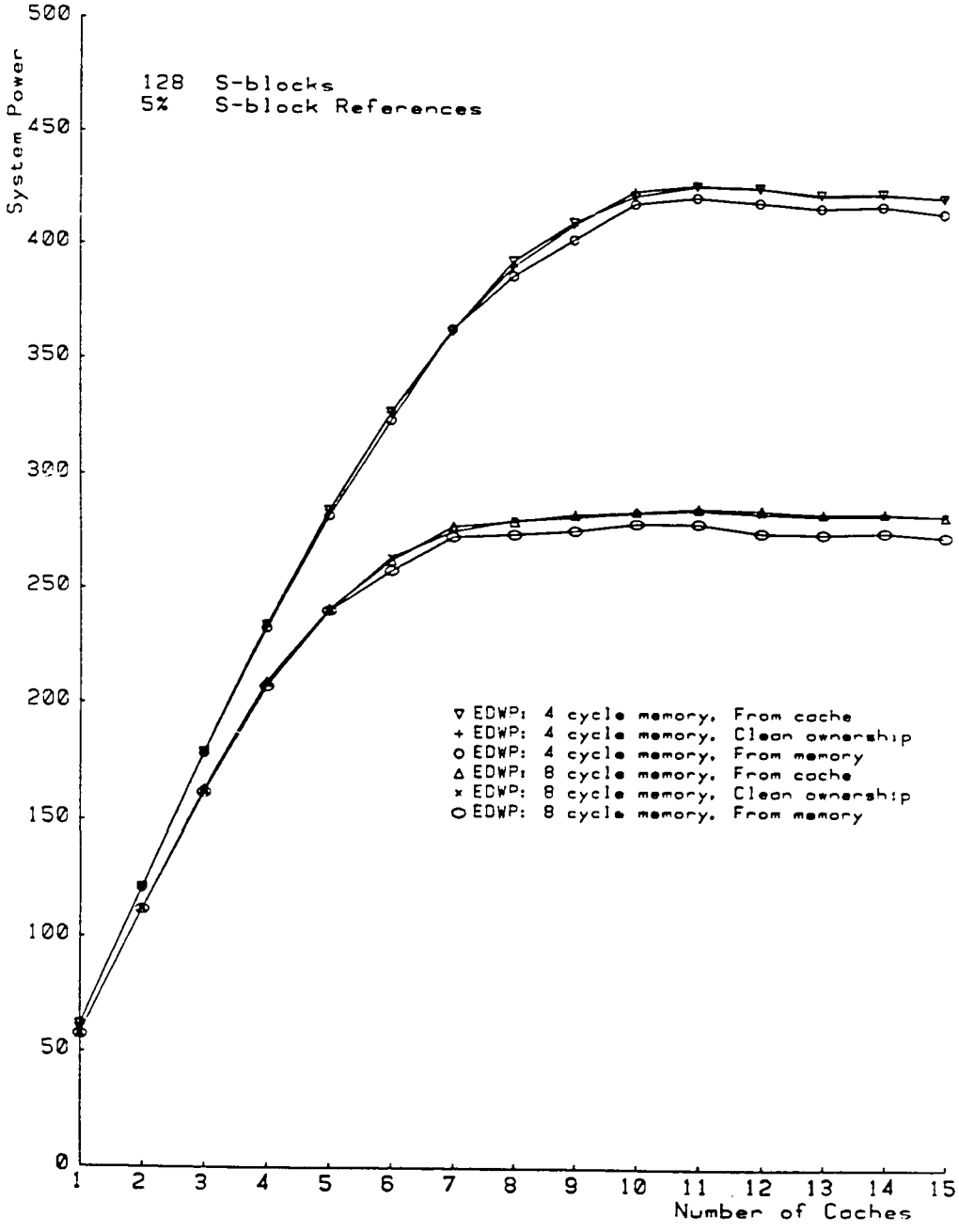


Figure 4.37: EDWP: advantage of clean ownership

In principle, invalidation could take place after any number of distributed writes with no intervening references. For example, invalidation on the first distributed write (with no REMOTE-WRITE states) results in an invalidation protocol. With one REMOTE-WRITE state, invalidation takes place on the second distributed write. To determine which number of states gives the best performance, three different versions of EDWP were simulated, using one, two, and three REMOTE-WRITE states. Figure 4.38 shows an example of the simulation results. As can be seen, using a single state results in more overhead than using two or three, as the traffic caused by misses resulting from the invalidations is far greater than the distributed write traffic saved by the early invalidation. Using a third REMOTE-WRITE state appears to offer little if any performance improvement. The decision to use two REMOTE-WRITE states in EDWP is based on these simulation results.

It can be observed that the performance difference between EDWP and the Dragon is greatest with 128 S-blocks in the simulation. With 16 S-blocks, sharing is high and there are many distributed writes, and it is very likely that blocks continue to be referenced in other caches. With 1024 S-blocks, there is little sharing and there are therefore few distributed writes, so elimination of any fraction will have little impact on overall performance. With 128 S-blocks, the level of sharing is quite high and therefore the distributed write traffic is also fairly high. But since S-block references are distributed over so many blocks, the probability that blocks continue to be referenced by other caches is relatively low. Hence, it is at this level of sharing that the elimination of unnecessary distributed writes has the greatest impact.

Although the figures indicate that the elimination of distributed writes has little impact on performance, the actual difference in performance could be considerably higher in a multiprocessor system with frequent task migration. The simulation does not directly include overhead resulting from task migration. The Dragon policy of continuing distributed writes until the block is replaced in the other caches will incur more overhead than the EDWP approach which limits the number of distributed writes.

4.7.3 Comparing Distributed Write and Invalidation

Figures 4.39 and 4.40 combine results of both invalidation and distributed write protocols. Included are the best and worst of the invalidation and distributed write approaches. Two additional protocols are also included for comparison purposes–a simple software approach, and a simple write-through scheme. In the write-through scheme, all writes require a write-through to main memory that invalidates all other cached copies. On write misses, blocks are not loaded into the

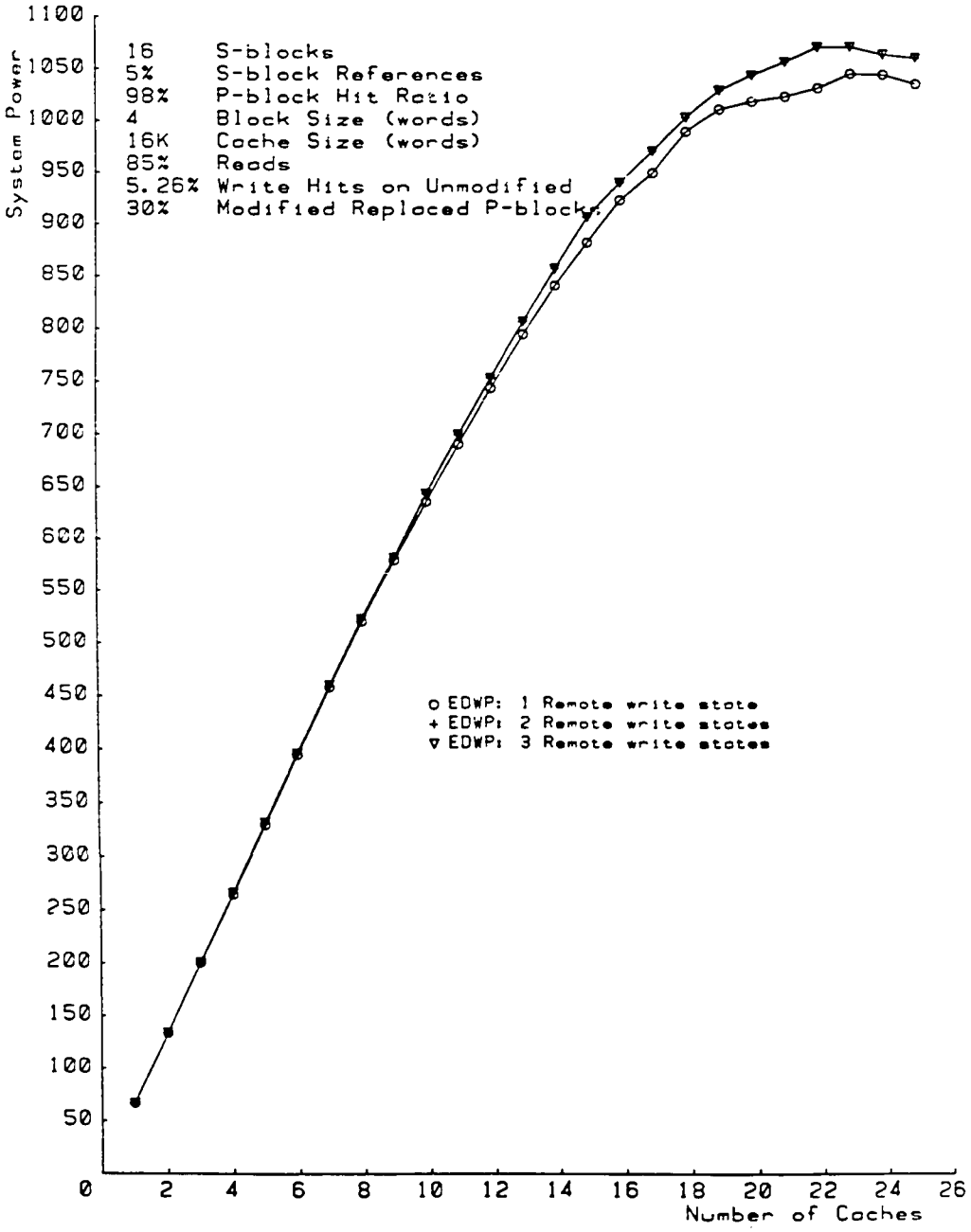


Figure 4.38: EDWP: effects of remote write states

cache. Read misses are always serviced from main memory since it is always up-to-date. The software scheme assumes that all blocks are tagged as shared or private. Private blocks are cached with no coherence overhead, while blocks tagged as shared are not allowed to be cached. All reads and writes for shared blocks must access main memory. Such an approach would also necessarily require either elimination of task migration or cache flushes on context switches, but this overhead is not reflected in the simulation.

As can be seen, the general performance of the distributed write schemes is far better than the invalidation schemes, although the performance of EIP is comparable to the Firefly. The software scheme performs reasonably well if there are very few S-block references. If the percentage of S-block references is high, the performance falls considerably–even though the actual sharing might be much lower. The write-through performance is well below all others including the Synapse, indicating that all write-back protocols offer a significant performance advantage over write-through.

If task migration overhead were included in the simulation, the relative performance of invalidation schemes would improve. In general, invalidation is more appropriate for sharing resulting from task migration where only one processor will continue to reference the block. On the other hand, distributed writes are more appropriate in the case of 'true sharing', where multiple processes access a shared variable. Of course, the two types of sharing are difficult to distinguish. Of the schemes described in this chapter, only the EDWP and CMU RWB use strategies to dynamically differentiate between the two. Later in this chapter, two additional schemes will be described that use different approaches to this problem. The first protocol reverses the steps of EDWP by first invalidating and then performing distributed writes if the block is shared. The second protocol makes use of software help to distinguish between task migration and actual sharing.

4.8 Implementation Considerations

Improvements in performance are generally the result of increased hardware complexity and cost. The complexity of the bus is an important consideration. As was previously mentioned, write-once is able to work with the existing Multibus protocol without modification. The Dragon and Firefly schemes require a bus with a dedicated line to detect sharing. Similarly, the Illinois and Futurebus protocols assume that a cache can detect whether a block came from memory or a cache, which could be implemented with an added bus line as with Dragon and Firefly. The EIP and EDWP both require two special bus lines–one to detect sharing and the other to implement clean ownership.

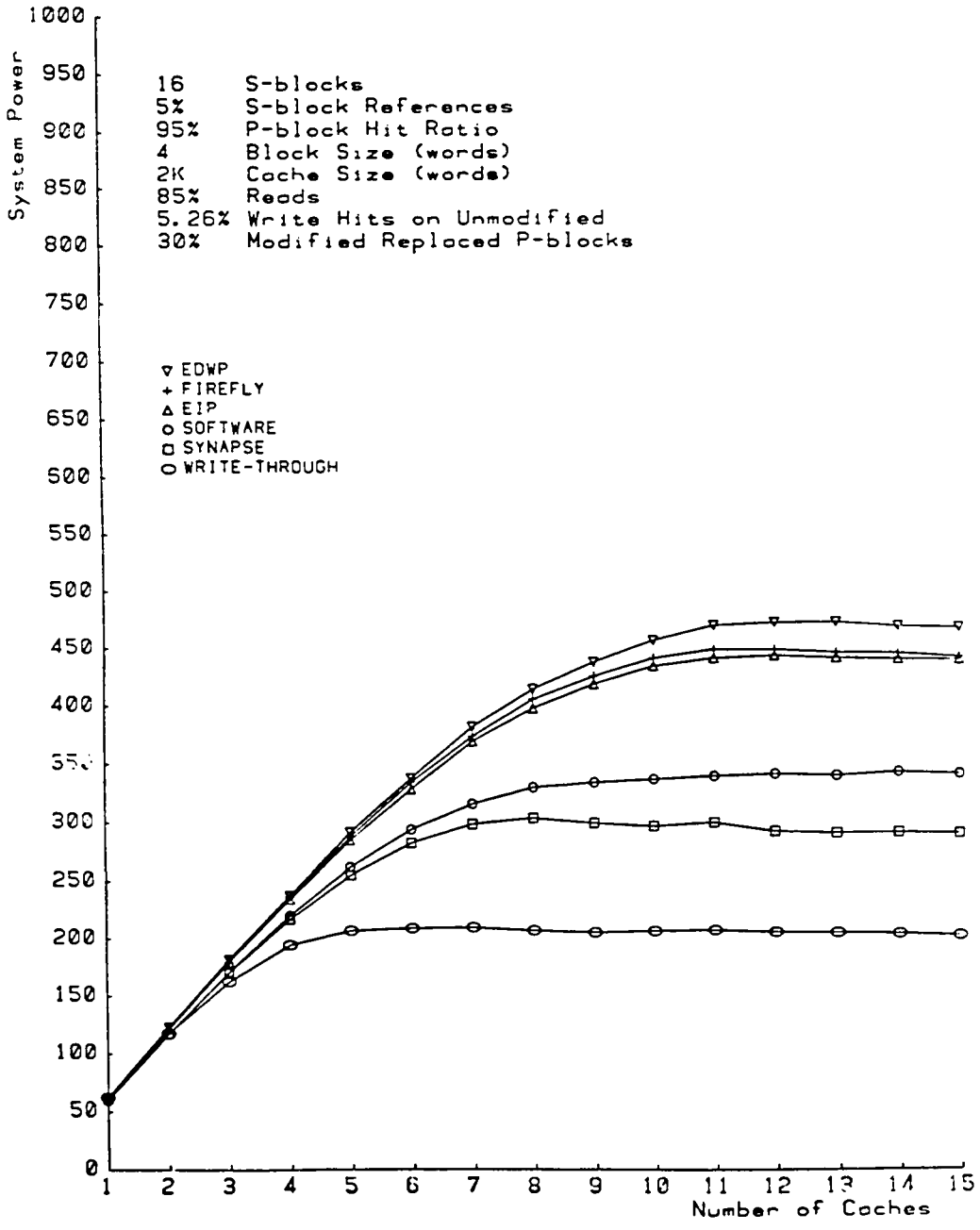


Figure 4.39: Comparison of inv. and dist. write, basic model

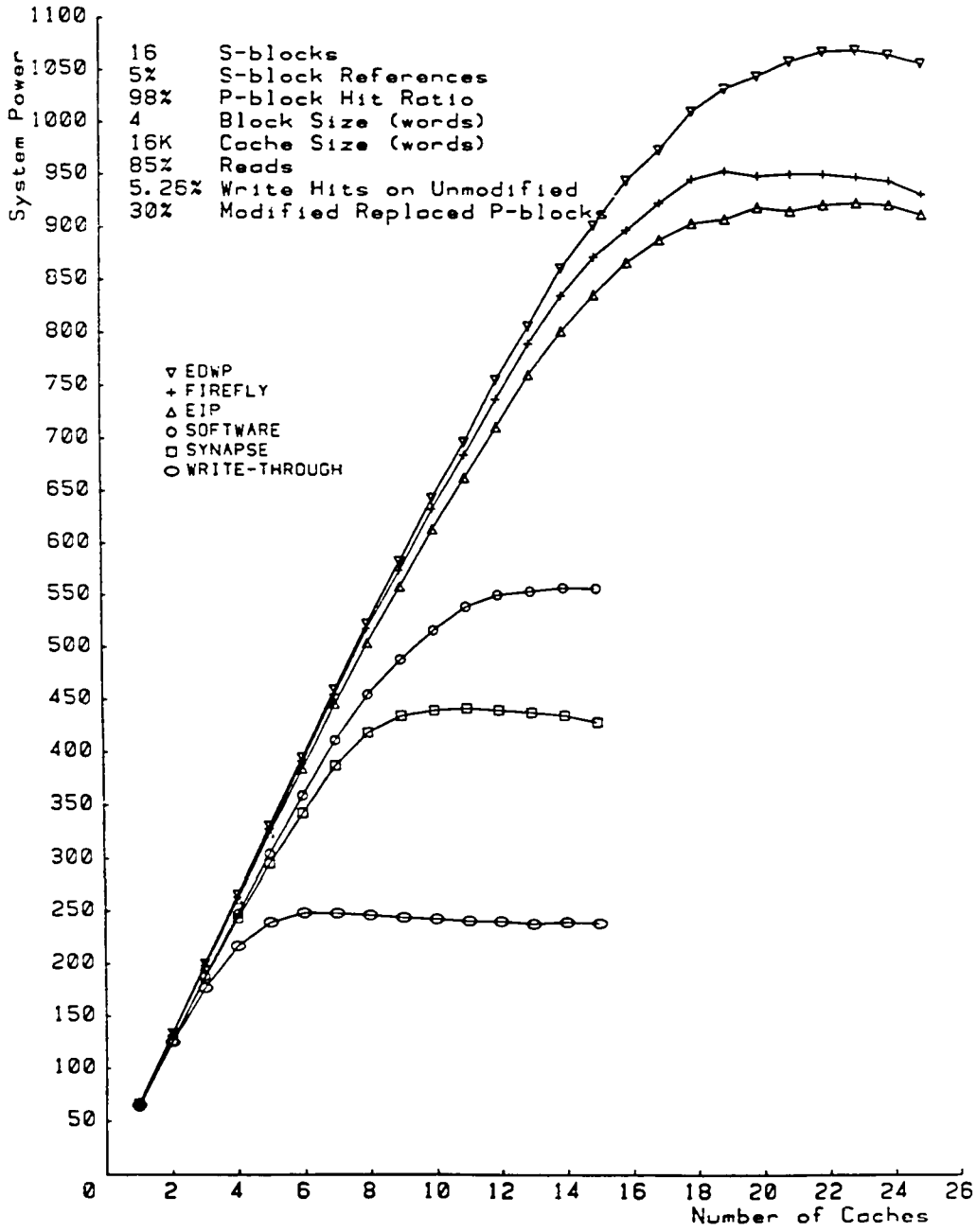


Figure 4.40: Comparison of inv. and dist. write, increased cache size

Both Illinois and Firefly always obtain clean blocks from other caches if they are cached. In the Illinois approach, exactly one cache will succeed in putting the data on the bus–the cache with the highest priority. Since that cache may be busy servicing a memory request, the bus arbiter or prioritizer might need to wait for it to respond, increasing the service time of the request. The Firefly assumes that *all* caches with a copy will succeed in putting the block on the bus, which requires a bus with fixed timing. These considerations, coupled with the possibility of slowing down the processors of those caches which succeed in putting data on the bus, have led some designers to conclude that it is more efficient to obtain the data from memory whenever possible. The concept of clean ownership simplifies these implementation concerns, since on all misses the block will be supplied by a unique source.

All protocols but Synapse assume that each cache has the capability to inhibit memory from responding to a request for a block when a modified copy of the block is present in that cache. The Synapse scheme uses a single bit in main memory for each block to indicate whether or not memory is to respond to requests for that block. This requires additional memory and specially designed memory controllers, but it avoids problems arising when the cache with the modified block is delayed in responding. This is particularly important in the Synapse implementation, because there are two system buses and each cache has two separate bus snoops. It is possible that a cache might be busy in activities on one bus, while at the same time the other bus has a transaction that the cache needs to respond to (such as providing a copy of a modified block). The bit in memory simplifies the concerns quite drastically in this case, since it is not critical that the cache respond within a particular time interval.

Additional capabilities of the bus are assumed by those schemes in which cache-to-cache transfers of modified blocks are written back to main memory at the same time (e.g., Illinois, Firefly, write-once). The added complexity of having three cooperating members on a bus can be avoided simply by performing the write-back as a separate bus operation but this results in lower performance. Note that schemes with a MOD-SHD state avoid the problem altogether since the block is not written back to memory at all. Synapse has no cache-to-cache transfer–the block can be loaded in the requesting cache only after the block is written back to main memory.

In our simplified model the bus remains busy until the entire memory cycle has completed, although in the case of a write, the cache is allowed to continue as soon as the data is put on the bus. One possible modification to our basic model would be to allow the bus to begin servicing the next transaction before a write has completed (assuming no contention for the same memory module). This would significantly reduce the cost of all writes, but it would not complicate coherence concerns. With this type of write implementation, the cost of a single word write could

approach that of an invalidation signal, increasing the relative performance of write-once, Firefly, and write-through.

In addition to the bus complexity, the complexity of the cache controller is an important consideration. In particular, the controller must maintain a local state associated with each block in the cache directory and complete transitions from state to state as defined by the protocol. While the size of the state information is not a concern (the most complex requires three bits for eight states, compared with two bits for three states in the simplest), the logic required to implement the state transitions varies greatly. In general, the complexity of the controller as a state machine corresponds to the complexity of the state diagrams for each protocol. The EIP and EDWP diagrams employ more local states and are much more complicated than the diagrams of the other protocols, and would therefore be more costly to implement.

4.9 Alternative Protocols

There are, of course, other hardware-based coherence protocol alternatives than those presented in this chapter. Although the EIP and EDWP exceed the performance of all protocols previously proposed, there is no reason to believe that they offer *optimal* performance in their respective classes. The development and analysis of additional shared bus protocols remains as a promising research topic.

Although an analysis of their performance is beyond the scope of this dissertation (as it would require a more sophisticated simulation model), two schemes are outlined here that are examples of protocols that could offer performance exceeding that of the schemes presented earlier in this chapter. To accurately evaluate their relative performance, it would be necessary to include the overhead resulting from task migration. These protocols are likely to demonstrate better performance on this type of sharing than the efficient distributed write protocols described earlier. But this increase in performance comes at the cost of slightly increased overhead on actual shared blocks in the first scheme, and at the cost of software assists in the second.

4.9.1 Invalidate, then Distributed Write

This is an outline of a scheme which first invalidates shared blocks when modified, and then uses distributed writes if the value continues to be referenced. This approach would be very effective if a significant portion of the sharing that takes place is the result of task migration. Possible states for blocks in local caches are:

1. INVALID. (INV)
2. UNMODIFIED-EXCLUSIVE-I. (UNMOD-EXC-I)
3. UNMODIFIED-SHARED-I. (UNMOD-SHD-I)
4. MODIFIED-SHARED-I. (MOD-SHD-I)
5. MODIFIED-EXCLUSIVE-I. (MOD-EXC-I)
6. UNMODIFIED-EXCLUSIVE-II. (UNMOD-EXC-II)
7. UNMODIFIED-SHARED-II. (UNMOD-SHD-II)
8. MODIFIED-SHARED-II. (MOD-SHD-II)
9. MODIFIED-EXCLUSIVE-II. (MOD-EXC-II)

The protocol uses two separate sets of states–I and II. In set I, write hits on unmodified blocks will cause invalidations. In set II, they will result in distributed writes. A transition from I to II takes place when misses occur on INV blocks present in the cache. Blocks remain in states from set II until they are completely replaced. All copies are either in set I or in set II at any given instant in time. For cache hits on INV blocks (normally treated as cache misses), separate miss signals are assumed. In the following discussion they will be referred to as INV read miss and INV write miss. Two special bus lines are assumed: *SHARED* and *DW*, the latter being used to signal that a block is in states in set II, where distributed writes are to be used.

The protocol works as follows:

- **Read miss.** Any cache with a MOD copy supplies the data and raises the *SHARED* line–otherwise the block comes from main memory. Any cache with a block in a II state also raises the *DW* line. Any caches with valid copies also raise the *SHARED* line. In addition, any cache with an EXC copy must change to the corresponding SHD state (i.e., UNMOD-EXC-I to UNMOD-SHD-I, MOD-EXC-II to MOD-SHD-II, etc.). The cache loading the block tests the *SHARED* and *DW* lines to determine the state of the block. If both are high, the block is loaded in UNMOD-SHD-II. If only *SHARED* is high, the state is UNMOD-SHD-I. If *SHARED* is low, *DW* must also be low and the state is UNMOD-EXC-I.

- **INV read miss.** Any caches with copies (including INV ones) raise the *SHARED* line. All caches with valid copies change from set I to set II (i.e., UNMOD-SHD-I to UNMOD-SHD-II, etc.) and also from EXC to SHD. Thus, all other copies are necessarily in UNMOD-SHD-II

or MOD-SHD-II after the INV read miss. The cache loads the block (from cache or memory as with regular read miss) in state UNMOD-SHD-II if *SHARED* is high, otherwise the block is loaded in state UNMOD-EXC-II. Any caches with an INV copy take the new data, update their local copy, and set their local state to UNMOD-SHD-II.

- **Write hit**. If the block is in either MOD-EXC state, the write takes place immediately in the local cache. If the state is either UNMOD-EXC, the state is changed to MOD-EXC and the write takes place directly. If the state is UNMOD-SHD-I or MOD-SHD-I, the write takes place after an invalidation signal is sent and the local state is changed to MOD-EXC-I. If the state is UNMOD-SHD-II or MOD-SHD-II, a distributed write takes place, updating all other copies and changing their states to UNMOD-SHD-II. The *SHARED* line is used to determine if sharing continues, just as in the other distributed write schemes. If caches continue to accept the new data, they raise the *SHARED* line and the new state in the writing cache is MOD-SHD-II. If the line is not raised, the new state is MOD-EXC-II.

- **Write miss**. All caches with II copies raise the *DW* line. The write miss will result in an exclusive copy (as other invalidation schemes) unless the block is known to be in set II and presumed shared. Any I copies that exist (after supplying the data, if a MOD copy exists) are changed to INV. If the block is in set II, EXC copies are set to SHD. The loading cache tests *DW* to determine its actions. If the line is high, other II copies exist and the block is loaded in state MOD-SHD-II and a distributed write occurs, updating all other copies and changing their states to UNMOD-SHD-II. If *DW* is not raised, the copy is unique and the state is set to MOD-EXC-I.

- **INV write miss**. All caches with copies raise the *SHARED* line. All states in set I are changed to set II, and all EXC copies are set to SHD. If the *SHARED* line is high, the block is loaded into the cache in state MOD-SHD-II and a distributed write takes place, setting all other states to UNMOD-SHD-II. If the line is low, the copy is unique and the state is set to MOD-EXC-II. As with the INV read miss, all caches with INV copies take the new value, raise the *SHARED* line, and set the state to UNMOD-SHD-II.

To accurately evaluate the relative performance of this new protocol, a higher level simulation would be required, including the overhead of task migration. For task migration overhead, it is as efficient as the invalidation techniques. For actual sharing, only one invalidation and subsequent miss in any cache are sufficient to cause a change to using distributed writes. Note that the first miss (either read or write) causes all caches to be reloaded with the shared data. This minimizes

the one-time penalty for invalidations on shared blocks.

The addition of UNMOD-SRC states and a *MODIFIED* bus line would allow the implementation of clean ownership. It was not included in the above description to avoid unnecessary complication, but it could be added if the performance increase is sufficient to warrant it.

4.9.2 A Protocol Using Software Hints

This is an outline of a protocol that could offer a very high level of performance using software tags as *hints*. The compiler generated tags would indicate whether the data is shared or private. These hints are different from the tags of software schemes because the protocol would still maintain coherence even if some section of data were incorrectly tagged, although the overall performance would suffer. Like the tags in Smith's OTI scheme described in Chapter 2, the tags in this approach could be maintained for each page, with a special entry in the TLB so the value of the tag could be known at the time of each attempted cache access.

The basic idea of this protocol is to use invalidation in the sharing of blocks tagged as private, and distributed writes with data tagged as shared. Like the efficient distributed write schemes, a *SHARED* bus line is assumed. The scheme uses the following states:

1. INVALID. (INV)
2. UNMODIFIED-EXCLUSIVE. (UNMOD-EXC)
3. UNMODIFIED-SHARED. (UNMOD-SHD)
4. MODIFIED-SHARED. (MOD-SHD)
5. MODIFIED-EXCLUSIVE. (MOD-EXC)

In this scheme, the main idea is to treat misses on shared blocks differently that misses on private blocks. In fact, there is a single shared block miss signal (for both reads and writes), and a single private block miss signal. The actions of the protocol are described as follows:

- **Read miss or write miss on private.** All private block misses cause a *exclusive load* to occur that results in an exclusive copy in the requesting cache. If a MOD-SHD or MOD-EXC copy exists, the cache with the copy supplies the data and set its state to INV. All caches with copies also set their states to INV. The block is always loaded in UNMOD-EXC on a read miss, and MOD-EXC on a write miss.
- **Read miss or write miss on shared.** All shared block misses cause a normal load operation to occur. If a MOD-SHD or MOD-EXC copy exists, the cache with the copy

supplies the data, raises the *SHARED* line, and sets its local state to MOD-SHD (if MOD-EXC). If no MOD copy exists, the block comes from main memory. In either case, all caches with copies raise the *SHARED* line and change to the appropriate SHD state if EXC. On a write miss, if *SHARED* is high when the block is loaded, the local state is set to MOD-SHD and a distributed write takes place. If *SHARED* is low, the local state is set to MOD-EXC and the write takes place directly. On a read miss, if *SHARED* is high, the local state is UNMOD-SHD. If the line is low the local state is set to UNMOD-EXC.

- **Write hit.** If the block is MOD-EXC the write may take place without delay. If the block is UNMOD-EXC the write may take place directly, accompanied by a state change to MOD-EXC. If the state is UNMOD-SHD or MOD-SHD a distributed write must take place. The new value is placed on the bus and taken by all other caches with copies, which change their states to UNMOD-SHD and raise *SHARED*. The cache performing the write tests the *SHARED* line after the distributed write completes. If it is low, the block is no longer shared and the state is changed to MOD-EXC. Otherwise at least one cache still has a copy and the state is set to MOD-SHD.

The software hints would provide the protocol with information that is difficult for schemes based entirely in hardware to determine: whether the observed sharing is truly shared data referenced by multiple processes, or whether it is private data present in multiple caches as a result of task migration. The use of the hint replaces the complicated mechanisms of other schemes that determine local usage–three distributed writes with no intervening references of EDWP, two consecutive writes in CMU RWB, and misses on INV blocks in the scheme in the previous section.

There are, of course, additions and modifications that could be made. One possibility would be to use a normal read miss signal (instead of an exclusive load) for read misses on private blocks. In this case, write hits on shared blocks would be treated differently, with invalidations on private blocks and distributed writes on shared. A second alternative would be the addition of an UNMOD-SRC state and *MODIFIED* line to implement clean ownership, as in EIP and EDWP.

This is not the first proposal to make use of compiler generated hints. In the paper describing the Berkeley protocol, it is observed that such hints, coupled with additional local state information (to add the equivalent of an UNMOD-EXC state) would improve performance. In particular, if private blocks were tagged as such, they would be loaded in this state causing other copies to be invalidated, just as on a write miss. Subsequent writes will not incur the overhead of invalidation signals. Like the scheme described here, the Berkeley scheme still functions correctly if the tags are marked incorrectly, although the performance suffers.

4.10 Additional Shared Bus Topics for Further Research

In addition to the discovery of fundamentally new protocol approaches, there are a number of important issues that should be more fully investigated that could affect the performance of all schemes. Examples of such investigations are schemes that do not hold the bus until each transaction is completed, and schemes that make use of multiple system buses.

In all of the schemes described in this chapter, the bus is held until the memory cycle has completed. The utilization of the bus could be reduced by allowing operations to overlap during the accessing of main memory and by using packet switching. Clearly the coherence concerns are complicated if overlapping operations should involve the same block. The most straightforward approach would be to prohibit each cache from submitting a request until all outstanding requests for that block have completed. How this might best be implemented, and whether or not it is the best approach to take would be important questions to address.

A similar topic is the complication of adding buses to the multiprocessor. In a system with multiple buses (as the Synapse), each cache unit must have bus watching logic for each bus. This can lead to serious race conditions if the same cache needs to respond to requests on more than one bus at the same time. Even worse, if all buses connect all caches to all memory modules (instead of each bus connecting to a subset of modules), it might be possible to have simultaneous transactions on all buses that involve the same block. In such a case, it might be necessary to implement an extra bit in memory for each block (as in the Synapse) to prevent problems occuring from pathological cases. While this might provide a solution to race conditions between cache and memory, it would not solve problems resulting from race conditions between caches. For example, simultaneous distributed writes for the same block on different buses might be seen in a different sequence at different caches, leaving them with different values for the same block. It appears that the use of multiple buses is also deserving of further investigation.

A research topic that has not been considered here is an investigation into *compatible* protocols–different protocols that can be used simultaneously in the same multiprocessor by different caches but still maintain consistency [SA86]. This topic is pursued specifically to consider the possibility of using processor-cache boards of different designs (but using a common system bus) at the same time in in the same system. For the protocols proposed in this dissertation, it is assumed that all caches are using the same coherence protocol. Whether or not these are compatible with other protocols in the sense described above is a question beyond the scope of this work.

Other important research topics relate to the modelling and analysis of multiprocessor performance. Future analysis of shared bus protocols would be more accurate if the simulation model

were extended and more detailed. In particular, the overhead of task migration could be included to make existing comparisons more accurate. In addition, the more complex simulation model could be used to determine the performance of the schemes described in the previous two sections. Comparison of hardware protocols and software schemes would be more feasible if the shared data class were separated into shared writable data, shared read-only data, and possibly non-cacheable semaphores or synchronization variables. The addition of these additional classes (each treated differently by some software schemes), combined with a higher level workload model (e.g., encapsulating shared variable accesses in critical sections), could allow a simulation model general enough to compare hardware and software schemes.

Validation of the existing simulation model and selected parameter ranges remains an important topic. The actual amount of sharing that will be observed in the normal operation of shared memory multiprocessors is very much a matter of debate. Any measurements of the reference behavior of shared variables would be very important. Specifically, any or all of the following measurements taken from a wide range of applications would be of interest: the percentage of references to shared variables, the locality of those references, the frequency of accessing synchronization variables associated with widely used resources, and the percent of sharing resulting from task migration. Ideally, the Dragon and Firefly workstation projects may soon provide us with some of this information.

Chapter 5

Protocols for General Interconnection Networks

5.1 General Approach

There are several key differences between protocols for shared buses and protocols for other interconnection networks. Without a common bus, cache controllers cannot observe each other's transactions, so it is not possible to maintain coherence with a mechanism located only at the cache level. The protocols described in this chapter include local state information, maintained by each cache, and global state information, maintained by each memory controller. Each protocol includes a specification of the communication necessary to keep the two sets of states consistent. As was the case with the shared bus schemes, the performance of these protocols depends on the minimization of global actions. The local state information is used to allow the cache to process most requests locally, although some require communication with the memory controller. The memory controller determines its actions based on the global state of the block and the type of cache request that is to be serviced. These controller actions may include sending commands to other caches, changing the global state, and returning a signal to the requesting cache.

With no shared bus to synchronize global actions, the cache coherence protocols for general interconnection networks must include provisions for asynchronous operation. The protocols presented in this chapter should work reliably in a packet-switching network, in which it would be possible to have several requests or messages in the network at the same time for the same block.

This condition can lead to race conditions and timing problems in the protocols, as will be seen. In general, the only assumptions made about the network is that it does not lose any messages, and that the message order from a source to a destination is preserved. In other words, if source A sends two signals to destination B, the signals will always arrive in the order in which they are sent. It is possible, however, to have an arbitrary delay in the sending of any particular signal, and it is also possible for destination C to receive a signal from source A before destination B receives a signal sent from source A that was sent before sending the signal to A.

There are two important consequences of the asynchronous operations of the protocols that should be mentioned here. First, all protocols are invalidation-based, allowing only a single writer and invalidating all other copies. Given the nature of the potential timing problems, an implementation of a distributed write protocol would be very difficult. Second, although these schemes also provide consistent caches, the nature of that consistency is weaker than that provided by the shared bus schemes. This topic is dealt with in detail in Chapter 6.

We begin this chapter by describing and discussing previously proposed protocols for general interconnection networks. Following this discussion, a new family of protocols is presented, based on a different organization of the global table. Simulation results of the schemes using a crossbar switch are presented and discussed. The chapter concludes with a discussion of possible extensions to these protocols and topics for further research.

5.2 Previously Proposed Protocols

Each of the schemes in this section is described using a uniform terminology. Each protocol description includes a local state list, a global state list, a list of request types from the cache to memory, and a list of commands and responses from memory to the cache. Following the lists of possible states and signals, the protocol specified actions of the multiprocessor are outlined for cache hits, cache misses, and replacement.

5.2.1 Full Map Approach

Proposed by Censier and Feautrier in 1978 [CP78], this scheme was to be implemented in the S-1 multiprocessor under development at Lawrence Livermore Laboratories. The protocol uses the following local states.

1. INVALID. (INV)
2. UNMODIFIED-SHARED. (UNMOD-SHD)

3. MODIFIED-EXCLUSIVE. (MOD-EXC)

The global state is encoded in an $n+1$ bit tag associated with each block in main memory, where n is the number of caches in the system. Each of the first n bits are called *cache bits* and the remaining bit is the *modified bit*. A cache bit is set if and only if the associated cache has a valid copy of the block. The modified bit is set if the copy in main memory is no longer up-to-date. If this bit is set, there is exactly one cache with a valid copy, and the local state of that copy is MOD-EXC. If the bit is not set, all valid copies (if any) will be UNMOD-SHD. Because the global state includes the identity of all caches with copies, this protocol is a *full map* approach.

Although the number of possible encodings of the $n+1$ bit tag is very large, they can be divided into the following three classes:

1. ABSENT. No cache bits are set. Modified bit is not set.

2. PRESENT+. One or more cache bits are set. Modified bit is not set.

3. PRESENTM. Exactly one cache bit is set. Modified bit is also set.

Throughout the following discussion, the global state is said to be ABSENT, PRESENT+, or PRESENTM. Implicitly, this includes the value of the cache bits indicating exactly which caches have copies.

For each of the following signals, **cache#** is the unique identity of the cache sending or receiving the signal, **block#** is the address of the block involved, and **data** indicates the actual transmission of the block. The memory controller sending or receiving the signal can be determined from the block address. The set of signals from the cache controller to the memory controller is the following:

- **Read-miss(cache#,block#)**. This signal requests the controller to transmit a copy of the specified block to the specified cache.

- **Write-miss(cache#,block#)**. This signal requests the memory controller to transmit an exclusive copy of the block.

- **Modify-request(cache#,block#)**. A write hit on an UNMOD-SHD block has occurred. This signal is a request to obtain permission to modify the block.

- **Replace-unmodified(cache#,block#)**. This signal informs the memory controller that an UNMOD-SHD block has been replaced in the cache.

- **Replace-modified(cache#,block#,data)**. This signal informs the memory controller that a MOD-EXC cache block has been replaced. It includes a block transfer to update main memory.

- **Update(cache#,block#,data).** This is a response to a memory controller request to update the main memory copy, and it includes a block transfer. The local state has been changed from MOD-EXC to UNMOD-SHD.

The set of signals from the memory controller to the cache are:

- **Read-data(cache#,block#,data).** This is a block transfer in response to read miss.
- **Write-data(cache#,block#,data).** This transfers an exclusive block copy in response to a write miss.
- **Modify-granted(cache#,block#).** This message grants permission to the requesting cache to proceed with the modification of an UNMOD-SHD block.
- **Write-back-shd(cache#,block#).** This signal requires the cache to perform a write-back of a MOD-EXC block, setting the local state to UNMOD-SHD.
- **Write-back-inv(cache#,block#).** This signal also requires the cache to perform a write-back, but the local state is set to INV upon completion.
- **Invalidation(cache#,block#).** This requires the cache to change the state from UNMOD-SHD to INV.

The operation of the protocol is described below. (Figure 5.1 and 5.2 contain state transition diagrams for the caches and memory controllers).

- **Read hit.** The data is returned to the processor. No action needs to be taken by the protocol.
- **Read miss.** The cache sends a **Read-miss** to the appropriate memory module. When it receives the signal, the memory controller tests the global state of the block. (The access time of the global table can be overlapped with the time to read the block, provided that the block read can be aborted if the modified bit is set and the memory copy is invalid.) If the global state is ABSENT or PRESENT+, the block may be sent directly using the **Read-data** signal. In addition, the global state is updated by setting the appropriate cache bit. If the state is PRESENTM, a valid copy must be obtained from the cache with the MOD-EXC copy, currently the only valid copy in the system. A **Write-back-shd** signal is sent to the cache, which sends an **Update** and sets its local state to UNMOD-SHD. When the **Update** is received, the memory controller clears the modified bit and sets the appropriate cache bit in the global state (thus changing the global state from PRESENTM to PRESENT+), and

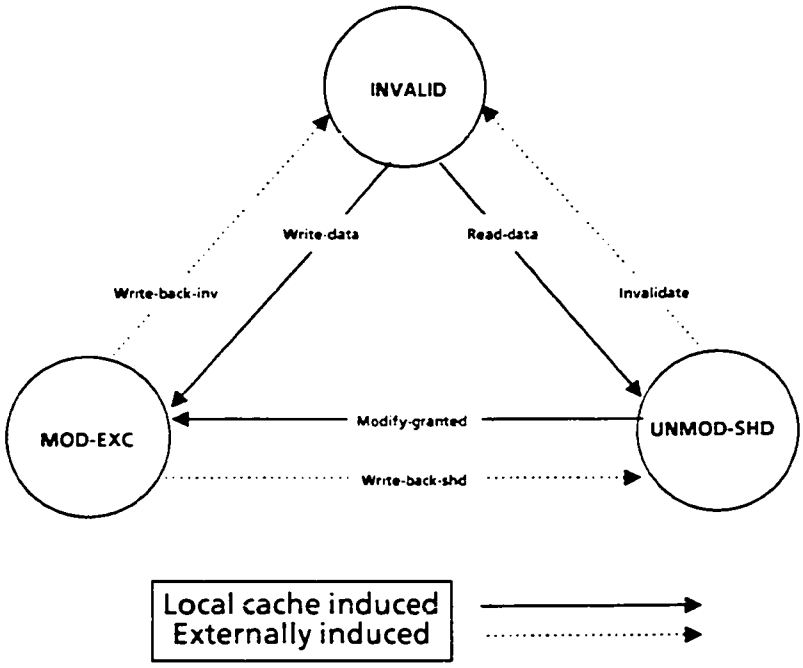


Figure 5.1: Full map protocol cache state transitions

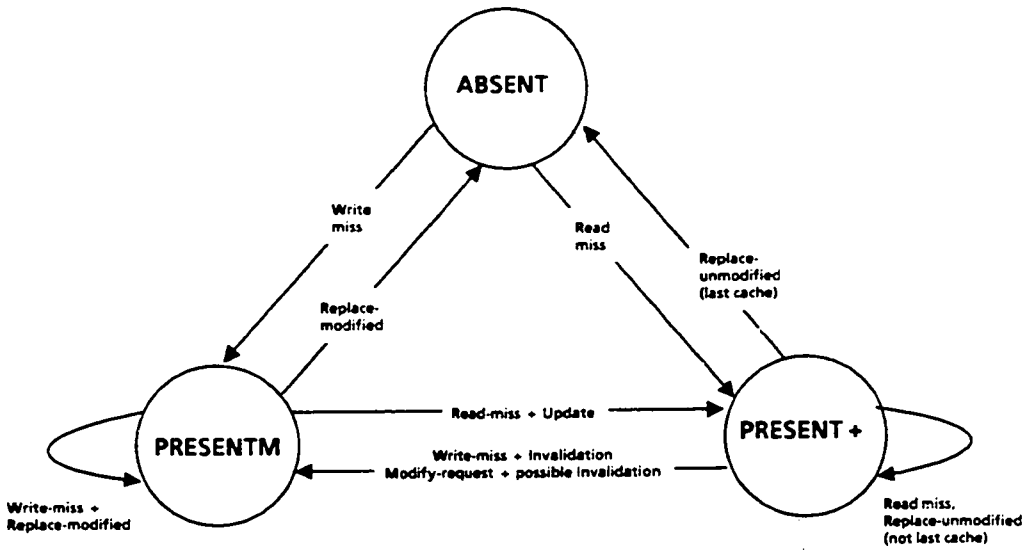


Figure 5.2: Full map protocol global state transitions

sends the **Read-data**. In all cases the block is loaded in state UNMOD-SHD. In general, during the time that the memory controller is waiting for a response from a cache, it could be allowed to process other incoming requests so long as they involve different blocks.

- **Write hit**. If the state is MOD-EXC, the write may proceed locally. If the state is UNMOD-SHD the cache must submit a **Modify-request** to the appropriate memory controller, which checks the global state of the block upon receiving the signal. The only possible global state of the block is PRESENT+, and it must be determined how many cache bits are set. For each cache bit that is set (other than the bit of the requesting cache), an **Invalidation** signal must be sent to that cache. After the necessary signals are sent, if any, the controller sends a **Modify-grant** to the requesting cache, clears the cache bits of all other caches, and sets the modified bit (making the global state PRESENTM). The cache receives the **Modify-grant**. performs the write, and sets the local state to MOD-EXC.

- **Write miss**. The cache sends a **Write-miss** to the controller, which receives the signal and checks the global state of the block. If the state is ABSENT, the block can be transferred immediately using the **Write-data** message. If the state is PRESENT+, all caches with the cache bits set must be sent an **Invalidation** signal first. If the state is PRESENTM the cache with the only copy must be sent a **Write-back-inv** signal and the controller must wait for the response. The cache with the MOD-EXC copy receives the **Write-back-inv**, sends a **Replace-modified** and sets the local state to INV. When the **Replace-modified** arrives at the controller, the block is sent to the requesting cache, and the global state is updated. In all cases the global state is set to PRESENTM (with the cache bit of the requesting cache set and the modified bit set). The block is always loaded in state MOD-EXC after a write miss.

- **Replacement**. If the block to be replaced is UNMOD-SHD, a **Replace-unmodified** signal is sent to the memory controller, which takes the signal and simply clears the appropriate cache bit for the block. The new global state is therefore PRESENT+ or ABSENT depending on the number of cache bits that are still set. If the replacement block is MOD-EXC, a **Replace-modified** must be sent to update main memory. The memory controller takes the data from the write-back, updates memory, and resets the global state to ABSENT by clearing the cache bit and the modified bit.

There are a number of subtle timing problems in the full map protocol that illustrate the complexity of a precise protocol specification. Some of these were not considered by Censier and

Feautrier in their protocol description. For example, consider the situation (illustrated in Figure 5.3) when caches A and B have UNMOD-SHD copies of block k and each tries to write the block at about the same time. A sends a **Modify-request**(A,k) and B sends a **Modify-request**(B,k) to memory controller X, which receives the request from A first. In processing A's request, the memory controller sends an **Invalidation**(B,k), which B receives while waiting for a **Modify-granted**(B,k). To resolve the situation correctly, cache B must treat the **Invalidation**(B,k) as a negative **Modify-granted**(B,k) by invalidating its copy and proceeding as if a write miss had occurred. However, Censier and Feautrier make no provision for treating an **Invalidation** signal as a negative response to a **Modify-request**–the protocol presented in their paper does not consider this possibility.

The above example leads to additional problems: After the initial **Modify-request**(A,k) is serviced and the **Modify-granted**(A,k) is sent, the controller will service the pending **Modify-request**(B,k) signal. At this point, cache B no longer has a valid copy of k. This must be determined by the memory controller by checking B's cache bit in the global state. Normally the bit will be set and the **Modify-request** may be processed normally. However, in this case, the cache bit is not set, and the **Modify-request** must be treated differently. In the paper describing this protocol, the authors do not include this necessary test in the procedure describing the actions of the memory controller in servicing a **Modify-request**. The sequence of actions given in this example would cause their protocol (as specified) to malfunction.

Receiving requests from caches which are no longer considered to have a valid copy when the request is processed is a problem that all general interconnection network protocols must deal with. Such signals will be referred to as *ghost* signals. More precisely, a ghost signal is a signal received from a cache that experiences a local state change (caused by the actions of another cache) after it sends the request.

There appear to be two different approaches that can be taken to resolve the problem in the above example. The first alternative is for the cache to submit a **Write-miss** when the **Invalidation** is received. The memory controller must test the cache bit of the cache from which a **Modify-request** is received. If the bit is not set, the signal must be ignored. A second alternative (used in the simulation of this protocol) is for the cache to do nothing besides invalidating the block when the **Invalidation** is received. In turn, when the **Modify-request** arrives at the memory controller from a cache for which the cache bit is not set, it is treated as a **Write-miss**, including the actions outlined above followed by a full block transfer. This second alternative would result in better performance, since it does not require additional signals and the associated delay time.

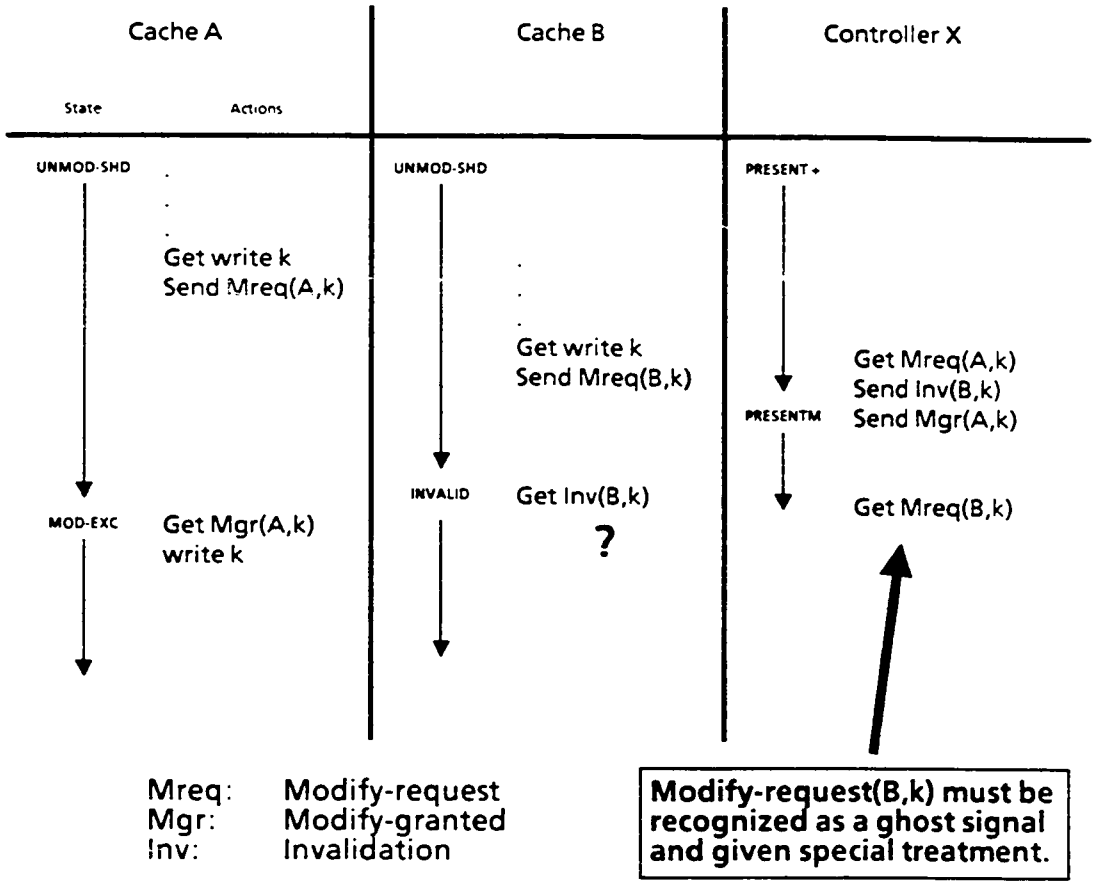


Figure 5.3: Ghost signal example

In addition to the ghost **Modify-request** mentioned above, it is possible for the protocol to generate ghost **Replace-unmodified** signals. Consider an example where block k is present in cache A in UNMOD-SHD state. At about the same time, cache A replaces the block and sends a **Replace-unmodified**(A,k) signal, and cache B sends either a **Write-miss**(B,k) or a **Modify-request**(B,k). If the request from B arrives before the signal from A, cache A will be sent an **Invalidation**(A,k) signal and its cache bit will be cleared in the global state. Then, when the **Replace-unmodified**(A,k) is serviced, the state of the cache bit indicates that the signal is to be ignored. Of course, the only effect of the **Replace-unmodified** is to clear the cache bit anyway, so the protocol is correct even if this is not recognized as a ghost signal.

It is also possible to have controller signals that arrive at caches that no longer have valid copies, although the cache bits remain set. This is possible if the cache has sent a **Replace-modified** or **Replace-unmodified** signal that has not yet arrived at the controller. The cache controller will receive the controller directive and attempt to match against the contents of the cache, but the match will be unsuccessful. It is important to note that unsuccessful matches are the result of a temporary inconsistency between the global and local states. They do not indicate that a system error has occurred, although they will often be associated with ghost signals.

There are only two types of ghost signals possible in the full map protocol: **Modify-request** and **Replace-unmodified**. These are the only actions that a cache can take with an UNMOD-SHD block. Ghost signals do not arise in the case of MOD-EXC blocks because the protocol enforces synchronization between the memory and cache controllers. The memory controller must wait for a response from the cache before it is allowed to continue, since a valid copy of the block must be obtained from the cache.

A final issue to consider is the difference between the **Replace-modified** and **Update** signals. Both include a write-back to update main memory, but the first indicates that the block is no longer in the cache, while a valid copy is retained if the second is sent. It is therefore necessary to treat the two differently. When the memory controller is waiting for a response from a **Write-back-shd** (sent after a read miss on a block MOD-EXC in another cache), it may receive either an **Update** or a **Replace-modified**. The former is normally received, but if the block is replaced before the **Write-back-shd** signal arrives at the cache, the latter will be seen instead. In their description of the protocol, Censier and Feautrier do not make provisions for the arrival of a **Replace-modified** when an **Update** is expected. This incorrectly leaves the cache bit set. Although this would introduce an unnecessary **Invalidation** signal when the block is next modified (which will be ignored by the cache that receives it), it does not lead to inconsistencies in cached data.

The main advantage of the full map protocol is that the state information provides complete information about which caches have copies of the block. Therefore, signals need only be sent to those caches with copies, minimizing the amount of traffic in the network, as well as the interruptions that may be necessary at each cache to service the signals. The time to access the global map can be overlapped with the regular block access on many transactions, so it need not substantially increase the time required to service each request.

The main disadvantage of this approach is the vast amount of memory required to implement the global table. As noted in Chapter 2, a global table using $n+1$ bits per tag can increase the size of main memory 13% or more. Particularly noteworthy is the low utilization of the bits–the information contained in the vast majority could, at any instant, be represented using much less space. For example, most of the memory blocks are not cached at any given instant, and the ABSENT state can be encoded in far fewer than $n+1$ bits for large n.

Censier and Feautrier state that the size of the global table can be reduced by using a single table entry of $n+1$ bits per *page*. Here a page is taken to mean a group of contiguous blocks in the same module, and this may be very different from the page in a virtual memory system. This technique would allow any number of caches to have UNMOD-SHD copies, but only a single cache could have *any* copies if any one block were MOD-EXC. The authors state that this technique would require block counts to be maintained at each cache to indicate the number of blocks present in the cache from each page. This would allow the single cache bit to be cleared in main memory only when the last copy of any block from that page is replaced in the cache. In addition to the complications to the design of the cache, this approach would also degrade performance. In particular, allowing only one writer per page could cause severe thrashing if the page included frequently accessed sections of writable shared data.

Another disadvantage, although less significant than the first, is that the size of the bit tags in the global table would limit the number of processor-cache pairs in the system, since each such pair would require an additional bit. While any system expansion would require changes in the interconnection network (for the non-bus systems considered here) and would therefore be costly, it should be noted that expanding a system using the full map protocol would also require substantial changes to the memory system (i.e., the global table would need to be replaced). In systems with networks that are relatively easy to extend, this consideration becomes much more important.

5.2.2 Extended Full Map Approach

This protocol was proposed by Yen and Fu [YK82, YYK85] and is very similar to the scheme of Censier and Feautrier. The essential difference between the two is the addition of a local state that allows the cache to complete writes on exclusive unmodified blocks without first obtaining write permission from the memory controller. As will be seen, the addition of this state (and associated signals) creates additional timing problems in the protocol. The following local states are used:

1. INVALID. (INV)
2. UNMODIFIED-SHARED. (UNMOD-SHD)
3. UNMODIFIED-EXCLUSIVE. (UNMOD-EXC)
4. MODIFIED-EXCLUSIVE. (MOD-EXC)

The global state is encoded using n+1 bits exactly as in the full map protocol. As before, the global state is one of the following:

1. ABSENT.
2. PRESENT+.
3. PRESENTM.

The requests from the cache controller to the memory controller are the same as in the full map protocol with one addition. The complete set includes:

- **Read-miss(cache#,block#)**.
- **Write-miss(cache#,block#)**.
- **Modify-request(cache#,block#)**.
- **Replace-unmodified(cache#,block#)**.
- **Replace-modified(cache#,block#,data)**.
- **Update(cache#,block#,data)**.
- **Exclusive-modified(cache#,block#)**. This signal informs the memory controller that an UNMOD-EXC copy has been modified and is now MOD-EXC.

The set of signals from the memory controller to the cache consists of:

- **Read-data-shd(cache#,block#,data)**. This response to a read miss informs the cache that the block is to be loaded in state UNMOD-SHD.
- **Read-data-exc(cache#,block#,data)**. This is also a reponse to a read miss, but now the block is to be loaded in state UNMOD-EXC.
- **Write-data(cache#,block#,data)**.
- **Modify-granted(cache#,block#)**.
- **Write-back-shd(cache#,block#)**.
- **Write-back-inv(cache#,block#)**.
- **Invalidation(cache#,block#)**.
- **Set-shared(cache#,block#)**. This instructs the cache to change the state from UNMOD-EXC to UNMOD-SHD.

Except as otherwise noted, the signals are identical to those described in the previous section.

Figures 5.4 and 5.5 contain transition diagrams for the local cache and global states. The protocol works as follows:

- **Read hit**. The data is returned to the processor. No other action is required.
- **Read miss**. A **Read-miss** is sent to the appropriate memory controller which tests the global state of the block. If the state is ABSENT, the block is sent using the **Read-data-exc** signal and the appropriate cache bit is set. If the state is PRESENT+ with exactly one cache bit set, that cache must be sent a **Set-shared** signal. After this is sent, or if two or more cache bits are set, the block is sent using **Read-data-shd** and the appropriate cache bit is set. If the state is PRESENTM, a **Write-back-shd** is sent requesting the cache with the modified copy to send an **Update** and change its local state to UNMOD-SHD. After the **Update** arrives, the memory controller clears the modified bit, sends the data using **Read-data-shd**, and sets the new cache bit. If the cache receives the read miss data with the **Read-data-exc** signal, the block is loaded in state UNMOD-EXC. Otherwise, the block is loaded in state UNMOD-SHD.
- **Write hit**. If the state is MOD-EXC, the write may proceed locally. If the state is UNMOD-EXC, the write may proceed immediately, accompanied by the sending of a **Exclusive-modified** signal to inform the memory controller to set the modified bit in the global state

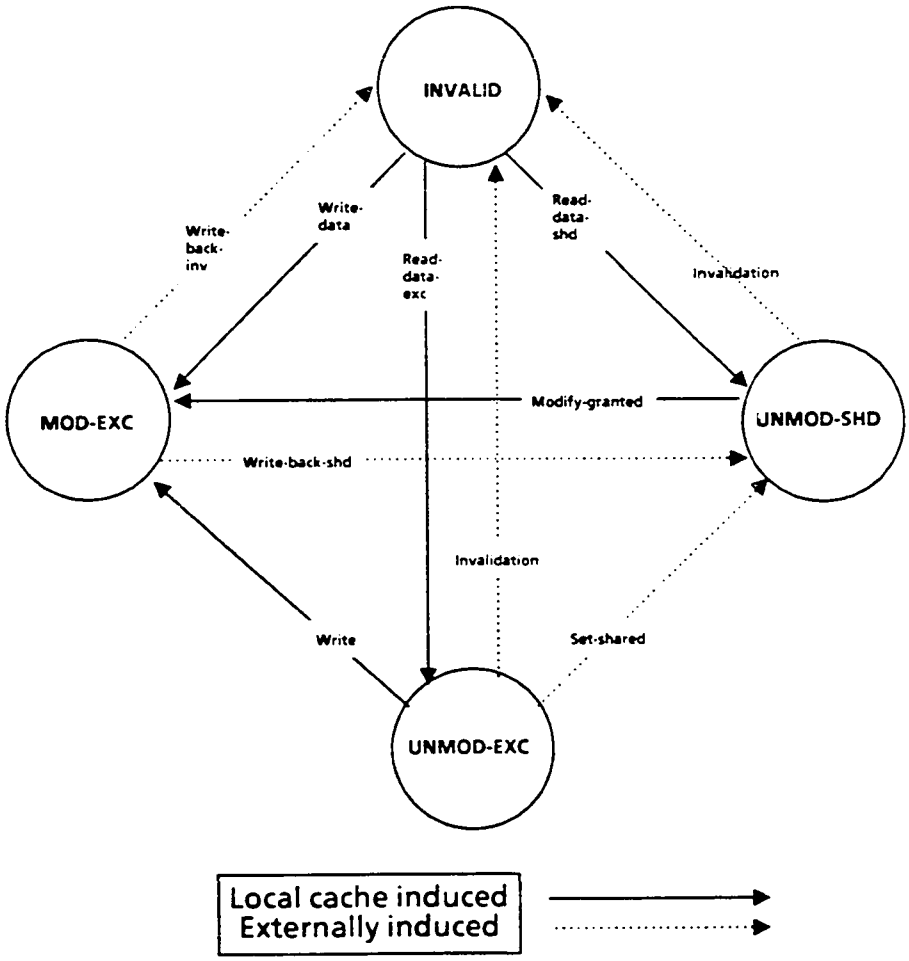


Figure 5.4: Extended full map protocol cache state transitions

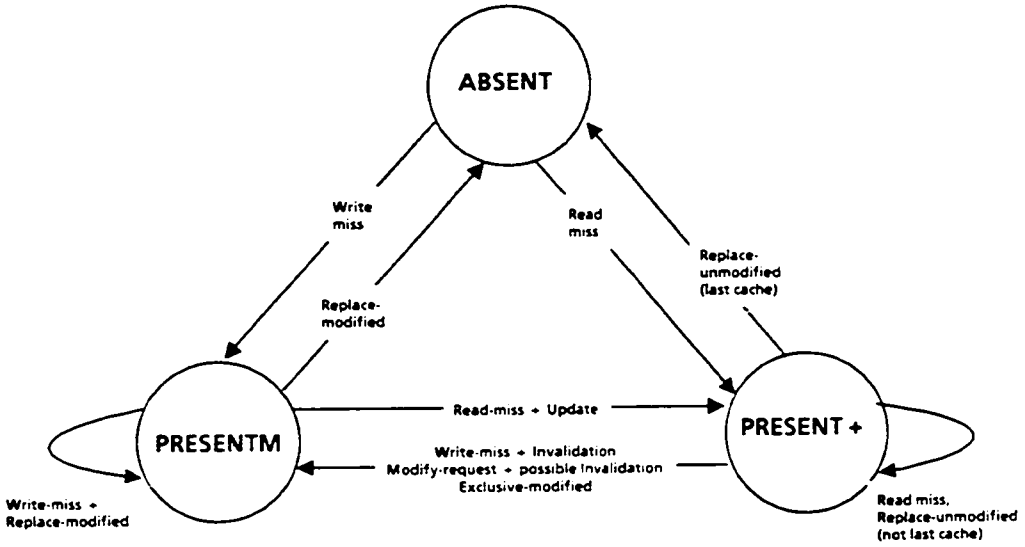


Figure 5.5: Extended full map protocol global state transitions

(changing from PRESENT+ to PRESENTM). If the state is UNMOD-SHD, a **Modify-request** must be sent. When the request arrives at the memory controller, the global state is read and all other caches with copies, if any, are sent **Invalidation** signals. Following this action, a **Modify-granted** is sent to the requesting cache. When this signal arrives, the cache performs the write and changes the state to MOD-EXC.

- **Write miss.** The cache sends a **Write-miss**. If the global state of the block is ABSENT, the memory controller processes the signal by immediately sending the data using the **Write-data** signal and setting the modified and cache bits. If the state is PRESENT+, all caches with copies must first be sent an **Invalidation** command and their cache bits must be cleared. If the state is PRESENTM a **Write-back-inv** signal must be sent. The cache with the MOD-EXC copy will respond with a **Replace-modified** and change the local state to INV. When the **Replace-modified** arrives, the controller takes the data and sends it to the requesting cache with a **Write-data**. After the data is sent, the global state is always PRESENTM with the cache bit set for the requesting cache. The block is always loaded in state MOD-EXC.

- **Replacement.** If the block selected for replacement is in state UNMOD-SHD or UNMOD-EXC a **Replace-unmodified** must be sent to clear the appropriate cache bit. If the local

state is MOD-EXC a **Replace-unmodified** must be sent to update main memory and reset the global state to ABSENT.

The addition of the UNMOD-EXC state creates some serious timing problems in the modified full map protocol. When a cache has a block in state UNMOD-EXC, there are four things that can happen: The cache can modify the block and send an **Exclusive-modified**, or it can replace the block and send a **Replace-unmodified**, or the cache can receive an **Invalidation** (sent by the controller as it services a **Write-miss**), or the cache can receive a **Set-shared** (sent by the controller servicing a **Read-miss**). The difficulties arise when the cache modifies the block locally (sending **Exclusive-modified**) at the same time that a **Set-shared** or an **Invalidation** is on its way to the cache. For example, assume that cache A has an UNMOD-EXC copy of k, and cache B has a write miss on the same block, as illustrated in Figure 5.6. At about the same time that B's **Write-miss**(B,k) is serviced by the memory controller, assume that A modifies its copy and sends an **Exclusive-modified**(A,k). If the above protocol description is followed as stated, the controller simply sends an **Invalidation**(A,k) and immediately sends a **Write-data**(B,k,*data*) to load the block into cache B. However, the copy it sends is already obsolete since a newer, modified version exists in the system.

In their paper outlining this protocol, Yen and Fu note that it is necessary to check that the exclusive copy has not yet been modified, but they do not tell exactly how this is to be accomplished. Nor do they describe the actions that must be taken if the copy actually is modified. In their paper they attempt to describe the actions of the cache and memory controllers in a single flow chart, thus ignoring problems that can arise from the asynchronous operation of the two units. In addition to the new concerns that result from the addition of the UNMOD-EXC copy, the scheme also has the same problems of the full map protocol. (Since the solutions are essentially the same, they are not repeated here.)

To correct this problem, it is necessary to wait for a response from the cache with the UNMOD-EXC copy before the requesting cache is allowed to proceed. A possible implementation–the approach taken in the simulation of this scheme–is to require an **Ack** (positive acknowledgement) from the cache with the exclusive copy when it is sent a **Set-shared** or an **Invalidation**. The block is not sent on to the cache requesting the block until the **Ack** is received. When the **Ack** arrives at the controller, it may proceed normally as in the protocol description above. However, while the controller is waiting for the **Ack**, it may instead receive an **Exclusive-modified** or a **Replace-unmodified**. The solutions for this situation are described below.

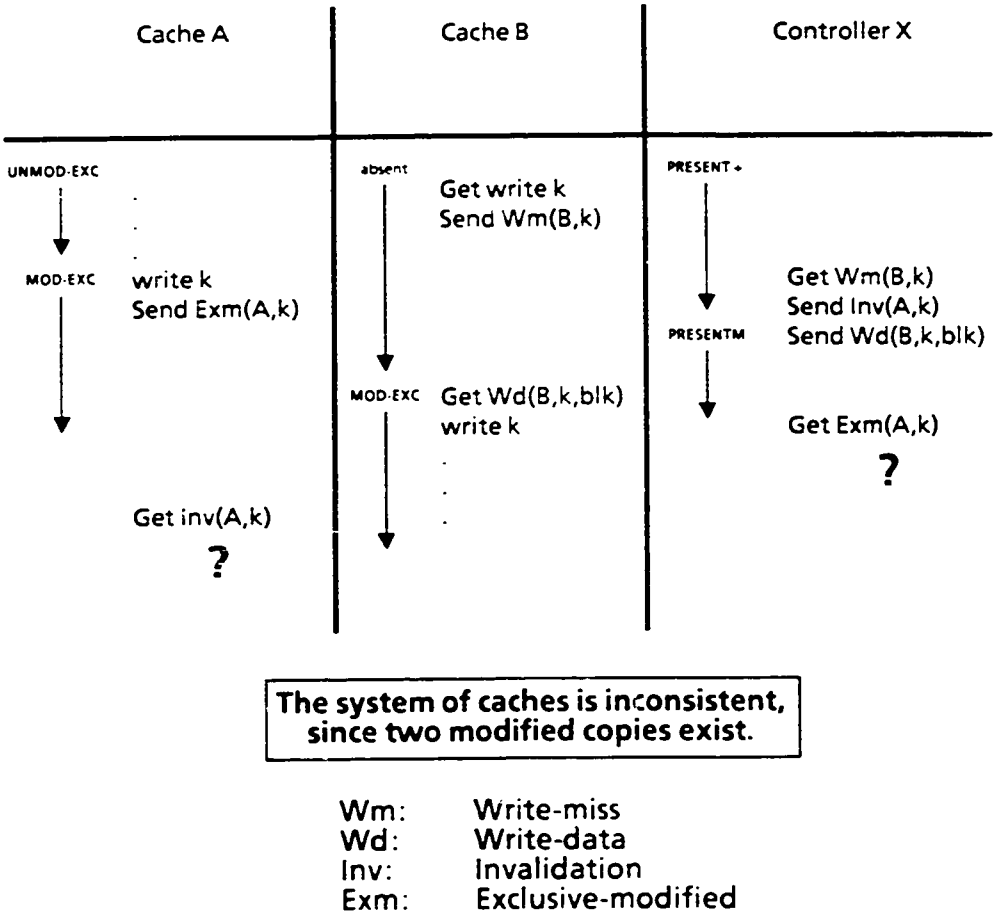


Figure 5.6: Race condition with UNMOD-EXC copy

Assume the controller is waiting after sending an **Invalidation** (and is therefore servicing a write miss). If a **Replace-unmodified** arrives, it is treated exactly as if it were an **Ack**. If an **Exclusive-modified** arrives, the global state is changed to PRESENTM (by setting the modified bit) and the controller simply proceeds as if it a **Write-back-inv** command had been sent–it does not need to send any other signal at this time. The cache with the exclusive copy will receive the **Invalidation** with a MOD-EXC copy and treat it as a **Write-back-inv** by doing a write-back and setting the state to INV. (It is possible for the cache to voluntarily replace the modified block in the meantime, but in both cases a **Replace-modified** is sent.)

Assume the controller sent a **Set-shared** (and is servicing a read miss). If a **Replace-unmodifed** arrives, the controller clears the cache bit and supplies the block to the requesting cache with a **Read-data-exc**, indicating that no other cached copies exist. If an **Exclusive-modified** signal arrives, the global state is set to PRESENTM and the controller proceeds as if a **Write-back-shd** signal had been sent. The cache with the exclusive copy will receive the **Set-shared** with a MOD-EXC copy and treat it as a **Write-back-shd** by sending an **Update** and setting the local state to UNMOD-SHD. When the **Update** arrives, the controller can clear the modified bit and send the block to the requesting cache with the **Read-data-shd** signal. (If the cache has already replaced the MOD-EXC copy, the controller will receive a **Replace-modified** instead of an **Update**, requiring the cache bit to be cleared and allowing the block to be sent with a **Read-data-exc** signal, since it will not be shared.)

The advantage of the modification to the full map protocol is that additional state information at the local cache allows modifications to all exclusive blocks without delay. This decreases the delay in writing private (or non-shared) blocks loaded into the cache on a read miss, since the write does not need to wait until the controller grants write permission. However, the memory controller still needs to be informed when such writes take place.

The disadvantage is that the protocol is made more complex–acknowledgements from caches with exclusive copies are necessary before proceeding. This increases the overhead of blocks that in an exclusive state in one cache and referenced in another; private block efficiency has been increased at the expense of additional overhead in some shared block transactions. Since the protocols have the same global table requirements, the large size and lack of expandability of both are also a concern.

5.3 New General Network Protocols

This section contains proposals for a family of protocols that are similar to the protocols of the previous section except that the size of each entry in the global table is smaller and of fixed length. Because the global state is encoded in a fewer number of bits, it necessarily provides less information than in the full map schemes. To reflect this loss of information, these schemes may be referred to as *partial map* protocols. In particular, these schemes do not maintain information about the identity of caches with copies. Since the memory controller cannot determine which caches have copies, some signals must be broadcast to all caches, requiring some form of broadcast mechanism from the interconnection network.

5.3.1 Basic 'Twobit' Scheme

We described the first partial map protocol in a paper published in 1984 [AJ84]. Since the time it was published, it has been shown that the protocol has problems potentially more serious than those of the full map protocols described in the previous section. Given certain assumptions about the maximum delay in the network, this basic protocol will work correctly, but it is not correct for any general interconnection network. However, it is presented here as originally proposed since it does work for a reasonably large class of networks, and because the other protocols in the partial map family are based on it. In addition, the problems of the basic protocol provide the motivation for the development of additional partial map protocols.

The protocol uses the same local states as in the full map protocol of Censier and Feautrier, namely:

1. INVALID. (INV)
2. UNMODIFIED-SHARED. (UNMOD-SHD)
3. MODIFIED-EXCLUSIVE. (MOD-EXC)

The global state is encoded in a two bit tag associated with each block in main memory. These two bits allow the encoding of the following four states:

1. ABSENT. The block is not present in any cache.
2. PRESENT1. The block is in exactly one cache and is not modified.
3. PRESENT*. The block is present in zero or more caches and is not modified.
4. PRESENTM. The block is present in exactly one cache and is modified.

The set of signals from the cache controller to the memory controller are identical to those of the full map protocol. These signals are:

- **Read-miss(cache#,block#)**.
- **Write-miss(cache#,block#)**.
- **Modify-request(cache#,block#)**.
- **Replace-unmodified(cache#,block#)**.
- **Replace-modified(cache#,block#,data)**.
- **Update(cache#,block#,data)**.

The set of signals from the memory controller to the cache controllers is very similar to those of the full map protocol, except that some must be broadcast to all caches. Each of these broadcast signals includes a **cache#** parameter, but it does not indicate where the signal is to be sent. Instead, the specified cache is to ignore the signal. (The reason for this will be made clear in the description of the protocol.) The signals are:

- **Read-data(cache#,block#,data)**.
- **Write-data(cache#,block#,data)**.
- **Modify-granted(cache#,block#)**.
- **Write-back-snd(cache#,block#)**. This is exactly the same as in the other protocols, except it is broadcast to all caches. Caches without a copy of the block will ignore the signal after failing to match on the block address. The specified cache will ignore the command.
- **Write-back-inv(cache#,block#)**. This is also broadcast to all caches. The specified cache will ignore the signal.
- **Invalidation(cache#,block#)**. This is broadcast to all caches, instructing all but the specified cache to set the local state to INV.

A state transition diagram for the global states is given in Figure 5.7; the cache state transition diagram is identical to the diagram in Figure 5.1. The actions of the protocol are described as follows:

- **Read hit**. The data is returned to the processor with no coherence protocol overhead.

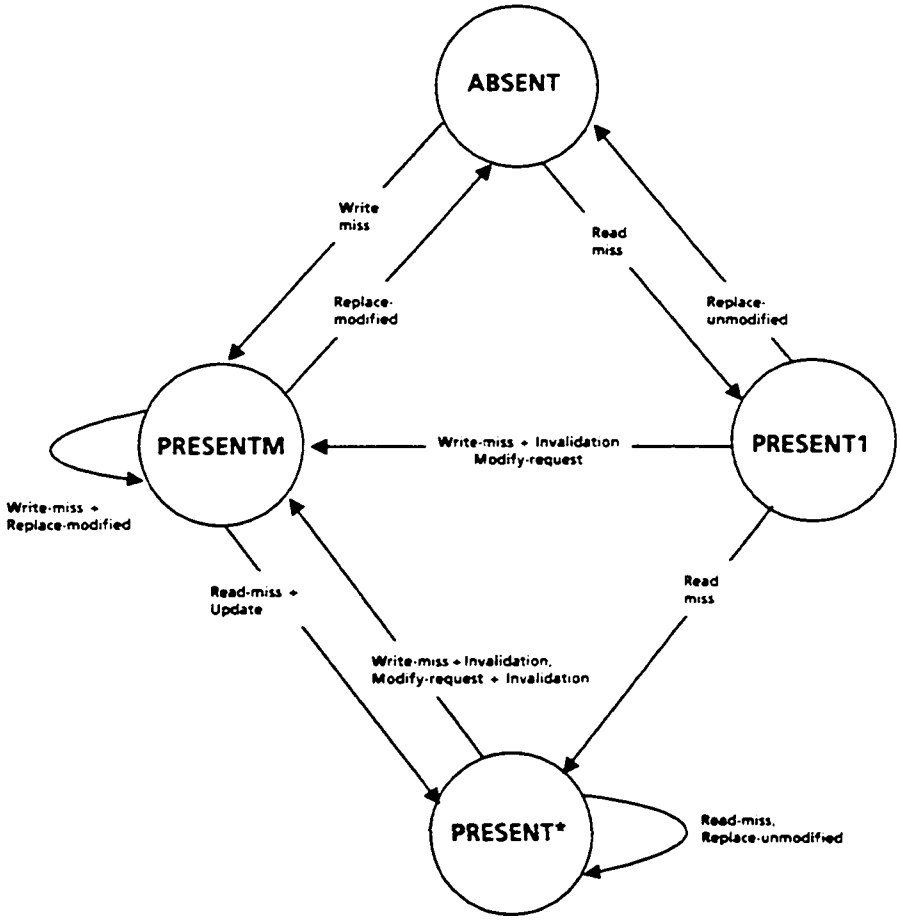


Figure 5.7: Twobit protocol global state transitions

- **Read miss.** The cache sends a **Read-miss** to the appropriate memory controller. When the controller receives the signal, it must check the global state of the block. If the global state is ABSENT, PRESENT1, or PRESENT*, the block may be sent immediately (using the **Read-data** signal). If the state is ABSENT, it is changed to PRESENT1. If the state is PRESENT1, it is changed to PRESENT*. If the state is already PRESENT*, it does not need to be updated. If the state is PRESENTM then a **Write-back-shd** signal must be broadcast to obtain a valid block copy from the cache with the modified block. The cache will receive the signal, set its local state to UNMOD-SHD, and send the valid contents of the block to the memory controller with an **Update**. When the controller receives the **Update**, it sends the block to the requesting cache and changes the global state to PRESENT*. If the cache replaces the block before the **Write-back-shd** arrives, the controller will receive a **Replace-modified** instead of an **Update**. In this case the global state is set to PRESENT1, since the block is no longer shared. In all cases the block is loaded into the cache in state UNMOD-SHD.

- **Write hit.** If the local state is MOD-EXC the write may proceed without any additional overhead. If the state is UNMOD-SHD, the write may take place only after write permission is obtained from the memory controller. The cache sends a **Modify-request** to the appropriate memory controller, which checks the global state of the block. The only possible global states are PRESENT1 and PRESENT*; if the global state is PRESENT1, the cache submitting the request has the only valid copy so write permission is immediately granted (in the form of a **Modify-grant**) and the state is changed to PRESENTM. If the global state is PRESENT*, other caches *may* have copies (unless since replaced) and the controller must broadcast an **Invalidation** signal, with the number of the requesting cache indicated so that it does not invalidate its copy. All caches but the specified cache will check their directory and invalidate their local copy, if one exists. The **Invalidation** signal must include a cache number so that the requesting cache does not invalidate its copy. After the **Invalidation** signal is sent and the global state is set to PRESENTM, write permission is sent to the requesting cache in the form of a **Modify-granted**. The cache receives the signal, performs the write, and changes the local state to MOD-EXC.

- **Write miss.** The cache sends a **Write-miss** to the controller, which then checks the global state of the block. If the state is ABSENT, the block may be transmitted directly (via a **Write-data**), accompanied by a state change to PRESENTM. If the global state is PRESENT1 or PRESENT*, an **Invalidation** signal is broadcast, causing all cached copies

to be invalidated. (The **cache#** parameter in the signal may be set to the number of the cache with the write miss, or it may be set to a value not matching any cache in the system.) Following the invalidation signal the **Write-data** may be sent to the requesting cache. If the state is PRESENTM, the controller must broadcast a **Write-back-inv** to get a valid and exclusive copy. The cache with the modified copy will receive the signal, set the local state to INV, and send the block with a **Replace-modified**. (Should the cache voluntarily replace the modified block before the **Write-back-inv**, the outcome is identical.) When the write-back arrives at the controller, the valid contents of the block are sent to the requesting cache. The global state remains PRESENTM. In all cases the block is loaded in state MOD-EXC.

- **Replacement**. UNMOD-SHD blocks replaced in the cache require a **Replace-unmodified** to be sent; the replacement of MOD-EXC blocks requires a **Replace-modified**. When the controller receives a **Replace-unmodified**, it checks the global state. If the state is PRESENT1, it is reset to ABSENT. If the state is PRESENT*, it is left unchanged, since it is not known how many caches (if any) still have a copy of the block. When the controller receives a **Replace-modified**, the global state must be PRESENTM and it is changed to ABSENT.

The problems in this protocol result from ghost signals, or signals from caches that no longer are considered to have a copy of the block when the signal arrives at the controller. Ghost signals in the full map scheme could be easily identified by checking the appropriate cache bit in the global state, but there is no cache bit provided in the partial map. Unless certain guarantees can be made about the maximum delay of signals from the cache to the controller, the arrival of ghost signals can cause the protocol to malfunction.

For example, in a network with an arbitrary transmission delay, the following problem can arise in the servicing of a **Replace-unmodified** signal (illustrated in Figure 5.8). Assume that cache A has an UNMOD-SHD copy of block k and the global state of k is PRESENT1. The controller services a **Write-miss**(B,k) from cache B, broadcasts a **Invalidation**(B,k) instructing all caches but B to invalidate their copies, and sends the block to B with a **Write-data**(B,k,*data*). The **Invalidation**(B,k) signal is slow in reaching A, and just before it arrives, A replaces the block and sends a **Replace-unmodified**(A,k). Cache B loads the block and quickly replaces it, sending a **Replace-modified**(B,k,*data*) which arrives at the controller before A's **Replace-unmodified**(A,k), which sets the global state to ABSENT. At about the same time, Cache C has a read miss and sends a **Read-miss**(C,k) which also arrives at the controller before A's **Replace-**

unmodified(A,*k*). The controller services the **Read-miss**(C,*k*) by sending **Read-data**(C,*k*,*data*) and changing the global state to PRESENT1. Then the ghost **Replace-unmodified**(A,*k*) arrives and the global state is incorrectly set to ABSENT, although C has a valid copy.

The following additional example, shown in Figure 5.9, illustrates a problem that arises with **Modify-request** signals (assuming an arbitrary transmission delay). Assume that the global state of block *k* is PRESENT* and that caches A and B have UNMOD-SHD copies. Cache A wants to write the block and submits a **Modify-request**(A,*k*) to the controller, which broadcasts an **Invalidation**(A,*k*) (instructing all caches but A to invalidate) and then sends a **Modify-granted**(A,*k*) giving permission to write. In addition the global state is changed to PRESENTM. Just before the **Invalidation**(A,*k*) signal (which is delayed arbitrarily) arrives at cache B, B sends a **Modify-request**(B,*k*). Shortly after granting write permission to A, the controller receives a **Read-miss**(C,*k*). In order to service this request, the controller broadcasts a **Write-back-shd**(C,*k*) to force A to send an **Update**(A,*k*,*data*), which arrives at the controller before the **Modify-request**(B,*k*). After the arrival of the write-back from A, the controller sends a **Read-data**(C,*k*,*data*) to load the block into cache C and changes the global state to PRESENT*. At this point, the ghost **Modify-request**(B,*k*) arrives, which is serviced by broadcasting an **Invalidation**(B,*k*) signal, invalidating all copies but B's, and by sending a **Modify-granted**(B,*k*), but B no longer has a copy of the block.

Modify-request and **Replace-unmodified** are the only possible ghost signals in this protocol. Both may arrive from caches that have since been sent **Invalidation** signals, either in servicing a **Modify-request** or a **Write-miss**. Since in either case the global state of the block will be set to PRESENTM, ghost signals can easily be identified *if they arrive while the global state is PRESENTM*. In other words, if the network parameters guarantee that all ghost signals will arrive before the *earliest possible time* that the cache with the modified block can either replace it voluntarily (and send a **Replace-modified**) or perform a write-back at the controller's request (and send an **Update**), then all ghost signals can be recognized and the scheme can be made to work. For the protocol to function correctly, all **Modify-request** and **Replace-unmodified** signals must receive special treatment when the state is PRESENTM. In this case, the **Replace-unmodified** signals can simply be ignored. There are two alternatives in dealing with the **Modify-request** signals; either they can be ignored by the memory controller with the cache (that is waiting for a **Modify-granted**) responsible to submit a **Write-miss** when the **Invalidation** arrives, or they can be treated as **Write-miss** signals by the controller without requiring the cache to resubmit any requests. (In the simulation of this scheme, *all* **Modify-request** signals are serviced by sending

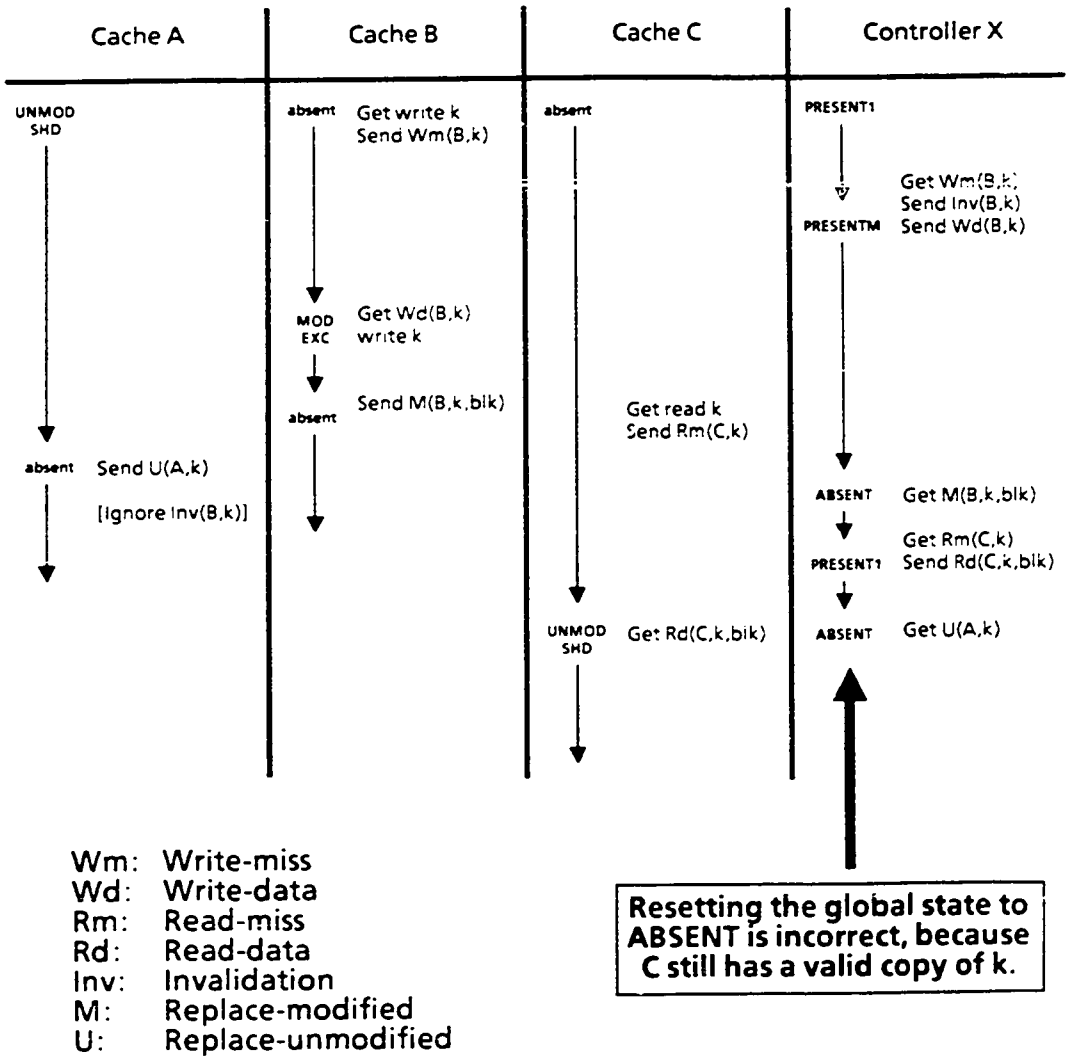


Figure 5.8: Example of ghost Replace-unmodified signal

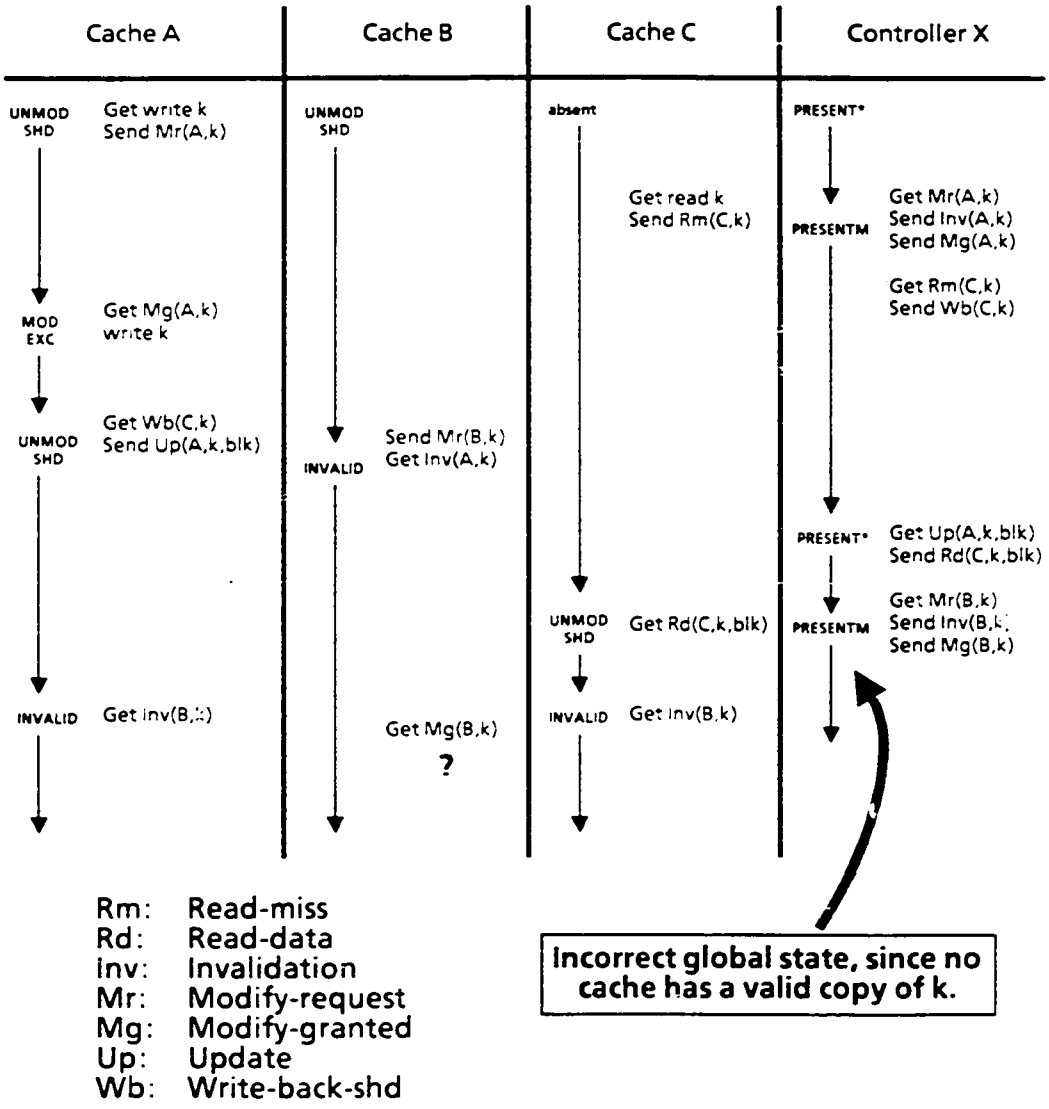


Figure 5.9: Example of ghost Modify-request signal

data when write permission is granted.)

There is, however, one major complication: a ghost signal may arrive at the controller after a legitimate request. Consider the following example (illustrated in Figure 5.10). Assume that block k is in global state PRESENT* and is present UNMOD-SHD in caches A and B. At about the same time, both A and B want to modify the block and submit **Modify-request** signals, and a third cache, C, sends a **Read-miss**. Assume that the requests arrive in the following order: **Modify-request**(A,k), **Read-miss**(C,k), and **Modify-request**(B,k). In servicing A's request a **Modify-granted**(A,k) is sent, following an **Invalidation**(A,k), which instructs all caches but A to invalidate k. Then C's request will be serviced by, first, checking the global state (which is PRESENTM), and second, broadcasting a **Write-back-shd**(C,k). Cache A will receive the signal and send an **Update**(A,k,(data)). According to the assumption above, the **Modify-request**(B,k) will arrive at the controller before the **Update**(A,k,*data*) arrives, but it must be recognized as a ghost signal *upon its arrival.* If it is simply inserted in a service queue, the global state will no longer be PRESENTM when it is serviced, and it cannot be recognized as a ghost signal. Therefore, for the protocol to function correctly, the controller must recognize and discard all incoming **Modify-request** and **Replace-unmodified** signals while waiting for an **Update** on the same block.

There are a number of modifications that can be made to the basic twobit protocol to avoid the problems described above. In the following sections, three protocols are described that are based on the twobit protocol, but without the same timing problems. Of particular interest is the fact that each of the following protocols has been proven correct. One proof is included in Chapter 8.

5.3.2 The 'Threestate' Scheme

This protocol was the first general interconnection network protocol proposed for which a proof of correctness was published [JC85, CGS86]. The essential idea of this approach is to eliminate the troublesome **Replace-unmodified** and **Modify-request** signals entirely. The protocol uses the same local states as the basic protocol, namely:

1. INVALID. (INV)
2. UNMODIFIED-SHARED. (UNMOD-SHD)
3. MODIFIED-EXCLUSIVE. (MOD-EXC)

The global state is one of:

1. ABSENT. The block is not present in any cache.

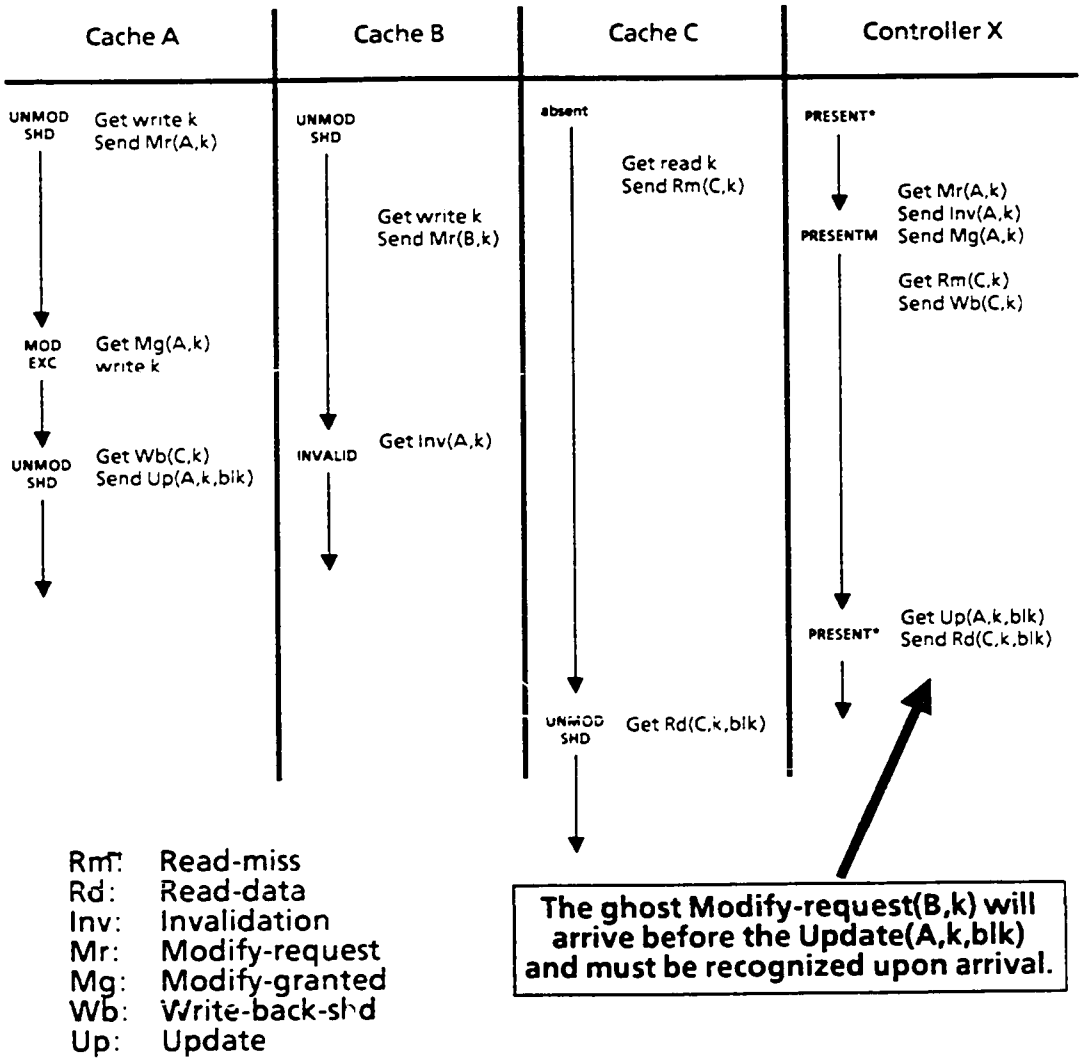


Figure 5.10: Recognizing ghost signals upon arrival

2. PRESENT*. The block is present in zero or more caches and is not modified.

3. PRESENTM. The block is present in exactly one cache and is modified.

As can be seen, the global state PRESENT1 is not used. Without the **Modify-request** and **Replace-unmodified** signals, there is no use for this state. (It is equally valid to see the problems of the twobit protocol as a consequence of having state PRESENT1, the elimination of which makes these two signals unnecessary. In any event, both signals and the state are eliminated in this protocol.)

The set of signals from the cache controller to the memory controller consists of:

- **Read-miss(cache#,block#)**.
- **Write-miss(cache#,block#)**.
- **Replace-modified(cache#,block#,data)**.
- **Update(cache#,block#,data)**.

The set of signals from the memory controller to the cache controllers is the following:

- **Read-data(cache#,block#,data)**.
- **Write-data(cache#,block#,data)**.
- **Write-back-shd(cache#,block#)**.
- **Write-back-inv(cache#,block#)**.
- **Invalidation(cache#,block#)**.

As in the twobit protocol, the last three signals are broadcast to all caches, and ignored by the cache whose number is included as a parameter.

A description of the function of the protocol follows below. The local and global state transitions are summarized in Figures 5.11 and 5.12.

- **Read hit**. The data is returned to the processor with no coherence protocol overhead.
- **Read miss**. The cache sends a **Read-miss** to the appropriate memory controller, which checks the global state of the block. If the global state is PRESENT*, the block may be sent immediately and no state change is required. If the state is ABSENT, the block can be sent immediately with a state change to PRESENT*. If the state is PRESENTM, a **Write-back-shd** signal must be broadcast. When it is received by the cache with the modified

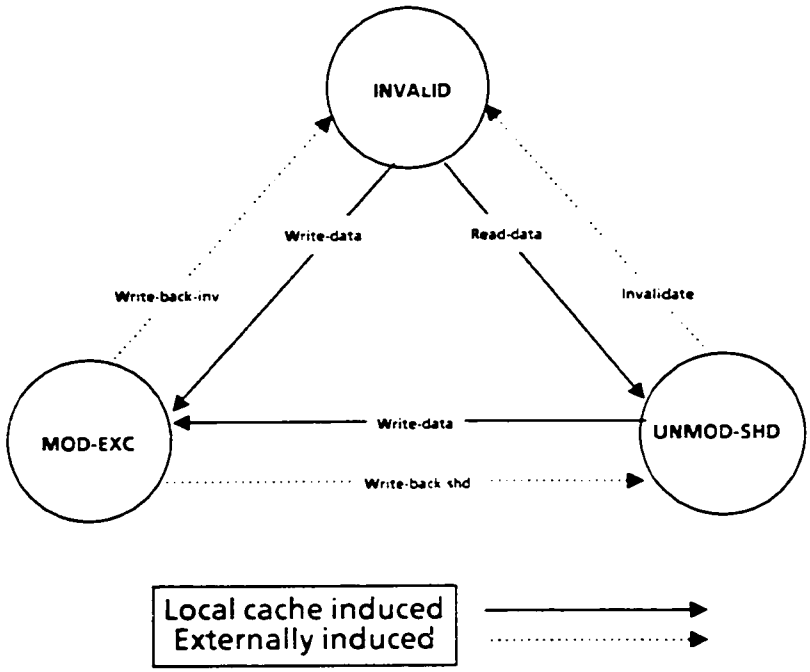


Figure 5.11: Threestate protocol cache state transitions

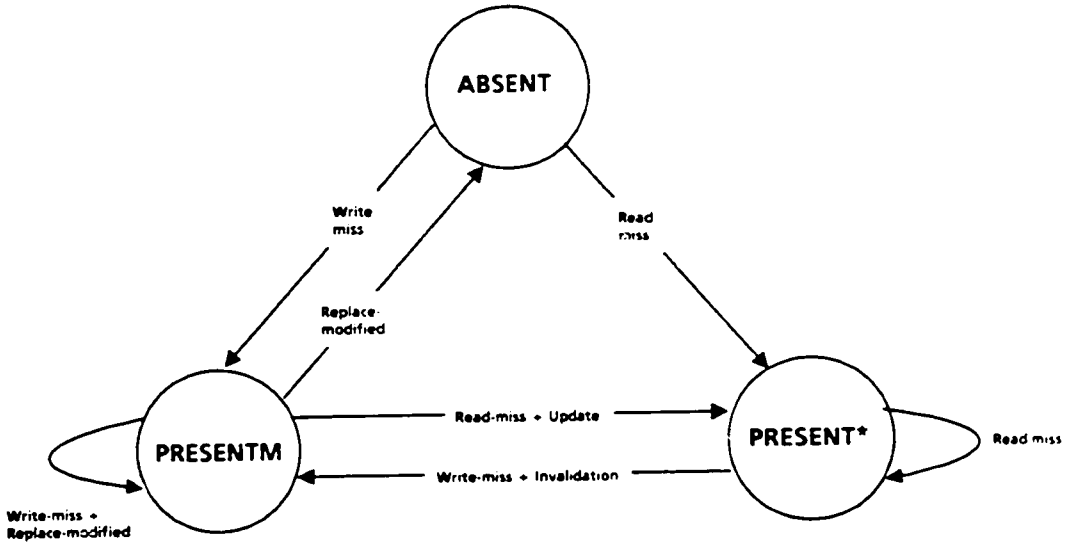


Figure 5.12: Threestate protocol global state transitions

copy, that cache will change the local state to UNMOD-SHD and an **Update** back to the memory controller. (Should the cache voluntarily replace the block before the signal arrives, the controller will receive a **Replace-modified** which is treated identically.) When the controller receives the **Update** (or **Replace-modified**) it sends the block to the requesting cache and changes the global state to PRESENT*. In all cases the block is loaded into the cache in state UNMOD-SHD.

- **Write hit**. If the local state is MOD-EXC the write may proceed without any additional overhead. If the state is UNMOD-SHD, the cache submits a **Write-miss** for the block, which will result in a **Write-data** from the controller. If the local copy is still UNMOD-SHD when the **Write-data** arrives, the transmitted contents of the block may be ignored. If the local state is INV, then the cache received an **Invalidation** signal in the interim, and the block contents from the **Write-data** must be loaded into the cache.

- **Write miss**. The cache sends a **Write-miss** to the controller, which checks the global state of the block. If the state is ABSENT, the block is transmitted directly (with a **Write-data**), and the state is changed to PRESENTM. If the state is PRESENT* an **Invalidation** signal is broadcast to all caches but the requesting cache, and then the block is sent and the state set to PRESENTM. If the state is PRESENTM, the controller broadcasts a **Write-back-inv**, and the cache with the modified copy receives the signal, sets the local state to INV, and sends the block with a **Replace-modified**. (If the block should be replaced voluntarily before the signal arrives at the cache, the same signal is sent.) When the write-back arrives at the controller, the valid contents of the block are sent to the requesting cache. The global state is left in PRESENTM. In all cases the block is loaded in state MOD-EXC.

- **Replacement**. Only if a MOD-EXC block is replaced is action required. The cache sends a **Replace-modified** to the appropriate cache controller, which changes the global state from PRESENTM to ABSENT.

The fact that ghost signals have been eliminated in this protocol can easily be verified by noting the four types of signals from the caches to the memory controller. Ghost signals are, by definition, signals sent from caches in which the local state of the block is changed between the time the request is sent and the time it arrives at the memory conroller. In this protocol, the only instance in which the cache sending the signal has a copy of the block after the signal is sent is when the cache has a write-hit on an unmodified block, but since this is treated as a write miss, the local state of the block is irrelevant.

The elimination of the **Replace-unmodified** signal increases the number of invalidation broadcasts necessary in servicing write misses, since many blocks will now be PRESENT* that would otherwise be ABSENT. In particular, read or write misses on blocks that are not present in any other cache will find blocks in either state ABSENT or PRESENT*. If the block has never been in a cache, or if it was MOD-EXC when replaced, the state will be ABSENT. If it was UNMOD-SHD when replaced, the global state will be PRESENT*. In the case of a write miss, this could result in a dramatic increase in the number of required **Invalidation** broadcasts. The elimination of the **Modify-request** signal results in the added overhead of full **Write-data** transfers, even though in most cases, the cache will still have a valid copy, and the data portion can be ignored.

While the elimation of **Replace-unmodified** signals may increase the amount of network traffic, it also reduces the total service demands of the memory controller, since it never needs to service **Replace-unmodified** signals. The additional **Invalidation** broadcasts that are required may be overlapped with other controller activities, so it requires no more time to service write misses on blocks in state PRESENT* that in state ABSENT. In summary, this protocol exchanges simplicity and reduced demands on the memory controller for an increase in the interconnection network traffic.

5.3.3 The 'Threebit' Scheme

This protocol uses the same three local states as the twobit protocol, namely:

1. INVALID. (INV)
2. UNMODIFIED-SHARED (UNMOD-SHD)
3. MODIFIED-EXCLUSIVE. (MOD-EXC)

The scheme derives its name from the length of the tag used to record the global state. Since there are a total of six states, the minimal tag length is three bits. The states it uses are:

1. R-ABSENT. Not present in any cache.
2. R-PRESENT1. Present in exactly one cache and unmodified.
3. R-PRESENTM. Present in exactly one cache and modified.
4. S-ABSENT. Not present in any cache.
5. S-PRESENT*. Present in zero or more caches and not modified.

6. S-PRESENTM. Present in exactly one cache and modified.

The main idea is to use two different sets of states; the R-states for regular private block transactions, and the S-states for transactions on shared blocks. Blocks are initially in R-states and remain there until they become shared, at which point they use S-states, which are identical in function to the states in the threestate protocol. The overhead of processing blocks in the S-states is higher than those in the R-states, so it is desirable to reset blocks to the R-states whenever possible. A technique using special hardware support to reset from the S-states to R-states will be described below.

The set of signals from the cache to the controller are the same as in the twobit protocol, namely:

- **Read-miss(cache#,block#).**
- **Write-miss(cache#,block#).**
- **Modify-request(cache#,block#).**
- **Replace-unmodified(cache#,block#).**
- **Replace-modified(cache#,block#,data).**
- **Update(cache#,block#,data).**

The set of signals from the cache to the controller is also the same.

- **Read-data(cache#,block#,data).**
- **Write-data(cache#,block#,data).**
- **Modify-granted(cache#,block#).**
- **Write-back-shd(cache#,block#).**
- **Write-back-inv(cache#,block#).**
- **Invalidation(cache#,block#).**

The local and global state transitions of the protocol are summarized in Figure 5.13 and 5.14. In the following discussion, the operation of the protocol is described. A proof of correctness for the scheme is given in the Appendix.

- **Read hit.** The data is returned to the processor with no coherence overhead.

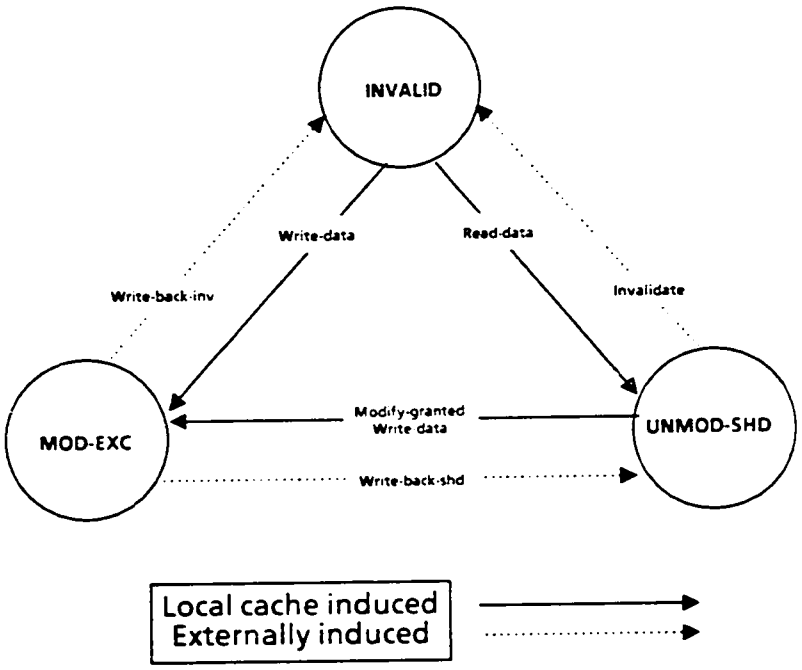


Figure 5.13: Threebit protocol cache state transitions

- **Read miss.** The cache sends a **Read-miss** signal to the controller which checks the global state. If the state indicates that the block is not modified, the block may be sent immediately with a **Read-data** accompanied by one of the following state changes: R-ABSENT to R-PRESENT1, R-PRESENT1 to S-PRESENT*, or S-ABSENT to S-PRESENT*. If the state is already S-PRESENT* it requires no change. If the block is R-PRESENTM or S-PRESENTM the cache with the modified copy must first perform a write back, so a **Write-back-shd** is sent. The cache with the MOD-EXC copy will set its local state to UNMOD-SHD and perform an **Update** when it receives the signal. The controller will receive the **Update**, change the local state to S-PRESENT*, and send the **Read-data**. If the controller should receive a **Replace-modified** while waiting for the update, the outcome is the same, unless the original global state was R-PRESENTM, in which case it is changed to R-PRESENT1 instead of S-PRESENT*. In all cases the cache loads the block in UNMOD-SHD.

- **Write hit.** If the local state is MOD-EXC, the write may proceed without delay. If the state is UNMOD-EXC, it can take place only after write permission is given by the controller. The cache submits a **Modify-request**, which the controller processes by first checking the

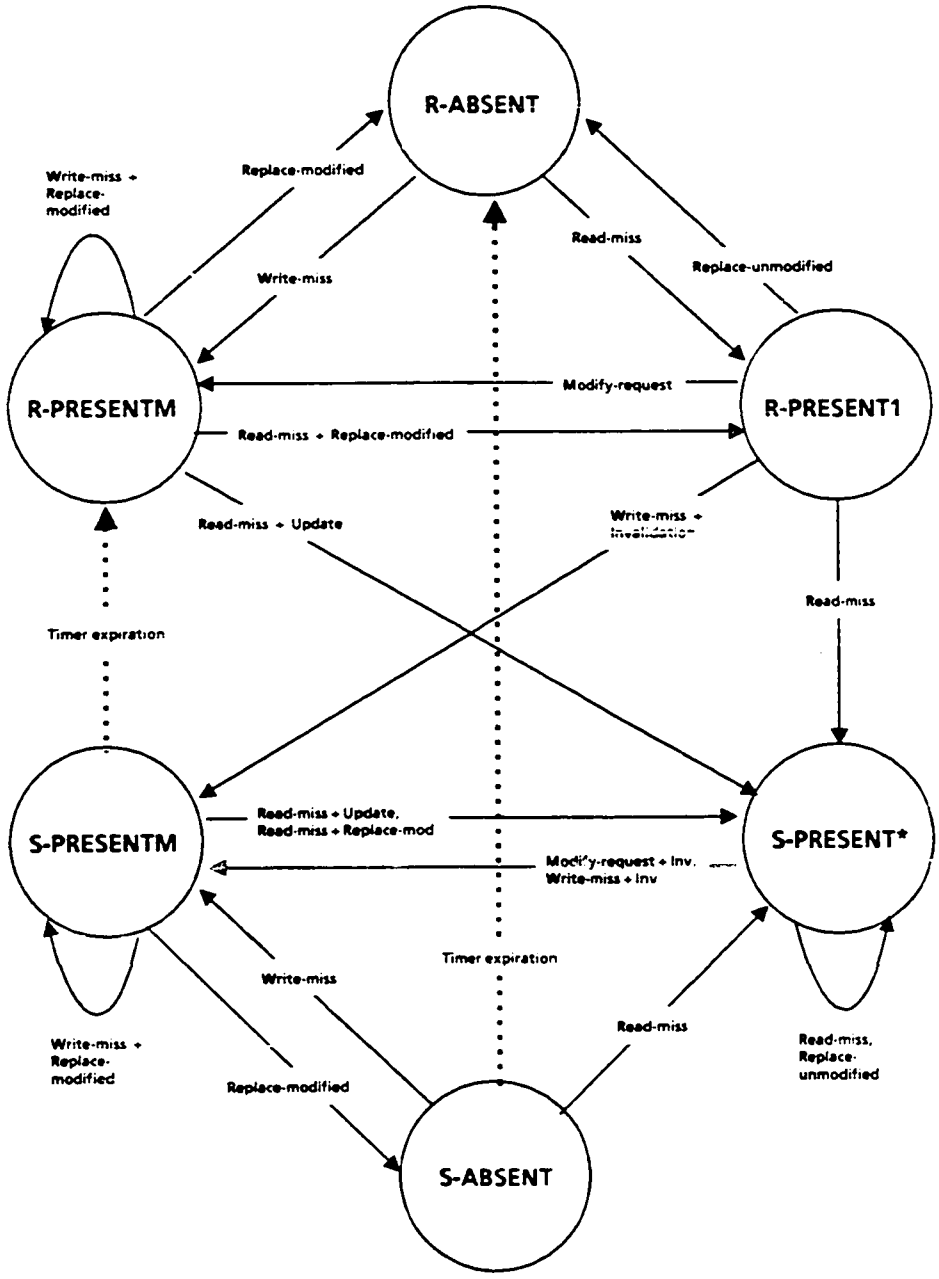


Figure 5.14: Threebit protocol global state transitions

global state. The only two possible states are R-PRESENT1 and S-PRESENT*; if the state is R-PRESENT1, the controller sends a **Modify-granted** and changes the state to R-PRESENTM. If the state is S-PRESENT*, the controller sends an **Invalidation** to all other caches and then sends a **Write-data** to the requesting cache, simultaneously granting write permission and providing a valid copy of the block. The controller also sets the global state to S-PRESENTM. Other caches waiting for a response from submitted **Modify-request** signals will receive the **Invalidation** signal and mark their copies as INV–they need do no more, since their **Modify-request** signal will be processed by the controller as a **Write-miss**. Caches waiting for a response to a **Modify-request** will receive either a **Modify-granted** or a **Write-data**. If the latter arrives and the local copy is not INV, the block data may be ignored. As soon as either signal arrives, the write is allowed to proceed and the local state is changed to MOD-EXC.

- **Write miss.** The cache sends a **Write-miss** signal to the appropriate controller, which checks the global state. If the block is not in any cache, the controller sends a **Write-data** signal immediately, and the state is changed to R-PRESENTM if it was R-ABSENT, and to S-PRESENTM if it was S-ABSENT. If the state is R-PRESENT1 or S-PRESENT*, an **Invalidation** signal must be broadcast, and the global state must be changed to S-PRESENTM as the block is sent to the requesting cache. If the state is R-PRESENTM or S-PRESENTM, the block must be written back before the controller can supply the data. A **Write-back-inv** signal is broadcast, which will result in a **Replace-modified** signal from the cache with the modified copy. When this signal arrives at the controller, the block is sent in a **Write-data** signal, and the global state remains unchanged. In all cases the block is loaded in state MOD-EXC.

- **Replacement.** When UNMOD-SHD blocks are replaced, the cache submits a **Replace-unmodified** signal. This only affects the global state if it is R-PRESENT1, in which case it is changed to R-ABSENT. It does not affect any other global state. When MOD-EXC blocks are replaced, the cache submits a **Replace-modified** signal, writing back the contents of the block to memory. If the state is R-PRESENTM it is reset to R-ABSENT. Otherwise, the global state must be S-PRESENTM and it is reset to S-ABSENT.

In general, the R-states are used as long as exactly zero or one caches have a copy of the block, and as long as there is no possibility of ghost signals. When the block is present in more than one cache at the same time (or if it was shared and the possibility of ghost signals exists), the S-states are used. There is little coherence overhead in the R-states, since blocks are not shared, but the

overhead increases in the S-states, where the actions correspond to those of the threestate scheme.

In the above protocol description, there is no provision for the state to change from an S-state to an R-state. Such a transition is only possible when it is certain that there are no ghost signals currently on their way to the controller. One possible solution is to add a special timing mechanism so that the controller can reset S-states to R-states, as the following example illustrates (indicated by the dotted lines in Figure 5.14). Let d be the maximum time round trip transmission time for a signal from the controller to a cache and back–given the configuration of this particular system. As each **Invalidation** signal is sent, the controller makes an entry in a small auxiliary table, recording the block number and the current time plus d. When the table entry equals the current time, if there have been no other transactions on the block in the meantime, the global state may be reset from S-PRESENTM (to which it is always set when an invalidation is sent) to R-PRESENTM. If the cache has voluntarily replaced the block (with a **Replace-modified**), but no other transactions have taken place for the block, the global state may also be reset from S-ABSENT to R-ABSENT. If any transactions (other than a voluntary write-back) take place before the time is reached, the entry is deleted from the auxiliary table.

The intent is to allow efficient private block transactions while ensuring that consistency is maintained even in the most pathological of cases involving shared blocks. Private blocks will remain in the R-states (where they are handled very efficiently) unless there is sharing due to task migration. In this case, the timer mechanism allows resetting from the S-states to the R-states. This solution is feasible provided that the size of the auxiliary table remains quite small. Each time the controller processes a request and the global state is an S-state, the address of the block involved must be matched against all entries in the table. If the number of entries is small, this search might be overlapped with the other actions performed by the controller, such as writing to or reading the block from memory. The number of entries is dependent on d; if d is very large, entries must be kept for a long time, and the number of entries needed in the table will increase. Since an entry must be created for each **Invalidation** signal that is broadcast, it follows that the frequency of such broadcasts must be fairly low if the number of table entries is to be kept small. The number of **Invalidation** signals sent depends in turn on the number of shared references–resulting from actual sharing or from task migration. The efficiency of this solution depends on the ability of the mechanism to reset blocks from S-states to the R-states. If they cannot be reset at the same rate that blocks change from the R-states to the S-states, the result is an increasing number of blocks in the S-states, accompanied with an increased frequency of **Invalidation** signals, and an increased likelihood of overflowing the auxiliary table. (Note that table entries are not necessary for correctness; the only purpose of the table is to improve performance. Therefore, in the case

of an overflow, additional entries to the table may simply be discarded.) The performance of this scheme is therefore extremely sensitive to the level of sharing and the maximum round trip signal delay, d, of the system network. For the scheme to work efficiently, the design of the controller must take into account information about the anticipated workload.

5.3.4 Extended 'twobit' scheme

This protocol has the potential to yield the best performance of all partial map protocols, yet the implementation cost is probably less than those of the other alternatives presented (especially considering such factors as the special timing mechanism required in the threebit scheme). The scheme uses the UNMOD-EXC state described in the modified full map approach. Although the use of this state introduced synchronization problems in that scheme, its use in the extended twobit protocol leads to a solution of the ghost signal problem. The local states used are:

1. INVALID. (INV)
2. UNMODIFIED-SHARED. (UNMOD-SHD)
3. UNMODIFIED-EXCLUSIVE. (UNMOD-EXC)
4. MODIFIED-EXCLUSIVE. (MOD-EXC)

The global state is one of the four used in the basic twobit protocol, namely:

1. ABSENT.
2. PRESENT1.
3. PRESENT*.
4. PRESENTM.

The requests from the cache controller to the memory controller are similar to those of the modified full map protocol, except that the **Modify-request** signal is not included, and an acknowledgement signal has been added. The complete set of signals includes:

- **Read-miss(cache#,block#).**
- **Write-miss(cache#,block#).**
- **Replace-unmodified(cache#,block#).**
- **Replace-modified(cache#,block#,data).**

- **Update(cache#,block#,data)**.
- **Exclusive-modified(cache#,block#)**.
- **Ack(cache#,block#)**. This signal acknowledges the arrival of a broadcast signal at the cache when it has an UNMOD-EXC copy.

The set of signals from the memory controller to the cache consists of:

- **Read-data-shd(cache#,block#,data)**.
- **Read-data-exc(cache#,block#,data)**.
- **Write-data(cache#,block#,data)**.
- **Write-back-shd(cache#,block#)**.
- **Write-back-inv(cache#,block#)**.
- **Invalidation(cache#,block#)**.
- **Set-shared(cache#,block#)**.

A description of the operation of the protocol is given below. The local and global state transitions are depicted in Figures 5.15 and 5.16.

- **Read hit**. The data is returned to the processor with no coherence overhead.
- **Read miss**. The cache sends a **Read-miss** to the controller, which checks the global state. If the state is ABSENT or PRESENT* the block may be sent without delay, using the **Read-data-exc** and **Read-data-shd** signals respectively. If the state is ABSENT, it is changed to PRESENT1. If it is PRESENT* it does not need to be changed. If the state is PRESENTM, the controller broadcasts a **Write-back-shd** signal. The cache with the modified copy will receive the signal and send an **Update** unless it has already replaced the block with a **Replace-modified**, in which case it ignores the signal. When the controller receives the **Update**, it changes the global state to PRESENT* and sends the block using **Read-data-shd**. If the controller should happen to receive a **Replace-modified** while waiting for the write-back, it changes the global state to PRESENT1 and sends the block using **Read-data-exc**. If the cache receives the block in a **Read-data-shd**, it loads the block in state UNMOD-SHD. Otherwise, the block arrives in a **Read-data-exc** and the block is loaded in state UNMOD-EXC.

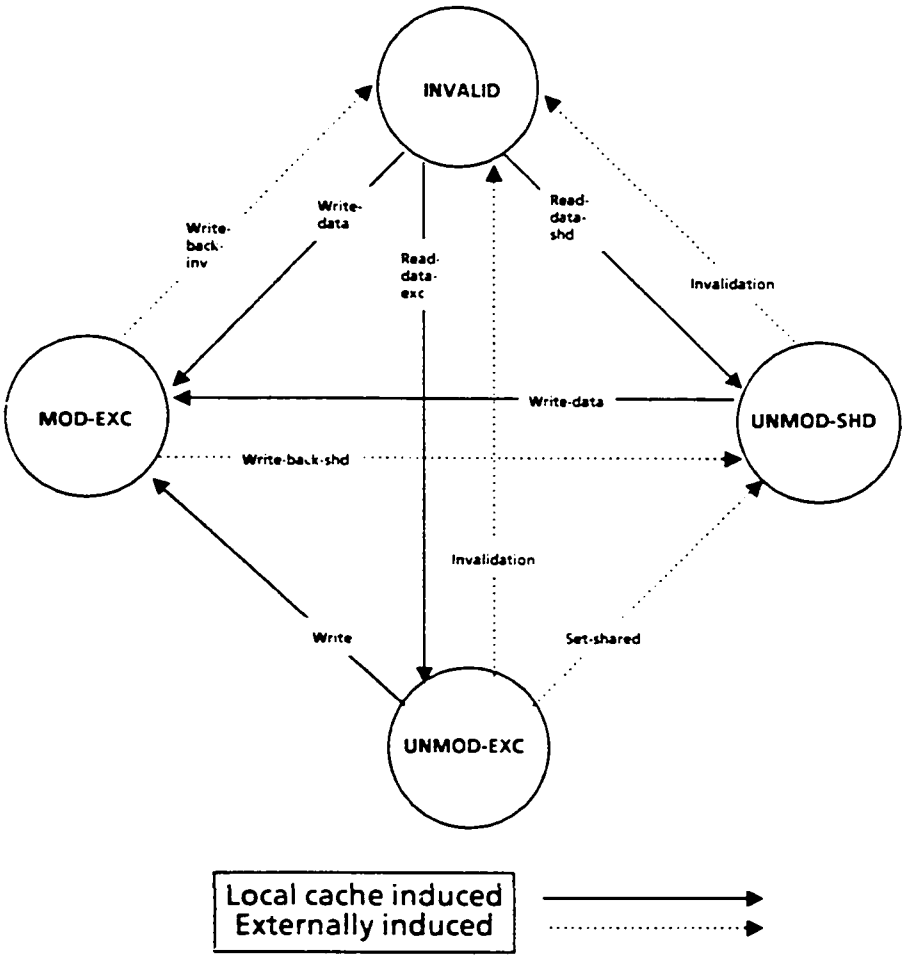


Figure 5.15: Extended twobit protocol cache state transitions

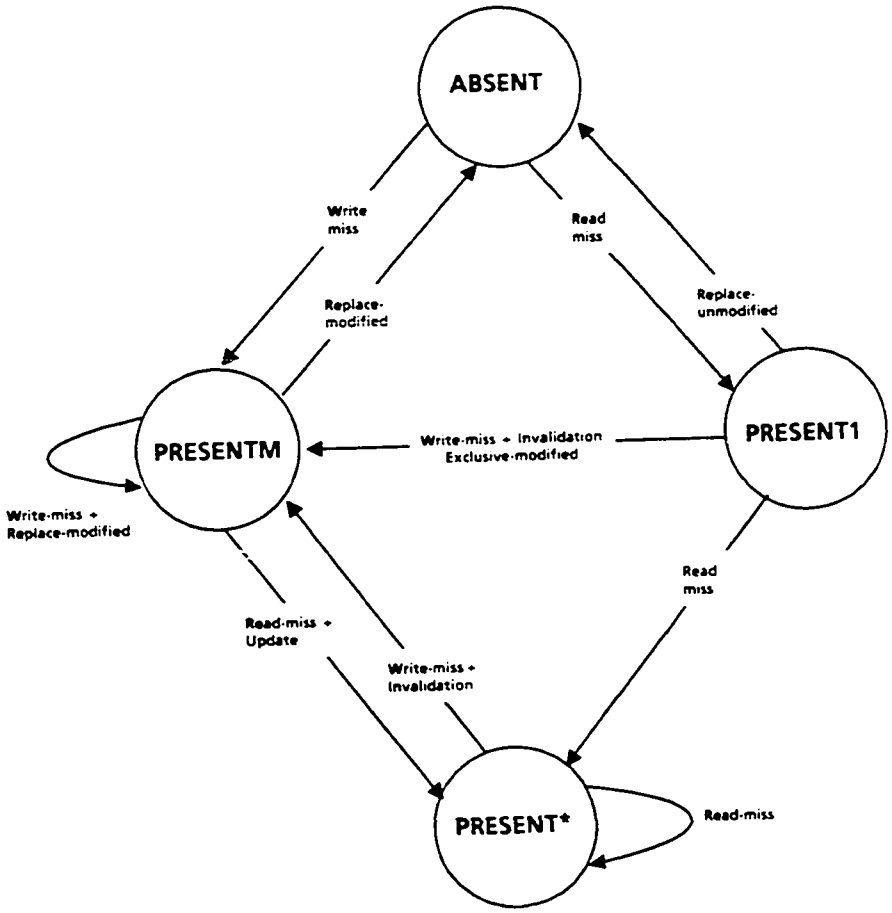


Figure 5.16: Extended twobit protocol global state transitions

If the global state is PRESENT1, the block cannot be sent until the cache with the exclusive copy is sent a **Set-shared** command and the cache responds with an **Ack** after setting its local state to UNMOD-SHD. While waiting for the **Ack**, the controller may receive a **Replace-unmodified** or an **Exclusive-modified** signal for the same block, submitted by the cache before the **Set-shared** command arrived. If the controller gets a **Replace-unmodified**, it sends a **Read-data-exc** to the requesting cache and leaves the global state PRESENT1 since the block is no longer shared. If the controller receives an **Exclusive-modified**, the global state is changed to PRESENTM and the controller proceeds as described above just after a **Write-back-shd** has been sent (by waiting for a write-back signal). The cache with the modified copy will receive the **Set-shared** command and treat is as a **Write-back-shd** by setting its local state to UNMOD-SHD and sending an **Update**. The cache knows to treat the **Set-shared** as a **Write-back-shd** because the local state is MOD-EXC.

- **Write hit**. If the local state is MOD-EXC, the modification can proceed without any additional overhead. If the state is UNMOD-EXC, the modification can take place immediately, accompanied by the sending of an **Exclusive-modified** signal to the controller. If the state is UNMOD-SHD, the write may proceed only after write permission is granted from the controller. To avoid problems with ghost **Modify-request** signals, the cache requests write permission by submitting a **Write-miss** just as if a write miss had occurred. The cache will receive write permission in the form of a **Write-data** signal. If the caches copy is still valid when this signal arrives, it may ignore the data portion and modify the block immediately, setting the local state to MOD-EXC.

- **Write miss**. The cache sends a **Write-miss** to the controller, which checks the global state. If the state is ABSENT, the block may be sent immediately accompanied by a state change to PRESENTM. If the block is PRESENT*, the block may be transmitted after an **Invalidation** signal is broadcast to all other caches. If the state is PRESENTM, the controller broadcasts a **Write-back-inv**, which the cache with the modified copy receives and obeys by sending a **Replace-modified** and setting the local state to INV. The outcome is exactly the same if the cache should happen to replace the block voluntarily before the **Write-back-inv** signal arrives. When the **Replace-modified** arrives at the controller, the global state is left PRESENTM and a **Write-data** is sent to the requesting cache. The cache always loads the data in state MOD-EXC.

  If the state is PRESENT1, the block cannot be sent until an **Invalidation** is sent and a response is received from the cache with the exclusive copy. The cache may send either

a **Replace-unmodified** or an **Exclusive-modified** before the signal arrives, or it may process it normally and respond with an **Ack**. The **Replace-unmodified** and **Ack** signals are treated identically, by setting the global state to PRESENTM and sending a **Write-data** to the requesting cache. If an **Exclusive-modified** signal arrives while the controller is waiting for a response, the global state is changed to PRESENTM and the controller proceeds as if a **Write-back-inv** signal had just been sent (and a write-back is expected). The cache with the modified copy will receive the **Invalidation** signal with the block in state MOD-EXC and treat it as a **Write-back-inv** by changing the state to INV and sending a **Replace-modified**. If the cache has already replaced the block voluntarily, the outcome is identical. For this to work correctly, the cache must always treat **Invalidation** signals as **Write-back-inv** when the local state is MOD-EXC.

- **Replacement**. When UNMOD-SHD blocks are replaced in the cache, no signal is required (since the global state must be PRESENT* and cannot be reset). If an UNMOD-EXC block is replaced, the cache must send a **Replace-unmodified**, which the controller will use to reset the state from PRESENT1 to ABSENT. If the replaced block is MOD-EXC, a **Replace-modified** signal is sent to update the block copy in main memory and to resent the global state from PRESENTM to ABSENT.

The protocol has no problems with ghost signals because it uses no **Modify-request** signals, and because **Replace-unmodified** signals are sent only if the local copy was UNMOD-EXC. Since other transactions on an exclusive block cannot proceed until the copy is no longer exclusive, there is no possibility of ghost **Replace-unmodified** signals. Potential timing problems with UNMOD-EXC blocks are resolved by forcing synchronization on the first transaction making the block shared, similar to the actions required with misses on MOD-EXC blocks except that no write-back is necessary.

The protocol has the potential for very good performance in the handling of private blocks. Allowing the immediate modification of all exclusive blocks reduces the coherence overhead on private blocks. The added overhead of treating write hits on UNMOD-SHD blocks as write misses makes the handling of shared blocks less efficient in this partial map protocol than in the modified full map protocol, but the overall performance is expected to be comparable to the full map protocols, assuming a low level of sharing.

5.4 Crossbar Simulation Results

Figures 5.17 through 5.24 show the simulation results of several of the protocols described in this chapter using a crossbar switch. Results are presented for the full map approach of Censier and Feautrier, the extended full map approach of Yen and Fu, the basic twobit protocol, the extended twobit protocol, the threestate protocol, and a simple software scheme. The simulated version of the twobit protocol avoids problems with ghost **Modify-request** signals by returning a valid copy of the block with the **Modify-granted** signal. (Other timing problems are avoided because of the limited delay and queueing in the crossbar.) The threebit protocol was not simulated, but its performance would be very similar to that of the twobit protocol, since the timing of the crossbar switch would allow S-states to be reset to R-states after very little delay. The software scheme is essentially identical to the shared bus software approach: blocks tagged as shared are not cached, and the other blocks are cached without any coherence overhead. It is included to give an idea of the potential relative performance of similar schemes. The indicated level of performance of this scheme is somewhat exaggerated, since the simulation does not include the overhead of eliminating task migration (or flushing the cache on each context switch) that would be necessary in a real implementation.

Simulation results are shown using four and eight memory modules with varying levels of sharing (a function of the percentage of S-block references and the number of S-blocks). In the crossbar simulation, the first system resources to become bottlenecks are the memory controllers; the bandwidth of the crossbar is not a limitation as was the case with a shared bus. Because the performance differences between the schemes are the greatest when the curves begin to level out (when the bottleneck begins to saturate), results are presented with a relatively small number of memory modules. A system using only four memory modules with 32 processors is probably not realistic, but the differences between the curves become smaller as the number of modules is increased (similar to the curves in Figures 5.21 and 5.24). A more realistic system using n processors and n memory modules (with an n by n crossbar) would reveal little if any differences between the schemes.

Because the memory is the first to saturate, the schemes are ranked in these results according to demands on the memory controllers, and not according to the amount of traffic passing through the switch. Very different results could be expected if an interconnection network was simulated with much bandwidth than a crossbar switch (e.g., multistage network), in which case the amount of traffic generated by the protocol would likely become much more important. As expected, the extended full map protocol improves on the performance of the basic full map approach. The

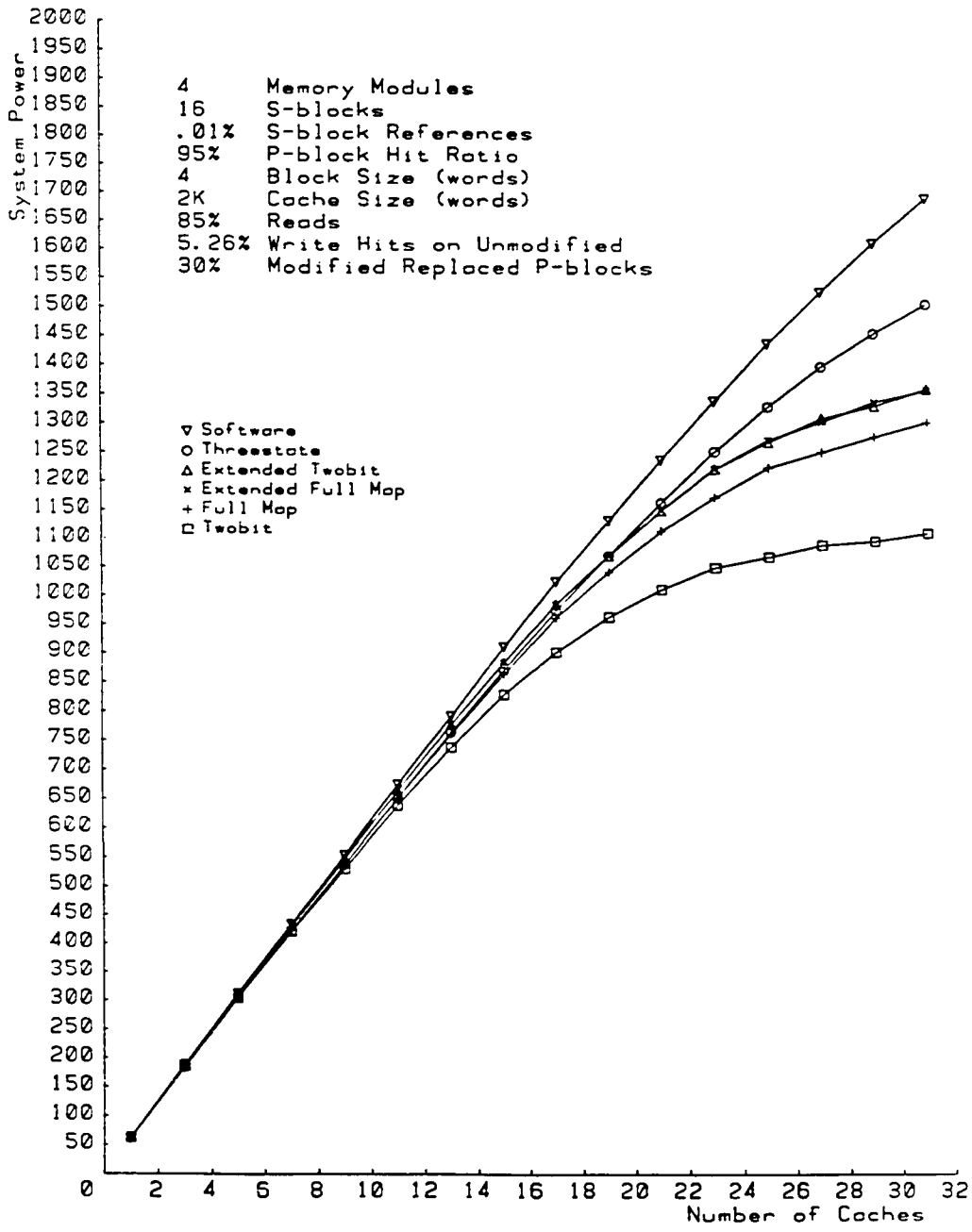


Figure 5.17: Four memory modules, negligible sharing

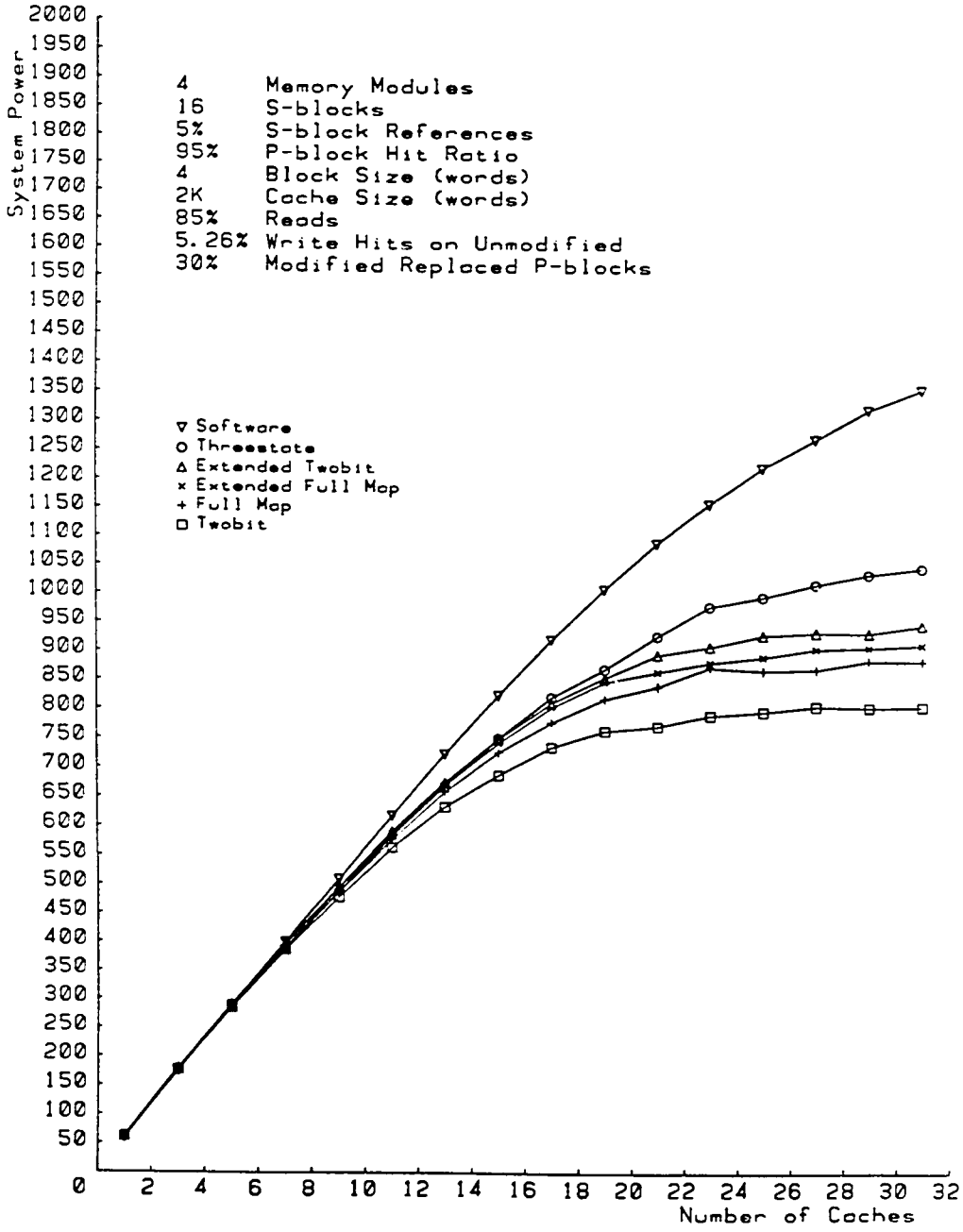


Figure 5.18: Four memory modules, basic model, 16 S-blocks

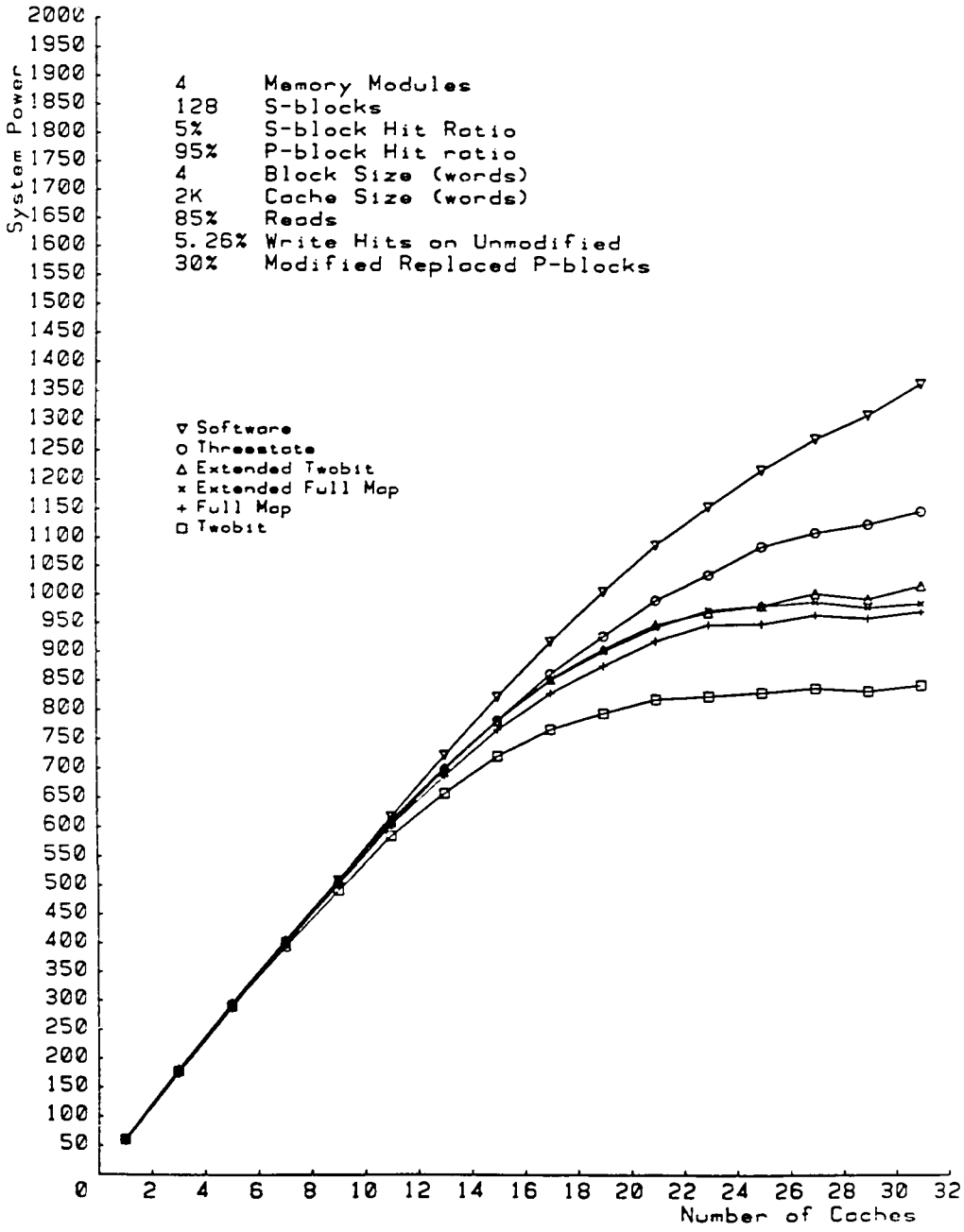


Figure 5.19: Four memory modules, basic model, 128 S-blocks

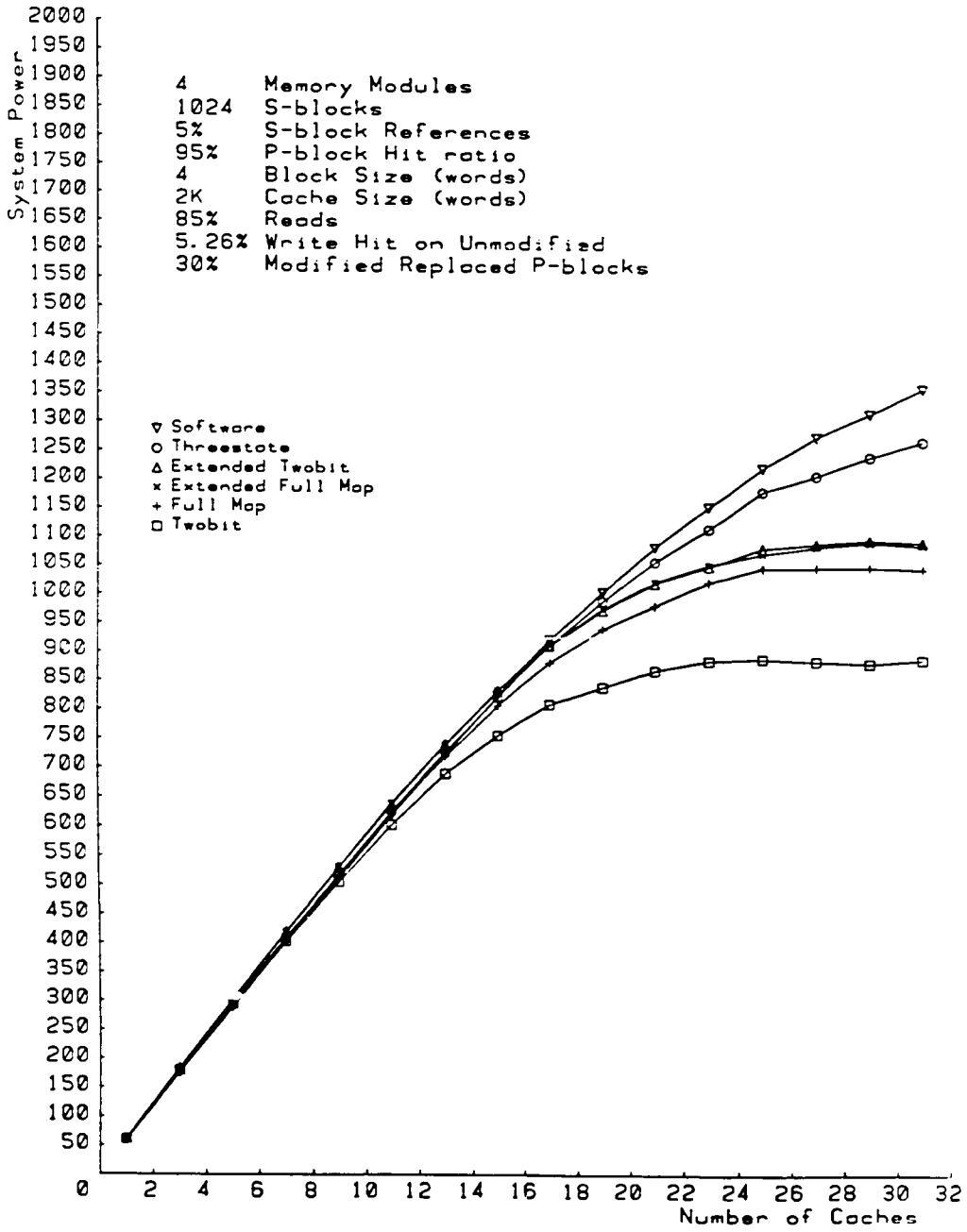


Figure 5.20: Four memory modules, basic model, 1024 S-blocks

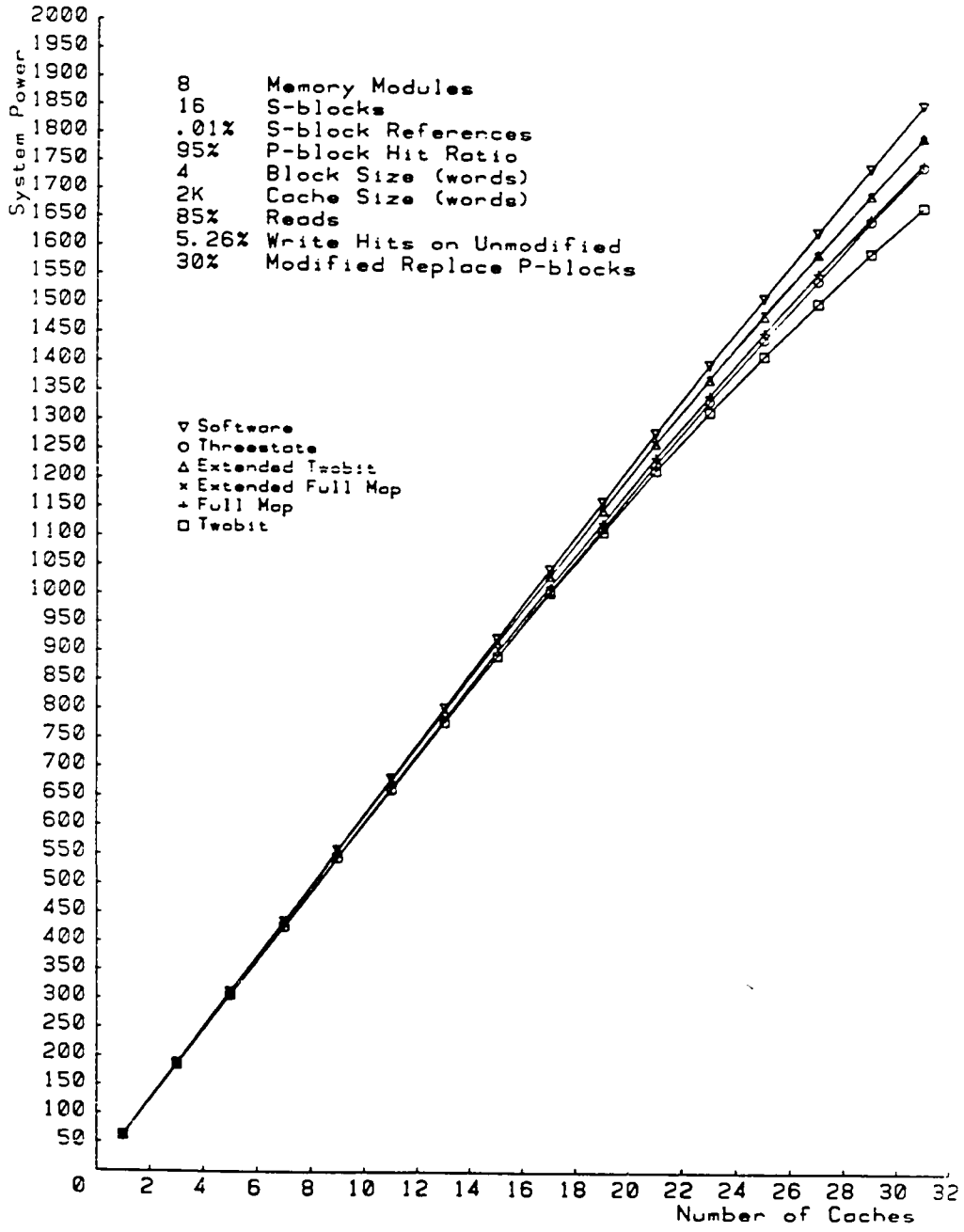


Figure 5.21: Eight memory modules, negligible sharing

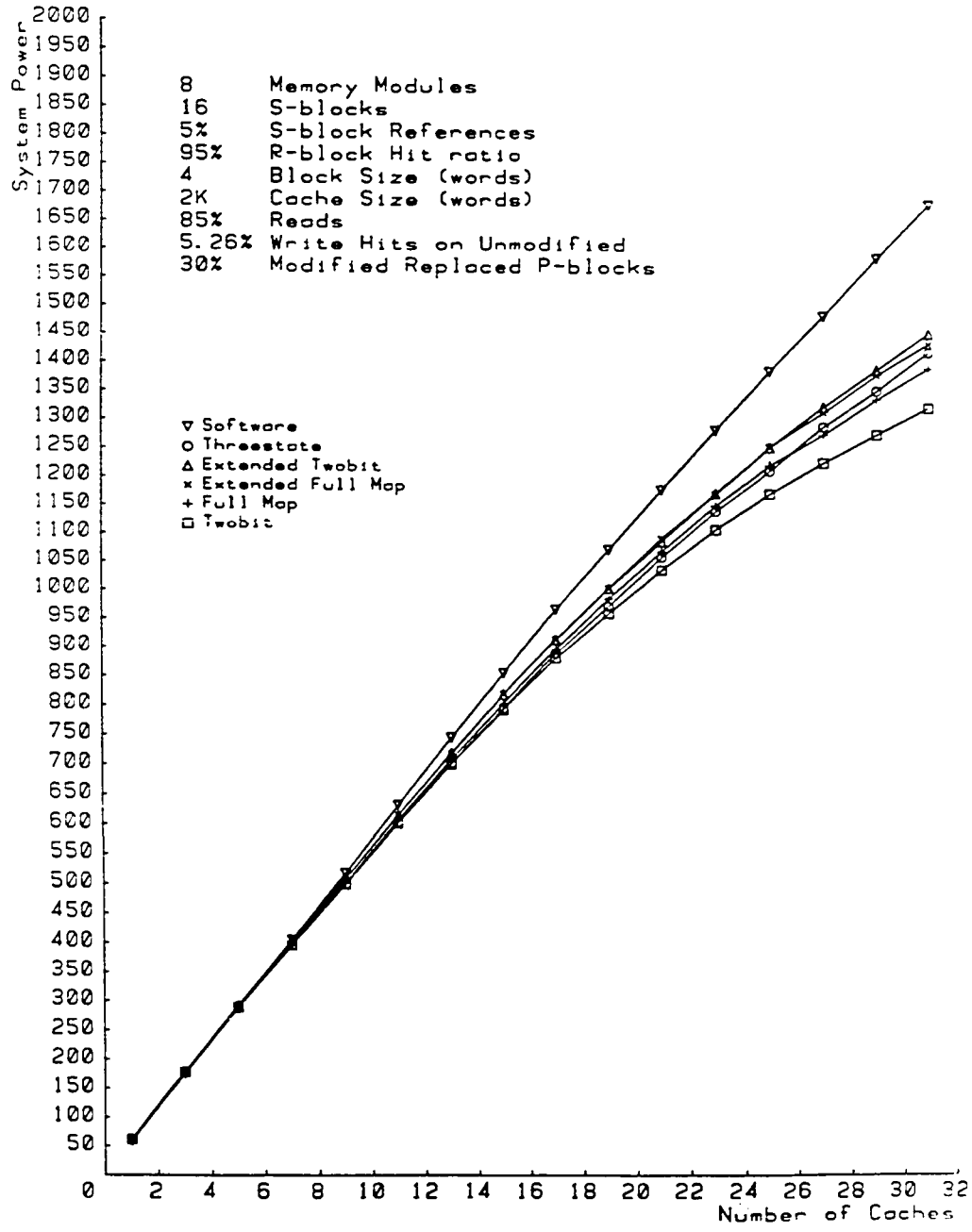


Figure 5.22: Eight memory modules, basic model, 16 S-blocks

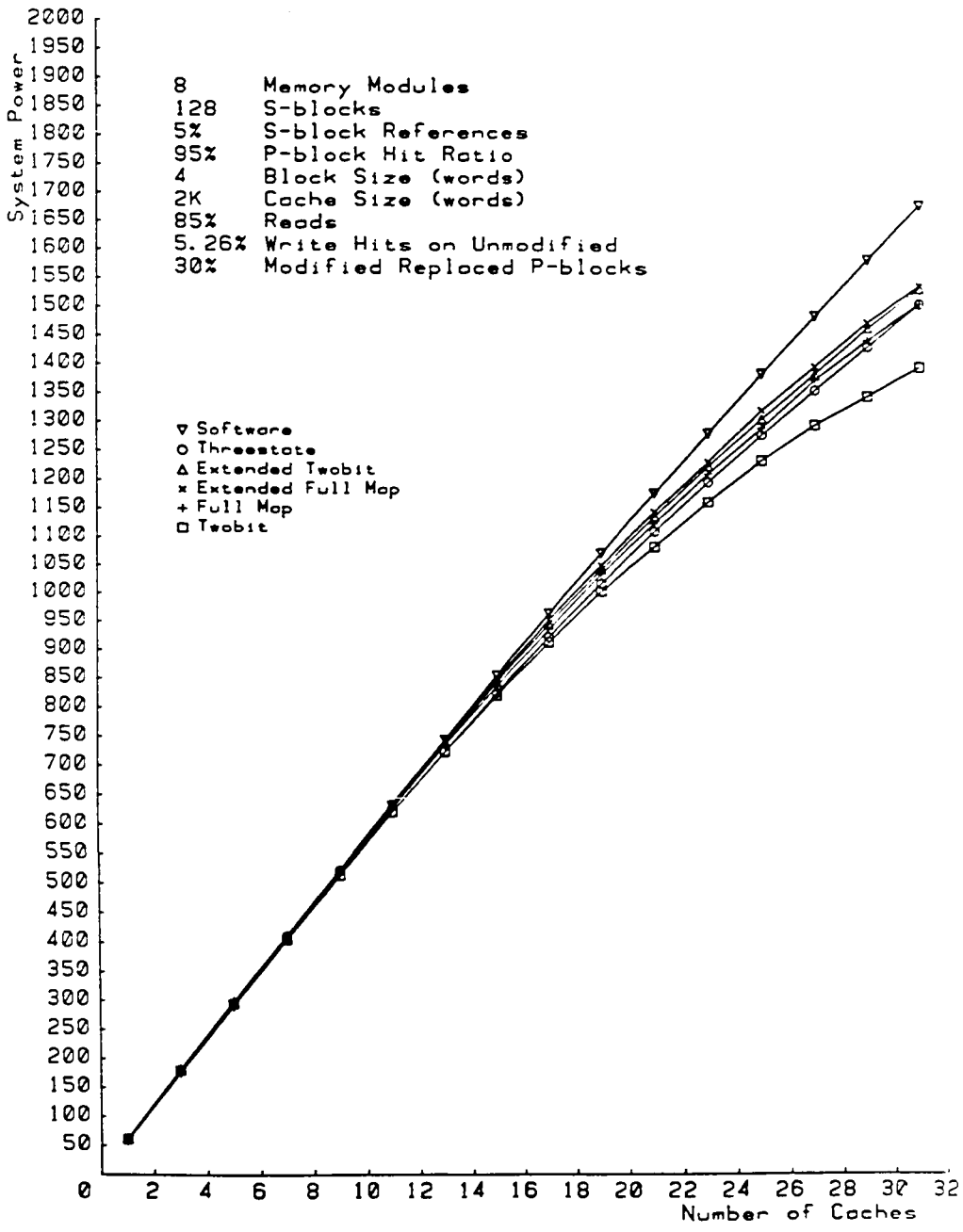


Figure 5.23: Eight memory modules, basic model, 128 S-blocks

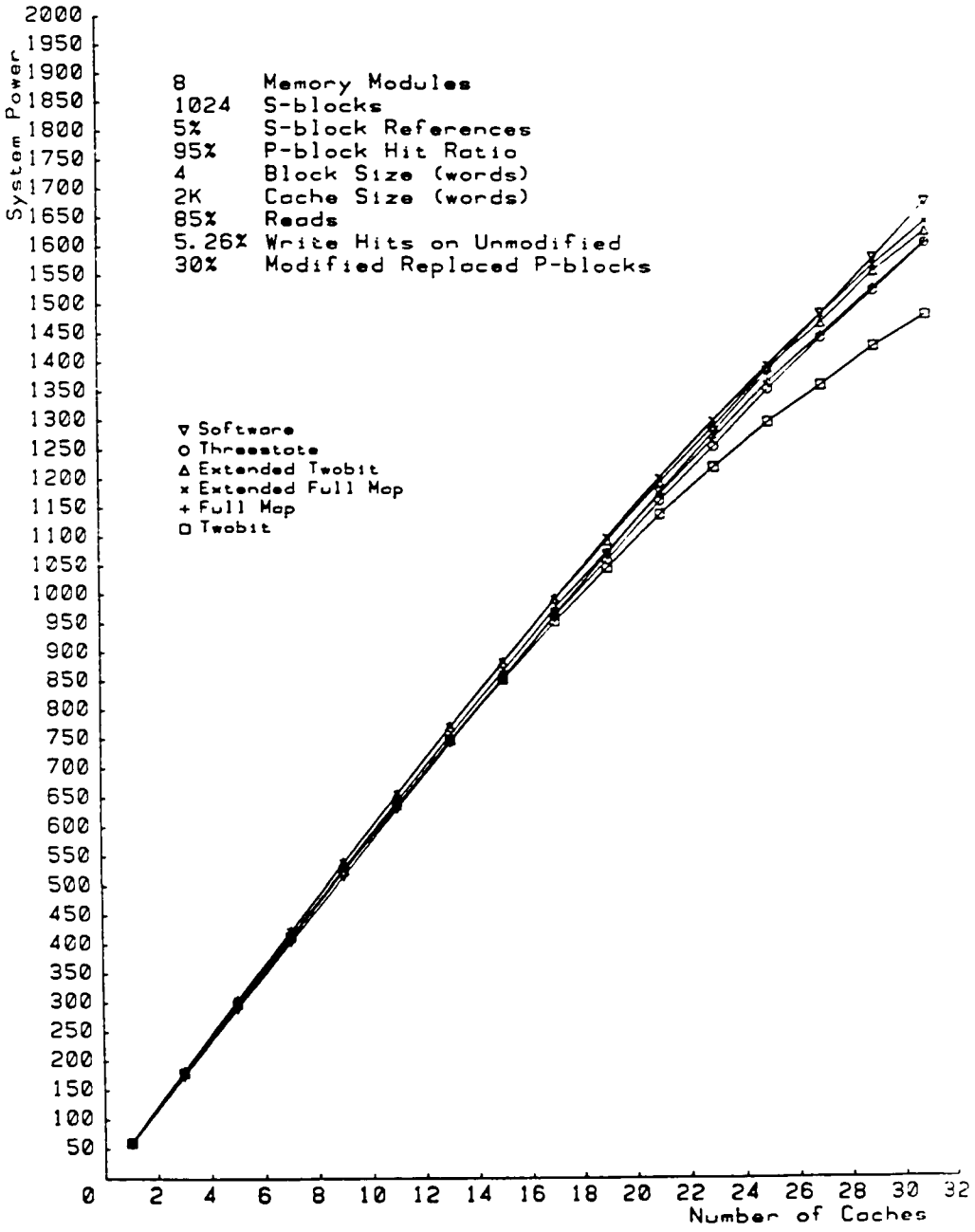


Figure 5.24: Eight memory modules, basic model, 1024 S-blocks

extended twobit protocol makes an even bigger improvement over the basic twobit approach, and gives performance comparable to the full map protocols. The high level of performance of the threestate protocol and the software solution is somewhat surprising, and the result of decreased service demands on the memory controllers (through the elimination of cache-controller service requests). This and other causes of the observed differences in protocol performance will be discussed in the following sections. As in Chapter 4, the separate issues of shared block and private block handling will be examined in detail.

5.4.1 Private Block Handling

Figure 5.17 displays the performance of the protocols with a negligible amount of sharing and four memory modules. The differences between the curves are the result of differences in the overhead of handling private blocks (neglecting effects due to task migration). As can be observed from the figure, the software scheme has the lowest overhead, followed by the threestate scheme. Consider the differences between the protocols in the following types of transactions on private blocks: read and write misses, write hits on unmodified blocks, and replacement.

Since the block is not present in any other caches, the time to load a block on a read or write miss is the same for all schemes (neglecting contention). While the delay observed by the cache is the same, and the transmission time is the same, there are slight differences in the total service time required at the memory controller. In the simulation, the total time to read and update the global map cannot be overlapped entirely with the time to read the block from memory unless the block size is larger than four words. Since the software scheme does not maintain a global state, it requires less service time of the memory controller. (In the simulation, the memory controller does not begin to process the next request until the map cycle and memory cycle associated with the previous request are complete.) There is one additional difference between the protocols, although it does not affect the memory controller service time. The threestate protocol will not find private blocks in state ABSENT on a miss unless they were modified when replaced the last time. (The probability of this occurring is approximated in the simulation by the percentage of modified P-blocks at replacement.) In those cases when the block was unmodified the last time it was replaced (and is therefore PRESENT*), a write miss requires the broadcast of an **Invalidation** signal. Since this signal can be sent while the global state is being updated and the block is being read from memory, it does not increase the service time of the request. It does, however, increase the amount of traffic through the switch.

In the case of write hits on unmodifed blocks, both the basic twobit protocol and the full map

protocol require the cache to send a **Modify-request** to the memory controller. In the full map protocol, the controller is able to send a **Modify-granted** after checking the global state, following which it must also update the global state. In the twobit protocol, the memory controller sends the valid block contents with write permission, so the controller must also read the block from memory. For blocks in state ABSENT (and block sizes of four words or less), this block load can be completely overlapped with the reading and updating of the global state. Therefore, the total memory controller service demand is the same, but the transfer time and the observed cache delay is longer with the twobit protocol. In the threestate protocol, the global state will be PRESENT* and an **Invalidation** signal must always be sent. In addition, the controller always transmits write permission via a **Write-data**, including a full block transfer. However, both the block read and the **Invalidation** broadcast can be overlapped with the reading and updating of the global state, so the service time of the threestate scheme is identical to the basic twobit protocol (although the switch traffic increases because of the added broadcasts).

In the extended twobit protocol and the extended full map protocol (both with UNMOD-EXC states), the write may proceed locally accompanied by the sending of a **Exclusive-modified** signal. When this signal is received by the memory controller, it is not necessary to check the global state; it is simply changed to PRESENTM. The overhead of both schemes is therefore the same, in terms of bus traffic generated and memory controller service requirements. In the software scheme the write takes place in the local cache without any traffic generated and without any service required of the memory controller.

The remaining difference between the schemes arises in the replacement of private blocks. In all approaches, if the selected block is modified, it must be written-back to update main memory. For this transaction, the demands on the memory controller are identical for all schemes, since the update of the map (for those schemes with a map) can be overlapped with the writing of the block to memory. In addition, the amount of switch traffic generated by the writeback in each protocol (dictated by the size of the block) is identical for all schemes. When an unmodified block is selected for replacement, the software and threestate schemes require no action to be taken, eliminating work for the controller, as well as reducing the switch traffic. All other protocols require the transmission of a **Replace-unmodified** signal. In the full map protocols, the memory controller can service the signal by simply clearing the associated bit–no read is required. In the extended twobit protocol, the controller is also able to update the map (to ABSENT) without first reading it, since the signal only comes from caches with an exclusive copy. However, in the basic twobit protocol, the memory controller must read the map before it can be updated, since it also receives **Replace-unmodified** signals when the global state is PRESENT* and cannot be updated.

In terms of memory controller demands, the software solution has the lowest overhead in all cases mentioned above, consistent with the simulation results. The threestate is the next best because it entirely avoids **Replace-unmodified** signals. Although these schemes increase the switch traffic, the bandwidth of the crossbar switch is sufficient to accept the increase without any noticeable degradation in performance. Following the software and threestate protocols come the extended twobit and extended full map protocols. The addition of the UNMOD-EXC state increases the efficiency of private block handling (with respect to the basic protocols without this state) by allowing a more efficient modification of blocks loaded on read misses. In essence, the UNMOD-EXC state grants the cache write-permission at any time, without the overhead of sending a **Modify-request** and waiting for a response as in the basic protocols. This is the sole cause of the observed difference in performance between the full map protocol and the extended full map protocol. The performance of the basic twobit protocol is lower than all other approaches as a result of treating each **Modify-request** signal as a **Write-miss**, and as a result of reading and updating the global state in servicing each **Replace-unmodified**.

5.4.2 Shared Block Handling

Figures 5.18 through 5.20 and 5.22 through 5.24 display the performance of the protocols with an increased level of sharing. In Figures 5.22 through 5.24, the simulation uses eight memory modules, and the differences in performance are much less than Figures 5.18 through 5.20 using four memory modules. A comparison of Figure 5.18 with Figure 5.17 shows that increasing the level of sharing significantly reduces the performance of all schemes, although the ordering of the schemes remains much the same. In the following discussion, differences between the protocols will be identified in the cases of read and write misses, write hits on unmodified shared blocks, and block replacement. Of course, none of these apply to the software scheme in which shared blocks are not cached. In this scheme, each shared block reference generates a main memory access.

In the case of misses on blocks present in other caches, there are a number of differences between the protocols. In the extended twobit protocol and in the extended full map protocol, when a cache has a UNMOD-EXC copy, the controller cannot complete the servicing of either read or write misses until an acknowledgement is received from that cache (in response to a **Set-shared** or an **Invalidation**). This added shared block overhead is the result of an optimization that improves private block handling, but it also degrades performance on shared blocks. This can be seen by comparing the full map protocol and the extended full map protocol in Figures 5.17 and 5.18. With a higher level of sharing, the difference between the two approaches decreases as

a result of this added synchronization overhead.

A second important difference between the schemes in the handling of read and write misses is that the partial map protocols require broadcasts instead of sending signals to specific caches because their identity is not known. In the full map protocols, only those caches with valid copies will be sent signals, but the signals must be sent sequentially. Therefore, in those cases when two or more signals must be sent, the time required for the controller to transmit the signals is longer than the time required for a broadcast (neglecting time spent waiting for the switch to become available). For this reason, the performance of the extended twobit protocol actually exceeds that of the extended full map protocol for high levels of sharing.

A third but much less significant difference is that the partial map protocols do not require the global state to be updated in servicing a read miss when the global state is PRESENT*. Using the parameters of the simulation, this saves a single cycle of the memory controllers time on each occurrence.

The only difference arising in the case of write hits on unmodifed blocks present in other caches is that all partial map protocols must treat the **Modify-request** as a **Write-miss**, including full data transfer. This increases the memory controller service requirements and the observed cache delay and it causes an increase in the traffic in the interconnection network.

Issues relating to block replacement are identical to those considered in the previous section with one exception. The extended twobit protocol does not require a **Replace-unmodified** signal when an UNMOD-SHD block is replaced.

For the results presented here, the ordering of the protocols is determined by the service requirements of the memory controller, since our parameters are set up in such a way that the memory controller saturates before the crossbar switch. Simulation of a system with only four memory modules is somewhat unrealistic in that the ratio of processors to memory modules becomes quite high. When the number of memory modules is increased to eight, the differences in performance with negligible sharing become much less pronounced, as all system components are far from saturation. If the protocols were simulated using an interconnection network with a lower bandwidth than the crossbar used here, the relative performance of the schemes would be more affected by the network traffic that they generate, and it is quite likely that a different ordering of the protocols would be the result. In particular, the additional broadcast traffic generated by the threestate scheme would make it much less efficient. The other partial map protocols would also experience a degradation in performance because of their broadcast overhead, although it is significantly less than the overhead of the threestate scheme. The performance of the software scheme would also be reduced at high levels of sharing, since traffic is generated at every shared

block reference.

5.5 Alternative Protocols

There are, of course, many alternatives to the cache coherence protocols described in this chapter. In particular, there are a number of enhancements that could be made to improve the performance of the partial map schemes described in the previous sections. The following sections describe three such extensions, namely: adding a special table for the memory controller, altering the number of bits in the global state, and using software hints. While none of these extensions is sufficient to correct the errors in an incorrect protocol (such as the ghost signals in the basic twobit scheme), each can be used to improve the performance of a correct partial map protocol.

5.5.1 Cache of Presence Bits

The first enhancement involves the addition of a special table, accessed by the memory controller, that contains an n bit tag for each entry in the table. Each of these bits functions as a cache bit in the full map approach, recording the identities of those caches with copies. However, in this scheme an entry is created in the special table only when the block is loaded into a cache. The table could be organized like a normal cache, in which each block is mapped to a set of entries which can be efficiently matched in parallel. Like the cache bits in the full map protocol, the bits in the special table entry would be used to reduce the number of broadcasts. If an entry exists for a particular block, the signals can be sent directly to the caches with copies, just as in the full map protocols. If no entry exists, broadcasts must take place, as in the basic partial map protocol.

The performance of the partial map protocols with this modification can be made to approach the performance of the full map protocols to any desired degree by ensuring a sufficiently high hit ratio on the special table. For example, if the table contains the desired block entry 90% of the time a broadcast is required, then 90% of the broadcast overhead will be eliminated. The actual size and organization of the presence bit table would require careful study to ensure a hit ratio sufficient to make the approach cost-effective based on the parameters of the system involved, including the number of caches in the system and the size of each (affecting the number of blocks that could be cached at any instant), and the number of blocks in each memory module.

This technique has the potential to yield very good performance, but it comes at the cost of the special table and additional memory controller complexity. It also has the disadvantage that the organization of the special table may limit the expansion of the system. Since each cache must have a bit in each entry in the special table, caches cannot be added to the system unless there

are sufficient cache bits in the table. Increasing the system size beyond this limit would require the replacement of the special tables. While this would not be required in the existing partial map protocols as presented, it would be considerably cheaper than a similar expansion using a full map protocol (since the number of entries requiring expansion or replacement would be far greater).

5.5.2 Two + Log(n) Bits

A second alternative is to expand the size of the global tag associated with each block in main memory, allowing the global state to contain more information. One interesting possibility is to add log(n) bits to each tag entry. This allows the encoding of the identity of a single cache or the number of caches with copies. For example, in the extended twobit protocol, the added field would contain no useful information when the global state is ABSENT, the identity of the cache with an exclusive copy when the state is PRESENT1 or PRESENTM, and the number of caches with copies when the state is PRESENT*. Recording the cache identity with exclusive copies eliminates all broadcasts except in the case of a **Write-miss** on PRESENT*. Maintaining a count of caches with copies in state PRESENT* allows the state to be reset to ABSENT when all copies are replaced (provided that that the protocol is extended to include a **Replace-unmodified** for UNMOD-SHD blocks).

This modification has the advantage of improving system performance without the addition of a special table as in the previous section. The disadvantages are the increased size and cost of the global map, and the limitation on system expansion. The latter concern is not as critical as in the full map protocols; the addition of a few extra bits to each tag entry would very likely allow any desired expansion, since each extra bit allows the number of caches to be doubled.

5.5.3 Twobit With Software Hints

As with the shared bus protocols, the addition of software hints allows some interesting modifications to the basic protocol. Hints from the system software can be very valuable in distinguishing between actual sharing and task migration. Assume, for example, that each block is tagged as one of the following: Shared-Read-Only, Shared-Writable, Private-Read-Only, or Private-Writable. (Assume also that the value of the tag can be obtained from the TLB entry associated with the virtual page containing the block, and that the contents are therefore known at each processor reference.) In the extended partial map protocol (with an UNMOD-EXC local state), the performance could be improved by treating read misses based on the block class as follows:

- **Shared-Read-Only**. Since the block will not be modified, there is no advantage to loading the block in UNMOD-EXC on a read miss. Instead, an UNMOD-SHD copy of the block is loaded and the global state is set to PRESENT*–even if no other copies are cached. (For the memory controller to do this requires a special read miss signal from the cache.) This avoids the additional synchronization necessary on the second read miss (waiting for an **Ack** from the cache with the exclusive copy), and it avoids the overhead of **Replace-unmodifed** signals. The global state will reach PRESENT* on the first miss and remain in that state with minimal coherence overhead.

- **Shared-Writable**. This class can be processed normally. There appears to be little advantage to taking a different approach. Always loading the block in UNMOD-SHD state is undesirable because it will incur additional overhead if it is modified. Always loading the block in UNMOD-EXC state is also undesirable because there may be several caches accessing the block and invalidation of their copies could cause severe thrashing.

- **Private-Read-Only**. As with Shared-Read-Only, the block should be loaded in UNMOD-SHD and the global state set to PRESENT* even if no other copies exist. (They may exist as a result of task migration, but since the block will never be modified, there is no disadvantage to leaving the state PRESENT*.) This also avoids all **Replace-unmodified** signals for these blocks.

- **Private-Writable**. If the global state indicates that another copy exists when a read miss on this type of block is serviced (requiring another special read miss signal), the other copy should be invalidated so that the requesting cache may load the block in UNMOD-EXC allowing subsequent modification with minimal overhead. (Since the block is private, the other copy is the result of task migration and may be invalidated without degrading the performance of the other processor.) For blocks in this class, the only possible global states are ABSENT, PRESENT1, and PRESENTM. At the time of the read miss, if another cache has a copy, that copy is always exclusive, and therefore the protocol requires a response from the cache (either a write-back or an acknowledgement) before the copy may be loaded in the requesting cache. It takes no extra time to send an **Invalidation** instead of a **Set-shared**, or a **Write-back-inv** instead of a **Write-back-shd**.

It is important to note that the software tags are used only to improve performance and not to provide correctness. While the hint must be correct to obtain the highest level of performance, its validity has no effect on the correctness of the protocol.

5.6 Other Topics For Future Research

There are many topics relating to cache coherence protocols for general interconnection networks that should be examined in more detail. One of the most important is the study of protocol performance on interconnection networks other than a crossbar. In a 'realistic' crossbar system (one in which neither the memory nor the switch become bottlenecks), little if any difference would likely be observed between the protocols. However, crossbars are very expensive to implement, and it is likely that multiprocessor systems with even a moderate number of processors will use a more cost-effective interconnection network. Simulation using omega, banyan, or delta networks [Fen81] with lower bandwidth than a crossbar could lead to a very different ordering of the protocols.

Another important performance consideration is the comparison of software approaches to the hardware protocols. It is generally accepted (even by advocates of software solutions [Smi85a]) that snooping protocols are best for shared bus multiprocessors, but the coherence overhead of protocols for general interconnection networks is higher. This can be seen by comparing the relative performance of the software shared bus scheme with the software scheme in this chapter. While the performance of the software approach was significantly lower than the better hardware-based shared bus protocols, the same approach results in performance exceeding that of all crossbar protocols (neglecting such limitations as task migration). In short, software oriented solutions are more appropriate for a non-bus multiprocessor, and an analysis of the relative performance of hardware and software approaches would be very important. This analysis would require the formulation of a different simulation workload model allowing higher level abstractions of shared variable references.

One intriguing possibility that deserves further investigation is the addition of a special bus connecting all caches (or processors) that could be used to implement a distributed write protocol. In Chapter 4, the performance of distributed write shared bus protocols was shown to be a definite improvement over even the most efficient of invalidation protocols. It is possible that a similar performance improvement could result from adding this feature to the protocols discussed in this chapter. The addition of a special bus would make the timing of the protocol very complicated. For example, it would be possible for cache A to perform a distributed write on the same block that B has just submitted a read miss for. If the controller has already sent the block to B, the new modification will not have affected the copy that arrives at B. If B is waiting for the block to be loaded when the distributed write occurs, it can solve the problem by taking the new data (for that portion of the block) and ignoring the corresponding part of the block when it arrives from the controller. However, if the distributed write takes place before the read miss occurs, cache B

cannot take the data and must rely on the memory controller to send a valid copy. For this to work correctly, the memory controller must be aware of A's modification before the read miss arrives from B. A solution to the problem would probably require the definition of additional global states involving distributed write permission and responsibilities. The use of software hints could prove to be very advantageous. In particular, distributed writes need be performed only on those blocks tagged as shared so the special bus is not saturated.

As was the case with the shared bus simulation results, actual run-time measurements gathered during the execution of a multiprocessor would be very valuable in verifying and validating the simulation model that was used. In addition, extensions of the simulation model could allow a more precise comparison of the performance of software schemes relative to the hardware protocols examined in this chapter.

Chapter 6

Definition of Coherence

One important topic to be discussed with respect to the protocols in the previous chapter is the definition of coherence. According to the definition given in Chapter 2, a system of caches is coherent if every read returns the value most recently written to that location by any processor in the system. This is an acceptable definition for shared bus systems, but it is not sufficient for general network protocols. Consider the following example, illustrated in Figure 6.1. Caches A and B have UNMOD-SHD copies of block *i*. Cache A wants to modify the block and submits a **Modify-request** to memory controller X, which services the request by sending an **Invalidation** to B and sending a **Modify-granted** to A. It is possible for A to receive the **Modify-granted** and complete the write before B receives the **Invalidation**. In this case, B will continue to read the old value until the **Invalidation** arrives.

This occurrence does not fall within the given definition of coherence, yet it does not affect the correct execution of the program. A satisfactory definition of coherence must make allowances for possible interconnection network delays as in the example above.

Definition 1 *A system of caches is said to be coherent with respect to block a if each cache in the system observes all modifications of a in the same order.*

A modification is *observed* by a cache when any subsequent read returns the newly written value. In all cache coherence protocols, the modification is observed when the cache receives an invalidation signal or a distributed write for the block, since these signals are processed before the cache services another processor reference. Definition 1 allows for an arbitrary delay in the sending and receiving of signals, so long as the observed order of writes remains the same in all caches for any given

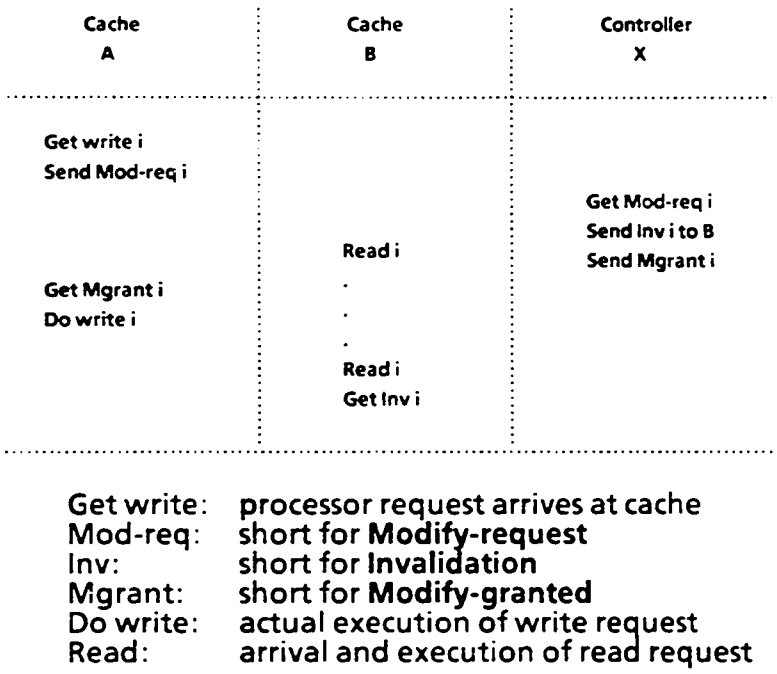


Figure 6.1: Delayed arrival of Invalidation signal

block. All of the protocols in Chapter 5 maintain coherence of each block by strictly regulating the permission to write that block.

It is important to contrast coherence with the stronger notion of sequential consistency, as defined by Lamport [Lam79].

Definition 2 *A system is sequentially consistent if 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.*

In multiprocessors with a general interconnection network, it is possible for the system to be coherent but not sequentially consistent. Consider the following example, illustrated in Figure 6.2. Caches A and B each have UNMOD-SHD copies of blocks i and j, which are located in memory modules X and Y respectively. Processor A's reference stream contains a write to i followed by a read from j, and processor B's stream has a write to j followed by a read from i. Assume that both writes happen at about the same time. Since both copies are shared, write permission must be obtained before the writes are allowed to proceed. The controllers will receive the requests for write permission, send **Invalidation** signals to the other caches with copies, and then send the requesting caches signals granting write permission. It is possible for A and B to both receive permission to write before either receives the incoming **Invalidation** signals, in which case the read of block i and the read of block j both return the values of the blocks before the write. It should be readily apparent that there is no way to serialize the two execution streams and obtain the same outcome. Therefore, the system is not sequentially consistent.

If, however, the coherence protocols are modified to ensure that submitted **Invalidation** signals *arrive* at the caches before write permission is granted, all references will be sequentially consistent [DSF86]. In general, the arrival of **Invalidation** signals can be guaranteed by using circuit switching, in which a path is established and held until the signal arrives at the cache (effectively what is done in a shared bus system), or by using packet switching and forcing the controller to wait for an acknowledgement from each of the caches before granting write permission. If, in the example given in the previous paragraph, the memory controllers do not grant write permission until acknowledgements are received, it will always be possible to serialize the memory references. For example, assume that cache A receives write permission and performs the write to block i and the read to block j before it receives the **Invalidation** signal for block j. Then it must be the case (as shown in Figure 6.3) that cache B receives the **Invalidation** for block i before it receives write permission for block j, since the **Invalidation** signal for block i must have been acknowledged before cache A received write permission for i, and since cache B cannot receive write permission

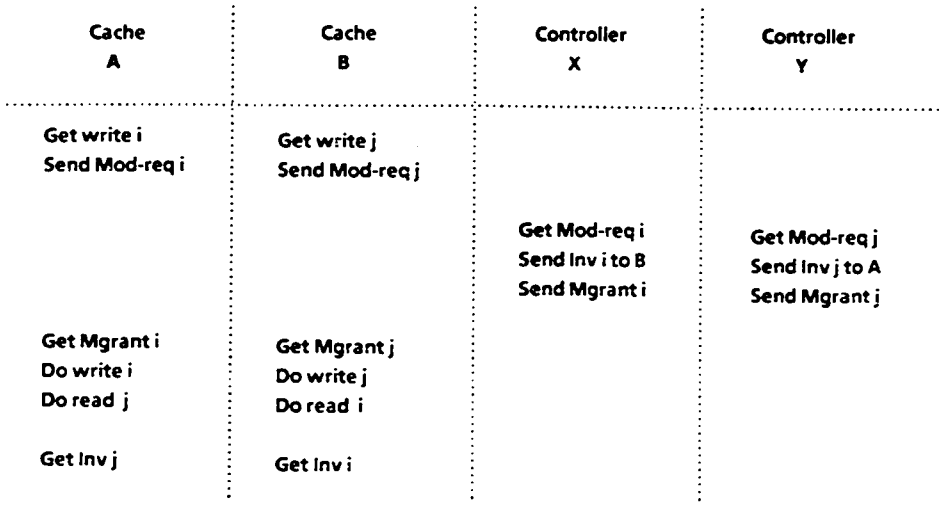


Figure 6.2: Example of coherence without sequential consistency

for block j until the **Invalidation** of j is acknowledged by cache A. It is therefore impossible for cache B to complete the read to block i before the **Invalidation** arrives, so that it will always return the newly written value. In this case, the references are easily serialized by first completing the two instructions of processor A and then the two instructions of processor B.

Although the addition of acknowledgment signals might be practical in full map protocols, it would be very difficult in the partial map protocols since it is never known which caches have copies and how many such acknowledgement signals would be returned. (Of course, each cache could return a signal whether it had the block or not, but this would increase the traffic in the switch and it would increase the service time of the request, since many such signals would have to be processed.) There are, however, other alternatives for maintaining sequential consistency. Before they are discussed, an alternative definition of sequential consistency will be presented, which is more convenient to use.

Collier has proven that if all processors observe all writes in the same order then sequential consistency is maintained [Col85]. This allows the following alternative definition of sequential consistency:

Definition 3 *A system is sequentially consistent if all caches observe all writes in the same order.*

This second definition is useful in reasoning about cache coherence protocols, since it is concerned with the ordering of writes, just as the definition of coherence (Definition 1). For example, it is easy to see that all shared bus protocols provide sequential consistency, because all caches observe

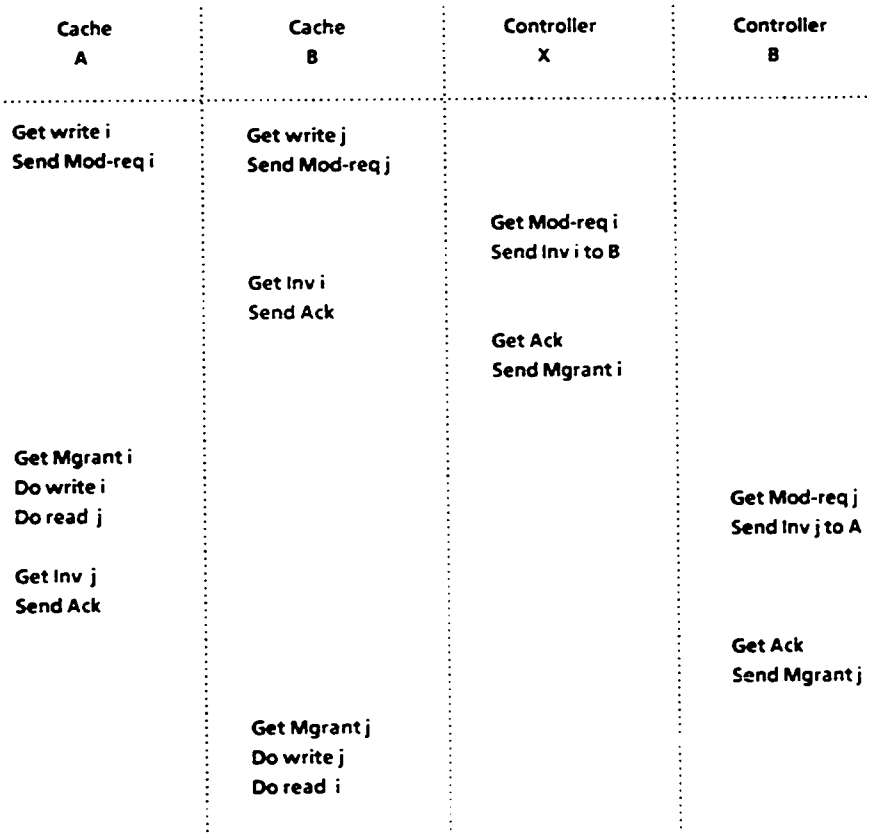


Figure 6.3: Acknowledgements result in sequential consistency

all writes to shared blocks in the same order that the bus was obtained. Similarly, the partial map protocols (as simulated on the crossbar switch) provide sequential consistency because the sending of a broadcast **Invalidation** signal requires the use of the entire switch. If two controllers attempt to send an **Invalidation** at the same time, one will succeed and the other must wait until the first has finished. Since all caches will observe the writes in the same order that they were sent, the system is sequentially consistent. In general, any protocol requiring the use of the entire switch to send signals will enforce a sequentiality of both sending and receiving the signals, resulting in sequential consistency.

It is also possible to obtain sequential consistency by adding a special bus to enforce a system wide ordering of invalidations. For example, assuming that a special bus connects all cache controllers, the protocols can be changed so that the controller no longer sends **Invalidation** signals. Instead, each time the protocol requires an invalidation, the controller sends this information to the cache receiving the write permission. Before the cache is allowed to complete the write, it must send an **Invalidation** signal on the bus. Access to the bus will determine the order in which the signals are observed by all caches. It would not be necessary to make the bus extremely fast or wide, since it would only be necessary to send the block address, and since a signal is only sent when required by the global state. The overall performance of the system would likely improve, since all invalidation traffic in the switch would be eliminated. Also, since all invalidations would be effectively broadcast to all caches anyway, there would be little value in maintaining the presence bits (indicating which caches have copies of the block) in the global state of the full map protocols. Adding snooping logic to the cache controllers would ensure that their performance was affected only if the invalidation involves a block in the local cache.

It is important to note that observing all writes in the same order at all caches is actually stronger than sequential consistency. In other words, it is possible to have sequential consistency without all caches observing all writes in the same order, as the following example illustrates. Assume that block i is present in caches A, C, and D, and that block j is present in caches B, C, and D. Assume also that **Invalidation** signals are individually sent and acknowledged before proceeding, but that the order of sending may vary. (Recall that controllers process requests in a FIFO order.) As shown in Figure 6.4, cache A receives a request from processor A to write block i, and at about the same time, cache B receives a request from processor B to write block j. Since the copies are shared, write permission must first be obtained from the memory controller, and the caches send **Modify-request** signals. The blocks are in different memory modules, so the requests are processed by different memory controllers. Assume that caches C and D receive the **Invalidation** signals for blocks i and j in different orders. The only problem that could arise

to prevent the memory references from being sequentializable is if, between the arrival of the two **Invalidation** signals, each cache is able to read the newly written value of the invalidated block (say i for cache C and j for cache D) and then read the value of the block to be invalidated (j for cache C and i for cache D). However, this cannot happen, since the read reference to the newly invalidated block will cause a **Read-miss** signal to be sent, which the memory controller cannot service until until the previous transaction is completed (including receiving all **Ack** signals and sending write permission). Therefore, the memory references are sequentially consistent, although the writes are observed in a different sequence,

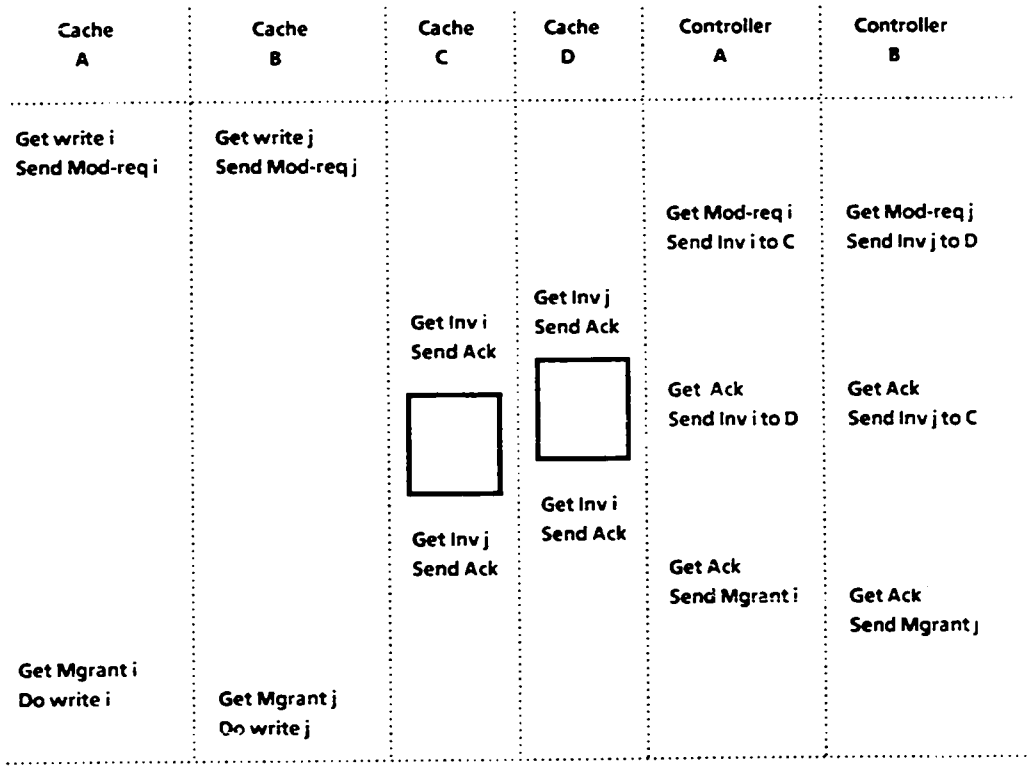
Sequential consistency is a stronger requirement than might normally be expected for the correct operation of a multiprocessor. The additional overhead to guarantee it may be avoided if the requirements are less strict, provided that the operation of the system remains reliable. One approach, described by Dubois and Briggs [DSF86], is to require all accesses to shared variables to be encapsulated within critical sections using special synchronizing variables for which sequential consistency must be enforced by the system. They provide the following definition (restated in the terms of this discussion):

Definition 4 *In a multiprocessor system, references are said to be weakly ordered if*

1. *references to global synchronizing variables are sequentially consistent, and if*
2. *no reference to a synchronizing variable is issued by any processor until all previous modifications to global data have been observed by all caches, and if*
3. *no reference to global data is issued by any processor until all previous modifications to synchronizing variables have been observed by all caches.*

In a system with weak ordering, an improvement in system performance (resulting from a relaxation of the requirements about sequential consistency for all references) is exchanged for the added complication of requiring the programmer to explicitly synchronize shared data references. However, the three stipulations in Definition 4 are not as easy to meet as might be hoped. In particular, meeting all conditions would seem to require exclusive access to the switch for all synchronizing variable references. Although weak ordering is not as strong a requirement as sequential consistency, it would appear that it is not significantly easier to implement, if at all.

We believe that acceptable multiprocessor operation can be obtained using something less than weak ordering. In particular, the definition of weak ordering restricts the accesses of independent synchronization variables by requiring that they be sequentially consistent. However, it is difficult to imagine a circumstance in which a race condition arises involving two or more synchronization variables. If such problems exist, they can easily be be resolved by introducing one



The boxes represent the interval where inconsistency appears to be possible with the following operations:

Cache C	Cache D
Observe write i	Observe write j
Read new i	Read new j
Read old j	Read old i
Observe write j	Observe write i

However, these sequences are impossible. Cache C cannot read the new value of i until cache D has observed the write to i. Similarly, cache D cannot read the new value of j until C has observed the write to j. In at least one of the caches, the initial read will be delayed until both writes have been observed, so the references can always be serialized.

Figure 6.4: Order of observing writes stronger than sequential consistency

or more new synchronizing variables and adding additional levels of synchronization. Therefore, any synchronization problem involving shared data accesses or synchronizing variable accesses can be programmed correctly so long as the underlying hardware correctly solves the race conditions arising in the execution of the synchronization primitives provided by the system. This support is provided by the cache coherence protocol, since it restricts write permission to a single cache at any time.

To illustrate the relationship between the cache coherence protocol and the use of synchronization primitives, it might be helpful to consider an example. Assume that GET-LOCK and RELEASE-LOCK operations are to be implemented using the processors instruction set, which contains a test-and-set (TAS) instruction. The RELEASE-LOCK operation is equivalent to a simple write, updating the value of the variable. The GET-LOCK operation requires the TAS, which is assumed to be implemented as a test-and-test-and-set (or TATAS, as described in Chapter 4) for performance reasons. Assume that GET-LOCK is to be implemented as a boolean function: it attempts to get the lock and returns success or failure depending on the outcome. The first step in the TATAS is a simple read to determine the value of the lock. If this test reveals that the lock is not available at this point, the function returns failure. If it succeeds, it proceeds to perform an atomic TAS in the local cache by requesting an exclusive copy (equivalent to requesting write permission) from the memory controller. (For those schemes with an UNMOD-EXC local state, this step is unnecessary if the block is in this state.) Once the local copy is guaranteed to be exclusive, the value of the lock is again tested. If this test succeeds, the lock is modified and the function returns true. If it fails, the function returns failure. Race conditions in modifying a lock are resolved by the cache coherence protocol, since it will grant write permission to one cache at a time.

Consider the following definition.

Definition 5 *A multiprocessor cache system is said to be strongly coherent if*

1. *the system of caches is coherent, and if*
2. *all caches observe each modification of each synchronizing variable only after all modifications to the global variables that it protects have been observed.*

It is claimed that strong coherence is sufficient for the correct operation of a multiprocessor system, assuming that all global data accesses are explicitly synchronized using locks. The first condition (coherence) guarantees that accesses to each individual lock will be well behaved (i.e., a single cache will be granted write permission), avoiding possible race conditions in attempting to enter critical sections. The second condition guarantees that all changes which take place within a critical section

will be observed by the time it is observed that the critical section has been exited. These two conditions seem to be the minimum required for the 'normal' operation of a multiprocessor system, yet they also appear to be sufficient. One possible approach to meeting the second condition is to require that each lock be stored in the same memory module as the shared variables that it protects. Since all required **Invalidation** signals will be sent by the same controller, and since the network preserves the order of messages from a given source to any given destination, all caches will observe all writes to the shared variables in the critical section before they observe the write indicating that the lock has been released.

In summary, there are several levels of consistency in a multiprocessor with private caches. At the lowest level is cache coherence, required for all higher levels of consistency, which states that references to any single memory location are well behaved. The next highest level is strong coherence, which states that the effects of a critical section are visible before the completion of the section becomes visible. Following strong coherence is weak ordering, which requires that all references to locks be serializable. The next level, sequential consistency, requires that all block references be serializable. The highest level is that of write ordering, in which all modifications appear to all caches in the same order. The higher levels are distinguished by low programming overhead and increased coherence protocol overhead. The lower levels increase programming overhead but require little, if any, changes to the coherence protocols.

Chapter 7

Protocol Extensions for Very Large Systems

The protocols presented in the previous chapters are feasible for a limited number of processors. In the shared bus protocols, the bus is likely to saturate with a fairly small number of processors. In the general network protocols, the system size is limited by the overhead of broadcasts and the size of the memory required to maintain the global states. With the possible exception of the software solutions mentioned in Chapter 2, none of the cache coherence protocols presented in the previous chapters are appropriate for a multiprocessor with hundreds or thousands of processors.

In this chapter, two versions of a protocol are presented for a multiprocessor with a two-level hierarchical interconnection network. The system organization consists of interconnected *clusters* of processors, as illustrated in Figure 7.1. Each cluster of processors shares a common bus and requires a cluster controller to perform the inter-cluster communication. A version of the protocol is described for both a bus and non-bus global inter-cluster interconnection network. This protocol is intended to serve as an example of the type of protocol that can be used in such a system, based on extensions to the protocols described in Chapters 4 and 5. Note that the cluster does not include local memory other than the private cache for each processor. Alternative cluster organizations are considered briefly at the conclusion of this chapter.

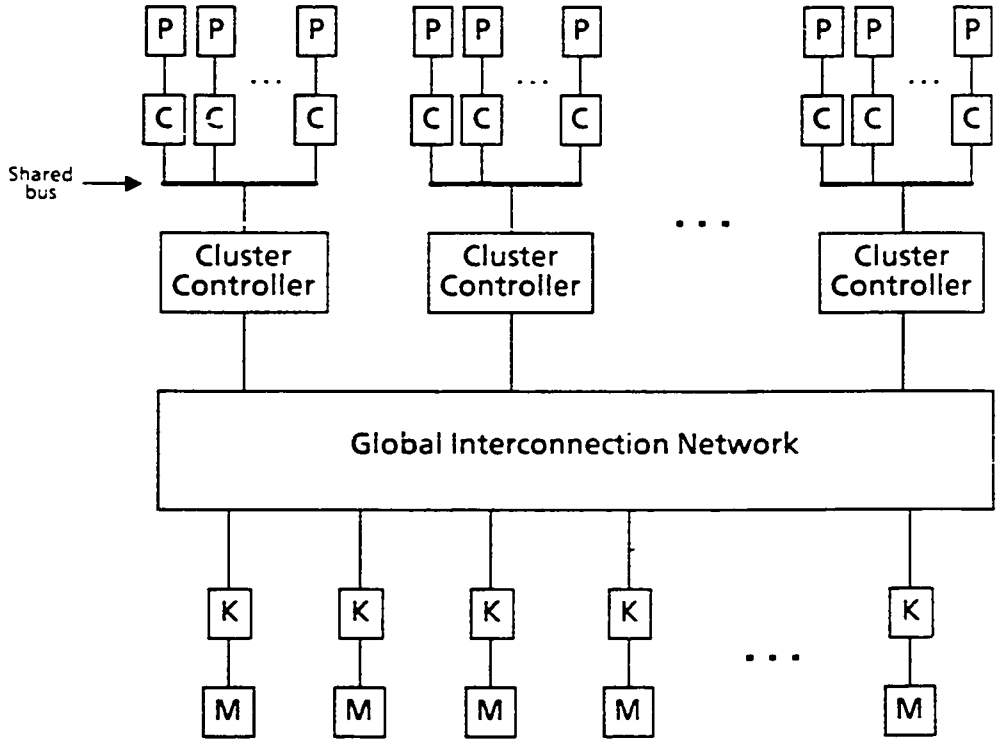


Figure 7.1: Two-level hierarchical multiprocessor

7.1 A Hierarchical Protocol with a Cluster Bus

When the local cluster network is a bus, it is possible to use an extension of the efficient snooping bus protocols to maintain coherence within the cluster. In particular, the use of an efficient distributed write approach within each cluster supports efficient sharing between caches in the same cluster; sharing between clusters is assumed to be rare, with the possible exception of read-only data and instructions. Each cluster is assumed to have a cluster controller on the data path connecting the local cluster bus to the global switch. While the design and function of the cluster controller depends on the organization of the global network, the operation of the protocol within each cluster can be described independently of that portion of the protocol operating in the global network. This *internal cluster protocol* is presented in the next section. Subsequent sections will present the remaining portion of the protocol for both shared bus and non-bus global interconnection networks.

7.1.1 The Internal Cluster Protocol

The following local cache states are used:

1. INVALID. (INV) Accesses to this block must be treated as cache misses.
2. UNMODIFIED-EXCLUSIVE. (UNMOD-EXC) This is the only cached copy in the entire system and it is not modified.
3. MODIFIED-EXCLUSIVE. (MOD-EXC) This is the only cached copy in the entire system and it is modified.
4. UNMODIFIED-SHARED-READ-ONLY. (UNMOD-SHD-R) Other copies of this block may exist in this and other clusters.
5. UNMODIFIED-SHARED-WRITABLE. (UNMOD-SHD-W) This copy is not modified, and no other cluster has a copy, but other caches within the cluster may have copies.
6. MODIFIED-SHARED. (MOD-SHD) The block was last modified in this cache, and no other cluster has a copy of the block, but other copies may exist within the local cluster.

A cluster is said to own a block if no other clusters have valid copies. Cluster ownership is indicated by blocks in state UNMOD-EXC, MOD-EXC, UNMOD-SHD-W, and MOD-SHD. If a block is shared between clusters, it is not owned by any cluster and all cached copies must be UNMOD-SHD-R.

The cluster protocol includes the following set of signals that are used on the cluster bus:

1. **Read-miss**. Signal sent by a cache to indicate read miss.
2. **Write-miss**. Signal sent by a cache to indicate write miss.
3. **Read-data**. Cluster controller response to read miss–includes valid block contents.
4. **Write-data**. Cluster controller response to write miss–includes valid block contents.
5. **Dist-write**. A distributed write to all caches in the cluster.
6. **Write-back**. Sends the contents of the block to update main memory.
7. **Exclusive-modified**. Informs the cluster controller that an UNMOD-EXC copy is being modified.
8. **Replace-unmodified**. Informs the cluster controller that an unmodified block has been replaced in the cache.
9. **Modify-request**. Requests cluster ownership so that a write can proceed.
10. **Modify-granted**. Indicates that cluster ownership has been obtained.
11. **Invalidation**. Commands all caches to invalidate their block copies.
12. **Write-back-inv**. Requests cache with modified copy to write back modified copy. All caches in the cluster are to invalidate the block.
13. **Write-back-shd**. Requests cache with modified copy to write back modified copy. All caches in the cluster are to change their local states to indicate that cluster ownership has been relinquished.

The cluster protocol requires two special bus lines: *SHARED*, and *CLUSTER-OWNERSHIP*, abbreviated *C-O* in the following discussion. (A *MODIFIED* bus line would also be required if clean-ownership is to be implemented.) The use of these lines will be outlined in the following discussion. Figure 7.2 contains a state transition diagram for the cluster portion of the protocol. (The remaining part, including the operation of the cluster controller will be described in subsequent sections). The cluster protocol works as follows:

- **Read hit**. The cache supplies the data to the processor. No other action is required.
- **Read miss**. The cache sends a **Read-miss** signal on the cluster bus. If another cache in the cluster has a copy of the block, it will supply it. (This may be implemented using clean

ownership, or using the techniques of the Illinois or Firefly protocols.) Obtaining the block from within the cluster whenever possible is very important in a hierarchical multiprocessor where the delay in accessing main memory is increased. If any cache in the cluster has the block in an exclusive state, it will supply the data and change its local state to a (intra-cluster) shared state (UNMOD-EXC to UNMOD-SHD-W, MOD-EXC to MOD-SHD). No other state changes are required. Any cache in the cluster that has the block raises the *SHARED* line. If the cluster owns the block (indicated by valid copies in states other than UNMOD-SHD-R), the caches also raise the *C-O* line. If no other caches in the cluster have a copy of the block, the cluster controller will respond to the request and supply a valid copy of the block in a **Read-data** signal. (How the controller accomplishes this will be described in a subsequent section.) As the data is supplied, the controller raises the *SHARED* line if another cluster has a valid copy of the block. If no copy exists in another cluster, the controller will raise the *C-O* line. The cache determines the state of its block from these lines. If *SHARED* and *C-O* are both high, the block is loaded in state UNMOD-SHD-W. If only *SHARED* is raised, the block is loaded in state UNMOD-SHD-R. If only *C-O* is raised, the block is loaded in state UNMOD-EXC. (It is not possible to have both lines remain low.)

- **Write hit.** Blocks in MOD-EXC may be modified without any additional overhead. Blocks in UNMOD-EXC may be modified after an **Exclusive-modified** signal is sent on the bus, accompanied by a state change to MOD-EXC. If the block is shared within the cluster and owned by the cluster (UNMOD-SHD-W or MOD-SHD), the cache performs a distributed write by obtaining the bus and sending a **Dist-write**, supplying the new contents to all other caches in the cluster. Caches with valid copies will take the new data, set their local states to UNMOD-SHD-W, and raise the *SHARED* line indicating that the block is still shared. The cache performing the write changes its local state to MOD-EXC if *SHARED* remains low throughout the distributed write. Otherwise the state is set to MOD-SHD. If the block is in state UNMOD-SHD-R, the write may proceed only after write permission is obtained. The cache sends a **Modify-request** to the cluster controller which responds (after some delay) with a **Modify-granted** indicating that the cluster has obtained ownership. All caches in the cluster observe the **Modify-granted** and change their local copies to UNMOD-SHD-W. The cache waiting to write then performs a distributed write as described above.

- **Write miss.** The cache sends a **Write-miss** on the bus. Any other caches in the cluster with a copy will respond by supplying the block and raising *SHARED* and *C-O* (unless UNMOD-SHD-R) as with a **Read-miss**. If a copy of the block is supplied by another cache but the

cluster does not have ownership for the block, the cache proceeds exactly as in the case of write hit on UNMOD-SHD-R described above. If no cache in the cluster has a valid copy, the request will be serviced by the cluster controller, which responds with a **Write-data**. It is not necessary to raise the *C-O* line if the block is supplied by the controller, since the the controller always obtains cluster ownership on a write miss. The local cache loads the block in state MOD-SHD and performs a distributed write if *SHARED* is high (other caches in cluster have copies). If *SHARED* is not raised, the block is loaded in state MOD-EXC and the write may proceed local to the cache.

- **Replacement**. When a cache replaces a blocks in state MOD-EXC or MOD-SHD, it must send a **Write-back** on the bus, which will be serviced by the cluster controller. Other caches with a valid copy of the block must raise the *SHARED* line, if any other copies exist. The block is then sent to the global switch by the cluster controller. The replacement of blocks in any other state requires the sending of a **Replace-unmodified** signal on the cluster bus. Other caches with valid copies will raise the *SHARED* line.

- **Response to controller signals**. The cluster controller may send any of the following three signals on the bus: **Write-back-inv**, **Write-back-shd**, and **Invalidation**. In the case of the first two signals, the cache with a MOD-EXC or MOD-SHD copy will respond by sending a **Write-back** on the bus. All cluster caches with copies will respond to either a **Write-back-inv** or an **Invalidation** by setting their local state to INV. All caches with copies will observe a **Write-back-shd** by setting the local state to UNMOD-SHD-R. Caches waiting for the cluster bus when one of these signals arrives must effectively restart the request currently being serviced. For example, if the cache is waiting for the bus to send a **Dist-write** on an UNMOD-SHD-W block and it observes a **Write-back-inv** for that block on the bus, it must first set the block's state to INV and then continue as if a write miss had occurred by sending a **Write-miss**.

The cluster protocol requires the **Exclusive-modified** and **Replace-unmodified** signal that were not needed in the snooping bus protocols. The sole purpose of these signals is to keep cache state information consistent with state information maintained by the cluster controller. (The function of the controller and the cluster state table is described in the next section.)

The efficiency of the system is increased if the cluster bus is released while waiting for a response from the cluster controller (in servicing a **Read-miss**, **Write-miss**, or **Modify-request**). This complicates the cluster bus protocol since bus transactions within the cluster will use circuit switching, while those involving the cluster controller will be packet switched. In the case of a

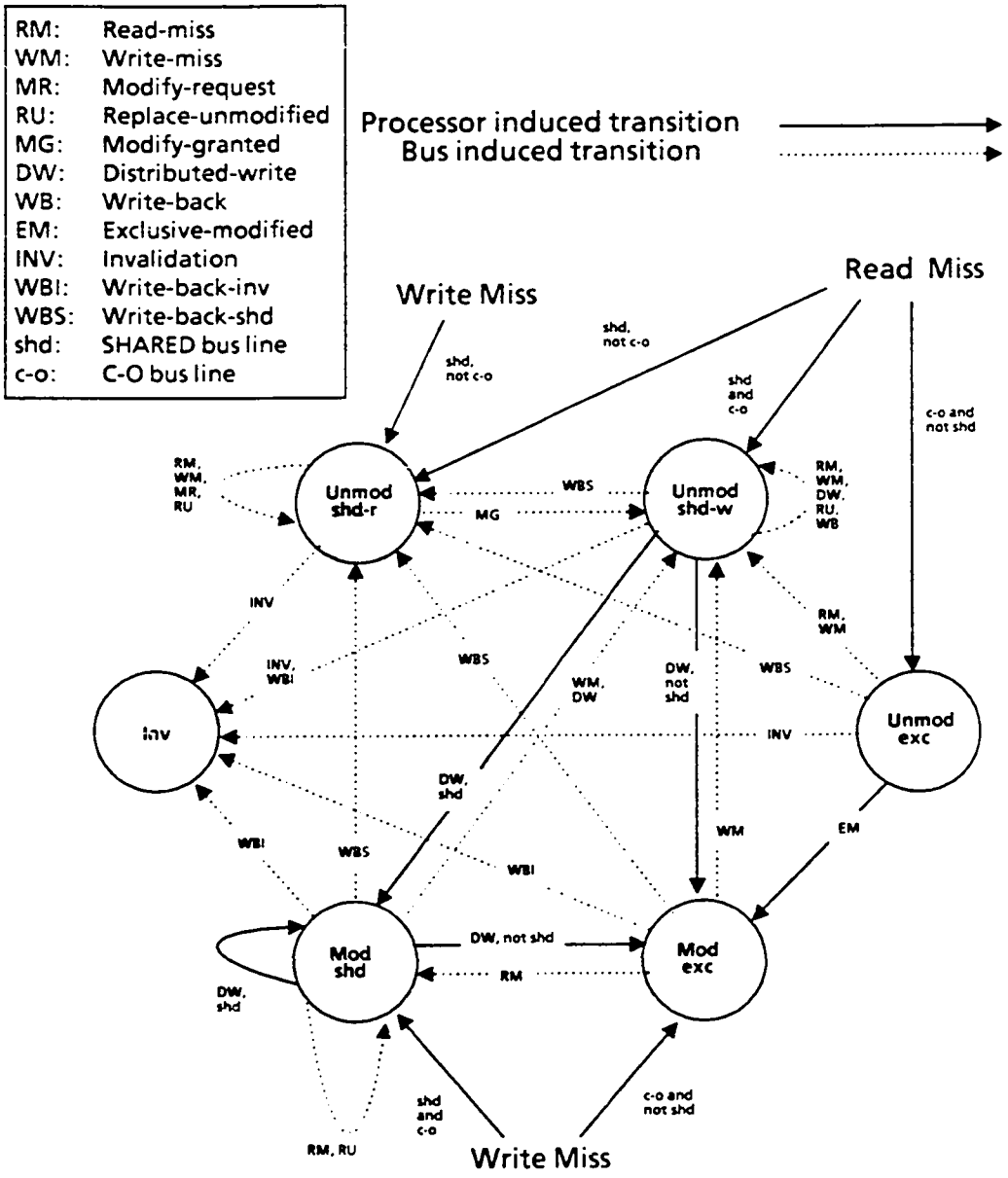


Figure 7.2: Cache state transition diagram

miss, the cache sending the signal will not know in advance the supplier of the block. If supplied by another cache in the cluster, the bus will be held for the entire transaction. However, if the miss is to be serviced by the cluster controller, the bus is held only until the cluster controller indicates that is is responding.

This approach leads to complications if the cluster controller receives requests for a block that currently has a transaction pending in the global network. Such requests may originate within the cluster or they may arrive via the global switch. Consider the case of external requests. If a **Read-miss** or **Write-miss** signal is pending, no copy of the block exists in the cluster and all external signals may be ignored by the cluster controller. If the pending signal is a **Modify-request**, it is possible for the cluster controller to receive a signal requiring invalidation of the cluster copies while the **Modify-request** is pending. As will be explained in the next section describing the functions of the cluster controller, it is able to respond with a **Write-data** signal if an invalidation occurred, and with a **Modify-granted** under normal conditions.

Signals arriving from within the cluster are more problematic. With any of the three types of pending requests, it is possible for another cache in the cluster to send a **Read-miss** or **Write-miss** signal. If the pending signal is a **Modify-request**, it is also possible to see **Modify-request** and **Replace-unmodified** signals from another cache. While it would be possible to allow certain combinations of requests to continue, the easiest way to avoid problems is for the cluster controller to abort cluster bus transactions involving blocks for which a global transaction is pending. This assumes that such an abort facility exists, and that aborted transactions are re-attempted after some delay, allowing the transactions of other caches (involving other blocks) to proceed in the meantime.

7.1.2 The Global Protocol Using a Shared Bus

If the global interconnection network consists of a shared bus, it is possible to implement the global portion of the protocol entirely at the cluster controller level, just as the shared bus protocols are implemented entirely at the cache controller level. The cluster controller serves as a gateway between the cluster and the global switch. Without an intelligent gateway, all cluster bus signals would also have to be sent on the global bus, and all global signals sent on the local bus, effectively reducing the switch to a single shared bus. To prevent unnecessary traffic between the two levels, each cluster controller must maintain a table of the state of all blocks cached in the cluster. One way in which the table could be organized is as a hash table (similar to the TLB in the IBM System 138 or the matching units in a dataflow system).

The global protocol is based on each cluster controller observing global bus transactions, matching against the local cluster contents, and taking actions as necessary to maintain the consistency of those blocks it has copies of. The speed of attempted matches is important in those cases when the main memory must be inhibited from responding (when a modified copy is present in another cluster). To avoid these problems, it is assumed that a single *owner bit* is associated with each block in main memory, indicating whether or not main memory is to respond to requests for this block. This relaxes the constraint that the cluster controller respond within a specified time interval and allows consideration of space efficient organizations of the cluster state table.

The general approach taken in this protocol is to make sharing within each cluster efficient and to assume that sharing between clusters is rare. Unless modified, blocks will come from main memory. (There would appear to be little, if any, improvement in speedup of loading a block from another cluster; in fact, it might be slower than main memory if the cluster bus of the supplying cluster is busy.) The concept of ownership in this protocol differs somewhat from the ownership concepts of other protocols. In this protocol, ownership is synonymous with exclusivity. This allows the local cluster to write owned blocks at any time without any global interaction.

The states associated with each block in the cluster table are prefixed with a 'C' to indicate that they are cluster states. If no state entry exists for a block, then it is not present in the cluster. To simplify the terminology, the state of such a block will be referred to as C-ABSENT. Implicitly, this means that no global table entry for the block exists in the cluster. Therefore, the following cluster states are possible:

1. C-ABSENT. The block is not present in the cluster.
2. C-OWNED-UNMODIFIED. (C-OWN-UNMOD) The cluster has ownership for the block and it is not modified (with respect to main memory).
3. C-OWNED-MODIFIED. (C-OWN-UNMOD) The cluster has ownership for the block and it is modified.
4. C-UNOWNED. The cluster has read-only copies of the block and does not have cluster ownership.

The set of signals used on the global bus is the following:

1. **G-Load**. Requests a block copy in response to a read miss.
2. **G-Exclusive-load**. Requests a block copy in response to a write miss.
3. **G-Ownership-claim**. Requests ownership (exclusive cluster copy) for the given block.

4. **G-Write-back**. Updates main memory but does not reset ownership bit.

5. **G-Replace-modified**. Updates main memory and resets ownership bit.

6. **G-Release-ownership**. Resets ownership bit.

The cluster controller must take actions on several types of cluster transactions. These actions are outlined in the following discussion. The operation of the global protocol requires the use of a special bus line to indicate sharing that will be referred to as *G-SHARED*. Figure 7.3 contains a state transition diagram for the cluster controller states.

- **Read-miss**. If no cache in the cluster responds to the miss (implying that the cluster state is C-ABSENT), the cluster controller indicates that it will service the miss. It then obtains the global bus and sends a **G-Load** signal for the appropriate block which will be serviced by main memory unless the owner bit is set. In this case, the owning cluster will respond with the contents of the block (or it will make it possible for memory to respond by overriding the owner bit if the copy in main memory is valid). When the block arrives at the controller, it releases the global bus and sends the data on the cluster bus in a **Read-data** signal. If *G-SHARED* was raised during the **G-Load** transaction, the controller sends the signal with *SHARED* high and *C-O* low, and it creates a cluster entry for the block in state C-UNOWNED. If *G-SHARED* was not raised, the block is sent with *SHARED* low and *C-O* high, and a C-OWN-UNMOD cluster state entry is created for the block. If *G-SHARED* was raised during the **G-Load**, the ownership bit associated with the block in main memory will be cleared. If *G-SHARED* was not raised, the requesting cluster becomes the owner and the ownership bit is set.

- **Write-miss**. If no cache in the cluster supplies the data, the cluster controller indicates that it is responding. The global bus is obtained and an **G-Exclusive-load** signal is sent for the block. If the owner bit for the block is not set, main memory will service the request, set the owner bit, and supply the data. If the block is owned by another cluster, that cluster controller will supply the data or override the owner bit so that memory may provide the block, as in the case of a **Read-miss**. (The actions of the other cluster controllers upon seeing a **G-Exclusive-load** on the global bus are described below.) When the data arrives at the controller, the global bus is released and the block is sent on the cluster bus using the **Write-data** signal. The cluster controller creates a C-OWN-MOD entry for the block in the cluster table. During the transaction on the global bus, the memory controller sets the owner bit for the block.

- **Modify-request**. The global bus is obtained and an **G-Ownership-claim** is sent. This signal has the effect of an invalidation signal on the global bus, and it sets the ownership bit for the block in main memory. Immediately after sending the signal, the cluster controller may send a **Modify-granted** on the cluster bus and change the state from C-UNOWNED to C-OWN-UNMOD; it will be set to C-OWN-MOD when the cache sends the **Dist-write**. If the controller observes an **G-Exclusive-load** or an **G-Ownership-claim** for the same block while it is waiting for access to the bus to sent the **G-Ownership-claim**, it must first respond to these signals as described below (including a deletion of the state entry for the block). It then proceeds as if servicing a **Write-miss** by sending a **G-Exclusive-load** request on the global bus and sending the data to the requesting cache via a **Write-data**, in addition to creating a C-OWN-MOD cluster table entry for the block. In this way, the data is transmitted (on the cluster bus) only when needed by the requesting cache.

- **Write-back**. If the *SHARED* line is raised, other copies of the block remain in the cluster. The cluster controller obtains the global bus and sends a **G-Write-back** to update main memory, and the cluster state is changed from C-OWN-MOD to C-OWN-UNMOD. The ownership bit in main memory is not cleared by the **G-Write-back**. If the *SHARED* line is low, no other copies of the block exist within the cluster and ownership may be relinquished. The cluster controller sends a **G-Replace-modified** which updates main memory and clears the ownership bit. In addition, the controller deletes the entry for the block in the cluster table.

- **Dist-write**. If the cluster state is C-OWN-UNMOD, it is changed to C-OWN-MOD. No other actions are required.

- **Exclusive-modified**. The cluster state is changed from C-OWN-UNMOD to C-OWN-MOD. No other actions are required.

- **Replace-unmodified**. If *SHARED* is raised, other copies of the block still exist in the cluster and no action is taken. (If the state is C-OWN-MOD *SHARED* will always be raised by the cache with the modified copy.) If *SHARED* is low and the cluster state is C-UNOWNED, the only action required is to delete the block entry from the cluster table. If *SHARED* is low and the cluster state is C-OWN-UNMOD, the controller sends a **G-Release-ownership** signal on the global bus which clears the ownership bit associated with the block in main memory.

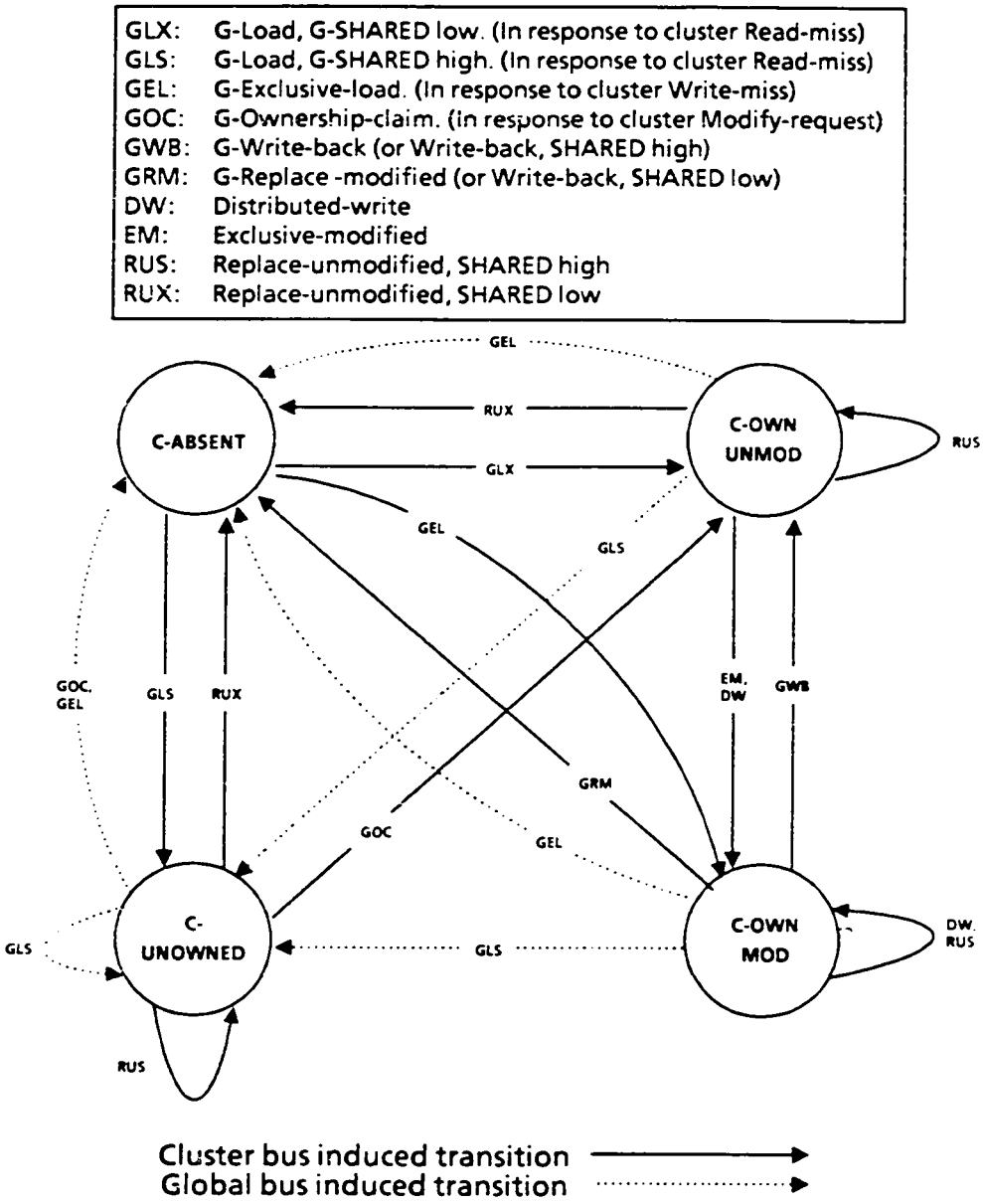


Figure 7.3: Cluster state transitions using a global bus

It is essential for the correct operation of the protocol that the cluster state information be consistent with the local cache information. This is the sole purpose of the **Exclusive-modified** and **Replace-unmodified** cluster signals. If no **Exclusive-modified** signal is sent before the modification proceeds, the cluster state might be C-OWN-UNMOD although a modified copy exists in one of its caches. Without the **Replace-unmodified** signal (and the associated raising of the *SHARED* line on the cluster bus), the cluster table entries for C-OWN-UNMOD and C-UNOWNED blocks would never be deleted until another cluster requested ownership. This could require maintaining an enormous number of block entries in the cluster table (limited only by the number of blocks in main memory!), most of which would be associated with blocks long since replaced in all caches in the cluster.

The cluster controller must also respond to signals observed on the global bus. Actions may be required on three different types of signals sent by other caches. No actions are required by a cluster controller if the block is not present in the cluster.

- **G-Load**. If the cluster state is C-OWN-MOD, the cluster controller sends a **Write-back-shd** signal on the cluster bus. (For performance reasons, the cluster controller should have priority in all bus operations so that it never waits longer than the completion of the current transaction.) The cache with the modified copy will supply the valid contents of the block and the cluster controller will place the block contents on the global bus. All caches in the cluster with copies will see the **Write-back-shd** signal and change their local states to UNMOD-SHD-R. In addition, the cluster controller sets the cluster state to C-UNOWNED and raises *G-SHARED*. If the cluster state is C-OWN-UNMOD, the cluster controller also sends a **Write-back-shd** on the cluster bus. No write back occurs, since no modified copy exists, but all caches with copies will change their local states to UNMOD-SHD-R. Since the block copy in main memory is up-to-date, it is possible to load the block from memory, assuming that the global bus has a signal to override the ownership bit so that memory can respond. The cluster controller changes the cluster state to C-UNOWNED and raises *G-SHARED*. If the local cluster state is already C-UNOWNED, no action is required of the cluster controller except raising *G-SHARED*. Note that the final state of all cached copies is UNMOD-SHD-R and the final state of the block in the cluster tables is C-UNOWNED. *G-SHARED* will be raised if at least one cluster has a copy of the block.

- **G-Exclusive-load**. If the local cluster state is C-UNOWNED, the controller sends an **Invalidation** signal for the block on the cluster bus and deletes the entry for the block in the cluster table. If the cluster state is C-OWN-MOD, the controller sends a **Write-back-**

inv signal on the cluster bus, causing the cache with the modified copy to supply the data and also causing the invalidation of all cached copies. When the valid block contents arrive from the cache, they are sent on the global bus and the entry for the block is deleted from the cluster table. If the cluster state is C-OWN-UNMOD, the controller sends an **Invalidation** signal on the cluster bus, signals an override of the memory ownership bit so the block may be loaded from main memory, and deletes the block entry from the cluster table. After observing an **G-Exclusive-load**, no other clusters in the system will have valid copies of the block.

- **G-Ownership-claim**. This signal is possible only if all copies in the system are UNMODSHD-R and if all cluster states are C-UNOWNED. Each cluster controller with a copy of the block sends an **Invalidation** signal for the block on its cluster bus and deletes the block entry from the cluster table.

7.1.3 The Global Protocol Using a General Interconnect

If the global interconnection network is not a shared bus, the global portion of the protocol must be similar to the protocols discussed in Chapter 5. In particular, it becomes necessary to maintain global state information for each block in main memory. While it would be possible to implement the protocol using a full map protocol, the following discussion assumes a partial map implementation for reasons of economy and expandability, although the number of clusters is unlikely to grow very large. This requires a broadcast mechanism in the global switch so that signals can be sent to all clusters. In order to filter out broadcast signals that do not pertain to blocks cached in the local cluster, it is useful to maintain the same cluster state information that was required in the global shared bus version of the protocol. In the cluster table, there exists a state entry for every block that is cached in the cluster. Using this information, the cluster controller allows only those signals involving blocks in the local cluster to affect the traffic in the cluster.

The states associated with each block in the cluster table are identical to the states in the previous section, namely:

1. C-ABSENT. The block is not present in the cluster.
2. C-OWNED-UNMODIFIED. (C-OWN-UNMOD) The cluster has ownership for the block and it is not modified (with respect to main memory).
3. C-OWNED-MODIFIED. (C-OWN-MOD) The cluster has ownership for the block and it is modified.

4. C-UNOWNED. The cluster has read-only copies of the block and does not have cluster ownership.

Associated with each block in main memory is one of the following states:

1. G-ABSENT. The block is not present in any cluster.

2. G-OWNED. The block is owned by exactly one cluster.

3. G-UNOWNED. The block is not owned and may be present in more than one cluster.

The global portion of the protocol combines the PRESENT1 and PRESENTM states of the partial map protocols into a single G-OWNED state. Within the cluster, a distinction is maintained between modified ownership and unmodified ownership, but no such distinction exists at the global state level. This combination of states allows the modification of private unmodified blocks without any additional global actions (although actions within the cluster are required). The handling of writable shared data is efficient within each cluster, at the expense of increased overhead of sharing between clusters, which is assumed to be very rare.

The signals that are sent from the cluster controllers to the memory controller in the global interconnection network are the following:

1. **G-Load**. Requests a block copy to service a read miss.

2. **G-Exclusive-load**. Requests a block copy with cluster ownership.

3. **G-Write-back**. Updates main memory without necessarily relinquishing ownership.

4. **G-Replace-modified**. Updates main memory and voluntarily relinquishes ownership.

5. **G-Release-ownership**. Voluntarily relinquishes ownership.

6. **G-Ack**. Acknowledges that owner cluster received broadcast signal.

The set of global network signals from the memory controller to the cluster controllers is the following:

1. **G-Data**. Supplies a block copy without cluster ownership.

2. **G-Data+ownership**. Supplies a block copy with cluster ownership.

3. **G-Invalidation**. Invalidates all copies of the block.

4. **G-Write-back-inv**. Requires write-back (if modified) and invalidation of all cluster copies.

5. **G-Write-back-shd**. Requires write-back (if modified) and return of cluster ownership.

The cluster controller must respond to signals from the cluster bus and signals from the global switch. The required actions for cluster transactions are described in the text below. Figure 7.4 contains a state transition diagram for the cluster states for both cluster and bus signals.

- **Read-miss**. If no caches in the cluster respond to the miss, the cluster controller indicates that it will service the miss. It then submits a **G-Load** request for the appropriate block into the global switch. After some delay, it receives either a **G-Data** packet or a **G-Data+ownership** packet. If the controller receives a **G-Data+ownership** signal, it creates a cluster table entry in state C-OWN-UNMOD and sends the block to the requesting cache with a **Read-data** signal with the *C-O* line raised to indicate cluster ownership. If the controller receives a **G-Data** signal, it creates a cluster table entry in state C-UNOWNED and sends the **Read-data** with *SHARED* raised.

- **Write-miss**. If no cache in the cluster supplies the data, the cluster controller indicates that it is responding and sends an **G-Exclusive-load** signal into the global switch. After some delay it will receive a **G-Data+ownership** packet, following which it creates a cluster table entry for the block in state C-OWN-MOD and sends the data in a **Write-data** signal on the cluster bus.

- **Modify-request**. A **G-Exclusive-load** is sent to the appropriate memory controller. In response, the cluster controller will always receive a **G-data+ownership**. It is necessary to include the data since the cluster copies may be invalidated while waiting for the response. If no **G-Invalidation** signal has been received when the block arrives, the controller responds with a **Modify-granted** on the cluster bus, setting the state to C-OWN-UNMOD and discarding the data. (It will be changed to C-OWN-MOD when the cache writes the block and sends a **Dist-write**.) If a **G-Invalidation** signal was received before the data arrives, the controller responds exactly as if a write miss had occurred by sending the data in a **Write-data** signal on the cluster bus. In this case, a C-OWN-MOD state table entry is created. (The previous state table entry was deleted upon the arrival of the **G-Invalidation** signal.)

- **Write-back**. If *SHARED* is raised (and other copies remain in the cluster), the cluster controller sends a **G-Write-back** to main memory and the cluster state is changed from C-OWN-MOD to C-OWN-UNMOD. If *SHARED* is not raised, the controller sends a **G-Replace-modified** signal and deletes the entry for the block in the cluster table.

- **Dist-write.** If the cluster state is C-OWN-UNMOD, it is changed to C-OWN-MOD. No other actions are required.

- **Exclusive-modified.** The cluster state is changed from C-OWN-UNMOD to C-OWN-MOD. No other actions are required.

- **Replace-unmodified.** If *SHARED* is raised on the cluster bus, other cached copies of the block remain in the cluster and no cluster controller action is required. If *SHARED* is not raised and the cluster state is C-UNOWNED, the controller need only delete the block entry in the cluster table. If *SHARED* is not raised and the cluster state is C-OWN-UNMOD, the controller sends a **G-Release-ownership** signal to the appropriate memory controller, which changes the global state. In addition, the cluster state entry for the block is deleted. If the cluster state is C-OWN-MOD, *SHARED* will always be raised by the cache with the modified copy, so no action is ever required.

The cluster controller must also respond to broadcast signals it receives from the global network. (Signals from the global switch that are not broadcast are always responses to cluster controller requests.) The actions it must take upon receiving the different types of broadcast signals are described as follows:

- **G-Invalidation.** The controller attempts to match the block address included in the command with the blocks in the cluster state table. If no entry exists, the signal is ignored. If an entry exists, it must be C-UNOWNED. In this case, the controller sends an **Invalidation** signal on the cluster bus and deletes the block entry in the cluster table.

- **G-Write-back-shd.** If no cluster table entry exists for the block, the signal is ignored. If an entry exists, it must be either C-OWN-MOD or C-OWN-UNMOD. If it is C-OWN-MOD, the controller sends a **Write-back-shd** on the cluster bus, causing the cache with the modified copy to send a **Write-back**, and also changing the state of all blocks in the cluster to UNMOD-SHD-R. When the controller receives the **Write-back**, it returns the valid block contents to the memory controller in a **G-Write-back** signal and changes the cluster state to C-UNOWNED. If the cluster state is C-OWN-UNMOD, it sends a **Write-back-shd** on the cluster bus. There is no modified copy in the cluster, so no cache will respond with a **Write-back**, but all caches with copies will change their states to UNMOD-SHD-R. Immediately after sending the **Write-back-shd** on the cluster bus, the controller responds with a **G-Ack** to the memory controller, indicating that the block may be safely loaded from main memory. In addition, the cluster state is changed to C-UNOWNED.

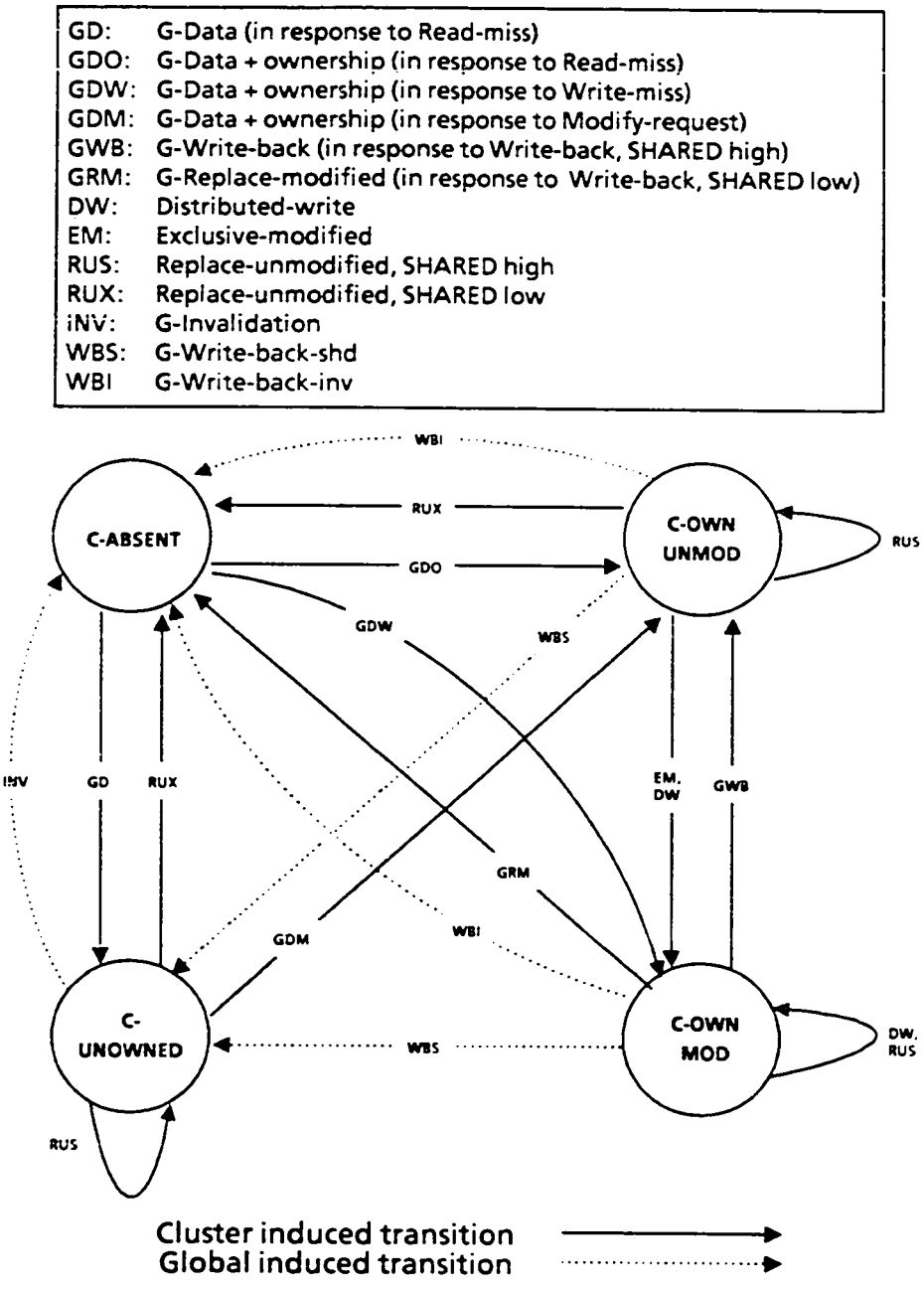


Figure 7.4: Cluster state transitions for global non-bus

- **G-Write-back-inv**. If no cluster table entry for the block exists, the command is ignored. If an entry exists, the only possible cluster states are C-OWN-UNMOD and C-OWN-MOD. If the state is C-OWN-MOD, the controller must send a **Write-back-inv** on the cluster bus, forcing the cache with the modified copy to send a **Write-back** and also causing the invalidation of all copies in the cluster. When the **Write-back** is received at the controller, it returns the block to the memory controller via a **G-Write-back** signal and deletes the cluster table entry for the block. If the cluster state is C-OWN-UNMOD, the controller sends an **Invalidation** signal on the cluster bus, deletes the cluster table entry for the block, and sends a **G-Ack** to the memory controller, signalling that the command was received and that the block may be loaded from main memory, which is up-to-date.

The memory controllers service requests from the cluster controllers. The actions it must take for each type of request are outlined below. A global state transition diagram is given in Figure 7.5.

- **G-Load**. The controller first determines the global state. If the global state is G-ABSENT, the block will be read from memory and sent in a **G-Data+ownership** packet and the global state will be changed to G-OWNED. If the global state is G-UNOWNED, other copies of the block may exist, and the requesting cluster will be sent the data in a **G-Data** packet (which does not include ownership). If the global state is G-OWNED, another cluster has ownership for the block, and the memory controller broadcasts a **G-Write-back-shd** signal to cause the cluster to relinquish ownership. In this case, there are four possible responses that the memory controller may observe, since the cluster copy may be C-OWN-MOD or C-OWN-UNMOD, and since the block may have been voluntarily replaced in either cluster state before the signal arrives.

  If the cluster copy was C-OWN-MOD, the memory controller will receive a **G-Write-back**. The memory is updated, the global state is sent to G-UNOWNED, and the block is sent to the requesting cluster using the **G-Data** signal. If the cluster copy had been C-OWN-MOD but the last copy had been replaced voluntarily before the **G-Write-back-shd** signal arrived, the memory controller will receive a **G-Replace-modified**. In this case, the memory is updated, the global state is left G-OWNED, and the block is sent in a **G-Data+ownership**. If the cluster copy was C-OWN-UNMOD, the controller will receive a **G-Ack**, indicating that that cluster ownership has been relinquished and that the memory copy is up-to-date. The controller reads the block from memory, sends it to the requesting cluster in a **G-Data** packet, and changes the global state to G-UNOWNED. If the cluster copy had been C-OWN-

UNMOD but the last copy was replaced before the **G-Write-back-shd** signal arrived, the controller will receive a **G-Release-ownership** signal. In this case, the controller reads the block from memory, sends it to the requesting cluster in a **G-Data+ownership** packet, and leaves the global state G-OWNED.

- **G-Exclusive-load**. The memory controller first checks the global state. If it is G-ABSENT, the block is sent directly from memory in a **G-Data+ownership** packet, and the global state is changed to G-OWNED. If the global state is G-UNOWNED, the controller broadcasts a **G-Invalidation** for the block, changes the global state to G-OWNED, and sends the block in a **G-Data+ownership** packet. If the global state is G-OWNED, the controller must broadcast a **G-Write-back-inv** to force the owning cluster to relinquish ownership. There are four possible responses that the memory controller may observe.

  If the cluster state of the owning cluster was C-OWN-MOD, the memory controller will receive a **G-Write-back**. If the cluster state had been C-OWN-MOD, but the last copy was voluntarily replaced before the arrival of the **G-Write-back-inv** at the cluster, the memory controller will receive a **G-Replace-modified**. If the cluster state was C-OWN-UNMOD, the memory controller will receive an **G-Ack** and load the block from memory. If the cluster state had been C-OWN-UNMOD, but the last copy in the cluster was replaced before the arrival of the **G-Write-back-inv** signal, the memory controller receives a **G-Release-ownership** and the block is loaded from memory. In neither write-back case is it necessary to update main memory, and in all cases the global state is G-OWNED upon completion. The block is always transmitted in a **G-Data+ownership** packet to the requesting cluster controller.

- **G-Write-back**. The memory controller takes the block contents and updates main memory. The global state is not changed, and no further action is required. (This assumes that the controller is not waiting for the block to be written back in order to process another request.)

- **G-Replace-modified**. The memory controller updates the block copy in main memory and changes the global state to G-ABSENT. No other action is required.

- **G-Release-ownership**. The memory controller changes the global state to G-ABSENT. No other action is required.

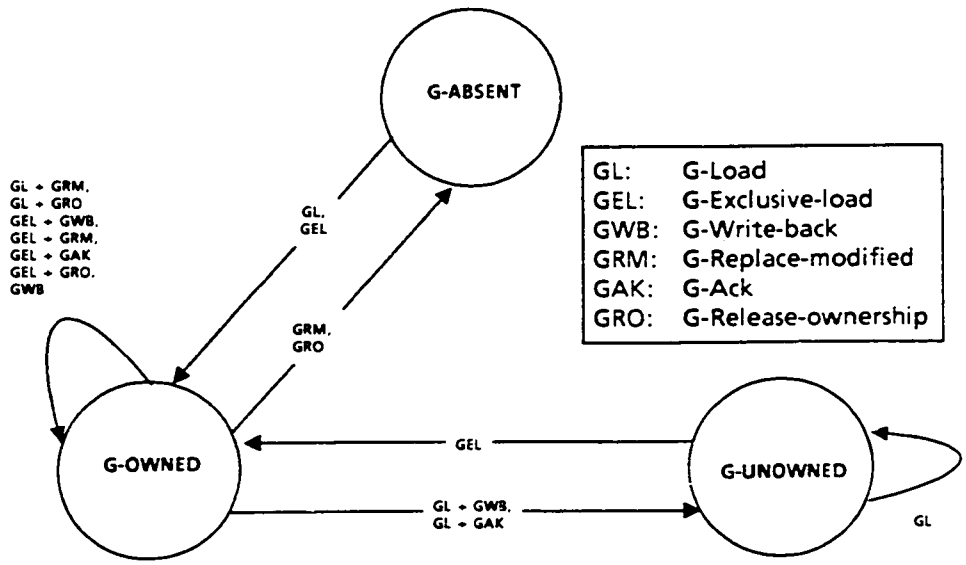


Figure 7.5: Global state transition diagram for global non-bus

7.2 Topics for Further Research

The overall performance of a system using the protocol presented in the previous section would depend on several factors, such as the tradeoff between the traffic on the cluster bus and the traffic in the global switch. The protocol relies on intelligence at the cluster controller level to minimize the effects of one on the other. Less intelligence at the controller level would reduce the implementation cost, but it would also reduce the efficiency if the overall traffic is increased. An accurate performance analysis of this (and other) protocols is an important area for future research. Such an analysis would require an accurate workload model reflecting inter- and intra-cluster shared reference behavior–much more complex than the workload model described in Chapter 3. In the ideal case, such a study would be general enough to allow comparison of hardware-based protocols with software approaches. The results of such a study would almost certainly lead to the development of more efficient protocols.

The protocol of the previous section is based on a cluster design using a shared bus. Hierarchical designs are also possible using non-bus organizations for the local cluster, as illustrated in Figure 7.6. Since the cluster has but one connection to the global network, the only advantage of a cluster switch connecting the n processors to m different paths would be if each of the m paths leads to a distinct memory element. These memories could be either shared local memories (accessible

only by caches within the cluster with addresses distinct from those in the global memory shared between clusters), or they might be shared caches, introducing a new level to the memory hierarchy. In the latter case, the problem of *multi-level cache coherence* is introduced. There has been comparatively little research on the subject of multi-level caches, and there exist no published coherence protocols for multi-level cache organizations. Maintaining consistency among all levels of the memory hierarchy becomes even more complex if the block size (or transfer size between levels) is allowed to vary, as might be desired for performance reasons. Multi-level caches are currently an important research topic.

Another important and related topic is the design of protocols for multiprocessors with additional levels in the hierarchical interconnection network. Discussions in this chapter have considered only a two-level hierarchy. A straightforward extension of the two-level protocols would require that complete caching information be maintained on each global data path for all clusters (and groups of clusters) that can be accessed by that path. For example, Figure 7.7 illustrates the organization of a three level hierarchy, in which clusters are connected together in *groups*. An adaptation of the two-level protocol would require the group controllers to maintain a state information about all blocks cached in the group. While this might be feasible for a two-level hierarchy, it becomes very impractical with additional levels. It appears likely that a protocol for a system with three or more levels would require a different approach.

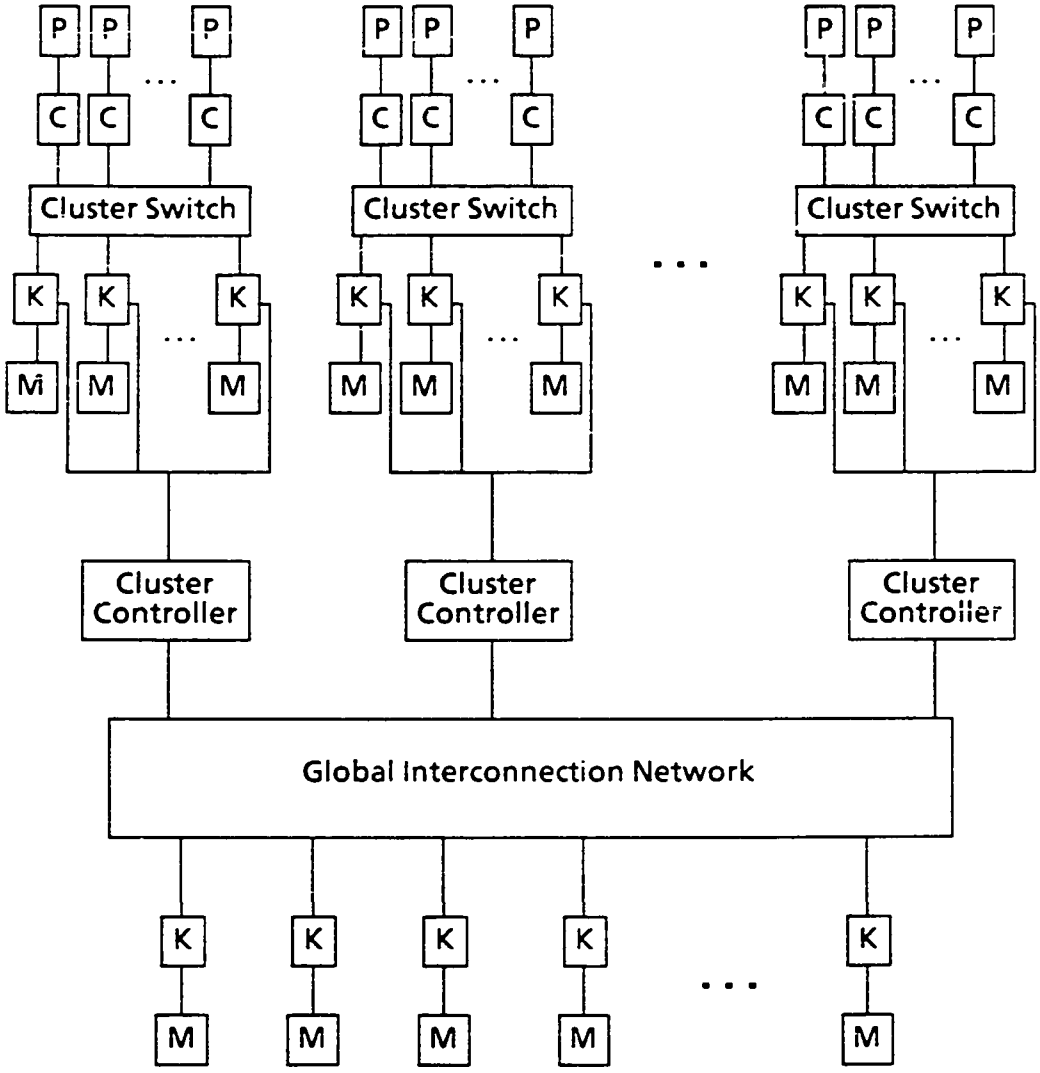


Figure 7.6: Non-bus cluster organization

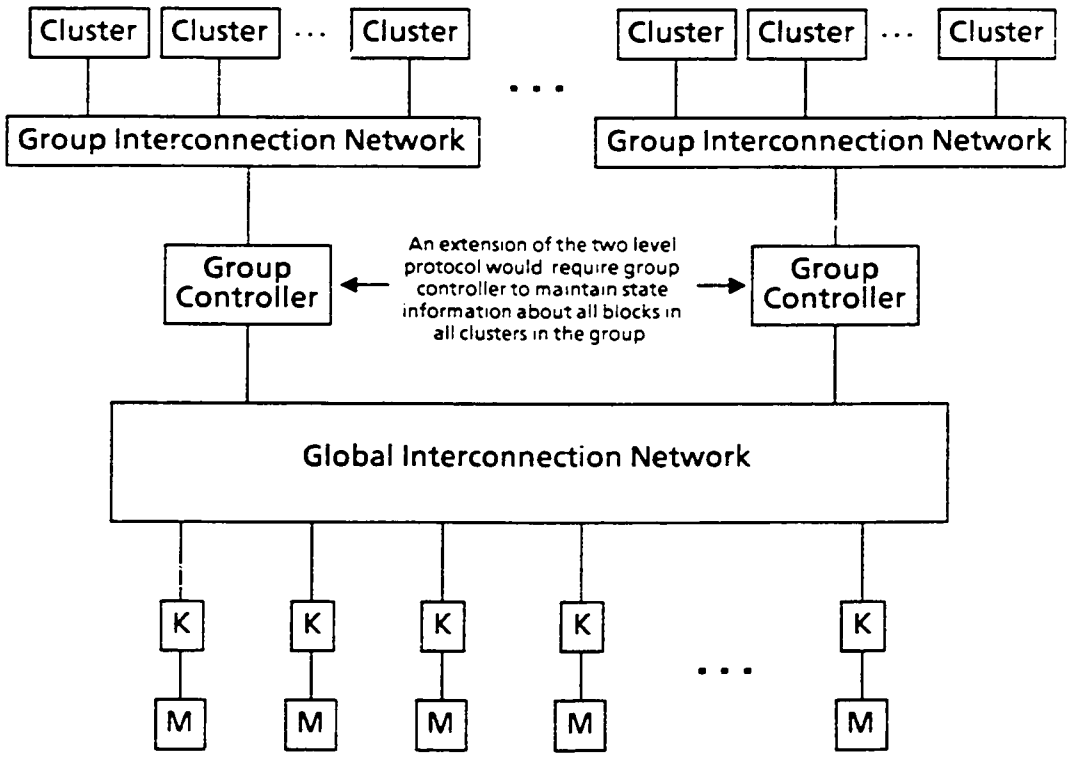


Figure 7.7: Three level hierarchy

Chapter 8

Correctness Proofs

Few proofs of correctness have been published for cache coherence protocols. While this may not be a serious problem for shared bus protocols because of their relatively simple synchronous operation, it is very important for general network protocols. In particular, many of the timing problems associated with the full map protocols in Chapter 5 would certainly have been discovered had the authors completed such a proof. Such proofs are considered essential for the partial map family of protocols, particularly in light of the fact that the original twobit scheme was incorrect as originally published.

Each of the alternative partial map protocols presented in Chapter 5 has been proven correct. A proof of correctness for the threestate protocol using invariants derived from Petri nets has been published [CGS86]. Correctness proofs have also been constructed for the threebit and extended twobit protocols.

The first section in this chapter contains a proof of correctness for the EIP and EDWP shared bus protocols. Following this section, a very different proof of correctness is given for the extended twobit protocol. The threebit protocol has been prover correct in a similar fashion, but the proof is not included. As can be seen, reasoning about shared bus protocols is much less complex than reasoning about general network protocols.

8.1 Shared Bus Protocols

Throughout the following, *up-to-date* means 'identical to the most recently written copy', and *dirty-owner* refers to a cache with a copy in state MOD-EXC or MOD-SHD. EXCLUSIVE states

are MOD-EXC and UNMOD-EXC, while the remaining valid states are SHARED.

It is assumed that the protocols are implemented as presented in Chapter 4. More specifically:

1. On all misses, the dirty owner always supplies the block and inhibits main memory from responding.
2. All caches with valid copies raise the *SHARED* line as specified by the protocol–on read miss for EIP, and on all misses and distributed-writes in EDWP.
3. In EIP the invalidation signal causes the invalidation of all cached copies in the system, with the exception of the cache sending the signal.
4. In EDWP the distributed-write signal either updates or invalidates all other cached copies in the system.

On the basis of these assumptions and the protocol description the following observations can be made:

1. **Each copy of a block present in more than one cache is always in a SHARED state.** From assumption 2 and the EIP protocol description, all caches determine sharing on read misses by the value of the *SHARED* line in the bus. In the EDWP the *SHARED* line is used to detect sharing on all misses and also to determine when sharing ceases on all distributed writes. Since all caches with copies participate in the raising of the *SHARED* line and change their state to the appropriate SHARED state, and since the cache loading the block (or performing the write) tests the same line, it is impossible for blocks to be present in more than one cache in any state other than a SHARED state.
2. **If there is no dirty owner, the contents of the block in main memory are up-to-date.** If the block has never been written this is clearly true. On each write, the cache performing the write becomes the dirty owner. The only way there can cease to be a dirty owner is to have the dirty owner write the block back, updating main memory.
3. **All valid cached copies are identical and up-to-date.** All writes modify the local cache copy, making that cache the dirty owner. In addition, if the block is in a SHARED state (which by observation 1 it must be if other copies exist) either an invalidation signal is sent (EIP) or a distributed-write is sent (EDWP). In either case after the write is completed, all copies are identical and up-to-date, since by assumptions 3 and 4 all other existing copies are updated, making them identical to the dirty owner's copy, or they are invalidated, eliminating all copies but the dirty owner's.

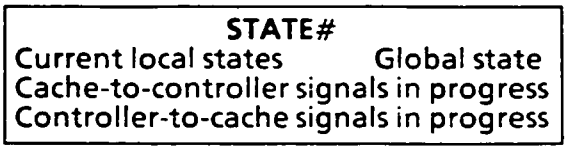


Figure 8.1: State description

THEOREM: On any processor read in EIP or EDWP, the value returned by the cache will be the most recently written value.

PROOF: If a valid copy of the block is already present in the cache, it must be up-to-date by observation 3. If no valid copy of the block is present, the resulting cache miss will cause a copy of the block to be loaded from another cache or memory. If the block is loaded from another cache, that copy must be up-to-date by observation 3. If the block is loaded from main memory, by assumption 1 there can be no dirty owner, so by observation 2 the copy in memory is up-to-date. Therefore the value read by the processor is always obtained from an up-to-date copy.

8.2 Extended Twobit Protocol

Because of the asynchronous nature of the protocol, the proof technique selected is an expansion of reachable states for a particular block (see Figure 8.2). Beginning from the state in which no caches have a copy of the block (corresponding to the system state when it is inititialized or powered on), all possible state transitions are considered. All possible 'legal' system states (i.e., reachable by a sequence of actions in the protocol) are included in the figure. All possible transitions are considered and none leads to an 'illegal' state–one in which the information is inconsistent. In addition, the state transition graph is *maximally strongly connected*, indicating that all states can be reached an infinite number of times.

This approach to proving the protocol correct has the advantage that it also completely specifies the protocol. Such a specification would be invaluable should an implementation ever be attempted.

Figure 8.1 describes the representation of each system state, which includes the local and global states for the block, in addition to the signals that have been sent but not yet received. State names are based on the page number (A, B, or C) and are numbered from left to right down the page for ease of reference.

The local states are described in a set of the form {(cache#: state)...}. Each state is one of:

- **A**: (not present or INVALID)
- **R**: (UNMODIFIED-SHARED)
- **X**: (UNMODIFIED-EXCLUSIVE)
- **W**: (MODIFIED-EXCLUSIVE)

To simplify the description, only states other than A will be shown. For example, {} means that no caches have a valid copy.

The global state is one of:

- **ABS**: ABSENT
- **P1**: PRESENT1
- **P***: PRESENT*
- **PM**: PRESENTM
- **P1R**: PRESENT1, waiting for a response so a **Read-miss** can be serviced.
- **P1W**: PRESENT1, waiting for a response so a **Write-miss** can be serviced.
- **PMR**: PRESENTM, waiting for a response so a **Read-miss** can be serviced.
- **PMW**: PRESENTM, waiting for a response so a **Write-miss** can be serviced.

The last four states have been added for convenience. In these states, the controller is restricted in the type of signals that it can service, since it can only process signals from the cache with the exclusive copy.

The set of signals from the cache to the controller has an entry of the form (cache#: signal1, signal2, ...) for every cache with an outstanding request. The ordering of signals from each cache must be strictly maintained. With an indefinite number of caches, it is possible to have an arbitrary number of incoming **Read-miss** and **Write-miss** signals at any instant; to simplify the representation, these are not explicitly represented. In each state, the set is assumed to have any number of these signals, although they are not listed. The signals are abbreviated as follows:

- **RM**: **Read-miss**
- **WM**: **Write-miss**
- **RU**: **Replace-unmodified**

- **WB**: **Replace-modified** (or write-back)
- **UP**: **Update**
- **XM**: **Exclusive-modified**
- **ACK**: Acknowledgement from cache with exclusive copy

The set of signals from the memory controller to the caches has an entry of the form (cache#: signal1, signal2, ...) for each cache that has not yet received a signal sent by the controller. As illustrated, there may be multiple signals in the set but their order must always be preserved. The signals are abbreviated as:

- **SD**: **Read-data-shd** (or shared data)
- **XD**: **Read-data-exc** (or exclusive data)
- **WD**: **Write-data**
- **WBS**: **Write-back-shd**
- **WBI**: **Write-back-inv**
- **INV**: **Invalidation**
- **SS**: **Set-shared**

In the state expansion graph, the arcs from state to state represent atomic actions that can be taken by the caches or the memory controllers. Each arc is labelled by the action that it represents. There are exactly three types of actions that are possible: actions initiated by a cache, actions taken by a cache in response to a memory controller signal, and memory controller actions taken in response to a cache controller signal. The following notation will be used:

- **C(action)**. Represents actions initiated by a cache. Since the incoming set of cache requests to the controller always includes an arbitrary number of **Read-miss** and **Write-miss** signals, the generation of read and write miss signals will not be explicitly considered. This leaves two possible actions that can cause a local state change: servicing a write to a previously unmodified block, and replacing a block in the cache. The first possibility will be written as C(WH) and is only possible when the local state is R or X. Replacement is always possible when a cache has a valid copy, and it will be written C(REP).
- **R(signal)**. Represents a cache receiving the specified signal and completing the prescribed actions. The identity of the cache can be determined from the context.

- **K(cache#:signal)**. Represents the memory controller's processing of the specified cache controller's request. If it is clear from the context which cache sent the request, the cache# will be omitted. This action is always completed in its entirety unless the signal is a miss and another cache has an exclusive copy, in which case the miss cannot be serviced until a response is received from the cache with the exclusive copy. While waiting for a response, the global state will be in one of the four added wait states. It is important to note that this arc does not necessarily represent the *arrival* of the signal at the controller; it is instead the processing of the signal, which could have arrived earlier and been waiting in a local buffer if the controller was busy.

Beginning from an initial state (A1) where the block is in state ABSENT and present in no caches, the entire state space is constructed by considering all possible actions of each of the above three types from each state. To limit the number of states, each is kept independent of the identity of the cache with the copy. For example, state B1 represents all possible system configurations in which a cache has an UNMOD-EXC copy of the block with no other signals outstanding (besides the implicit **Read-miss** and **Write-miss** signals from other caches).

The following additional explanations may be helpful in understanding the transitions.

1. Signals sent to a cache that has replaced the block may be ignored. Hence, in states like A7, the **Set-shared** signal may be deleted from the set of outgoing controller signals.

2. The controller may not service miss signals when the block is in one of the four global wait states.

3. The replacement of a block has the expected results. For an R copy, only the local state is changed. For an X copy, the cache sends a **Replace-unmodified** signal. For a W copy, a **Replace-modified** is required.

4. Write hits on previously unmodified blocks also have the expected results. If the block is in state R a **Write-miss** signal is sent. If the block is in state X the state is changed to W and an **Exclusive-modified** signal is sent.

5. If the global state is P1R or P1W and the controller receives an **Exclusive-modified** signal, the controller sets the state to PMR or PMW respectively and no other action is taken.

6. Transitions such as from state A8 to C7 are allowed because the **Set-shared** signal will be treated as a **Write-back-shd** signal when the cache has a W copy. Transitions such as

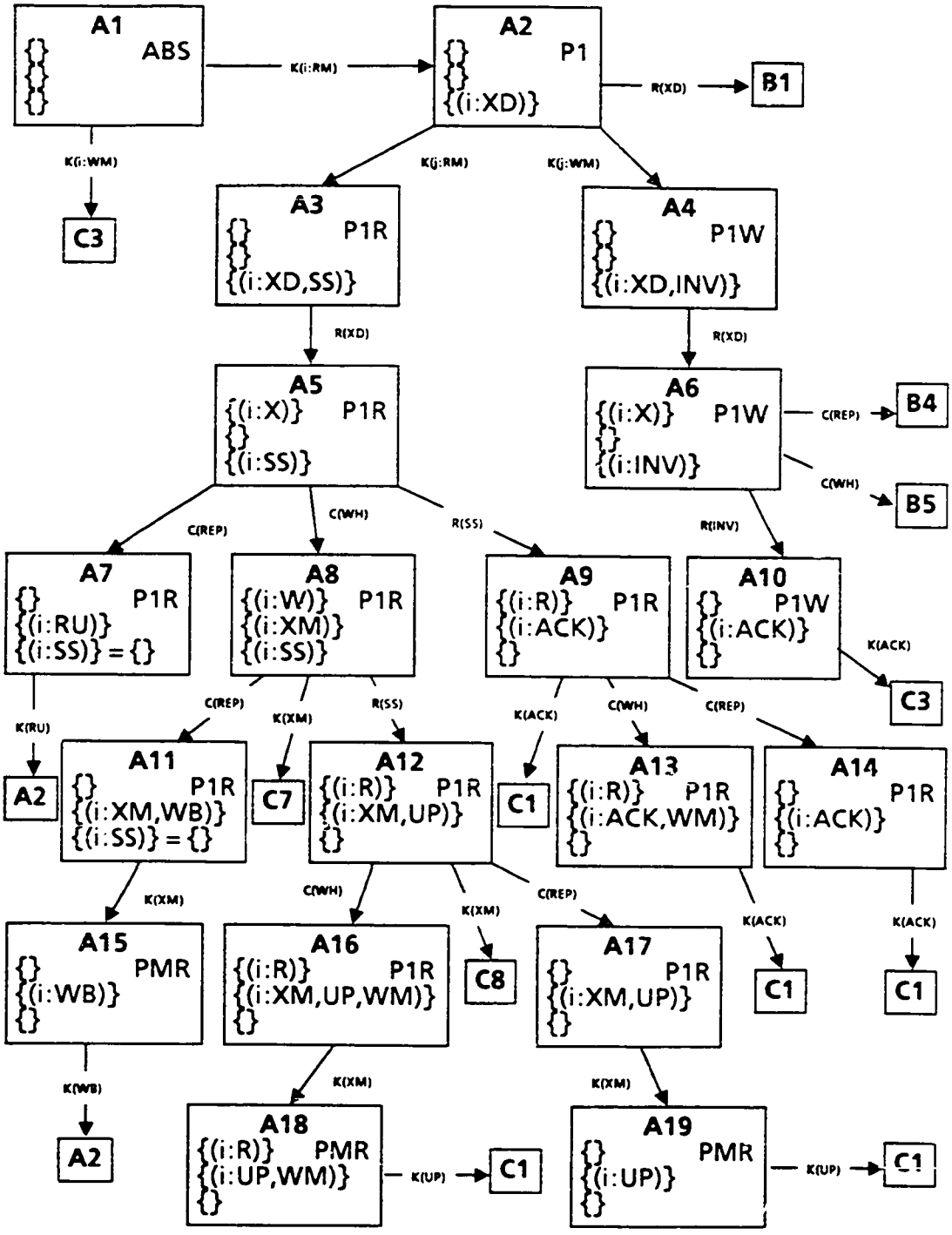


Figure 8.2: Extended twobit protocol proof of correctness, section A

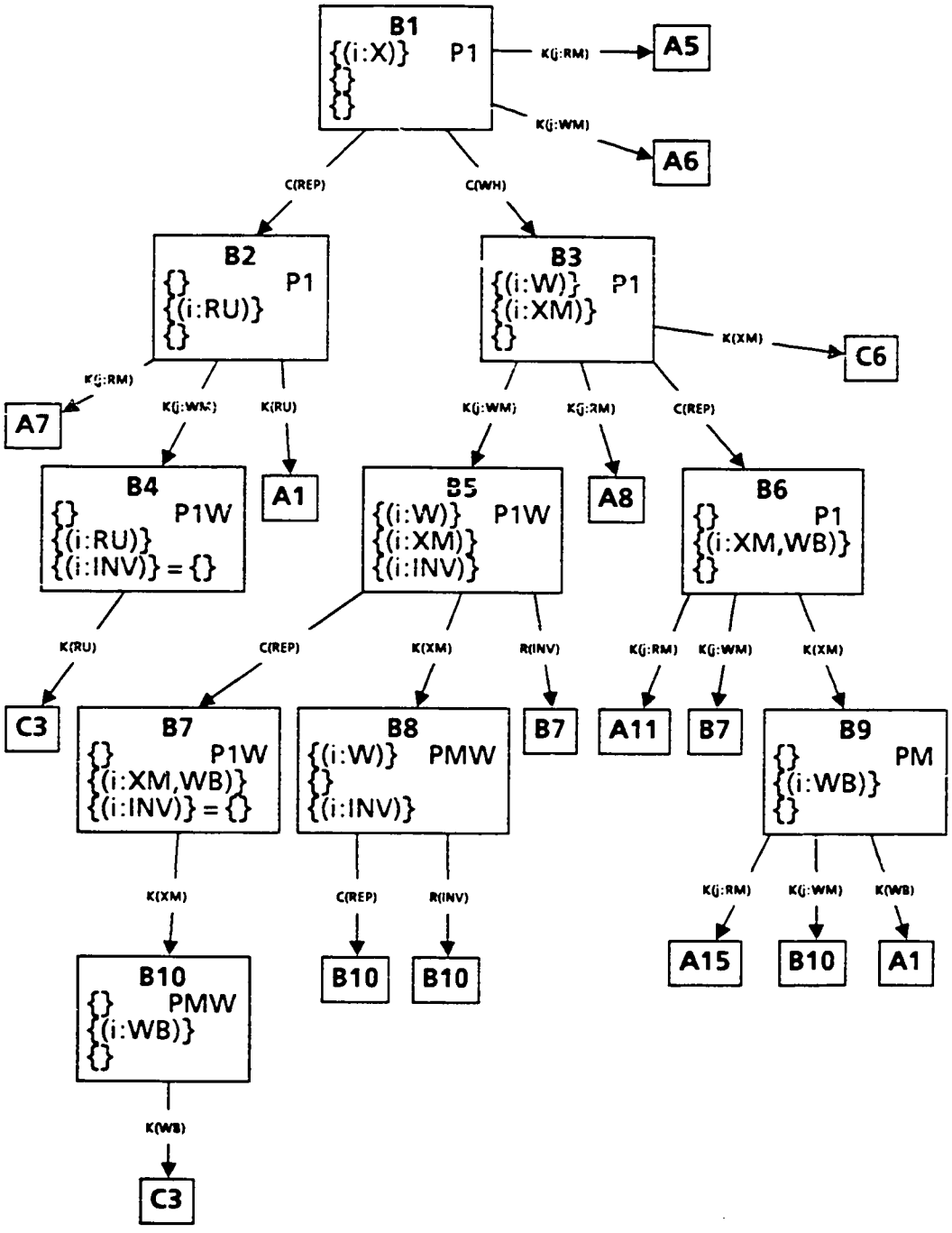


Figure 8.3: Extended twobit protocol proof of correctness, section B

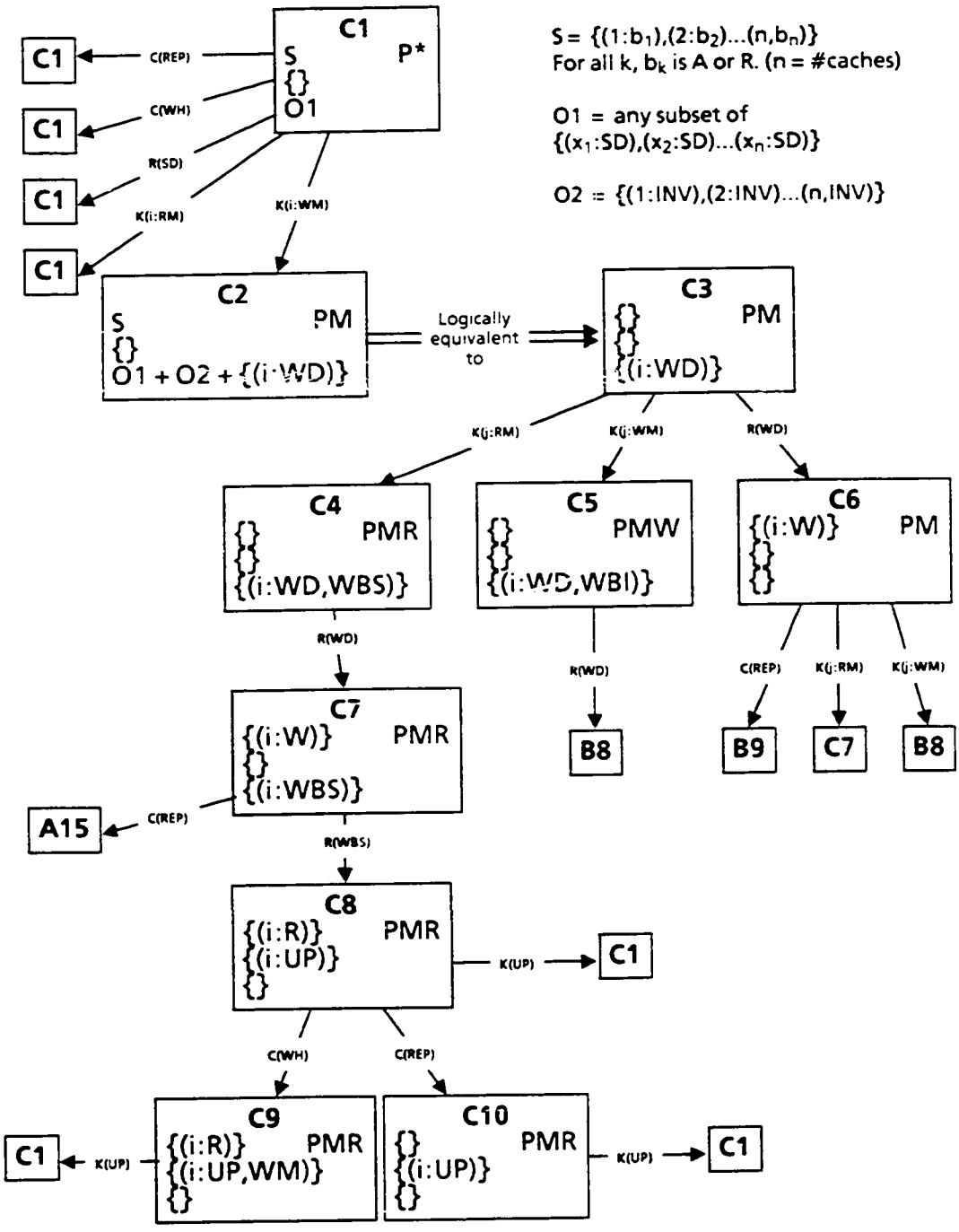


Figure 8.4: Extended twobit protocol proof of correctness, section C

C5 to B8 are allowed because the **Invalidation** and **Write-back-inv** signals are treated identically when the cache has a W copy.

7. In the transition from B9 to B10, a **Write-back-inv** signal is sent, but can be ignored, since the cache no longer has a copy of the block.

State C1 represents all possible configurations with a P* global state. As indicated in the figure, any number of caches may have copies of the block in state R (with the remainder in state A) and there may be any number of **Read-data-shd** packets on their way to caches. The only path out of C1 is for the controller to service a **Write-miss**, in which case an **Invalidation** signal is sent to all caches. As indicated by the special arc from C2 to C3, the system state can be simplified by treating the **Invalidation** signals at each cache. This is permissible because there are only two actions affecting the state that a cache with an R copy can take before the signal arrives. These are: write hit and replacement of the block. However, a write hit on an R block is treated as a write miss, and therefore already considered in the controller's incoming queue, and the replacement of an R block merely changes the local state to A.

It can be observed that there are no dead ends in the graph, implying that it is possible to change from any state to any other state, given the necessary sequence of actions. At each state, all possible atomic actions (with respect to the protocol) are considered. Since the initial state is the state expected when the system is powered on, the operation of the protocol is entirely within the indicated state graph, and the protocol is therefore correct.

Chapter 9

Summary and Conclusions

This dissertation has examined the cache coherence problem in shared-memory multiprocessors, an important problem in the structure of a memory hierarchy for these machines. The use of caches is essential in reducing the effective memory access time in shared bus systems, and is one of the most promising alternatives in the case of non-bus interconnected systems. The presence of private caches introduces a problem of consistency between the multiple copies that must be dealt with. The spectrum of possible solutions to this problem for the two types of architectures mentioned above has been examined, including software and hardware approaches. The relative merits of each approach were considered, and it was concluded that the most promising of the schemes are those implemented entirely in hardware based on a protocol between the cache controllers and (optionally) the memory controllers. However, a comprehensive analysis of the relative performance and cost-effectiveness of software approaches and hardware protocols is beyond the scope of this dissertation. The focus of this study has been to evaluate the performance and (to a lesser extent) implementation costs associated with previously proposed cache coherence protocols in shared bus systems, and to make improvements wherever possible, as well as to design, evaluate, and prove correct hardware-efficient protocols for interconnected systems.

In order to evaluate the performance of the protocols, a realistic simulation model was developed. The most critical element in the simulation was the workload model, which was based on one previously proposed, but with significant extensions. The simulation method chosen was trace-driven simulation, although a synthetic trace was used in the absence of true multiprocessor traces. The reference stream of each processor was modelled as the merging of two reference

streams–one representing blocks local to a process, and the other representing blocks referenced by other processors. By combining the effects of the two types of references, the workload model provides sufficient detail to illuminate the differences in the performance of the cache coherence protocols. Each protocol for the two general types of systems was simulated in detail, including all the necessary communication between processors, caches, and memories.

Cache coherence protocols were shown to divide naturally into two different categories, namely snooping protocols and non-snooping protocols. The snooping protocols assume a shared bus organization, while the other protocols may work for any interconnection network. The class of snooping protocols was further divided into invalidation protocols and distributed-write protocols. In each of these subclasses, all previously proposed solutions were described in a uniform fashion. By themselves, these descriptions are an important contribution, since they provide the basis necessary for a meaningful comparison of the schemes.

Within each of the invalidation and distributed-write subclasses, a new protocol was presented incorporating new features and new combinations of the features found in other protocols. Simulation results were presented that show that the performance of these new protocols surpasses that of the existing schemes, albeit with increased implementation cost. The simulation results were discussed in detail, identifying those features that hurt or help the performance of each protocol. In addition, simulation results were presented comparing the relative performance of the invalidation and distributed-write protocols. These results provide strong evidence of the benefit of updating instead of invalidating other copies when the level of sharing is moderate to high. The correctness of each of the new protocols was demonstrated with a proof.

In the class of non-snooping protocols, the previously proposed solutions to the problem were presented and pathological cases leading to incorrect protocols (as published) were discussed. A new class of solutions was presented using more space efficient bit maps. Performance figures for the protocols were presented based on simulation results for a multiprocessor with a crossbar switch. Unlike the shared-bus results which were ranked according to traffic generated, the non-snooping protocols were ranked according to demands on the memory controller. With the bandwidth of the crossbar switch and a reasonable number of memory modules, the difference between the performance of the various schemes is quite small, particularly with little sharing. The extension of the simulation results for a more cost effective type of interconnection was cited as an important topic for future research. With the exception of the original twobit protocol (which is not correct without additional restrictions), each of the partial map protocols presented have been proven correct. A proof of correctness for one of the schemes has been included in Chapter 8. The technique used to prove the protocol correct is noteworthy because it also provides a complete

specification.

The nature of the consistency that coherence protocols provide has been precisely defined, and its relationship to other types of consistency has been examined. Synchronous shared bus protocols were shown to provide naturally a stronger level of consistency than the asynchronous general interconnection protocols. While many levels of consistency can be defined, the ordering of writes required by coherence was shown to be mandatory for all other levels of consistency.

Another important contribution was the presentation of the first cache coherence protocol for a hierarchical system composed of clusters connected to global memory via an interconnection network. The proposed protocol was developed to determine the type of extensions that are necessary for a system of this type. We show that it is possible to use an efficient distributed-write intra-cluster protocol to promote sharing within a cluster, while at the same time using a global invalidation protocol. The development and analysis of schemes of this type should be considered further. In particular, the relative merits of approaches using software support need to be studied.

Although nearly all the effort was focused on protocols relying only on the hardware, mention was also made of improvements that could result from software assistance, specifically in the form of hints about whether the block is shared or not. For both the snooping and non-snooping categories, protocols were outlined that could make use of software supplied information to improve efficiency. The development and evaluation of other schemes of this type is also an important area for additional research.

Another important topic for further study, related to the simulation modelling of the protocols, is the validation of the parameters used in the simulation. This is likely to take place only when multiprocessor traces become available. It would also be useful if the simulation model were extended to allow the comparison of hardware and software approaches. Although simple software solutions were simulated, they are very different from the software solutions presented by other authors that were described in Chapter 2.

Bibliography

[AJ84] J. Archibald and J.-L.Baer. An economical solution to the cache coherence problem. In *Proc. of 11th Int. Symp. on Computer Architecture*, pages 355–362, IEEE, 1984.

[BA86] P. Bitar and A.Despain. Multiprocessor cache synchronization. In *Proc. of 13th Int. Symp. on Computer Architecture*, pages 424–433, IEEE, 1986.

[Bae80] J.-L. Baer. *Computer Systems Architecture.* Computer Science Press, Rockville,Md., 1980.

[Bit85] P. Bitar. *Fast Synchronization for Shared-Memory Multiprocessors.* Technical Report TR 85.11, RIACS, NASA Ames Research Center, 1985.

[BLPS] B.M. Bean, K. Langston, R. Partridge, and K.-B. Sy. Bias filter memory for filtering out unnecessary interrogations of cache directories in a multiprocessor system. United States Patent 4,142,234, February 17, 1979.

[CGS86] C. Chatelain, C. Girault, and S.Haddad. Specification and properties of a cache coherence protocol model. In *7th European Workshop on Application and Theory of Petri Nets*, 1986.

[Col85] W.W. Collier. Reasoning about parallel architectures. 1985. Submitted to Journal of the ACM.

[CP78] L.M. Censier and P.Feautrier. A new solution to coherence problems in multicache systems. *IEEE TC*, C-27(12):1112–1118, December 1978.

[DF82] M. Dubois and F.Briggs. Effects of cache coherency in multiprocessors. *IEEE TC*, C-31(11):1083–1099, November 1982.

[DSF86] M. Dubois, C. Scheurich, and F.Briggs. Memory access buffering in multiprocessors. In *Proc. of 13th Int. Symp. on Computer Architecture*, pages 434–442, IEEE, 1986.

[Fen81] T.Y. Feng. A survey of interconnection networks. *Computer*, 14(12):12–30, December 1981.

[Fra84] S.J. Frank. Tightly coupled multiprocessor systems speeds memory access times. *Electronics*, 57(1):164–169, January 1984.

[Goo83] J.R. Goodman. Using cache memory to reduce processor-memory traffic. In *Proc. of 10th Int. Symp. on Computer Architecture*, pages 124–131, IEEE, 1983.

[Goo86] J.R. Goodman. Cache memory optimization to reduce processor/memory traffic. *Journal of VLSI and Computer Systems*, 2(1), 1986. In press.

[JC85] Baer J.-L. and C.Girault. A Petri Net model for a solution to the cache coherence problem. In *1st Int. Conf. on Supercomputing Systems*, pages 680–689, IEEE, 1985.

[JP80] A.K. Jones and P.Schwartz. Experience using multiprocessor systems: a status report. *Computing Surveys*, 12(2):121–166, June 1980.

[KEW*85] R. Katz, S. Eggers, D.A. Wood, C. Perkins, and R.G.Sheldon. Implementing a cache consistency protocol. In *Proc. of 12th Int. Symp. on Computer Architecture*, pages 276–283, IEEE, 1985.

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE TC*, C-28(9), September 1979.

[McC84] E. McCreight. *The Dragon Computer System An Early Overview*. Technical Report, Xerox Corp., September 1984.

[PJ84] M. Papamarcos and J.Patel. A low overhead coherence solution for multiprocessors with private cache memories. In *Proc. of 11th Int. Symp. on Computer Architecture*, pages 348–354, IEEE, 1984.

[RZ84] L. Rudolph and Z.Segall. Dynamic decentralized cache schemes for MIMD parallel processors. In *Proc. of 11th Int. Symp. on Computer Architecture*, pages 340–347, IEEE, 1984.

[SA86] P. Sweazey and A.J.Smith. A class of compatible cache consistency protocols and their support by the IEEE Futurebus. In *Proc. of 13th Int. Symp. on Computer Architecture*, IEEE, 1986.

[Sat80] M. Satyanaranayan. Commercial multiprocessing systems. *Computer*, 13(5):75–96, 1980.

[Sei85] C. Seitz. The cosmic cube. *CACM*, 28(1):22–33, 1985.

[SFD77] R.J. Swan, S.H. Fuller, and D.P.Siewiorek. Cm*- a modular multimicroprocessor. In *Proc. AFIPS 1977 Nat. Comp. Conf*, pages 637–644, AFIPS, 1977.

[Smi82] A.J. Smith. Cache memories. *Computing Surveys*, 14(3):473–530, September 1982.

[Smi85a] A.J. Smith. Cache evaluation and the impact of workload choice. In *Proc. of 12th Int. Symp. on Computer Architecture*, pages 64–73, IEEE, 1985.

[Smi85b] A.J. Smith. CPU cache consistency with software support and using "one time identifiers". In *Proc. of Pacific Computer Communications Symp.*, 1985.

[Sny82] L. Snyder. Introduction to the configurable, highly parallel computer. *Computer*, 15(1):47–56, January 1982.

[Spe82] A.Z. Spector. Performing remote operations efficiently on a local computer network. *CACM*, 25(4):246–260, April 1982.

[Tan76] C.K. Tang. Cache system design in the tightly coupled multiprocessor system. In *Proc. 1976 AFIPS Nat. Comp. Conf.*, pages 749–753, AFIPS, 1976.

[Tha84] C. Thacker. July 1984. Private communication, Digital Equipment Corp.

[VM86] M.K. Vernon and M.A.Holliday. Performance analysis of multiprocessor cache consistency protocols using generalized timed petri nets. In *Proc. of PERFORMANCE 86 and ACM Sigmetrics 1986*, pages 9–17, 1986.

[Wid79] L.C. Widdoes. *S-1 Multiprocessor Architecture (MULT-1D).* Lawrence Livermore Laboratory, 1979.

[YK82] W.C. Yen and K.S.Fu. Coherence problem in a multicache system. In *Proc. of 1982 Int. Conf. on Parallel Processing*, pages 332–339, IEEE, 1982.

[YYK85] W.C. Yen, D.W.L. Yen, and K.S.Fu. Data coherence problem in a multicache system. *IEEE TC*, C-34(1):56–65, January 1985.

Vita

James K Archibald was born May 1, 1958 in Rexburg, Idaho. He graduated from Weber High School in Ogden, Utah in 1976. In 1981 he received the B.S. degree (summa cum laude in Mathematics) from Brigham Young University. He received an M.S. in Computer Science from the University of Washington in 1983.